



Caché SQL Reference

Version 2017.2
2020-06-26

Caché SQL Reference

Caché Version 2017.2 2020-06-26

Copyright © 2020 InterSystems Corporation

All rights reserved.

InterSystems, InterSystems IRIS, InterSystems Caché, InterSystems Ensemble, and InterSystems HealthShare are registered trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

Email: support@InterSystems.com

Table of Contents

About This Book	1
Symbols and Syntax Conventions	3
Symbols Used in Caché SQL	4
Syntax Conventions Used in this Manual	9
SQL Commands	11
ALTER TABLE	12
ALTER USER	19
ALTER VIEW	21
CALL	23
CASE	27
%CHECKPRIV	30
CLOSE	33
COMMIT	35
CREATE DATABASE	37
CREATE FUNCTION	39
CREATE INDEX	44
CREATE METHOD	51
CREATE PROCEDURE	57
CREATE QUERY	65
CREATE ROLE	70
CREATE TABLE	72
CREATE TRIGGER	93
CREATE USER	102
CREATE VIEW	104
DECLARE	111
DELETE	114
DISTINCT	122
DROP DATABASE	127
DROP FUNCTION	129
DROP INDEX	131
DROP METHOD	134
DROP PROCEDURE	136
DROP QUERY	138
DROP ROLE	140
DROP TABLE	142
DROP TRIGGER	145
DROP USER	147
DROP VIEW	148
FETCH	150
FROM	153
GRANT	163
GROUP BY	170
HAVING	174
INSERT	181
INSERT OR UPDATE	196
INTO	200

%INTRANSACTION	204
JOIN	205
LOCK	213
OPEN	217
ORDER BY	218
REVOKE	224
ROLLBACK	229
SAVEPOINT	232
SELECT	234
SET OPTION	252
SET TRANSACTION	256
START TRANSACTION	260
TOP	265
TRUNCATE TABLE	269
UNION	273
UNLOCK	278
UPDATE	280
USE DATABASE	293
VALUES	294
WHERE	297
WHERE CURRENT OF	306
SQL Predicate Conditions	309
Overview of Predicates	310
ALL	316
ANY	318
BETWEEN	319
%CONTAINS	322
%CONTAINSTERM	326
EXISTS	329
%FIND	330
FOR SOME	332
FOR SOME %ELEMENT	334
IN	337
%INLIST	340
%INSET	344
IS JSON	346
IS NULL	348
LIKE	349
%MATCHES	353
%PATTERN	356
SOME	359
%STARTSWITH	360
SQL Aggregate Functions	367
Overview of Aggregate Functions	368
AVG	373
COUNT	376
%DLIST	381
JSON_ARRAYAGG	384
LIST	388
MAX	391

MIN	393
STDDEV, STDDEV_SAMP, STDDEV_POP	395
SUM	397
VARIANCE, VAR_SAMP, VAR_POP	399
XMLAGG	401
SQL Functions	405
ABS	406
ACOS	407
%ALPHAUP	408
ASCII	410
ASIN	411
ATAN	412
CAST	413
CEILING	419
CHAR	420
CHARACTER_LENGTH	421
CHARINDEX	423
CHAR_LENGTH	425
COALESCE	427
CONCAT	430
CONVERT	431
COS	436
COT	437
CURDATE	438
CURRENT_DATE	439
CURRENT_TIME	440
CURRENT_TIMESTAMP	442
CURTIME	445
DATABASE	447
DATALENGTH	448
DATE	449
DATEADD	451
DATEDIFF	455
DATENAME	460
DATEPART	464
DAY	468
DAYNAME	469
DAYOFMONTH	471
DAYOFWEEK	473
DAYOFYEAR	476
DECODE	478
DEGREES	480
%EXACT	481
EXP	483
%EXTERNAL	485
\$EXTRACT	487
\$FIND	490
FLOOR	493
GETDATE	494
GETUTCDATE	496

GREATEST	498
HOUR	500
IFNULL	502
INSTR	506
%INTERNAL	508
ISNULL	510
ISNUMERIC	513
JSON_ARRAY	515
JSON_OBJECT	518
\$JUSTIFY	521
LAST_DAY	524
LAST_IDENTITY	525
LCASE	527
LEAST	528
LEFT	530
LEN	531
LENGTH	532
\$LENGTH	534
\$LIST	537
\$LISTBUILD	542
\$LISTDATA	545
\$LISTFIND	547
\$LISTFROMSTRING	549
\$LISTGET	551
\$LISTLENGTH	554
\$LISTSAME	557
\$LISTTOSTRING	559
LOG	561
LOG10	562
LOWER	563
LPAD	564
LTRIM	566
%MINUS	567
MINUTE	569
MOD	571
MONTH	572
MONTHNAME	574
NOW	576
NULLIF	578
NVL	580
%OBJECT	583
%ODBCIN	584
%ODBCOUT	585
%OID	586
PI	587
\$PIECE	588
%PLUS	592
POSITION	593
POWER	595
QUARTER	596
RADIANS	598

REPEAT	599
REPLACE	600
REPLICATE	602
REVERSE	603
RIGHT	605
ROUND	606
RPAD	609
RTRIM	610
SEARCH_INDEX	611
SECOND	612
SIGN	614
%SIMILARITY	616
SIN	618
SPACE	619
%SQLSTRING	620
%SQLUPPER	622
SQRT	625
SQUARE	626
STR	627
STRING	628
%STRING	630
STUFF	633
SUBSTR	635
SUBSTRING	637
SYSDATE	639
TAN	640
TIMESTAMPADD	641
TIMESTAMPDIFF	644
TO_CHAR	646
TO_DATE	655
TO_NUMBER	661
TO_TIMESTAMP	663
\$TRANSLATE	668
TRIM	670
TRUNCATE	672
%TRUNCATE	675
\$TSQL_NEWID	677
UCASE	678
UNIX_TIMESTAMP	679
UPPER	681
%UPPER	683
USER	685
WEEK	686
XMLCONCAT	688
XMLELEMENT	689
XMLFOREST	693
YEAR	696
SQL MultiValue Support Functions	699
\$MVFMT	700
\$MVFMTS	702

\$MVICONV	704
\$MVICONVS	705
\$MVOCONV	706
\$MVOCONVS	708
%MVR	709
SQL Unary Operators	711
- (Negative)	712
+ (Positive)	713
SQL Reference Material	715
Data Types	716
Date and Time Constructs	732
Default user name and password	735
Field constraint	736
Reserved words	737
Special Variables	739
String Manipulation	741

List of Tables

Table B-1: SQL Equality Comparison Predicates	176
Table B-2: SQL Equality Comparison Predicates	301
Table B-3: SQL Substring Predicates	302
Table C-1: LIKE Wildcard Characters	349

About This Book

This book provides reference material for various elements of Caché SQL: DDL and DML commands, functions, and predicate conditions, and a list of the symbols, data types, and reserved words used in Caché SQL.

This book contains the following sections:

- [Symbols](#)
- [SQL Commands](#)
- [SQL Predicate Conditions](#)
- [SQL Aggregate Functions](#)
- [SQL Functions](#)
- [SQL MultiValue Support Functions](#)
- [SQL Unary Operators](#)
- [SQL Reference Material](#)

There is also a detailed [Table of Contents](#).

Other related topics in the Caché documentation set are:

- [Using Caché SQL](#) provides in-depth material on SQL components and features, executing SQL queries, error and transaction processing.
- [Caché SQL Optimization Guide](#) describes how to optimize a table definition by [defining and building indices](#), how to use [Tune Table](#) to optimize table metadata based on typical data, and how to [optimize query execution](#) using cached queries, ShowPlan, frozen plans, and other optimization techniques.
- [Using Caché with JDBC](#) describes how to access Caché tables from external applications via JDBC.
- [Using Caché with ODBC](#) describes how to access Caché tables from external applications via ODBC.
- [Caché Advanced Configuration Settings Reference](#) describes the [SQL configuration settings](#).
- [Caché Error Reference](#) lists the [SQLCODE](#) error messages.

For general information, see [Using InterSystems Documentation](#).

Symbols and Syntax Conventions

Symbols Used in Caché SQL

A table of characters used in Caché SQL as operators, etc.

Table of Symbols

The following are the literal symbols used in Caché SQL. (This list does not include symbols indicating [format conventions](#), which are not part of the language.) There is a separate table for [symbols used in ObjectScript](#).

The name of each symbol is followed by its ASCII decimal code value.

Symbol	Name and Usage
[space] or [tab]	<i>White space (Tab (9) or Space (32))</i> : One or more whitespace characters between keywords, identifiers, and variables.
!	<i>Exclamation mark (33)</i> : OR logical operator in between predicates in condition expressions. Used in the WHERE clause, the HAVING clause, and elsewhere.
!=	<i>Exclamation mark/Equal sign</i> : Is not equal to comparison condition .
"	<i>Quotes (34)</i> : Encloses a delimited identifier name. Encloses a string literal (not recommended, use single quotes instead). In Dynamic SQL used to enclose literal values for class method arguments, such as SQL code as a string argument for the %Prepare() method, or input parameters as string arguments for the %Execute() method. In %PATTERN used to enclose a literal value within a pattern string. For example, '3L1"L".L' (meaning 3 lowercase letters, followed by the capital letter "L", followed by any number of lowercase letters). In XMLELEMENT used to enclose a tag name string literal.
""	<i>Double quotes</i> : A literal quotes character within a delimited identifier .
#	<i>Pound sign (35)</i> : Valid identifier name character (not first character). With spaces before and after, modulo arithmetic operator . For Embedded SQL, ObjectScript macro preprocessor directive prefix. For example, #Include. In SQL Shell the # command is used to recall statements from the SQL Shell history buffer.
\$	<i>Dollar sign (36)</i> : Valid identifier name character (not first character). First character of some Caché extension SQL functions.
\$\$	<i>Double dollar sign</i> : used to call a ObjectScript user-defined function (also known as an extrinsic function).

Symbol	Name and Usage
%	<p><i>Percent sign (37)</i>: Valid first character for identifier names (first character only).</p> <p>First character of some Caché SQL extensions to the SQL standard, including string collation functions (%SQLUPPER), aggregate functions (%DLIST), and predicate conditions (%STARTSWITH).</p> <p>First character of %ID, %TABLENAME, and %CLASSNAME keywords in SELECT.</p> <p>First character of some privilege keywords (%CREATE_TABLE, %ALTER) and some role names (%All).</p> <p>First character of some Embedded SQL system variables (%ROWCOUNT, %msg).</p> <p>Data type max length indicator: CHAR(%24)</p> <p>LIKE condition predicate multi-character wildcard.</p>
%%	<p><i>Double percent sign</i>: Prefix for the pseudo-field reference variable keywords: %%CLASSNAME, %%CLASSNAMEQ, %%ID, and %%TABLENAME, used in ObjectScript computed field code and trigger code.</p>
&	<p><i>Ampersand (38)</i>: AND logical operator in WHERE clause and other condition expressions.</p> <p>\$BITLOGIC bitstring And operator.</p> <p>Embedded SQL invocation prefix: &sql(<i>SQL commands</i>).</p>
'	<p><i>Single quote character (39)</i>: Encloses a string literal.</p>
"	<p><i>Double single quote characters</i>: An empty string literal.</p> <p>A literal single quote character within a string value. For example: 'can''t'</p>
()	<p><i>Parentheses (40,41)</i>: Encloses comma-separated lists. Encloses argument(s) of an SQL function. Encloses the parameter list for a procedure, method, or query. In most cases, the parentheses must be specified, even if no arguments or parameters are supplied.</p> <p>In a SELECT DISTINCT BY clause, encloses an item or comma-separated list of items used to select unique values.</p> <p>In a SELECT statement, encloses a subquery in the FROM clause. Encloses the name of a predefined query used in a UNION.</p> <p>Encloses host variable array subscripts. For example, INTO :var(1), :var(2)</p> <p>Encloses embedded SQL code: &sql(code)</p> <p>Used to enforce precedence in arithmetic operations: 3+(3*5)=18. Used to group predicates: WHERE NOT (Age<20 AND Age>12).</p>
(())	<p><i>Double Parentheses</i>: suppress literal substitution in cached queries. For example, SELECT TOP ((4)) Name FROM Sample.Person WHERE Name %STARTSWITH (('A')). Optimizes WHERE clause selection of a non-null outlier value.</p>

Symbol	Name and Usage
*	<p><i>Asterisk (42)</i>: A wildcard, indicating “all” in the following cases: In SELECT retrieve all columns: SELECT * FROM table. In COUNT, count all rows (including nulls and duplicates). In GRANT and REVOKE, all basic privileges, all tables, or all currently defined users.</p> <p>In %MATCHES pattern string a multi-character wildcard.</p> <p>Multiplication arithmetic operator.</p>
/	<p><i>Asterisk slash</i>: Multi-line comment ending indicator. Comment begins with /.</p>
*=	<p><i>Asterisk, equal sign</i>: In WHERE clause, a Right Outer Join.</p>
+	<p><i>Plus sign (43)</i>: Addition arithmetic operator. Unary positive sign operator.</p>
,	<p><i>Comma (44)</i>: List separator, for example, multiple field names.</p> <p>In data size definition: NUMERIC (precision,scale).</p>
–	<p><i>Hyphen (minus sign) (45)</i>: Subtraction arithmetic operator. Unary negative sign operator.</p> <p>SQLCODE error code prefix: –304.</p> <p>Date delimiter.</p> <p>In %MATCHES pattern string a range indicator specified within square brackets. For example, [a–m].</p>
—	<p><i>Double hyphen</i>: Single-line comment indicator.</p>
→	<p><i>Hyphen, greater than (arrow)</i>: implicit join arrow syntax.</p>
.	<p><i>Period (46)</i>: Used to separate parts of multipart names, such as qualified table names: schema.tablename, or column names: tablename.fieldname</p> <p>Decimal point for numeric literals in American numeric format.</p> <p>Date delimiter for Russian, Ukrainian, and Czech locales: DD.MM.YYYY</p> <p>Prefixed to a variable or array name, specifies passing by reference: .name</p> <p>%PATTERN pattern string multi-character wildcard.</p>
/	<p><i>Slash (47)</i>: Division arithmetic operator.</p> <p>Date delimiter.</p>
/*	<p><i>Slash asterisk</i>: Multi-line comment begins indicator. Comment ends with */.</p>
:	<p><i>Colon (58)</i>: Host variable indicator prefix: :var</p> <p>A time delimiter for hours, minutes, and seconds. In CAST and CONVERT functions, an optional thousandth-of-a-second delimiter.</p> <p>In trigger code a prefix indicating an ObjectScript label line.</p> <p>In CREATE PROCEDURE ObjectScript code body, a macro preprocessor directive prefix. For example, :#Include.</p>
::	<p><i>Double colon</i>: In trigger code this doubled prefix indicates that the identifier (::name) beginning that line is a host variable, not a label line.</p>

Symbol	Name and Usage
;	<i>Semicolon (59)</i> : SQL end of statement delimiter in procedures , methods , queries , and trigger code . Accepted as an optional end of statement delimiter by DDLImport() . Otherwise, Caché SQL does not use or allow a semicolon at the end of an SQL statement.
<	<i>Less than (60)</i> : Less than comparison condition .
<=	<i>Less than or equal to</i> : Less than or equal to comparison condition .
<>	<i>Less than/Greater than</i> : Is not equal to comparison condition .
=	<i>Equal sign (61)</i> : Equal to comparison condition . In WHERE clause, an Inner Join .
=*	<i>Equal sign, asterisk</i> : In WHERE clause, a Left Outer Join .
>	<i>Greater than (62)</i> : Greater than comparison condition .
>=	<i>Greater than or equal to</i> : Greater than or equal to comparison condition .
?	<i>Question mark (63)</i> : In Dynamic SQL, an input parameter variable supplied by the Execute method. In %MATCHES pattern string a single-character wildcard. In SQL Shell the ? command displays help text for SQL Shell commands.
@	<i>At sign (64)</i> : Valid identifier name character (not first character).
E, e	<i>The letter "E" (69, 101)</i> : Exponent indicator . %PATTERN code specifying any printable character.
[<i>Open square bracket (91)</i> : Contains predicate . Used in the WHERE clause, the HAVING clause, and elsewhere.
[]	<i>Open and close square brackets</i> : In %MATCHES pattern string, encloses a list or range of match characters. For example, [abc] or [a-m].
\	<i>Backslash (92)</i> : Integer division arithmetic operator . In %MATCHES pattern string an escape character.
]	<i>Close square bracket (93)</i> : Follows predicate . Used in the WHERE clause, the HAVING clause, and elsewhere.
^	<i>Caret (94)</i> : In %MATCHES pattern string a NOT character. For example, [^abc].
_	<i>Underscore (95)</i> : Valid first (or subsequent) character for identifier names. Valid first character for certain user names (but not passwords). Used in column names to represent embedded serial class data: SELECT Home_State, where Home is a field that references a serial class and State is a property defined in that serial class. LIKE condition predicate single-character wildcard.
{ }	<i>Curly braces (123, 125)</i> : Enclose ODBC scalar functions: {fn name(...)}. Enclose time and date construct functions: {d 'string'}, {t 'string'}, {ts 'string'}. Enclose ObjectScript code in procedures , methods , queries , and trigger code .

Symbol	Name and Usage
	<p data-bbox="357 205 954 237"><i>Double vertical bar (124):</i> Concatenation operator.</p> <p data-bbox="357 258 1385 390">Compound ID indicator. Used by Caché as a delimiter between multiple properties in a generated compound object ID (a concatenated ID). This can be either an IDKey index defined on multiple properties (<code>prop1 prop2</code>), or an ID for a parent/child relationship (<code>parent child</code>). Cannot be used in IDKEY field data.</p>

Syntax Conventions Used in this Manual

Specifies conventions used in the SQL Command Reference.

Description

The following are the format conventions used in this manual. These format characters explain usage; they are *not* specified when coding an SQL program. For a table of the symbols that are used in SQL coding, refer to the [SQL Symbols](#) table.

Symbol	Meaning
[<i>nnnn</i>]	An argument enclosed in square brackets is optional. Specify none or one.
{ <i>nnnn</i> }	An argument enclosed in curly braces is optional, and may be repeated multiple times. Specify none, one, or more than one. Curly braces are also used as literal characters, for example in ODBC scalar functions with the form: {fn FUNCTION(<i>arg</i>)}
<i>mmmm</i> <i>nnnn</i>	A vertical bar means OR. Specify either one or the other.
...	An ellipsis indicates an unspecified portion of a complete SQL statement. It can also be used to specify repetition: <i>var1, var2, ...</i>
::=	Is equivalent to.

If an argument appears as an "*item-list*", then the argument can consist of one or more of the particular *items* delimited by a particular character. A cross-reference from an *item-list* points to the page for *item* itself.

If an argument appears as an "*item-commalist*", then the argument can consist of one or more of the particular *items* delimited by a comma. A cross-reference from an *item-commalist* points to the page for *item* itself.

When an item is listed in bracketed parentheses, such as [() *identifier* ()] then the *pair* of parentheses (as a unit) is optional.

SQL Commands

ALTER TABLE

Modifies a table.

```
ALTER TABLE table alter-action

where alter-action is one of the following:
  ADD [(add-action {,add-action} [ ])] |
  DROP drop-action |
  DELETE drop-action |
  ALTER [COLUMN] identifier alter-column-action |
  MODIFY modification-spec

add-action ::=
  [CONSTRAINT table]
  [(FOREIGN KEY (identifier-commalist)
   REFERENCES table (identifier-commalist)
   [referential-action] [ ])]
  |
  [(UNIQUE (identifier-commalist) [ ])]
  |
  [(PRIMARY KEY (identifier-commalist) [ ])]
  |
  DEFAULT [(default-spec [ ])] FOR identifier
  |
  [COLUMN] [(identifier datatype [sqlcollation]
   [%DESCRIPTION literal]
   [DEFAULT [(default-spec [ ])] ]
   [UNIQUE] [NOT NULL]
   [REFERENCES table (identifier-commalist) ]
   [ ])]

drop-action ::=
  FOREIGN KEY identifier |
  PRIMARY KEY |
  CONSTRAINT identifier |
  [COLUMN] identifier [RESTRICT | CASCADE]

alter-column-action ::=
  datatype |
  [SET] DEFAULT [(default-spec[ ])] | DROP DEFAULT |
  NULL | NOT NULL |
  COLLATE sqlcollation

modification-spec ::=
  identifier [datatype]
  [DEFAULT [(default-spec[ ])] ]
  [CONSTRAINT identifier] [NULL] [NOT NULL]

sqlcollation ::=
  { %ALPHAUP | %EXACT | %MINUS | %MVR | %PLUS | %SPACE |
    %SQLSTRING [(maxlen)] | %SQLUPPER [(maxlen)] |
    %STRING [(maxlen)] | %TRUNCATE[(maxlen)] | %UPPER }
```

Arguments

<i>table</i>	The name of the table to be altered. The table name can be qualified (schema.table), or unqualified (table). An unqualified table name takes the system-wide default schema name . Schema search path values are not used.
<i>identifier</i>	The name of the column to be modified. For further details on valid identifiers , see the “Identifiers” chapter of <i>Using Caché SQL</i> .
<i>identifier-commalist</i>	The name of a column or a comma-separated list of columns. An <i>identifier-commalist</i> must be enclosed in parentheses, even when only a single column is specified. For further details on valid identifiers , see the “Identifiers” chapter of <i>Using Caché SQL</i> .
<i>datatype</i>	A valid Caché SQL data type. For a list of valid data types , see the SQL reference material at the end of this manual.
<i>default-spec</i>	A default data value automatically supplied for this field, if not overridden by a user-supplied data value. Allowed values are: a literal value; one of the following keyword options (NULL, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_DATE, CURRENT_TIME, and CURRENT_TIMESTAMP); or an OBJECTSCRIPT expression. Do not use the SQL zero-length string as a default value. For further details, see CREATE TABLE .
COLLATE <i>sqlcollation</i>	<i>Optional</i> — Specify one of the following SQL collation types: %EXACT, %MINUS, %PLUS, %SPACE, %SQLSTRING, %SQLUPPER, %TRUNCATE, or %MVR. The default is the namespace default collation (%SQLUPPER, unless changed). The %ALPHAUP, %STRING, and %UPPER collation types are deprecated and should not be used. %SQLSTRING, %SQLUPPER, %STRING, and %TRUNCATE may be specified with an optional maximum length truncation argument, an integer enclosed in parentheses. The percent sign (%) prefix to these collation parameter keywords is optional. The COLLATE keyword is optional. For further details refer to Table Field/Property Definition Collation in the “Collation” chapter of <i>Using Caché SQL</i> .

Description

An **ALTER TABLE** statement modifies a table; it can add elements, remove elements, or modify existing elements. You can only perform one type of operation in each **ALTER TABLE** statement. However, an **ALTER TABLE ADD** statement can add multiple columns and/or constraints to a table. The **ALTER TABLE DROP** statement and the **ALTER TABLE DELETE** statement are synonyms.

To determine if a specified table exists in the current namespace, use the `$$SYSTEM.SQL.TableExists()` method.

Privileges and Locking

The **ALTER TABLE** command is a privileged operation. Prior to using **ALTER TABLE** it is necessary for your process to have either %ALTER_TABLE administrative privilege or an %ALTER object privilege for the specified table. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has %ALTER privilege by invoking the `%CHECKPRIV` command. You can determine if a specified user has %ALTER privilege by invoking the `$$SYSTEM.SQL.CheckPriv()` method. You can use the **GRANT** command to assign %ALTER_TABLE or %ALTER privileges, if you hold appropriate granting privileges. In embedded SQL, you can use the `$$SYSTEM.Security.Login()` method to log in as a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql( )
```

You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

ALTER TABLE cannot be used on a [table created by defining a persistent class](#), unless the table class definition includes [\[DdlAllowed\]](#). Otherwise, the operation fails with an SQLCODE -300 error with the %msg DDL not enabled for class 'Schema.tablename' >

The **ALTER TABLE** statement acquires a table-level lock on *table*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **ALTER TABLE** operation. When **ALTER TABLE** locks the corresponding class definition, it uses the [SQL Lock Timeout setting](#) for the current process.

ADD COLUMN Restrictions

ALTER TABLE can add a single column, or can add a comma-separated list of columns.

If you attempt to add a field to a table through an **ALTER TABLE** *tablename* ADD COLUMN statement:

- If a column of that name already exists, the statement fails with an SQLCODE -306 error.
- If the statement specifies a NOT NULL constraint on the column *and* there is no default value for the column, then the statement fails if data already exists in the table. This is because, after the completion of the DDL statement, the NOT NULL constraint is not satisfied for all the pre-existing rows. This generates the [error code](#) SQLCODE -304 (Attempt to add a NOT NULL field with no default value to a table which contains data).
- If the statement specifies a NOT NULL constraint on the column *and* there is a default value for the column, the statement updates any existing rows in the table and assigns the default value for the column to the field.
- If the statement DOES NOT specify a NOT NULL constraint on the column *and* there is a default value for the column, then there are no updates of the column in any existing rows. The column value is NULL for those rows.

To change this default NOT NULL constraint behaviors, refer to the COMPILEMODE=NOCHECK option of the [SET OPTION](#) command.

If you specify an ordinary data field named “ID” and the [RowID](#) field is already named “ID” (the default), the ADD COLUMN operation succeeds. **ALTER TABLE** adds the ID data column, and renames the RowId column as “ID1” to avoid duplicate names.

Adding an Integer Counter

If you attempt to add an integer counter field to a table through an **ALTER TABLE** *tablename* ADD COLUMN statement:

- You can add an [IDENTITY](#) field to a table if the table does not have an IDENTITY field. If the table already has an IDENTITY field, the ALTER TABLE operation fails with an SQLCODE -400 error with a %msg such as the following: ERROR #5281: Class has multiple identity properties: 'Sample.MyTest::MyIdent2'. When you use ADD COLUMN to define this field, Caché populates existing data rows for this field using the corresponding [RowID](#) integer values.

If **CREATE TABLE** defined a [bitmap extent index](#) and later you add an IDENTITY field to the table, and the IDENTITY field is not of type %BigInt, %Integer, %SmallInt, or %TinyInt with a MINVAL of 1 or higher, and there is no data in the table, the system automatically drops the bitmap extent index.

- You can add one or more [SERIAL](#) (%Library.Counter) fields to a table. When you use ADD COLUMN to define this field, existing data rows are NULL for this field. You can use **UPDATE** to supply values to existing data rows that are NULL for this field; you cannot use **UPDATE** to change non-NULL values.
- You can add a [ROWVERSION](#) field to a table if the table does not have a ROWVERSION field. If the table already has a ROWVERSION field, the ALTER TABLE operation fails with an SQLCODE -400 error with a %msg such as the following: ERROR #5320: Class 'Sample.MyTest' has more than one property of type %Library.RowVersion. Only one is allowed. Properties: MyVer, MyVer2. When you use ADD COLUMN to define this field, existing data rows are NULL for this field; you cannot update ROWVERSION values that are NULL.

ALTER COLUMN Restriction

You cannot change the [data type](#) of a column that contains data if this change would result in stream data being typed as non-stream data or non-stream data being typed as stream data. Attempting to do so results in an SQLCODE -374 error. If there is no existing data, this type of datatype change is permitted.

If you change the [collation type](#) for a column that contains data, you must [rebuild all indices](#) for that column.

DROP COLUMN Restrictions

Deleting a column definition does not delete any data that has been stored in that column from the data map.

Deleting a column definition does not delete the corresponding column-level privileges. For example, the privilege granted to a user to insert, update, or delete data on that column. This has the following consequences:

- If a column is deleted, and then another column with the same name is added, users and roles will have the same privileges on the new column that they had on the old column.
- Once a column is deleted, it is not possible to revoke object privileges for that column.

For these reasons, it is generally recommended that you use the [REVOKE](#) command to revoke column-level privileges from a column before deleting the column definition.

You cannot drop a column if that column appears in an index, or is defined in a foreign key constraint or other unique constraint. Attempting a DROP COLUMN for that column fails with an SQLCODE -322 error. See [DROP INDEX](#).

You cannot drop a column if that column is used in COMPUTECODE or in a COMPUTEONCHANGE clause. Attempting to do so results in an SQLCODE -400 error.

ADD PRIMARY KEY Restrictions

You cannot add a primary key constraint to an existing field if that field contains non-unique data or permits NULL values.

If you add a primary key constraint to an existing field, the field may also be automatically defined as an IDKey index. This depends on whether data is present and upon a configuration setting established in one of the following ways:

- The SQL [SET OPTION](#) PKEY_IS_IDKEY statement.
- The `$$SYSTEM.SQL.SetDDLPrimaryKeyNotIDKey()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Are Primary Keys Created through DDL not ID Keys**. If set to “Yes” (1), when a Primary Key constraint is specified through DDL it does not automatically become the IDKey index in the class definition. If “No” (0), it does become the IDKey index. Setting this value to “No” can give better performance, but means that the Primary Key fields cannot be updated.

The default is “Yes” (1). If this option is set to “No” (0), and the field does not contain data, the primary key index is also defined as the IDKey index. If this option is set to “No”, and the field *does* contain data, the IDKey index is not defined.

If [CREATE TABLE](#) defined a [bitmap extent index](#) and later you use [ALTER TABLE](#) to add a primary key that is also the IDKey, the system automatically drops the bitmap extent index.

ADD PRIMARY KEY When Already Exists

What happens when you try to add a primary key to a table that already has a defined primary key is configuration-dependent. By default, Caché rejects an attempt to define a primary key when one already exists and issues an SQLCODE -307 error. You can set this behavior as follows:

- The `$$SYSTEM.SQL.SetDDLNo307()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a `Suppress SQLCODE=-307 Errors` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow Create Primary Key Through DDL When Key Exists**.

The default is “No” (0). This is the recommended setting for this option.

If this option is set to “Yes” (1), an **ALTER TABLE ADD PRIMARY KEY** causes Caché to remove the primary key index from the class definition, and then recreates this index using the specified primary key field(s).

However, even if this option is set to allow the creation of a primary key when one already exists, you cannot recreate a primary key index if it is also the IDKEY index and the table contains data. Attempting to do so generates an SQLCODE -307 error.

ADD FOREIGN KEY Restrictions

By default, you cannot have two foreign keys with the same name. Attempting to do so generates an SQLCODE -311 error. This option is configurable using:

- The `$$SYSTEM.SQL.SetDDLNo311()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a `Suppress SQLCODE=-311 Errors` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow DDL ADD Foreign Key Constraint When Foreign Key Exists**.

The default is “No” (0). This is the recommended setting for this option. When “Yes” (1), you can add a foreign key through DDL even if one with the same name already exists. When “No” (0), this action generates an SQLCODE -311 error.

Your table definition should not have two foreign keys with different names that reference the same *identifier-commalist* field(s) and perform contradictory referential actions. In accordance with the ANSI standard, Caché SQL does not issue an error if you define two foreign keys that perform contradictory referential actions on the same field (for example, ON DELETE CASCADE and ON DELETE SET NULL). Instead, Caché SQL issues an error when a **DELETE** or **UPDATE** operation encounters these contradictory foreign key definitions.

An ADD FOREIGN KEY that specifies a non-existent foreign key field generates an SQLCODE -31 error.

An ADD FOREIGN KEY that references a non-existent parent key table generates an SQLCODE -310 error. An ADD FOREIGN KEY that references a non-existent field in an existing parent key table generates an SQLCODE -316 error. If you do not specify a parent key field, it defaults to the ID field.

Before issuing an ADD FOREIGN KEY, the user must have [REFERENCES privilege](#) on the table being referenced or on the columns of the table being referenced. REFERENCES privilege is required if the **ALTER TABLE** is executed via Dynamic SQL or xDBC.

An ADD FOREIGN KEY that references a field (or combination of fields) that can take non-unique values generates an SQLCODE -314 error, with additional details available through %msg.

An ADD FOREIGN KEY is constrained when data already exists in the table. To change this default constraint behavior, refer to the `COMPILEMODE=NOCHECK` option of the [SET OPTION](#) command.

When you define an ADD FOREIGN KEY constraint for a single field and the foreign key references the idkey of the referenced table, Caché converts the property in the foreign key into a reference property. This conversion is subject to the following restrictions:

- The table must contain no data.
- The property on the foreign key cannot be of a persistent class (that is, it cannot already be a reference property).
- The data types and data type parameters of the foreign key field and the referenced idkey field must be the same.

- The foreign key field cannot be an **IDENTITY** field.

For further information on foreign keys, refer to the **CREATE TABLE** command, and to the “Using Foreign Keys” chapter in *Using Caché SQL*.

DROP CONSTRAINT Restrictions

By default, you cannot drop a unique or primary key constraint if it is referenced by a foreign key constraint. Attempting to do so results in an SQLCODE -317 error. To change this default foreign key constraint behavior, refer to the **COMPILE-MODE=NOCHECK** option of the **SET OPTION** command.

The effects of dropping a primary key constraint depend on the setting of the Are Primary Keys also ID Keys setting (as described above):

- If the PrimaryKey index is not also the IDKey index, dropping the primary key constraint drops the index definition.
- If the PrimaryKey index is also the IDKey index, and there is no data in the table, dropping the primary key constraint drops the entire index definition.
- If the PrimaryKey index is also the IDKey index, and there *is* data in the table, dropping the primary key constraint just drops the PRIMARYKEY qualifier from the IDKey index definition.

DROP CONSTRAINT When Non-Existent

What happens when you try to drop a field constraint on a field that does not have that constraint depends on a configuration setting.

- The **\$\$SYSTEM.SQL.SetDDLNo315()** method call. To determine the current setting, call **\$\$SYSTEM.SQL.CurrentSettings()**, which displays a Suppress SQLCODE=-315 Errors setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow DDL DROP of Non-constraint**.

The default is “No” (0). By default, Caché rejects an attempt to drop a constraint that does not exist and issues an SQLCODE -315 error. However, if this option is set to “Yes” (1), an **ALTER TABLE DROP CONSTRAINT** causes Caché to perform no operation and not issue an error message.

Examples

The following example uses two embedded SQL programs to create a table, populate two rows, and then alter the table definition. The **ALTER TABLE** command creates the ColorPreference column and populates it with the value 'Blue' for the two pre-existing rows of the table.

To demonstrate this, please run the two embedded SQL programs in the order shown. (It is necessary to use two embedded SQL programs here because embedded SQL cannot compile an **INSERT** statement unless the referenced table already exists.)

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(DROP TABLE SQLUser.MyStudents)
  IF SQLCODE=0 { WRITE !,"Deleted table" }
  ELSE { WRITE "DROP TABLE error SQLCODE=",SQLCODE }
&sql(CREATE TABLE SQLUser.MyStudents (
  Id      INT NOT NULL,
  Name    VARCHAR(35),
  DOB     DATE,
  CONSTRAINT MyStudentsPK PRIMARY KEY (Id) )
)
  IF SQLCODE=0 { WRITE !,"Created table" }
  ELSE { WRITE "CREATE TABLE error SQLCODE=",SQLCODE }
```

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
NEW SQLCODE, %msg
&sql(INSERT INTO SQLUser.MyStudents (Id, Name, DOB)
VALUES (1, 'David Vanderbilt', 46639))
IF SQLCODE=0 { WRITE !, "Inserted data in table" }
ELSE { WRITE !, "SQLCODE=", SQLCODE, ": ", %msg }
&sql(INSERT INTO SQLUser.MyStudents (Id, Name, DOB)
VALUES (2, 'Mary Smith', 49759))
IF SQLCODE=0 { WRITE !, "Inserted data in table" }
ELSE { WRITE !, "SQLCODE=", SQLCODE, ": ", %msg }
&sql(ALTER TABLE SQLUser.MyStudents
ADD COLUMN ColorPreference %String NOT NULL DEFAULT 'Blue')
IF SQLCODE=0 {
WRITE !, "Altered table, SQLCODE=", SQLCODE }
ELSEIF SQLCODE=-306 {
WRITE !, "SQLCODE=", SQLCODE, ": ", %msg }
ELSE { WRITE "SQLCODE error=", SQLCODE }
```

To view the data, go to the Management Portal, select the Globals option for the SAMPLES namespace. Scroll to “SQLUser.MyStudentsD” and click the Data option.

See Also

- [CREATE TABLE, DROP TABLE](#)
- [JOIN](#)
- [SELECT](#)
- [INSERT, UPDATE, INSERT OR UPDATE, DELETE](#)
- “Defining Tables” chapter in *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

ALTER USER

Changes a user's password.

```
ALTER USER user-name IDENTIFY BY password
ALTER USER user-name IDENTIFIED BY password
```

Arguments

<i>user-name</i>	The name of an existing user whose password is to be changed. If support for delimited identifiers is on and the user name begins with an underscore, you must place the user name in quotation marks. User names are not case-sensitive.
<i>password</i>	The new password for the user. A password must be at least 3 characters and cannot exceed 32 characters. Passwords are case-sensitive. Passwords can contain Unicode characters.

Description

The **ALTER USER** command allows you to change a user's password. You can always change your own password. To change another user's password, you must have the %Admin_Secure:USE system permission.

The IDENTIFY BY and IDENTIFIED BY keywords are synonyms.

The *user-name* must be an existing user. Specifying a non-existent user generates an SQLCODE -400 error with a %msg such as the following: ERROR #838: User badname does not exist.

A *password* can be a string literal, a numeric, or an identifier. A string literal must be enclosed in quotes, and can contain any combination of characters, including blank spaces. A numeric or an identifier does not have to be enclosed in quotes. A numeric must consist of only the characters 0 through 9. An [identifier](#) must start with a letter (uppercase or lowercase) or a % (percent symbol); this can be followed by any combination of letters, numbers, or any of the following symbols: _ (underscore), & (ampersand), \$ (dollar sign), or @ (at sign).

ALTER USER does not issue an error code if the new password is identical to the existing password. It sets SQLCODE = 0 (Successful Completion).

You can also change a user password using the **\$\$SYSTEM.Security.ChangePassword()** method:

```
$$SYSTEM.Security.ChangePassword(args)
```

Privileges

The **ALTER USER** command is a privileged operation. Prior to using **ALTER USER** in embedded SQL, it is necessary to be logged in as a user with appropriate privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation). Use the **\$\$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql( )
```

You must have the %Service_Login:Use privilege to invoke the **\$\$SYSTEM.Security.Login** method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

Examples

The following embedded SQL example changes the password of user Bill from “temp_pw” to “pw4AUser”:

```
Main
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(CREATE USER Bill IDENTIFY BY temp_pw)
IF SQLCODE=0 { WRITE !,"Created user" }
ELSE { WRITE "CREATE USER error SQLCODE=",SQLCODE,! }
&sql(ALTER USER BILL IDENTIFY BY pw4AUser)
IF SQLCODE=0 { WRITE !,"Altered user password" }
ELSE { WRITE "ALTER USER error SQLCODE=",SQLCODE,! }
Cleanup
SET toggle=$RANDOM(2)
IF toggle=0 {
  &sql(DROP USER Bill)
  IF SQLCODE=0 { WRITE !,"Dropped user" }
  ELSE { WRITE "DROP USER error SQLCODE=",SQLCODE }
}
ELSE {
  WRITE !,"No drop this time"
  QUIT
}
```

As stated above, if support for [delimited identifiers](#) is on and the user name begins with an underscore, you must place the user name in quotation marks, such as:

```
ALTER USER "_ADMIN" IDENTIFY BY myPW4now
```

See Also

- SQL statements: [CREATE USER](#), [DROP USER](#), [GRANT](#), [REVOKE](#)
- “[Users, Roles, and Privileges](#)” chapter of *Using Caché SQL*
- ObjectScript: [\\$ROLES](#) and [\\$USERNAME](#) special variables
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

ALTER VIEW

Modifies a view.

```
ALTER VIEW view-name [(column-commalist)] AS query [WITH READ ONLY]
ALTER VIEW view-name [(column-commalist)] AS query [WITH [level] CHECK OPTION]
```

Arguments

<i>view-name</i>	The view being modified, which has the same naming rules as a table name . A view name can be qualified (schema.viewname), or unqualified (viewname). An unqualified view name takes the system-wide default schema name .
<i>column-commalist</i>	<i>Optional</i> — The column names that compose the view. If not specified here, the column names can be specified in the <i>query</i> , as shown below.
<i>query</i>	The result set (from a query) that serves as a the basis for the view.
WITH READ ONLY	<i>Optional</i> — Specifies that no insert, update, or delete operations can be performed through this view upon the table on which the view is based. The default is to permit these operations through a view, subject to the constraints described below.
WITH <i>level</i> CHECK OPTION	<i>Optional</i> — Specifies how insert, update, or delete operations are performed through this view upon the table on which the view is based. The <i>level</i> can be the keywords LOCAL or CASCADED. If no <i>level</i> is specified, the WITH CHECK OPTION default is CASCADED. For further details, refer to CREATE VIEW .

Description

The **ALTER VIEW** command allows you to modify a [view](#). A view is based on the result set from a query consisting of a **SELECT** statement or a **UNION** of two or more **SELECT** statements. See [CREATE VIEW](#) for further information on using a **UNION**.

To determine if a specified view exists in the current namespace, use the **\$SYSTEM.SQL.ViewExists()** method.

The optional *column-commalist* specifies the names of the columns included in the view. They must correspond in number and sequence with the table columns specified in the **SELECT** statement. You can also specify these view column names as column name aliases in the **SELECT** statement. If you specify neither, the table column names are used as the view column names.

These two ways to specify view column names are shown in the following examples:

```
ALTER VIEW MyView (MyViewCol1,MyViewCol2,MyViewCol3) AS
SELECT TableCol1, TableCol2, TableCol3 FROM MyTable
```

is the same as:

```
ALTER VIEW MyView AS SELECT TableCol1 AS ViewCol1,
TableCol2 AS ViewCol2,
TableCol3 AS ViewCol3
FROM MyTable
```

The column specification replaces any prior columns specified for the view.

A view query cannot contain [host variables](#) or include the [INTO](#) keyword. If you attempt to reference a host variable in *query*, the system generates an SQLCODE -148 error.

Privileges

The **ALTER VIEW** command is a privileged operation. Prior to using **ALTER VIEW** it is necessary for your process to have either %ALTER_VIEW administrative privilege or an %ALTER object privilege for the specified view. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has %ALTER privilege by invoking the %CHECKPRIV command. You can determine if a specified user has %ALTER privilege by invoking the \$SYSTEM.SQL.CheckPriv() method. You can use the **GRANT** command to assign %ALTER_VIEW or %ALTER privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the \$SYSTEM.Security.Login() method to log in as a user with appropriate privileges:

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( )
```

You must have the %Service_Login:Use privilege to invoke the \$SYSTEM.Security.Login method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

Examples

The following examples create a view then alter that view. Programs are provided to query the view and to delete the view. Note that altering the view replaces the column list with a new column list; it does not preserve the prior column list.

```
IF $SYSTEM.SQL.ViewExists("MassFolks")
  {WRITE "View already exists, please run DROP VIEW" QUIT}
&sql(CREATE VIEW MassFolks (vFullName) AS
  SELECT Name FROM Sample.Person WHERE Home_State='MA')
IF SQLCODE=0 { WRITE !,"Created a view",! }
ELSE { WRITE "CREATE VIEW error SQLCODE=",SQLCODE }
```

```
SELECT * FROM MassFolks
```

```
IF 0=$SYSTEM.SQL.ViewExists("MassFolks")
  {WRITE "View doesn't exist" QUIT}
&sql(ALTER VIEW MassFolks (vMassAbbrev,vCity) AS
  SELECT Home_State,Home_City FROM Sample.Person WHERE Home_State='MA')
IF SQLCODE=0 { WRITE !,"Altered view",! }
ELSE { WRITE "ALTER VIEW error SQLCODE=",SQLCODE }
```

```
DROP VIEW MassFolks
```

The following embedded SQL example alters a view using a query WITH CHECK OPTION:

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(ALTER VIEW Sample.MyTestView AS
  SELECT FIRSTWORD FROM Sample.MyTest WITH CHECK OPTION)
IF SQLCODE=0 { WRITE !,"Altered view" }
ELSE { WRITE "ALTER VIEW error SQLCODE=",SQLCODE }
```

See Also

- [CREATE VIEW DROP VIEW GRANT](#)
- “Defining Views” chapter in *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

CALL

Invokes a stored procedure.

```
CALL procname(arg1,arg2,...) [USING contextvar]
retval=CALL procname(arg1,arg2,...) [USING contextvar]
```

Arguments

<i>procname</i>	The name of a stored procedure. May be a fully qualified name (including schema), or an unqualified name for which Caché supplies the schema name, as described below.
<i>arg1</i>	<i>Optional</i> — The actual argument list. One, or more values to pass to a stored procedure, specified in order as a comma-separated list. The parentheses are required, even when no arguments are specified.
USING <i>contextvar</i>	<i>Optional</i> — <i>contextvar</i> specifies a descriptor area variable that receives the procedure context object generated by the procedure call. If omitted, the default is %sqlcontext.
<i>retval</i>	<i>Optional</i> — A variable specified to receive the procedure return value. Can contain a single value, not a result set. Can be specified as a local variable, a host variable, or a question mark (?) argument.

Description

A **CALL** statement invokes a query exposed as an SQL stored procedure. The *procname* must be an existing stored procedure in the current namespace. If Caché cannot locate *procname*, it generates an SQLCODE -428 error. The *procname* must be a Stored Procedure with SqlProc=True. Refer to [SqlProc](#) in *Caché Class Definition Reference*.

The *procname* is not case-sensitive. You must append the argument parentheses to the *procname*, even if you are not specifying any arguments. Failing to do so results in an SQLCODE -1 error.

The *procname* can be either qualified (`schema.procname`) or unqualified (`procname`). Caché locates the match for an unqualified *procname* in a schema, using either the [system-wide default schema name](#), or (if provided) a schema name from the [schema search path](#). If Caché cannot locate the specified procedure using either the schema search path or the system-wide schema default, it generates an SQLCODE -428 error.

The **CALL** *arg* arguments are optional; the parentheses are required. This comma-separated list is known as the actual argument list, which must match in number and in sequence the formal argument list in the procedure definition. You may specify fewer actual argument values than the formal arguments defined in the stored procedure. If you specify more actual argument values than the formal arguments defined in the stored procedure, the system generates an SQLCODE -370 error. This error message specifies the name of the stored procedure, the number of arguments specified, and the number of arguments defined in the stored procedure.

You can omit trailing arguments; any missing trailing arguments are undefined and take default values. You can specify an undefined argument within the argument list by specifying a placeholder comma. For example, (*arg1,,arg3*) passes three arguments, the second of which is undefined. Commonly, undefined arguments take a default value that was specified when defining the stored procedure. If no default is defined, an undefined argument takes NULL. For further details refer to [NULL and the Empty String](#) in *Using Caché SQL*.

If you specify an argument value that does not match the data type defined in the stored procedure that argument takes NULL, even if a default value is defined. For example, a stored procedure defines an argument as `IN numarg INT`

DEFAULT 99. If **CALL** specifies a numeric argument, that *arg* value is used. If **CALL** omits the argument, the defined default is used. However, if **CALL** specifies a non-numeric argument, NULL is used, not the defined default.

For further details on stored procedures, refer to the [CREATE PROCEDURE](#) command.

From Embedded SQL

ObjectScript embedded SQL can either issue a **CALL** statement, or use the **DO** command to invoke the underlying routine or method.

Using Embedded SQL, you can supply argument values to **CALL** as literals or by using any combination of :name [host variables](#) or question mark (?) [input parameters](#), as follows:

```
SET a=7,b="A",c=99
&sql(CALL MyProc(:a,:b,:c))

&sql(CALL MyProc(?, :b, ?))
```

The initial invocation of a **CALL** statement in Embedded SQL creates an %sqlcontext variable, by default. Subsequent iterations use this existing %sqlcontext variable. This means that multiple iterations accumulate results in %sqlcontext that could potentially result in a <STORE> error. If a **CALL** statement is to be iterated repeatedly, you can explicitly specify the %sqlcontext variable in the USING clause. When a procedure context is specified in the USING clause Caché issues a [NEW](#) on that procedure context each time it is invoked.

A host variable used for an output *arg* can be a single value, an array reference, an oref.property reference, or a multidimensional oref.property reference.

You can return a value from a **CALL** statement by using either a host variable or a question mark (?):

```
&sql(:rtnval=CALL MyProc())

&sql(?=CALL MyProc())
```

The **CALL** return value must be a single value. You cannot return a result set from a **CALL** statement in Embedded SQL. Attempting to use `retval=CALL` syntax for a procedure that does not return a value generates an SQLCODE -371 error.

You can use the [#SQLCompile Path](#) macro directive to provide a schema search path that **CALL** can use to resolve an unqualified procedure name. If no [#SQLCompile Path](#) directive is defined, Caché uses the schema that maps to the package of the current class. If Caché cannot locate the specified procedure using either the schema search path or the schema default, it generates an SQLCODE -428 error.

For further details, refer to the [Embedded SQL](#) chapter of *Using Caché SQL*.

From Dynamic SQL

The following Dynamic SQL example calls the Stored Procedure `Sample.PersonSets`, which performs two queries on the `Sample.Person` table. The Stored Procedure arguments specify the WHERE clause values for these two queries. The first argument specifies to return all records in the first query where Name starts with *arg1* (in this case, the letter “M”). The second argument specifies to return all records in the second query where `Home_State = arg2` (in this case, “MA”):

```
ZNSPACE "Samples"
SET mycall = "CALL Sample.PersonSets(?, 'MA')"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(mycall)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("M")
IF rset.%SQLCODE '= 0 {WRITE "SQL error=", rset.%SQLCODE QUIT}
DO rset.%Display()
```

The following Dynamic SQL example also calls the Stored Procedure `Sample.PersonSets`, returning the result sets for each query separately. The `%Next()` method iterates through the first query result set. The `%MoreResults()` method accesses the result set for the second query. If there were more than two queries, `%MoreResults()` would access each result set in turn.

```

ZNSPACE "Samples"
SET mycall = "CALL Sample.PersonSets(?, 'MA')"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(mycall)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("M")
IF rset.%SQLCODE '= 0 {WRITE "SQL error=", rset.%SQLCODE QUIT}
FirstResultSet
WHILE rset.%Next() {
  WRITE "Name: ", rset.%Get("Name")
  IF rset.%Get("Spouse") {
    WRITE " Spouse: ", rset.%Get("Spouse"), !}
  ELSE {WRITE " unmarried", !}
}
WRITE !, "1st row count=", rset.%ROWCOUNT, !!
SecondResultSet
WHILE rset.%MoreResults() {
  DO rset.%CurrentResult.%Display()
}

```

Note that it is important to check the %SQLCODE value set by the **CALL** execution before invoking %Next(). Invoking the %Next() method sets %SQLCODE, overwriting the prior **CALL** %SQLCODE value. If %Next() receives no result set data, it sets %SQLCODE=100. It does not distinguish between an empty result set (no rows selected) and a nonexistent result set due to an error in **CALL** processing.

For further details on %SQL.Statement and on how to display a list of formal parameters and other metadata for a stored procedure, refer to the “Using Dynamic SQL” chapter of *Using Caché SQL*. The [Returning the Full Result Set](#) section of the “Using Dynamic SQL” chapter provides further information and examples of the %Display() method. The [Returning Specific Values from the Result Set](#) section of the “Using Dynamic SQL” chapter provides further information and examples of the %Next() and %Get() methods.

From ObjectScript

Rather than calling stored procedures directly from embedded SQL, you can invoke stored procedures through ObjectScript calls to the class methods that contain them. In this case, you have to manage the parameters, and with query-based stored procedures, the separate methods have to be called and the fetch loop managed.

For example, to call a method exposed as a stored procedure called UpdateAllAvgScores that has no arguments, the code is:

```

NEW phnd
SET phnd=##class(%SQLProcContext).%New()
DO ##class(students).UpdateAllAvgScores(phnd)
IF phnd.%SQLCODE {QUIT phnd.%SQLCODE}
USE 0
WRITE !, phnd.%ROWCOUNT, " Rows Affected"

```

When specifying a procedure’s arguments in the call statement, you must *not* specify the %Library.SQLProcContext parameter if the procedure has an explicitly defined %Library.SQLProcContext parameter. The handling of this parameter is done automatically.

In the following example, the stored procedure takes two arguments. It has an explicitly defined procedure context.

```

ZNSPACE "Samples"
NEW phnd
SET phnd=##class(%SQLProcContext).%New()
SET rtn=##class(Sample.ResultSets).PersonSets("D", "NY")
IF phnd.%SQLCODE {QUIT phnd.%SQLCODE}
DO %sqlcontext.%Display()
WRITE !, "All Done"

```

To call a stored procedure that has been implemented as a query, you must call all three methods:

```

NEW qhnd
DO ##class(students).GetAvgScoreExecute(.qhnd, x1)
NEW avgrow, AtEnd
SET avgrow=$lb(" ")
SET AtEnd=0
DO ##class(students).GetAvgScoreFetch(.qhnd, .avgrow, .AtEnd)
SET x5=$lg(avgrow, 1)
DO ##class(students).GetAvgScoreClose(qhnd)

```

If a query-based stored procedure is to be nested within a number of other stored procedures, it is useful to write a wrapper method to hide all of this.

From ODBC or JDBC

Caché fully supports **CALL** syntax as defined by the ODBC 2.x and JDBC 1.0 standards. In JDBC, you can invoke **CALL** through the methods of the `CallableStatement` class. In ODBC, there are APIs. The **CALL** syntax and semantics are exactly the same for JDBC and ODBC. Further, they are processed in the same way: both drivers parse the statement text and, if the statement is **CALL**, they directly invoke the special methods on the server side, bypassing the SQL engine.

If class `PERSON` has a stored procedure called `SP1`, then you can call this from an ODBC or JDBC client (such as Microsoft Query) as follows:

```
retcode = SQLExecDirect(hstmt, "{?=call PERSON_SP1(?,?)}", SQL_NTS);
```

Caché conforms to the ODBC standard in its structure for calling stored procedures. See the relevant documentation for more information on that standard.

With ODBC only, Caché allows relaxed syntax for calls, so there do not need to be curly braces around **CALL** or parentheses around parameters. (Since this is good programming form, the above example uses them.)

Again, with ODBC only, Caché allows modified syntax for using default parameters, so that **CALL SP** is different from **CALL SP()**. The second form implies passing of a default parameter — as does `CALL SP (,)` or `SP(, ? ,)` or other such syntax. In that sense, the parenthesized form of **CALL** is different from non-parenthesized.

See Also

- SQL statements: [CREATE PROCEDURE](#) [CREATE QUERY](#) [CREATE METHOD](#)
- ObjectScript: [DO](#) command
- “[Defining and Using Stored Procedures](#)” chapter in *Using Caché SQL*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

CASE

Chooses one of a specified set of values depending on some condition.

```

CASE
  WHEN search_condition THEN value_expression
  [ WHEN search_condition THEN value_expression ... ]
  [ ELSE value_expression ]
END

CASE value_expression
  WHEN value_expression THEN value_expression
  [ WHEN value_expression THEN value_expression ... ]
  [ ELSE value_expression ]
END

```

Arguments

<i>search_condition</i>	An SQL boolean expression.
<i>value_expression</i>	An SQL expression (such as a literal value or field name.)

Description

The **CASE** expression allows you to make comparison tests on series of values, returning when it encounters the first match.

The **CASE** expression comes in two forms: Simple and Searched.

The Simple **CASE** expression tests a series of value *expressions* (specified by a **WHEN** clause) to see if they are equal to a given value expression:

```

SELECT
CASE Field1
  WHEN 1 THEN 'ONE'
  WHEN 2 THEN 'TWO'
  ELSE NULL
END
FROM MyTable

```

The value associated with the first matching expression is returned as the value of the **CASE** expression.

Numeric *value_expression* values may have different data types. The data type returned is the type most compatible with all of the possible result values, the data type with the highest [data type precedence](#). For numeric *value_expression* values **CASE** returns the largest length, precision, and scale from all of the possible result values. A result value of NULL has the lowest data type precedence; however, if all result values are NULL, the data type returned is VARCHAR.

The Searched **CASE** expression tests a series of search *conditions* (specified by a **WHEN** clause), finds the first **WHEN** condition that evaluates to true, and returns the value associated with it:

```

SELECT
CASE
  WHEN Field1 = 1 THEN 'ONE'
  WHEN Field1 = 2 THEN 'TWO'
  ELSE NULL
END
FROM MyTable

```

With either form of **CASE** expression, you can use an **ELSE** clause to specify what value to return if none of the **WHEN** clause conditions are true. If you omit the **ELSE** clause and none of the **WHEN** clause conditions are true, **CASE** returns NULL.

A **CASE** comparison tests for NULL must use the IS NULL or IS NOT NULL keyword phrase. NULL is *not* a data value (it represents the absence of a value). For this reason, any equality or arithmetic test for NULL always return false. A **CASE**

expression that compares NULL and any data value always returns false. For example, NULL < 1 and NULL > 1 both return false. A **CASE** expression that equates NULL with NULL also returns false.

The end of a **CASE** expression is marked by an END token.

You can use the **GREATEST** and **LEAST** functions to return the value from a series of values that is the largest or smallest (collates highest or lowest) of the provided values.

Examples

The following query is an example of a Simple **CASE** expression, where specified field values are replaced by supplied values. Note the use of the RetireAge column alias after the END keyword; the optional AS keyword is omitted in this example:

```
SELECT Name,
CASE Age
  WHEN 65 THEN 'Retire this year'
  WHEN 64 THEN 'Retire next year'
  ELSE 'Past retirement age ' || Age
END RetireAge
FROM Sample.Person
WHERE Age > 63
ORDER BY Age
```

The following query is another example of a Simple **CASE** expression. This query labels rows with certain Home_State values as either “Northern NE” or “Southern NE”, and sets all other Home_State values in this column to NULL. It uses the As clause to label this column as “NewEnglanders”, and also displays Names and the original Home_State values. The resulting rows are ordered first by the NewEnglanders column (in descending order), and within this alphabetically by Home_State, and then by Name.

```
SELECT Name,
CASE Home_State
  WHEN 'VT' THEN 'Northern NE'
  WHEN 'NH' THEN 'Northern NE'
  WHEN 'ME' THEN 'Northern NE'
  WHEN 'MA' THEN 'Southern NE'
  WHEN 'CT' THEN 'Southern NE'
  WHEN 'RI' THEN 'Southern NE'
  ELSE NULL
END AS NewEnglanders, Home_State
FROM Sample.Person
ORDER BY NewEnglanders DESC, Home_State, Name
```

The following query is an example of a Searched **CASE** expression. It uses logical operators (greater than (>), logical AND (&), logical OR (!)) to specify a boolean statement for each **WHEN** clause. The first **WHEN** clause that tests True sets the value expression that follows the **THEN** keyword. In this example, the Age and Home_State field values are used to identify three types of Yankees: Old Yankees, Yankees (residents of the six New England states), and likely fans of the New York Yankees baseball team:

```
SELECT Name,
CASE
WHEN Age > 55 & Home_State = 'VT'
  ! Home_State='ME' ! Home_State='NH'
  ! Home_State='MA' ! Home_State='CT'
  ! Home_State='RI'
THEN 'Old Yankee'
WHEN Home_State = 'VT'
  ! Home_State='ME' ! Home_State='NH'
  ! Home_State='MA' ! Home_State='CT'
  ! Home_State='RI'
THEN 'Yankee'
WHEN Home_State='NY' THEN 'Yankees Fan'
ELSE Home_State
END AS Yankees
FROM Sample.Person
```

The following example shows that any comparison with NULL always returns false:

```

SELECT TOP 5 Name,
CASE NULL
  WHEN NULL THEN 'Null = Null'
  WHEN 0 THEN 'Null = 0'
  WHEN '' THEN 'Null = empty string'
  WHEN CHAR(0) THEN 'Null = CHAR(0)'
  ELSE 'Null Arithmetic Invalid'
END
FROM Sample.Person

```

The following example shows how to use **CASE** with a field that has NULLs:

```

SELECT TOP 20 Name,
CASE
  WHEN FavoriteColors IS NULL THEN 'No Colors'
  ELSE $LISTTOSTRING(FavoriteColors,':')
END
FROM Sample.Person

```

CASE is not limited to use in queries, as shown in the following example:

```

ZNSPACE "SAMPLES"
SET myin=3
SET myin(1) = "INSERT INTO SQLUser.MyStudents (Name,PxTs) VALUES "
SET myin(2) = "(CASE ? WHEN 'a' THEN 'Alice' WHEN 'b' THEN 'Barney' ELSE 'Unknown' END,"
SET myin(3) = "CURRENT_TIMESTAMP)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myin)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("a")
DO rset.%Display()

```

See Also

- SQL functions: [DECODE](#), [GREATEST](#), [LEAST](#), [NULLIF](#), [COALESCE](#)
- ObjectScript function: [\\$CASE](#)

%CHECKPRIV

Checks whether the user holds a specified privilege.

```
%CHECKPRIV [GRANT OPTION FOR | ADMIN OPTION FOR] syspriv [,syspriv]
%CHECKPRIV [GRANT OPTION FOR] objpriv ON object
%CHECKPRIV column-privilege (column-list) ON table
```

Arguments

GRANT OPTION FOR	<i>Optional</i> — This keyword phrase specifies checking whether the current user holds the WITH GRANT OPTION privilege on the specified privilege(s). A %CHECKPRIV with this option <i>does not</i> check whether the user holds the specified privilege(s) itself.
ADMIN OPTION FOR	<i>Optional</i> — This keyword phrase specifies checking whether the current user can grant the specified system privilege(s) to other users or roles. A %CHECKPRIV with this option <i>does not</i> check whether the user holds the specified privilege(s) itself.
<i>syspriv</i>	<p>A system privilege, or a comma-separated list of system privileges. The available <i>syspriv</i> options include sixteen object definition privileges and four data modification privileges.</p> <p>The object definition privileges are: %CREATE_FUNCTION, %DROP_FUNCTION, %CREATE_METHOD, %DROP_METHOD, %CREATE_PROCEDURE, %DROP_PROCEDURE, %CREATE_QUERY, %DROP_QUERY, %CREATE_TABLE, %ALTER_TABLE, %DROP_TABLE, %CREATE_VIEW, %ALTER_VIEW, %DROP_VIEW, %CREATE_TRIGGER, %DROP_TRIGGER. Alternatively, you can specify %DB_OBJECT_DEFINITION, which tests all 16 object definition privileges.</p> <p>The data modification privileges are the %NOCHECK, %NOINDEX, %NOLOCK, %NOTRIGGER privileges for INSERT, UPDATE, and DELETE operations.</p>
<i>objpriv</i>	An object privilege associated with a specified <i>object</i> . The available options are: %ALTER, DELETE, SELECT, INSERT, UPDATE, EXECUTE, and REFERENCES.
<i>object</i>	The name of the object for which the <i>objpriv</i> is being checked.
<i>column-privilege</i>	A column-level privilege associated with one or more listed columns. Available options are SELECT, INSERT, UPDATE, and REFERENCES.
<i>column-list</i>	A list of one or more column names for which privilege assignment is being checked, separated by commas and enclosed in parentheses. A space may be included or omitted between the <i>column-privilege</i> name and the opening parenthesis.

<i>table</i>	The name of the table or view that contains the <i>column-list</i> columns. A table name or view name can be qualified (schema.tablename), or unqualified (tablename). An unqualified name takes the system-wide default schema name . A schema search path cannot be present when invoking %CHECKPRIV .
--------------	---

Description

%CHECKPRIV can be used in two ways:

- To determine if the current user holds a system privilege of a specified type.
- To determine if the current user holds a user privilege of a specified type on a specified object. These objects can include table-level privileges on tables or views, column-level privileges on specified columns, and privileges on stored procedures.

%CHECKPRIV enables you to check whether a privilege is held. It does not enforce privileges. At runtime privileges are enforced through ODBC/JDBC and through Dynamic SQL (for example, through the Management Portal **Execute SQL Statement**). Privileges are not enforced for Embedded SQL. **%CHECKPRIV** is primarily used for Embedded SQL.

If the user holds the specified privilege, **%CHECKPRIV** sets SQLCODE=0. If the user does not hold the specified privilege, **%CHECKPRIV** sets SQLCODE=100.

Because **%CHECKPRIV** requires access to the SQLCODE 100 value (an SQLCODE status value, not an SQLCODE error value) to determine its result, it cannot be directly used by JDBC and other clients that can only distinguish error or no error status.

Because **%CHECKPRIV** prepares and executes quickly, and is generally run only once, Caché does not create a cached query for **%CHECKPRIV**.

You can determine if a specified user has a specified table-level privilege by invoking the **\$\$SYSTEM.SQL.CheckPriv()** method.

Embedded SQL and Privileges

Privileges are not automatically checked or enforced for [Embedded SQL](#). Therefore, an Embedded SQL program should (in most cases) call **%CHECKPRIV** before attempting a privileged operation, such as an update:

```
SET name="fred",age=99
SET SQLCODE=""
&sql(%CHECKPRIV UPDATE ON Sample.Person)
IF SQLCODE=100 {
  WRITE !,"No UPDATE privilege"
  QUIT }
ELSEIF SQLCODE < 0 {
  WRITE !,"Unexpected SQL error: ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"Proceeding with UPDATE" }
&sql(UPDATE Sample.Person SET Name=:name,Age=:age)
IF SQLCODE=0 { WRITE !,"UPDATE successful" }
ELSE { WRITE "UPDATE error SQLCODE=",SQLCODE }
```

Examples

The following Embedded SQL example checks whether the current user holds a specific object privilege for a specific table:

```
&sql(%CHECKPRIV UPDATE ON Sample.Person)
IF SQLCODE=0 {WRITE "Have privilege"}
ELSEIF SQLCODE=100 {WRITE "Do not have privilege"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE}
```

The following Embedded SQL example checks whether the current user holds system privileges on the three table operations for all tables:

```
&sql(%CHECKPRIV %CREATE_TABLE,%ALTER_TABLE,%DROP_TABLE)
IF SQLCODE=0 {WRITE "Have privileges"}
ELSEIF SQLCODE=100 {WRITE "Do not have one or more privileges"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE}
```

The following Embedded SQL example checks whether the current user holds all 16 object definition privileges. The SQLCODE value is set to either 0 (holds all 16 privileges) or 100 (does not hold one or more of the 16 privileges):

```
&sql(%CHECKPRIV %DB_OBJECT_DEFINITION)
IF SQLCODE=0 {WRITE "Have all privileges"}
ELSEIF SQLCODE=100 {WRITE "Do not have one or more privileges"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE}
```

The following Embedded SQL example checks whether the current user can grant the %CREATE_TABLE privilege to other users or roles:

```
&sql(%CHECKPRIV ADMIN OPTION FOR %CREATE_TABLE)
IF SQLCODE=0 {WRITE "Have admin option on privilege"}
ELSEIF SQLCODE=100 {WRITE "Do not have admin option on privilege"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE}
```

The following Embedded SQL example checks whether the current user holds the specified column-level privileges. Following the name of the privilege, specify the name of a column (or a comma-separated list of columns) in parentheses:

```
&sql(%CHECKPRIV UPDATE(Name,Age) ON Sample.Person)
IF SQLCODE=0 {WRITE "Have privilege on all specified columns"}
ELSEIF SQLCODE=100 {WRITE "Do not have privilege on one or more specified columns"}
ELSE {WRITE "Unexpected SQLCODE error: ",SQLCODE}
```

See Also

- SQL statements: [GRANT REVOKE](#)
- “Users, Roles, and Privileges” chapter of *Using Caché SQL*
- ObjectScript: [\\$ROLES](#) and [\\$USERNAME](#) special variables

CLOSE

Closes a cursor.

```
CLOSE cursor-name
```

Arguments

<i>cursor-name</i>	The name of the cursor to be closed. The cursor name was specified in the DECLARE statement. Cursor names are case-sensitive.
--------------------	--

Description

A **CLOSE** statement shuts down an open [cursor](#). It releases the current result set and frees any cursor locks held on the rows on which the cursor is positioned. However, **CLOSE** does not delete the cursor; it leaves the data structures accessible for reopening, but fetches and positioned updates are not allowed until the cursor is reopened. This behavior is demonstrated by the following command sequences:

- **DECLARE c1, OPEN c1, FETCH c1, CLOSE c1** is the standard sequence.
- **DECLARE c1, OPEN c1, CLOSE c1, OPEN c1** reopens the declared cursor c1.
- **DECLARE c1, OPEN c1, CLOSE c1, DECLARE c1, OPEN c1** reopens the cursor specified in the first **DECLARE**, the second **DECLARE** is ignored.
- **DECLARE c1, OPEN c1, FETCH c1, CLOSE c1, OPEN c1, FETCH c1** cause both fetch operations to retrieve the same record.

CLOSE must be issued on an open cursor. Issuing a **CLOSE** on a cursor that has only been declared (but not opened), or on a cursor that has already been closed results in an `SQLCODE -102` error. Issuing a **CLOSE** on a non-existent cursor — for example, a cursor that differs from the defined cursor in letter case — results in a `<SYNTAX>` error.

The *cursor-name* is not namespace-specific. Changing the current namespace has no effect on use of a declared cursor. The only namespace consideration is that **FETCH** must occur in the namespace that contains the table(s) being queried.

Note that, as an SQL statement, **CLOSE** is only supported from Embedded SQL. Equivalent operations are supported through ODBC using the ODBC API.

CLOSE does not support the `#SQLCompile Mode=Deferred` preprocessor directive. Attempting to use Deferred mode with a **DECLARE**, **OPEN**, **FETCH**, or **CLOSE** cursor statement generates a `#5663` compilation error.

Example

The following Embedded SQL example shows a cursor (named EmpCursor) being opened and closed:

```
SET name="LastName,FirstName",state="##"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name, Home_State
    INTO :name,:state FROM Sample.Employee
    WHERE Home_State %STARTSWITH 'A')
WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
QUIT:(SQLCODE'=0)
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"After FETCH error code: ",SQLCODE
WRITE !,"After FETCH row count: ",%ROWCOUNT
&sql(CLOSE EmpCursor)
WRITE !,"After CLOSE error code: ",SQLCODE
WRITE !,"After CLOSE row count: ",%ROWCOUNT
WRITE !,"AFTER: Name=",name," State=",state
```

Note that after closing the cursor, the host variables remain set to the last fetched data values, and %ROWCOUNT remains set to the number of rows retrieved. However, the SQLCODE value at the end of the fetch (SQLCODE=100) is overwritten by the SQLCODE value for the **CLOSE** (SQLCODE=0).

The following Embedded SQL example shows that a cursor persists across namespaces. This cursor is declared in SAMPLES, opened in DOCBOOK, fetched in SAMPLES, and closed in USER. Note that the **FETCH** must be executed in the namespace that contains Sample.Employee:

```
&sql(USE DATABASE "USER")
WRITE $ZNSPACE,!
&sql(DECLARE NSCursor CURSOR FOR SELECT Name INTO :name FROM Sample.Employee)
&sql(USE DATABASE DOCBOOK)
WRITE $ZNSPACE,!
&sql(OPEN NSCursor)
QUIT:(SQLCODE'=0)
&sql(USE DATABASE SAMPLES)
WRITE $ZNSPACE,!
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH NSCursor)
      QUIT:SQLCODE
      WRITE "Name=",name,! }
&sql(USE DATABASE "USER")
WRITE $ZNSPACE,!
&sql(CLOSE NSCursor)
WRITE "Close SQLCODE: ",SQLCODE,!
```

See Also

- [DECLARE, FETCH, OPEN](#)
- [SQL Cursors](#) in the “Using Embedded SQL” chapter of *Using Caché SQL*

COMMIT

Commits work performed during a transaction.

```
COMMIT [WORK]
```

Description

A **COMMIT** statement commits all work completed during the current [transaction](#), resets the transaction level counter, and releases all locks established. This completes the transaction. Work committed cannot be rolled back.

COMMIT and **COMMIT WORK** are equivalent statements; both versions are supported for compatibility.

A transaction is defined as the operations since and including the **START TRANSACTION** statement. A **COMMIT** restores the transaction level counter (*\$TLEVEL*) to its state immediately prior to the **START TRANSACTION** statement that initialized the transaction. (Because Caché SQL does not support nested transactions, issuing additional **START TRANSACTION** statements within a transaction has no effect on the transaction initialization point.)

A single **COMMIT** causes all savepoints within the transaction to be committed.

A **START TRANSACTION** statement is used to explicitly begin a new transaction. However, use of **START TRANSACTION** is optional. If transaction processing is activated, the first database operation following a **COMMIT** implicitly begins a new transaction. A **COMMIT** statement is not meaningful if either transaction processing is not in effect, or transaction processing is in effect with automatic commits. If no transaction is in progress, a **COMMIT** completes successfully (SQLCODE 0), but performs no operation.

The effects of a **COMMIT** on queries are determined by the current isolation level. These transaction parameters can be set using either the **SET TRANSACTION** or **START TRANSACTION** command.

An SQLCODE -400 is issued if a transaction operation fails to complete successfully.

ObjectScript and SQL Transactions

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with **TSTART**. If you need compatibility with the SQL standard, use **START TRANSACTION**.

ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

If a transaction involves SQL update statements, then the transaction should be started by the SQL **START TRANSACTION** statement and committed with the SQL **COMMIT** statement. Methods that use **TSTART/TCOMMIT** nesting can be included in the transaction, as long as they don't initiate the transaction. Methods and stored procedures should not normally use SQL transaction control statements, unless, by design, they are the main controller of the transaction. Stored procedures should not normally use SQL transaction control statements, because these stored procedures are normally called from ODBC/JDBC, which has its own model of transaction control.

Examples

The following Embedded SQL example demonstrates how a **COMMIT** restores the transaction level counter (*\$TLEVEL*) to the level immediately prior to the **START TRANSACTION**, regardless of how many **SAVEPOINTS** have been established within the transaction. Note that the second **START TRANSACTION** in this program is a no-op which has no effect on *\$TLEVEL*:

```

&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT b)
WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION) /* Performs no operation */
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT c)
WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL

```

The following Embedded SQL example demonstrates that the first **COMMIT** statement commits the entire transaction and that extra **COMMIT** statements have no effect and do not result in an error:

```

&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT) /* Performs no operation */
WRITE !,"Commit again, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT) /* Performs no operation */
WRITE !,"Commit again, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL

```

See Also

- SQL commands: [ROLLBACK SAVEPOINT SET TRANSACTION START TRANSACTION \\$TLEVEL](#)
- [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.
- ObjectScript command: [TCOMMIT](#)

CREATE DATABASE

Creates a database (namespace).

```
CREATE DATABASE dbname [ON DIRECTORY pathname]
  [WITH [ENCRYPTED_DB] [GLOBAL_JOURNAL_STATE [=] {YES | NO}] ]
```

Arguments

<i>dbname</i>	The name of the database (namespace) to be created.
<i>pathname</i>	<i>Optional</i> — The root pathname location for the databases, specified as a quoted string. The C and D directories are created as subdirectories of this root path. The default is to create the database in the mgr directory.
WITH ENCRYPTED_DB	<i>Optional</i> — Specifies whether or not the database is encrypted. The default is not encrypted.
WITH GLOBAL_JOURNAL_STATE	<i>Optional</i> — Specifies whether or not the database is journaled. YES specifies that the database is journaled (which is recommended). NO specifies that the database is not journaled. The equal sign (=) is optional. The default is journaled.

Description

The **CREATE DATABASE** command creates a namespace and two associated databases. This allows you to create a namespace within SQL.

The specified *dbname* is the name of the created namespace and the directory that contains the corresponding database files. Namespace names are not case-sensitive. A *dbname* follows the naming conventions for an [SQL identifier](#), with the following additional restrictions:

- An underscore (`_`) character is not permitted as the first character of *dbname* (but may be used elsewhere within the name). The `@`, `#`, and `$` characters are not permitted in *dbname*. Attempting to include these invalid characters in *dbname* generates an SQLCODE -343 error.
- A hyphen (`-`) character is not permitted in *dbname* (hyphen is not a valid SQL identifier character). However, a namespace name created by other means can include a hyphen character.
- A *dbname* cannot be longer than 63 characters; specifying a longer *dbname* generates an SQLCODE -400 fatal error with the appropriate %msg.

If the specified *dbname* namespace already exists, Caché issues an SQLCODE -341 error.

You can specify neither, either, or both WITH options: ENCRYPTED_DB and/or GLOBAL_JOURNAL_STATE. If you specify both, they are separated by a space, as follows: WITH ENCRYPTED_DB GLOBAL_JOURNAL_STATE=NO.

By default, **CREATE DATABASE** creates two databases in the mgr directory with the *dbname* name subdirectory containing two subdirectories, C (code) and D (data). Each of these subdirectories contains a CACHE.DAT file, a cache.lck file, and an empty stream folder. For example, on a Windows system, CREATE DATABASE Barney would create the namespace BARNEY and the following database files:

```
C:\InterSystems\Cache\mgr\Barney\C containing CACHE.DAT, cache.lck, stream folder
C:\InterSystems\Cache\mgr\Barney\D containing CACHE.DAT, cache.lck, stream folder
```

The C (code) directory is used for the namespace routines database. The D (data) directory is used for the namespace globals database.

The optional `ON DIRECTORY pathname` clause allows you to specify a different location for the database files, rather than a directory with the same name as the namespace. For example:

```
CREATE DATABASE Flintstone ON DIRECTORY 'C:\InterSystems\Cache\mgr\Fred'
```

If you specify a *pathname* that already exists, Caché issues an `SQLCODE -341` error.

The **CREATE DATABASE** command is a privileged operation. Prior to using **CREATE DATABASE**, it is necessary to be logged in as a user with the `%Admin_Manage` resource. Failing to do so results in an `SQLCODE -99` error (Privilege Violation).

Use the `$$SYSTEM.Security.Login()` method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login( "_SYSTEM", "SYS" )
&sql( )
```

You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to `%SYSTEM.Security` in the *InterSystems Class Reference*.

You can also create a namespace from the Management Portal. Select **System Administration, Configuration, System Configuration, Namespaces** to list the existing namespaces. At the top of this table of existing namespaces you can click **Create New Namespace**.

See Also

- [DROP DATABASE](#) command
- [USE DATABASE](#) command

CREATE FUNCTION

Creates a function as a method in a class.

```
CREATE FUNCTION name(parameter_list) [characteristics]
  [ LANGUAGE SQL ]
  BEGIN
  code_body ;
  END

CREATE FUNCTION name(parameter_list) [characteristics]
  LANGUAGE OBJECTSCRIPT
  { code_body }
```

Arguments

<i>name</i>	The name of the function to be created as a method in a stored procedure class. The <i>name</i> must be a valid identifier . A procedure name can be qualified (schema.procname), or unqualified (procname). An unqualified procedure name takes the system-wide default schema name . The <i>name</i> must be followed by parentheses, even if no parameters are specified.
<i>parameter_list</i>	<i>Optional</i> — A list of parameters used to pass values to the function. The parameter list is enclosed in parentheses, and parameters in the list are separated by commas. The parentheses are mandatory, even when no parameters are specified.
<i>characteristics</i>	<i>Optional</i> — One or more keywords specifying the characteristics of the function. Permitted keywords are FOR, FINAL, PRIVATE, PROCEDURE, RETURNS, SELECTMODE. Multiple characteristics are separated by whitespace (a space or line break). Characteristics can be specified in any order.
LANGUAGE OBJECTSCRIPT LANGUAGE SQL	<i>Optional</i> — The programming language used for <i>code_body</i> . Specify LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.
<i>code_body</i>	The program code for the method. SQL program code is prefaced with a BEGIN keyword and concludes with an END keyword. Each complete SQL statement within <i>code_body</i> ends with a semicolon (;). ObjectScript program code is enclosed in curly braces. ObjectScript code lines must be indented.

Description

The **CREATE FUNCTION** statement creates a function as a method in a class. This class method is projected as an SQL Stored Procedure. You can also use the [CREATE PROCEDURE](#) statement to create a method which is projected as an SQL Stored Procedure. **CREATE FUNCTION** should be used when the method is to return a value, but it can be used to create a method that does not return a value.

In order to create a function, you must have %CREATE_FUNCTION administrative privilege, as specified by the [GRANT](#) command.

For information on calling SQL functions from within SQL statements, refer to [User-defined Functions](#) in the “Querying the Database” chapter of *Using Caché SQL*. For calling SQL stored procedures in a variety of contexts, refer to the [CALL](#) statement.

Arguments

name

The name of the function (method) to be created. This name may be unqualified (StoreName) and take the [system-wide default schema name](#), or qualified by specifying the schema name (Patient.StoreName). You can use the `$$SYSTEM.SQL.DefaultSchema()` method to determine the current system-wide default schema name. The initial system-wide default schema name is `SQLUser` corresponds to the class package name `User`.

Note that the FOR characteristic (described below) overrides the class name specified in *name*. If a method with this name already exists, the operation fails with an SQLCODE -361 error.

The name of the generated class is the package name corresponding to the schema name, followed by a dot, followed by “func”, followed by the specified *name*. For example, if the unqualified function name `RandomLetter` takes the initial default schema `SQLUser`, the resulting class name would be: `User.funcRandomLetter`. For further details, see [SQL to Class Name Transformations](#) in the “Defining and Using Stored Procedures” chapter of *Using Caché SQL*.

Caché SQL does not allow you to specify a duplicate function name that differs only in letter case. Specifying a function name that differs only in letter case from an existing function name results in an SQLCODE -400 error.

parameter-list

A list of parameters used to pass values to the function. The parameter list is enclosed in parentheses, and parameter declarations in the list are separated by commas. Each parameter declaration in the list consists of (in order):

- An optional keyword specifying whether the parameter mode is IN (input value), OUT (output value), or INOUT (modify value). If omitted, the default parameter mode is IN.
- The parameter name. Parameter names are case-sensitive.
- The [data type](#) of the parameter.
- *Optional*: A default value for the parameter. You can specify the DEFAULT keyword followed by a default value; the DEFAULT keyword is optional. If no default is specified, the assumed default is NULL.

The following example specifies two input parameters, both of which have default values. The optional DEFAULT keyword is specified for the first parameter, omitted for the second parameter:

```
CREATE FUNCTION RandomLetter(IN firstlet CHAR DEFAULT 'A',IN lastlet CHAR 'Z')
BEGIN
-- SQL program code
END
```

characteristics

The available keywords are as follows:

FOR <i>className</i>	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the function name. The class name specified in the FOR clause overrides a class name specified by qualifying the function name.
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.
PRIVATE	Specifies that the method can only be invoked by other methods of its own class or subclasses. By default, a method is public, and can be invoked without restriction. This restriction is inherited by subclasses.
PROCEDURE	Specifies that the method is projected as an SQL stored procedure. Stored procedures are inherited by subclasses. Because CREATE FUNCTION always projects an SQL stored procedure, this keyword is optional. This keyword can be abbreviated as PROC.
RETURNS <i>datatype</i>	Specifies the data type of the value returned by a call to the method. If RETURNS is omitted, the method cannot return a value. This specification is inherited by subclasses, and can be modified by subclasses. This <i>datatype</i> can specify type parameters such as MINVAL, MAXVAL, and SCALE. For example RETURNS DECIMAL(19,4). Note that when returning a value, Caché ignores the length of <i>datatype</i> ; for example, RETURNS VARCHAR(32) can receive a string of any length that is returned by a call to the method.
SELECTMODE <i>mode</i>	Only used when LANGUAGE is SQL (the default). When specified, Caché adds an <code>#SQLCOMPILE SELECT=<i>mode</i></code> statement to the corresponding class method, thus generating the SQL statements defined in the method with the specified SELECTMODE. The possible <i>mode</i> values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is LOGICAL.

The SELECTMODE clause is used for **SELECT** query operations and for **INSERT** and **UPDATE** operations. It specifies the compile-time select mode. The value that you specify for SELECTMODE is added at the beginning of the ObjectScript class method code as: `#SQLCompile Select=mode`. For further details, see [#SQLCompile Select](#) in the “ObjectScript Macros and the Macro Preprocessor” chapter of *Using Caché ObjectScript*.

- In a **SELECT** query, the SELECTMODE specifies the mode in which data is returned. If the *mode* value is LOGICAL, then logical (internal storage) values are returned. For example, dates are returned in \$HOROLOG format. If the *mode* value is ODBC, logical-to-ODBC conversion is applied, and ODBC format values are returned. If the *mode* value is DISPLAY, logical-to-display conversion is applied, and display format values are returned. If the *mode* value is RUNTIME, the display mode can be set (to LOGICAL, ODBC, or DISPLAY) at execution time.
- In an **INSERT** or **UPDATE** operation, the SELECTMODE RUNTIME option supports automatic conversion of input data values from a display format (DISPLAY or ODBC) to logical storage format. This compiled display-to-logical data conversion code is applied only if the select mode setting when the SQL code is executed is LOGICAL (which is the default for all Caché SQL execution interfaces).

When the SQL code is executed, the %SQL.Statement class *%SelectMode* property specifies the execution-time select mode, as described in “[Using Dynamic SQL](#)” chapter of *Using Caché SQL*. For further details on SelectMode options, refer to “[Data Display Options](#)” in the “Caché SQL Basics” chapter of *Using Caché SQL*.

LANGUAGE

A keyword clause specifying the language you are using for *code_body*. Permitted clauses are LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.

code_body

The program code for the method to be created. You specify this code in either SQL or ObjectScript. The language used must match the LANGUAGE clause. However, code specified in ObjectScript can contain embedded SQL.

Caché uses the code you supply to generate the actual code of the method. If the code you specify is SQL, Caché provides additional lines of code when generating the method that embed the SQL in an ObjectScript “wrapper,” provide a procedure context handler (if necessary), and handle return values. The following is an example of this Caché-generated wrapper code:

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql( SELECT col FROM tbl )
QUIT $GET(title)
```

If the code you specify is OBJECTSCRIPT, the ObjectScript code must be enclosed in curly braces. All code lines must be indented from column 1, except for labels and macro preprocessor directives. A label or macro directive must be prefaced by a colon (:) in column 1.

For ObjectScript code, you must explicitly define the “wrapper” (which NEWs variables, and uses QUIT to exit and (optionally) to return a value upon completion).

When a stored procedure is called, an object of the class %Library.SQLProcContext is instantiated in the %sqlcontext variable. This procedure context handler is used to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server).

%sqlcontext consists of several properties, including an Error object, the SQLCODE error status, the SQL row count, and an error message. The following example shows the values used to set several of these:

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

The values of SQLCODE and %ROWCOUNT are automatically set by the execution of an SQL statement. The %sqlcontext object is reset before each execution.

Alternatively, an error context can be established by instantiating a %SYSTEM.Error object and setting it as %sqlcontext.Error.

An SQLCODE -361 error is generated if the specified function already exists.

Executing a User-defined Function

You can execute a function in a **SELECT** statement, such as the following:

```
SELECT StudentName,StudentAge,SQLUser.HalfAge() AS HalfTheAge
FROM SQLUser.MyStudents
```

An SQLCODE -359 error is generated if the function does not exist.

An SQLCODE -149 error is generated if the execution of the function results in a error. The type of error is described in %msg.

Examples

The following example creates the RandomLetter() function (method) stored as a procedure that generates a random capital letter. You can then invoke this function in a **SELECT** statement. A **DROP FUNCTION** is provided to delete the RandomLetter() function.

```

CREATE FUNCTION RandomLetter()
RETURNS INTEGER
PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
SET x=$RANDOM(90)
IF x<65 {GOTO Top}
ELSE {QUIT $CHAR(x)}
}

SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetter()

DROP FUNCTION RandomLetter

```

The following example creates a function that invokes ObjectScript code, which in turn contains embedded SQL:

```

&sql(CREATE FUNCTION TraineeName(
SSN VARCHAR(11),
OUT Name VARCHAR(50) )
PROCEDURE
RETURNS VARCHAR(30)
FOR SQLUser.MyStudents
LANGUAGE OBJECTSCRIPT
{
NEW SQLCODE,%ROWCOUNT
SET Name=""
&sql(SELECT Name INTO :Name FROM Sample.Employee
WHERE SSN = :SSN)
IF $GET(%sqlcontext)!='' {
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
QUIT Name
})
IF SQLCODE=0 { WRITE !,"Created a function" QUIT}
ELSE { WRITE !,"CREATE FUNCTION error: ",SQLCODE," ",%msg,!
&sql(DROP FUNCTION TraineeName FROM SQLUser.MyStudents) }
IF SQLCODE=0 { WRITE !,"Dropped a function" QUIT}
ELSE { WRITE !,"Drop error: ",SQLCODE }

```

It uses the %sqlcontext object, and sets its %SQLCODE and %ROWCOUNT properties using the corresponding SQL variables. Note the curly braces enclosing the ObjectScript code following the function's LANGUAGE OBJECTSCRIPT keyword. Within the ObjectScript code there is [Embedded SQL](#) code, marked by &sql and enclosed in parentheses.

See Also

- [DROP FUNCTION](#)
- “[Defining and Using Stored Procedures](#)” chapter in *Using Caché SQL*.

CREATE INDEX

Creates an index for a table.

```
CREATE [UNIQUE | BITMAP | BITMAPEXTENT | BITSlice ] INDEX index-name
      ON [TABLE] table-name
      (field-name, ...)
      [WITH DATA (datafield-name, ...)]
```

Arguments

UNIQUE	<p><i>Optional</i> — A constraint that ensures there will not be two rows in the table with identical values in all the fields in the index. You cannot specify this keyword for a bitmap or bitslice index.</p> <p>The UNIQUE keyword can be followed by (or replaced by) the CLUSTERED or NONCLUSTERED keywords. These keywords are no-ops; they are provided for compatibility with other vendors.</p>
BITMAP	<p><i>Optional</i> — Indicates that a bitmap index should be created. A bitmap index enables rapid queries on fields with a small number of distinct values.</p>
BITMAPEXTENT	<p><i>Optional</i> — Indicates that a bitmapextent index should be created. At most one bitmapextent index can be created for a table. No <i>field-name</i> is specified with BITMAPEXTENT.</p>
BITSlice	<p><i>Optional</i> — Indicates that a bitslice index should be created. A bitslice index enables very fast evaluation of certain expressions, such as sums and range conditions. This is a specialized index type, which should only be used to solve very specific problems.</p>
<i>index-name</i>	<p>The index being defined. The name is an identifier. For further details, see the “Identifiers” chapter of <i>Using Caché SQL</i>.</p>
<i>table-name</i>	<p>The name of an existing table for which the index is being defined. You cannot create an index for a view. A <i>table-name</i> can be qualified (schema.table), or unqualified (table). An unqualified table name takes the system-wide default schema name.</p>
<i>field-name</i>	<p>One or more field names that serve as the basis for the index. Field names must be enclosed in parentheses. Multiple field names are separated by commas.</p> <p>Each field name can be followed by an ASC or DESC keyword. These keywords are no-ops; they are provided for compatibility with other vendors.</p>
WITH DATA (<i>datafield-name</i>)	<p><i>Optional</i> — One or more field names to be defined as Data properties for the index. Field names must be enclosed in parentheses. Multiple field names are separated by commas. You cannot specify a WITH DATA clause when specifying a BITMAP or BITSlice index.</p>

See additional compatibility syntax below.

Description

CREATE INDEX creates a sorted index on the specified field (or fields) of the named table. Caché uses indices to improve performance of query operations. Caché automatically maintains indices during **INSERT**, **UPDATE**, and **DELETE** operations, and this index maintenance may negatively affect performance of these data modification operations.

You can create an index using the **CREATE INDEX** command or by adding an index definition to a class definition, as described in the [Defining and Building Indices](#) chapter of *Caché SQL Optimization Guide*. You can delete an index by using the **DROP INDEX** command.

CREATE INDEX can be used to create any of the following three types of index:

- A regular index (**Type=index**): Specify either **CREATE INDEX** (for non-unique values) or **CREATE UNIQUE INDEX** (for unique values).
- A [bitmap index](#) (**Type=bitmap**): Specify **CREATE BITMAP INDEX**.
- A [bitslice index](#) (**Type=bitslice**): Specify **CREATE BITSlice INDEX**.

You can also define an index using the %Dictionary.IndexDefinition class.

Privileges and Locking

The **CREATE INDEX** command is a privileged operation. Prior to using **CREATE INDEX** it is necessary for your process to have either %ALTER_TABLE administrative privileges or the %ALTER privilege for the specified table. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has %ALTER privilege by invoking the %CHECKPRIV command. You can determine if a specified user has %ALTER privilege by invoking the \$SYSTEM.SQL.CheckPriv() method. You can use the **GRANT** command to assign these privileges, if you hold appropriate granting privileges.

CREATE INDEX cannot be used on a [table created by defining a persistent class](#), unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg DDL not enabled for class 'Schema.tableName'>

The **CREATE INDEX** statement acquires a table-level lock on *table-name*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **CREATE INDEX** operation. **CREATE INDEX** maintains a lock on the corresponding class definition until the completion of the create index operation, including the population of the index data.

Options Supported for Compatibility Only

Caché SQL accepts the following **CREATE INDEX** options for parsing purposes only, to aid in the conversion of existing SQL code to Caché SQL. These options do not provide any actual functionality.

```
CLUSTERED | NONCLUSTERED
owner.catalog.
ASC | DESC
```

The following is an example showing the placement of these no-op keywords:

```
CREATE UNIQUE CLUSTERED INDEX index-name ON TABLE owner.catalog.schema.table
(field1 ASC, field2 DESC)
```

Index Name

The name of an index must be unique within a given table. Index names follow [identifier](#) conventions, subject to the restrictions below. By default, index names are simple identifiers. An index name should not exceed 128 characters. Index names are not case-sensitive.

Caché uses the name you supply (which it refers to as the “SqlName”) to generate a corresponding index name in the class and the global. This index name contains only alphanumeric characters (letters and numbers) and is a maximum of 96 characters in length. To generate an index name, Caché first strips punctuation characters from the SqlName you supply, and then generates a unique identifier of 96 (or less) characters, substituting a capital letter for the final character when this is needed to create a unique index name.

- An index name can be the same as a field, table, or view name, but such name duplication is not advised.

- An index name (after punctuation stripping) must begin with a letter. Either the first character of the index name or the first character after initial punctuation characters must be a letter. A valid letter is a character that passes the `$ZNAME` test. If the first character is a punctuation character (`%` or `_`) and the second character is a number, Caché appends a lowercase “n” as the first character of the stripped name.
- An index name (after punctuation stripping) must be unique. If you specify a duplicate index name, the system generates an `SQLCODE -324` error. If you specify an index name that differs only in punctuation characters from an existing index name, Caché substitute a capital letter (beginning with “A”) for the final character to create a unique index name. For this reason, it is not advisable (though possible) to create index names that differ only in their punctuation characters.
- An index name may be much longer than 31 characters, but index names that differ in their first 31 alphanumeric characters are much easier to work with.

What happens when you try to create an index with the same name as an existing index is described below.

Existing Index

By default, Caché rejects an attempt to create an index that has the same name as an existing index for that table and issues an `SQLCODE -324` error. This behavior is configurable, as follows:

- The `$SYSTEM.SQL.SetDDLNo324()` method call. To determine the current setting, call `$SYSTEM.SQL.CurrentSettings()`, which displays a `Suppress SQLCODE=-324 Errors` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow DDL CREATE INDEX for Existing Index**.

The default is “No” (0). By default, Caché rejects an attempt to create an index with the name of an existing index for that table and issues an `SQLCODE -324` error. This is the recommended setting for this option.

If this option is set to “Yes” (1), Caché deletes the existing index from the class definition and then recreates it by performing the **CREATE INDEX**. It deletes the named index from the table specified in **CREATE INDEX**. This option permits the delete/recreate of a **UNIQUE** constraint index (which cannot be done using a **DROP INDEX** command). To delete/recreate a primary key index, refer to the **ALTER TABLE** command.

However, even if this option is set to allow the recreating of an existing index, you cannot recreate an **IDKEY** index if the table contains data. Attempting to do so generates an `SQLCODE -324` error.

Table Name

You must specify the name of an existing table.

- If *table-name* is a nonexistent table, **CREATE INDEX** fails with an `SQLCODE -30` error, and sets %msg to `Table 'SQLUSER.MYTABLE' does not exist.`
- If *table-name* is a view, **CREATE INDEX** fails with an `SQLCODE -30` error, and sets %msg to `Attempt to CREATE INDEX 'My_Index' on view SQLUSER.MYVIEW failed. Indices only supported for tables, not views..`

Creating an index modifies the table’s definition; if you do not have permission to change the table definition, **CREATE INDEX** fails with an `SQLCODE -300` error, and sets %msg to `DDL not enabled for class 'schema.tablename'.`

Field Names

You must specify at least one field name to index on. Specify a field name or a comma-separated list of field names enclosed in parentheses. Duplicate field names are permitted and preserved in the index definition. Specifying more than one field may improve performance of **GROUP BY** operations, for example, group by state and then by city within each state. Generally, you should avoid indexing on a field or fields that have large amounts of duplicate data. For example, in a database of people indexing on a Name field would be appropriate because most names are unique. Indexing on a State

field would (in most cases) not be appropriate because of the large number of duplicate data values. The fields you specify must exist in the table. Specifying a nonexistent field generates an SQLCODE -31 error.

In addition to ordinary data fields, you can use **CREATE INDEX** to create an index:

- On a [SERIAL field](#) (a %Counter field).
- On an [IDENTITY field](#).
- On the [ELEMENTS](#) or [KEYS](#) value for a collection.

You cannot create an index on a [stream value field](#).

You cannot create an index with multiple IDKEY fields if one of the IDKEY fields (properties) is SQL Computed. This limitation does not apply to a single field IDKEY index. Because multiple IDKEY fields in an index are delimited using the “||” (double vertical bar) characters, you cannot include this character string in IDKEY field data.

Field in an Embedded Object (%SerialObject)

To index a field in an embedded object, you create an index in the table (%Persistent class) referencing that embedded object. In **CREATE INDEX** the *field-name* specifies the name of the referencing field in the table (%Persistent object) joined by an underbar to the field name in the embedded object (%SerialObject), as shown in the following example:

```
CREATE INDEX StateIdx ON TABLE Sample.Person (Home_State)
```

Here *Home* is a field in Sample.Person that references the embedded object Sample.Address, which contains the *State* field.

Only those embedded object records associated with the persistent class referencing property are indexed. You cannot index a %SerialObject property directly.

For further details on defining embedded objects (also known as serial objects) refer to [Embedded Object \(%SerialObject\)](#); for further details on indexing a property (field) defined in an embedded object, refer to [Indexing an Embedded Object \(%SerialObject\) Property](#).

WITH DATA Clause

Specifying this clause may allow a query to be resolved by only reading the index, which greatly reduces the amount of disk I/O, improving performance.

You should specify the same field in the *field-name* and the WITH DATA *datafield-name* if *field-name* uses [string collation](#); this allows retrieval of the uncollated value without having to go to the [master map](#). If the value in *field-name* does not use string collation there is no advantage to specifying this field in the WITH DATA *datafield-name*.

You can specify fields in WITH DATA *datafield-name* that are not indexed. This allows more queries to be satisfied from the index without going to the master map. The tradeoff is how many indices you want to maintain; and that adding data to an index makes it quite a bit larger, which will slow down operations that don't need the data.

The UNIQUE Keyword

Using the UNIQUE keyword, you can specify that each record in the index has a unique value. More specifically, this ensures that no two records within the index (and hence in the table that contains the index) can have the same *collated* value. By default, most indices use uppercase [string collation](#) (to make searches not case-sensitive). In this case, the values “Smith” and “SMITH” are considered to be equal and not unique. **CREATE INDEX** cannot specify non-default index string collation. You can specify a different string collation for individual indices by [defining the index in the class definition](#).

You can change the [namespace default collation](#) to make fields/properties case-sensitive by default. Changing this option requires recompiling all classes and rebuilding all indices in the namespace. Go to the Management Portal, select the **Classes**

option, select the namespace for your stored queries and use the **Compile** option to recompile the corresponding classes. Then rebuild all indices. They will be case-sensitive.

CAUTION: Do not rebuild indices while the table's data is being accessed by other users. Doing so may result in inaccurate query results.

The **BITMAP** Keyword

Using the **BITMAP** keyword, you can specify that this index will be a bitmap index. A bitmap index consists of one or more bit strings in which the bit position represents the row id, and each bit value represents the presence (1) or absence (0) of a specific value for the field in that row (or the value for the combined *field-name* fields). Caché SQL maintains these positional bits (as compressed bit strings) when inserting, updating, or deleting data; there is no significant difference in the performance of **INSERT**, **UPDATE**, or **DELETE** operations between using a bitmap index and a regular index. A bitmap index is highly efficient for many types of query operations. They have the following characteristics:

- You can only define bitmap indices in tables (classes) that either use system-assigned ID with positive integer values, or use an **IDKEY** to define custom ID values when the **IDKEY** is based on a single property with type `%Integer` and `MINVAL > 0`, or type `%Numeric` with `SCALE = 0` and `MINVAL > 0`. You can use the **SYSTEM.SQL.SetBitmapFriendlyCheck()** method to set a systemwide configuration parameter to check at compile time for this restriction. You can use **SYSTEM.SQL.GetBitmapFriendlyCheck()** to determine the current configuration of this option.

You can only define a bitmap index for tables that use default (`%CacheStorage`) structure. Tables with compound keys, such as a child table, cannot use a bitmap index. If you use **DDL** (as opposed to using class definitions) to create a table, it meets this requirement and you can make use of bitmap indices.

- A bitmap index should only be used when the number of possible distinct field values is limited and relatively small. For example, a bitmap index is a good choice for a field for gender, or nationality, or timezone. A bitmap should not be used on a field with the **UNIQUE** constraint. A bitmap should not be used if a field can have more than 10,000 distinct values, or if multiple indexed fields can have more than 10,000 distinct values.
- Bitmap indices are very efficient when used in combination with logical **AND** and **OR** operations in a **WHERE** clause. If two or more fields are commonly queried in combination, it may be advantageous to define bitmap indices for those fields.

For more details, refer to the “[Bitmap Indices](#)” section of the [Defining and Building Indices](#) chapter of *Caché SQL Optimization Guide*.

The **BITMAPEXTENT** Keyword

A bitmap extent index is a bitmap index for the table itself. Caché SQL uses this index to improve performance of **COUNT(*)**, which returns the number of records (rows) in the table. A table can have, at most, one bitmap extent index. Attempting to create more than one bitmap extent index results in an **SQLCODE -400** error with the `%msg ERROR #5445: Multiple Extent indices defined: DDLBEIndex`.

At Caché 2015.2 and subsequent, all tables defined using **CREATE TABLE** automatically define a bitmap extent index. This automatically-generated index is assigned the Index Name `DDLBEIndex` and the SQL MapName `%%DDLBEIndex`. A table defined as a class may have a bitmap extent index defined with an Index Name and SQL MapName of `$ClassName`.

You can use **CREATE BITMAPEXTENT INDEX** to add a bitmap extent index to a table, or to rename an automatically-generated bitmap extent index. The *index-name* you specify should be the class name corresponding to the *table-name* of the table. This becomes the SQL MapName for the index. No *field-name* or **WITH DATA** clause can be specified.

The following example creates a bitmap extent index with Index Name `DDLBEIndex` and the SQL MapName `Patient`. If `Sample.Patient` already had a `%%DDLBEIndex` bitmap extent index, this example renames that index to SQL MapName `Patient`:

```
ZNSPACE "Samples"
&sql(CREATE BITMAPEXTENT INDEX Patient ON TABLE Sample.Patient)
WRITE !,"SQL code: ",SQLCODE
```

For more details, refer to the “[Bitmap Extent Index](#)” section of the [Defining and Building Indices](#) chapter of *Caché SQL Optimization Guide*.

The BITSLICE Keyword

Using the BITSLICE keyword, you can specify that this index will be a bitslice index. A bitslice index is used exclusively for numeric data which is used in calculations. A bitslice index represents each numeric data value as a binary bit string. Rather than indexing a numeric data value using a boolean flag (as in a bitmap index), a bitslice index creates a bit string for each numeric value, a separate bit string for each record. This is a highly specialized type of index that should only be used for fast aggregate calculations. For example, the following would be a candidate for a bitslice index:

```
SELECT SUM(Salary) FROM Sample.Employee
```

You can create a bitslice index for a string data field, but the bitslice index will represent these data values as canonical numbers. In other words, any non-numeric string, such as “abc” will be indexed as 0. This type of bitslice index could be used to rapidly count records that have a value for a string field and not count those that are NULL.

A bitslice index should not be used in a WHERE clause, because they are not used by the SQL query optimizer.

Populating and maintaining a bitslice index using INSERT, UPDATE, or DELETE operations is significantly slower than using a bitmap index or a regular index. Using several bitslice indices, and/or using a bitslice index on a field that is frequently updated may have a significant performance cost.

A bitslice index can only be used for records that have system-assigned row Ids with positive integer values. A bitslice index can only be used on a single *field-name*. You cannot specify a WITH DATA clause.

For more details, refer to the “[Bitslice Indices](#)” section of the [Defining and Building Indices](#) chapter of *Caché SQL Optimization Guide*.

Rebuilding an Index

Creating an index using the **CREATE INDEX** statement automatically builds the index. However, there are cases when you may wish to explicitly rebuild an index.

CAUTION: You must take additional steps when rebuilding an index if the table’s data is being accessed by other users. Failing to do so may result in inaccurate query results. For more details, refer to [Building Indices on an Active System](#).

To rebuild all indices for an inactive table, execute the following:

```
SET status = ##class(mySchema.mytable).%BuildIndices()
```

By default, this command purges the indices prior to rebuilding them. You can override this purge default and use the **%PurgeIndices()** method to explicitly purge specified indices. If you call **%BuildIndices()** for a range of ID values, Caché does not purge indices by default.

You can also purge/rebuild specified indices:

```
SET status = ##class(mySchema.mytable).%BuildIndices($ListBuild("NameIDX", "SpouseIDX"))
```

You may want to purge/rebuild an index if the index is corrupt or to change the case sensitivity of the index, as described above. To [recompress a bitmap index](#), use the %SYS.Maint.Bitmap methods, rather than purge/rebuild.

You can also use the Management Portal to rebuild all of the indices for a specified class (table).

For more details, refer to [Building Indices](#) in the “Defining and Building Indices” chapter of *Caché SQL Optimization Guide*.

Examples

The following embedded SQL example creates a table named Fred, and then creates an index named "FredIndex" (by stripping out the punctuation from the supplied name "Fred_Index") on the Lastword and Firstword fields of the Fred table.

```
&sql(CREATE TABLE Fred (
TESTNUM      INT NOT NULL,
FIRSTWORD    CHAR (30) NOT NULL,
LASTWORD     CHAR (30) NOT NULL,
CONSTRAINT FredPK PRIMARY KEY (TESTNUM))
)
IF SQLCODE=0 { WRITE !,"Table created" }
ELSEIF SQLCODE=-201 { WRITE !,"Table already exists" }
ELSE { WRITE !,"SQL table create error code is: ",SQLCODE
      QUIT }
&sql(CREATE INDEX Fred_Index
      ON TABLE Fred
      (LASTWORD,FIRSTWORD))
IF SQLCODE=-324 {
  WRITE !,"Index already exists"
  QUIT }
ELSEIF SQLCODE=0 { WRITE !,"Index created" }
ELSE { WRITE !,"SQL index create error code is: ",SQLCODE
      QUIT }
```

The following example creates an index, named "CityIndex" on the City field of the Staff table:

```
CREATE INDEX CityIndex ON Staff (City)
```

The following example creates an index, named "EmpIndex" on the EmpName field of the Staff table. The UNIQUE constraint is used to avoid having rows with identical values in the fields:

```
CREATE UNIQUE INDEX EmpIndex ON TABLE Staff (EmpName)
```

The following example creates a bitmap index, named "SKUIndex" on the SKU field of the Purchases table. The BITMAP keyword indicates that this is a bitmap index:

```
CREATE BITMAP INDEX SKUIndex ON TABLE Purchases (SKU)
```

See Also

- [DROP INDEX](#) command
- [SEARCH_INDEX](#) function
- "Defining Tables" chapter in *Using Caché SQL*
- "Defining and Building Indices" chapter in *Caché SQL Optimization Guide*
- "Using Indices" in the "Optimizing Query Performance" chapter in *Caché SQL Optimization Guide*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

CREATE METHOD

Creates a method in a class.

```
CREATE [STATIC] METHOD name (parameter_list)
  [ characteristics ]
  [ LANGUAGE SQL ]
  BEGIN
  code_body ;
  END

CREATE [STATIC] METHOD name (parameter_list)
  [ characteristics ]
  LANGUAGE OBJECTSCRIPT
  { code_body }
```

Arguments

<i>name</i>	The name of the method to be created in a stored procedure class. The <i>name</i> must be a valid identifier . A procedure name can be qualified (schema.procname), or unqualified (procname). An unqualified procedure name takes the system-wide default schema name . The <i>name</i> must be followed by parentheses, even if no parameters are specified.
<i>parameter_list</i>	<i>Optional</i> — A list of parameters to pass to the method. The parameter list is enclosed in parentheses, and parameters in the list are separated by commas. The parentheses are mandatory, even when no parameters are specified.
<i>characteristics</i>	<i>Optional</i> — One or more keywords specifying the characteristics of the method. Permitted keywords are RETURNS, FOR, FINAL, PRIVATE, PROCEDURE, SELECTMODE. You can specify the <i>characteristics</i> keyword phrase RESULT SETS, DYNAMIC RESULT SETS, or DYNAMIC RESULT SETS <i>n</i> , where <i>n</i> is an integer. These phrases are synonyms; the DYNAMIC keyword and the <i>n</i> integer are no-ops provided for compatibility. Multiple characteristics are separated by whitespace (a space or line break). Characteristics can be specified in any order.
LANGUAGE OBJECTSCRIPT LANGUAGE SQL	<i>Optional</i> — The programming language used for <i>code_body</i> . Specify LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.
<i>code_body</i>	The program code for the method. SQL program code is prefaced with a BEGIN keyword and concludes with an END keyword. Each complete SQL statement within <i>code_body</i> ends with a semicolon (;). ObjectScript program code is enclosed in curly braces. ObjectScript code lines must be indented.

Description

The **CREATE METHOD** statement creates a class method. This class method may or may not be a stored procedure. To create a method in a class that is exposed as an SQL stored procedure, you must specify the PROCEDURE keyword. By

default, **CREATE METHOD** does not create a method which is also a stored procedure; the [CREATE PROCEDURE](#) statement always creates a method which is also a stored procedure.

The optional **STATIC** keyword is provided to clarify that the method created is a static (class) method, not an instance method. This keyword provides no actual functionality.

In order to create a method, you must have `%CREATE_METHOD` administrative privilege, as specified by the [GRANT](#) command. If you are attempting to create a method for an existing class with a defined owner, you must be logged in as the owner of the class. Otherwise, the operation fails with an `SQLCODE -99` error.

The following two examples both show the creation of the same class method. The first example uses **CREATE METHOD**, the second defines the class method in the class `User.Letters`:

```
CREATE METHOD RandCaseLetter(IN caps CHAR)
  RETURNS INTEGER
  PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
  IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
    ELSE {GOTO Top}}
  ELSEIF caps="L" {SET x=$RANDOM(123) IF x>97 {QUIT $CHAR(x)}
    ELSE {GOTO Top}}
  ELSE {QUIT "case must be 'U' or 'L'"}
}

Class User.Letters Extends %Persistent [ DdlAllowed ]
{
  ClassMethod RandCaseLetter(caps) As %String [ SqlName = RandomLetter, SqlProc ]
  {
    Top
    IF caps="U" {SET x=$RANDOM(91) IF x>64 {QUIT $CHAR(x)}
      ELSE {GOTO Top}}
    ELSEIF caps="L" {SET x=$RANDOM(123) IF x>97 {QUIT $CHAR(x)}
      ELSE {GOTO Top}}
    ELSE {QUIT "case must be 'U' or 'L'"}
  }
}
```

For information on calling methods from within SQL statements, refer to [User-defined Functions](#) in the “Querying the Database” chapter of *Using Caché SQL*. For calling SQL stored procedures in a variety of contexts, refer to the [CALL](#) statement.

Arguments

name

The name of the method to be created. This name may be unqualified (StoreName) and take the [system-wide default schema name](#), or qualified by specifying the schema name (Patient.StoreName). You can use the `$$SYSTEM.SQL.DefaultSchema()` method to determine the current system-wide default schema name. The initial system-wide default schema name is `SQLUser` which corresponds to the class package name `User`.

Note that the **FOR** characteristic (described below) overrides the class name specified in *name*. If a method with this name already exists, the operation fails with an `SQLCODE -361` error.

The name of the generated class is the package name corresponding to the schema name, followed by a dot, followed by “meth”, followed by the specified *name*. For example, if the unqualified method name `RandomLetter` takes the initial default schema `SQLUser`, the resulting class name would be: `User.methRandomLetter`. For further details, see [SQL to Class Name Transformations](#) in the “Defining and Using Stored Procedures” chapter of *Using Caché SQL*.

Caché SQL does not allow you to specify a duplicate method name that differs only in letter case. Specifying a method name that differs only in letter case from an existing method name results in an `SQLCODE -400` error.

parameter-list

A list of parameters used to pass values to the method. The parameter list is enclosed in parentheses, and parameter declarations in the list are separated by commas. The parentheses are mandatory, even when specifying no parameters. Each parameter declaration in the list consists of (in order):

- An optional keyword specifying whether the parameter mode is IN (input value), OUT (output value), or INOUT (modify value). If omitted, the default parameter mode is IN.
- The parameter name. Parameter names are case-sensitive.
- The [data type](#) of the parameter.
- *Optional:* A default value for the parameter. You can specify the DEFAULT keyword followed by a default value; the DEFAULT keyword is optional. If no default is specified, the assumed default is NULL.

The output value from a method is automatically converted from Logical format to Display/ODBC format.

An input value to a method is, by default, not converted from Display/ODBC format to Logical format. However, input display-to-logical conversion can be configured systemwide using the `$$SYSTEM.SQL.SetSQLFunctionArgConversion()` method. You can use `$$SYSTEM.SQL.GetSQLFunctionArgConversion()` to determine the current configuration of this option.

The following example specifies two input parameters, both of which have default values. The optional DEFAULT keyword is specified for the first parameter, omitted for the second parameter:

```
CREATE METHOD RandomLetter(IN firstlet CHAR DEFAULT 'A',IN lastlet CHAR 'Z')
BEGIN
-- SQL program code
END
```

characteristics

The available keywords are as follows:

FOR <i>className</i>	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the method name. The class name specified in the FOR clause overrides a class name specified by qualifying the method name.
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.
PRIVATE	Specifies that the method can only be invoked by other methods of its own class or subclasses. By default, a method is public, and can be invoked without restriction. This restriction is inherited by subclasses.
PROCEDURE	Specifies that the method is an SQL stored procedure. Stored procedures are inherited by subclasses. (This keyword can be abbreviated as PROC.)
RESULT SETS DYNAMIC RESULT SETS [<i>n</i>]	Specifies that the method created will contain the ReturnResultsets keyword. All forms of this <i>characteristics</i> phrase are synonyms.
RETURNS <i>datatype</i>	Specifies the data type of the value returned by a call to the method. If RETURNS is omitted, the method cannot return a value. This specification is inherited by subclasses, and can be modified by subclasses. This <i>datatype</i> can specify type parameters such as MINVAL, MAXVAL, and SCALE. For example RETURNS DECIMAL(19,4). Note that when returning a value, Caché ignores the length of <i>datatype</i> ; for example, RETURNS VARCHAR(32) can receive a string of any length that is returned by a call to the method.
SELECTMODE <i>mode</i>	Only used when LANGUAGE is SQL (the default). When specified, Caché adds an #SQLCOMPILE SELECT=mode statement to the corresponding class method, thus generating the SQL statements defined in the method with the specified SELECTMODE. The possible <i>mode</i> values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is LOGICAL.

If you specify a query keyword (such as CONTAINSID or RESULTS) that is not valid for a method, the system generates an SQLCODE -47 error. If you specify a duplicate query keyword (such as FINAL FINAL), the system generates an SQLCODE -44 error.

The SELECTMODE clause is used for **SELECT** query operations and for **INSERT** and **UPDATE** operations. It specifies the compile-time select mode. The value that you specify for SELECTMODE is added at the beginning of the ObjectScript class method code as: [#SQLCompile Select=mode](#). For further details, see [#SQLCompile Select](#) in the “ObjectScript Macros and the Macro Preprocessor” chapter of *Using Caché ObjectScript*.

- In a **SELECT** query, the SELECTMODE specifies the mode in which data is returned. If the *mode* value is LOGICAL, then logical (internal storage) values are returned. For example, dates are returned in \$HOROLOG format. If the *mode* value is ODBC, logical-to-ODBC conversion is applied, and ODBC format values are returned. If the *mode* value is DISPLAY, logical-to-display conversion is applied, and display format values are returned. If the *mode* value is RUNTIME, the display mode can be set (to LOGICAL, ODBC, or DISPLAY) at execution time.
- In an **INSERT** or **UPDATE** operation, the SELECTMODE RUNTIME option supports automatic conversion of input data values from a display format (DISPLAY or ODBC) to logical storage format. This compiled display-to-logical data conversion code is applied only if the select mode setting when the SQL code is executed is LOGICAL (which is the default for all Caché SQL execution interfaces).

When the SQL code is executed, the %SQL.Statement class *%SelectMode* property specifies the execution-time select mode, as described in “Using Dynamic SQL” chapter of *Using Caché SQL*. For further details on SelectMode options, refer to “Data Display Options” in the “Caché SQL Basics” chapter of *Using Caché SQL*.

LANGUAGE

A keyword clause specifying the language you are using for *code_body*. Permitted clauses are LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.

code_body

The program code for the method to be created. You specify this code in either SQL or ObjectScript. The language used must match the LANGUAGE clause. However, code specified in ObjectScript can contain embedded SQL.

Caché uses the code you supply to generate the actual code of the method.

If the code you specify is SQL, Caché provides additional lines of code when generating the method that embed the SQL in an ObjectScript “wrapper,” provide a procedure context handler (if necessary), and handle return values. The following is an example of this Caché-generated wrapper code:

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql( SELECT col FROM tbl )
QUIT $GET(title)
```

If the code you specify is OBJECTSCRIPT, the ObjectScript code must be enclosed in curly braces. All code lines must be indented from column 1, except for labels and macro preprocessor directives. A label or macro directive must be prefaced by a colon (:) in column 1.

For ObjectScript code, you must explicitly define the “wrapper” (which NEWs variable and uses QUIT exit and (optionally) to return a value upon completion).

The method can be exposed as a stored procedure by specifying the PROCEDURE keyword. When a stored procedure is called, an object of the class %Library.SQLProcContext is instantiated in the %sqlcontext variable. This procedure context handler is used to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server).

%sqlcontext consists of several properties, including an Error object, the SQLCODE error status, the SQL row count, and an error message. The following example shows the values used to set several of these:

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

The values of SQLCODE and %ROWCOUNT are automatically set by the execution of an SQL statement. The %sqlcontext object is reset before each execution.

Alternatively, an error context can be established by instantiating a %SYSTEM.Error object and setting it as %sqlcontext.Error.

Examples

The following example uses **CREATE METHOD** with SQL code to generate the method UpdateSalary in the class Sample.Employee:

```
CREATE METHOD UpdateSalary ( IN SSN VARCHAR(11), IN Salary INTEGER )
FOR Sample.Employee
BEGIN
    UPDATE Sample.Employee SET Salary = :Salary WHERE SSN = :SSN;
END
```

The following example creates the RandomLetter() method stored as a procedure that generates a random capital letter. You can then invoke this method as a function in a **SELECT** statement. A **DROP METHOD** is provided to delete the RandomLetter() method.

```
CREATE METHOD RandomLetter()
RETURNS INTEGER
PROCEDURE
LANGUAGE OBJECTSCRIPT
{
:Top
SET x=$RANDOM(91)
IF x<65 {GOTO Top}
ELSE {QUIT $CHAR(x)}
}

SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetter()

DROP METHOD RandomLetter
```

The following

The following Embedded SQL example uses **CREATE METHOD** with ObjectScript code to generate the method `TraineeTitle` in the class `SQLUser.MyStudents` and returns a *Title* value:

```
&sql(CREATE METHOD TraineeTitle(
IN SSN VARCHAR(11),
INOUT Title VARCHAR(50) )
RETURNS VARCHAR(30)
FOR SQLUser.MyStudents
LANGUAGE OBJECTSCRIPT
{
NEW SQLCODE,%ROWCOUNT
&sql(SELECT Title INTO :Title FROM Sample.Employee
WHERE SSN = :SSN)
IF $GET(%sqlcontext)'= "" {
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
QUIT
})
IF SQLCODE=0 { WRITE !,"Created a method" QUIT}
ELSEIF SQLCODE=-361 { WRITE !,"Method already exists SQLCODE: ",SQLCODE
&sql(DROP METHOD TraineeTitle FROM SQLUser.MyStudents)
IF SQLCODE=0 { WRITE !,"Dropped a method" QUIT}}
ELSE { WRITE !,"SQL error: ",SQLCODE }
```

It uses the `%sqlcontext` object, and sets its `%SQLCODE` and `%ROWCOUNT` properties using the corresponding SQL variables. Note the curly braces enclosing the ObjectScript code following the method's `LANGUAGE OBJECTSCRIPT` keyword. Within the ObjectScript code there is [Embedded SQL](#) code, marked by `&sql` and enclosed in parentheses.

See Also

- [CALL](#)
- [CREATE PROCEDURE](#)
- [DROP METHOD](#)
- “[Defining and Using Stored Procedures](#)” chapter in *Using Caché SQL*.

CREATE PROCEDURE

Creates a method or query which is exposed as an SQL stored procedure.

```
CREATE PROCEDURE procname(parameter_list)
  [ characteristics ]
  [ LANGUAGE SQL ]
BEGIN
code_body ;
END

CREATE PROCEDURE procname(parameter_list)
  [ characteristics ]
  LANGUAGE OBJECTSCRIPT
  { code_body }

CREATE PROC procname(parameter_list)
  [ characteristics ]
  [ LANGUAGE SQL ]
BEGIN
code_body ;
END

CREATE PROC procname(parameter_list)
  [ characteristics ]
  LANGUAGE OBJECTSCRIPT
  { code_body }
```

Arguments

<i>procname</i>	The name of the procedure to be created in a stored procedure class. The <i>procname</i> must be a valid identifier . A procedure name can be qualified (schema.procname), or unqualified (procname). An unqualified procedure name takes the system-wide default schema name . The <i>procname</i> must be followed by parentheses, even if no parameters are specified.
<i>parameter_list</i>	<i>Optional</i> — A list of zero or more parameters to pass to the procedure. The parameter list is enclosed in parentheses, and parameters in the list are separated by commas. The parentheses are mandatory, even when no parameters are specified. Each parameter consists of (in order): an optional IN, OUT, or INOUT keyword; the variable name; the data type; and an optional DEFAULT clause.
<i>characteristics</i>	<i>Optional</i> — One or more keywords specifying the characteristics of the procedure. When creating a method, permitted keywords are FINAL, FOR, PRIVATE, RETURNS, SELECTMODE. When creating a query, permitted keywords are CONTAINID, FINAL, FOR, RESULTS, SELECTMODE. You can specify the <i>characteristics</i> keyword phrase RESULT SETS, DYNAMIC RESULT SETS, or DYNAMIC RESULT SETS <i>n</i> , where <i>n</i> is an integer. These phrases are synonyms; the DYNAMIC keyword and the <i>n</i> integer are no-ops provided for compatibility. Multiple characteristics are separated by whitespace (a space or line break). Characteristics can be specified in any order.
LANGUAGE OBJECTSCRIPT LANGUAGE SQL	<i>Optional</i> — A keyword clause specifying the programming language used for <i>code_body</i> . Specify LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.

<i>code_body</i>	<p>The program code for the procedure.</p> <p>SQL program code is prefaced with a BEGIN keyword and concludes with an END keyword. Each complete SQL statement within <i>code_body</i> ends with a semicolon (;).</p> <p>ObjectScript program code is enclosed in curly braces. ObjectScript code lines must be indented.</p>
------------------	---

Description

The **CREATE PROCEDURE** statement creates a method or a query which is automatically exposed as an SQL stored procedure. A stored procedure can be invoked by all processes in the current namespace. Stored procedures are inherited by subclasses.

- If **LANGUAGE SQL**, the *code_body* must contain a **SELECT** statement in order to generate a query exposed as a stored procedure. If the code does not contain a **SELECT** statement, **CREATE PROCEDURE** creates a method.
- If **LANGUAGE OBJECTSCRIPT**, the *code_body* must call **Execute()** and **Fetch()** methods in order to generate a query exposed as a stored procedure. It may also call **Close()**, **FetchRows()**, and **GetInfo()** methods. If the code does not call **Execute()** and **Fetch()**, **CREATE PROCEDURE** creates a method.

By default, **CREATE PROCEDURE** creates a method exposed as a stored procedure.

To create a method not exposed as a stored procedure, use the **CREATE METHOD** or **CREATE FUNCTION** statement. To create a query not exposed as a stored procedure, use the **CREATE QUERY** statement. These statements can also be used to create a method or query exposed as a stored procedure by specifying the **PROCEDURE** characteristic keyword.

In order to create a procedure, you must have **%CREATE_PROCEDURE** administrative privilege, as specified by the **GRANT** command. If you are attempting to create a procedure for an existing class with a defined owner, you must be logged in as the owner of the class. Otherwise, the operation fails with an **SQLCODE -99** error.

A stored procedure is executed using the **CALL** statement.

For information on calling methods from within SQL statements, refer to **User-defined Functions** in the “Querying the Database” chapter of *Using Caché SQL*.

Arguments

procname

The name of the method or query to be created as a stored procedure. This name may be unqualified (StoreName) and take the **system-wide default schema name**, or qualified by specifying the schema name (Patient.StoreName). You can use the **\$\$SYSTEM.SQL.DefaultSchema()** method to determine the current system-wide default schema name. The initial system-wide default schema name is **SQLUser** which corresponds to the class package name **User**.

Note that the **FOR** characteristic (described below) overrides the class name specified in *procname*. If a procedure with this name already exists, the operation fails with an **SQLCODE -361** error.

The name of the generated class is the package name corresponding to the schema name, followed by a dot, followed by “proc”, followed by the specified *procname*. For example, if the unqualified procedure name **RandomLetter** takes the initial default schema **SQLUser**, the resulting class name would be: **User.procRandomLetter**. For further details, see **SQL to Class Name Transformations** in the “Defining and Using Stored Procedures” chapter of *Using Caché SQL*.

Caché SQL does not allow you to specify a *procname* that differs only in letter case. Specifying a *procname* that differs only in letter case from an existing procedure name results in an **SQLCODE -400** error.

If the specified *procname* already exists in the current namespace, the system generates an SQLCODE -361 error. To determine if a specified *procname* already exists in the current namespace, use the `$$SYSTEM.SQL.ProcedureExists()` method.

Note: Caché SQL procedure names and Caché TSQL procedure names share the same set of names. Therefore, you cannot create an SQL procedure that has the same name as a TSQL procedure in the same namespace. Attempting to do so results in an SQLCODE -400 error.

The name of a procedure must be followed by parameter parentheses.

parameter_list

A list of parameters used to pass values to the method or query. The parameter list is enclosed in parentheses, and parameter declarations in the list are separated by commas. The parentheses are mandatory, even if you specify no parameters.

Each parameter declaration in the list consists of (in order):

- An optional keyword specifying whether the parameter mode is IN (input value), OUT (output value), or INOUT (modify value). If omitted, the default parameter mode is IN.
- The parameter name. Parameter names are case-sensitive.
- The [data type](#) of the parameter.
- *Optional:* A default value for the parameter. You can specify the DEFAULT keyword followed by a default value; the DEFAULT keyword is optional. If no default is specified, the assumed default is NULL.

The following example creates a stored procedure with two input parameters, both of which have default values. One input parameter specifies the optional DEFAULT keyword, the other input parameter omits this keyword:

```
CREATE PROCEDURE AgeQuerySP(IN topnum INT DEFAULT 10,IN minage INT 20)
  BEGIN
  SELECT TOP :topnum Name,Age FROM Sample.Person
  WHERE Age > :minage ;
  END
```

The following example is functionally identical to the example above. The optional DEFAULT keyword is omitted:

```
CREATE PROCEDURE AgeQuerySP(IN topnum INT 10,IN minage INT 20)
  BEGIN
  SELECT TOP :topnum Name,Age FROM Sample.Person
  WHERE Age > :minage ;
  END
```

The following are all valid **CALL** statements for this procedure: `CALL AgeQuerySP(6,65);` `CALL AgeQuerySP(6);` `CALL AgeQuerySP(,65);` `CALL AgeQuerySP()`.

The following example creates a method exposed as a stored procedure with three parameters:

```
CREATE PROCEDURE UpdatePaySP
  (IN Salary INTEGER DEFAULT 0,
  IN Name VARCHAR(50),
  INOUT PayBracket VARCHAR(50) DEFAULT 'NULL')
  BEGIN
  UPDATE Sample.Person SET Salary = :Salary
  WHERE Name=:Name ;
  END
```

A stored procedure does not perform automatic format conversion of parameters. For example, an input parameter in ODBC format or Display format remains in that format. It is the responsibility of the code that calls the procedure, and the procedure code itself, to handle IN/OUT values in a format appropriate to the application, and to perform any necessary conversions.

Because the method or query is exposed as a stored procedure, it uses a procedure context handler to pass the procedure context back and forth between the procedure and its caller. When a stored procedure is called, an object of the class

`%Library.SQLProcContext` is instantiated in the `%sqlcontext` variable. This is used to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server).

`%sqlcontext` consists of several properties, including an Error object, the `SQLCODE` error status, the SQL row count, and an error message. The following example shows the values used to set several of these:

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

The values of `SQLCODE` and `%ROWCOUNT` are automatically set by the execution of an SQL statement. The `%sqlcontext` object is reset before each execution.

Alternatively, an error context can be established by instantiating a `%SYSTEM.Error` object and setting it as `%sqlcontext.Error`.

characteristics

Different *characteristics* are used for creating a method than those used to create a query.

If you specify a *characteristics* that is not valid, the system generates an `SQLCODE -47` error. Specifying duplicate *characteristics* results in an `SQLCODE -44` error.

The available method *characteristics* keywords are as follows:

Method Keyword	Meaning
FOR <i>className</i>	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the method name. The class name specified in the FOR clause overrides a class name specified by qualifying the method name. If you specify the class name using the FOR <code>my.class</code> syntax, Caché defines the class method with <code>Sqlname=procname</code> . Therefore, the method should be invoked as my.procname() (<i>not my.class_procname()</i>).
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.
PRIVATE	Specifies that the method can only be invoked by other methods of its own class or subclasses. By default, a method is public, and can be invoked without restriction. This restriction is inherited by subclasses.
RESULT SETS DYNAMIC RESULT SETS [<i>n</i>]	Specifies that the method created will contain the ReturnResultsets keyword. All forms of this <i>characteristics</i> phrase are synonyms.
RETURNS <i>datatype</i>	Specifies the data type of the value returned by a call to the method. If RETURNS is omitted, the method cannot return a value. This specification is inherited by subclasses, and can be modified by subclasses. This <i>datatype</i> can specify type parameters such as MINVAL, MAXVAL, and SCALE. For example <code>RETURNS DECIMAL(19,4)</code> . Note that when returning a value, Caché ignores the length of <i>datatype</i> ; for example, <code>RETURNS VARCHAR(32)</code> can receive a string of any length that is returned by a call to the method.
SELECTMODE <i>mode</i>	Only used when LANGUAGE is SQL (the default). When specified, Caché adds an <code>#SQLCOMPILE SELECT=mode</code> statement to the corresponding class method, thus generating the SQL statements defined in the method with the specified SELECTMODE. The possible <i>mode</i> values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is LOGICAL.

The available query *characteristics* keywords are as follows:

Query Keyword	Description
CONTAINID <i>integer</i>	Specifies which field, if any, returns the ID. Set CONTAINID to the number of the column that returns the ID, or 0 if no column returns the ID. Caché does not validate that the named field actually contains the ID, so a user error here results in inconsistent data.
FOR <i>className</i>	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the method name. The class name specified in the FOR clause overrides a class name specified by qualifying the method name.
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.
RESULTS (<i>result_set</i>)	Specifies the data fields in the order that they are returned by the query. If you specify a RESULTS clause, you must list all fields returned by the query as a comma-separated list enclosed in parentheses. Specifying fewer or more fields than are returned by the query results in a SQLCODE -76 cardinality mismatch error. For each field you specify a column name (which will be used as the column header) and a data type. If LANGUAGE SQL, you can omit the RESULTS clause. If you omit the RESULTS clause, the ROWSPEC is automatically generated during class compilation.
SELECTMODE <i>mode</i>	Specifies the mode used to compile the query. The possible values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is RUNTIME.

The SELECTMODE clause is used for **SELECT** query operations and for **INSERT** and **UPDATE** operations. It specifies the compile-time select mode. The value that you specify for SELECTMODE is added at the beginning of the ObjectScript class method code as: #SQLCompile Select=*mode*. For further details, see [#SQLCompile Select](#) in the “ObjectScript Macros and the Macro Preprocessor” chapter of *Using Caché ObjectScript*.

- In a **SELECT** query, the SELECTMODE specifies the mode in which data is returned. If the *mode* value is LOGICAL, then logical (internal storage) values are returned. For example, dates are returned in \$HOROLOG format. If the *mode* value is ODBC, logical-to-ODBC conversion is applied, and ODBC format values are returned. If the *mode* value is DISPLAY, logical-to-display conversion is applied, and display format values are returned. If the *mode* value is RUNTIME, the display mode can be set (to LOGICAL, ODBC, or DISPLAY) at execution time.
- In an **INSERT** or **UPDATE** operation, the SELECTMODE RUNTIME option supports automatic conversion of input data values from a display format (DISPLAY or ODBC) to logical storage format. This compiled display-to-logical data conversion code is applied only if the select mode setting when the SQL code is executed is LOGICAL (which is the default for all Caché SQL execution interfaces).

When the SQL code is executed, the %SQL.Statement class *%SelectMode* property specifies the execution-time select mode, as described in “[Using Dynamic SQL](#)” chapter of *Using Caché SQL*. For further details on SelectMode options, refer to “[Data Display Options](#)” in the “Caché SQL Basics” chapter of *Using Caché SQL*.

The RESULTS clause specifies the results of a query. The SQL data type parameters in the RESULTS clause are translated into corresponding Caché data type parameters in the query’s ROWSPEC. For example, the RESULTS clause RESULTS (Code VARCHAR(15)) generates a ROWSPEC specification of ROWSPEC = “Code:%Library.String(MAXLEN=15)”.

LANGUAGE

A keyword clause specifying the language you are using for *code_body*. Permitted clauses are LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.

code_body

The program code for the method or query to be created. You specify this code in either SQL or ObjectScript. The language used must match the LANGUAGE clause. However, code specified in ObjectScript can contain embedded SQL. Caché uses the code you supply to generate the actual code of the method or query.

- SQL program code is prefaced with a BEGIN keyword, followed by the SQL code itself. At the end of each complete SQL statement, specify a semicolon (;). A query contains only one SQL statement—a **SELECT** statement. You can also create procedures that insert, update, or delete data. SQL program code concludes with an END keyword.

Input parameters are specified in the SQL statement as [host variables](#), with the form :name. (Note that you *should not* use question marks (?) to specify input parameters in the SQL code. The procedure will successfully build, but when it is called these parameters cannot be passed or take default values.)

- ObjectScript program code is enclosed within curly braces: { code }. Lines of code must be indented. If specified, a label or a #Include preprocessor command must be prefaced by a colon and appear in the first column, as shown in the following example:

```
CREATE PROCEDURE SP123()
  LANGUAGE OBJECTSCRIPT
  {
  :Top
  :#Include %occConstant
    WRITE "Hello World"
    IF 0=$RANDOM(2) { GOTO Top }
    ELSE {QUIT $$$OK }
  }
```

The system automatically includes %occInclude. If program code contains Caché Macro Preprocessor statements (# commands, ## functions, or \$\$\$macro references) the processing and expansion of these statements is part of the procedure's method definition, and get processed and expanded when the method is compiled. For more details on preprocessor commands, see [ObjectScript Macros and the Macro Preprocessor](#) in *Using Caché ObjectScript*.

Caché provides additional lines of code when generating the procedure that embed the SQL in an ObjectScript “wrapper,” provide a procedure context handler, and handle return values. The following is an example of this Caché-generated wrapper code:

```
NEW SQLCODE,%ROWID,%ROWCOUNT,title
&sql(
  -- code_body
)
QUIT $GET(title)
```

If the code you specify is OBJECTSCRIPT, you must explicitly define the “wrapper” (which NEWs variable and uses **QUIT val** to return a value upon completion).

Examples

The examples that follow are divided into those that use an SQL *code_body*, and those that use an ObjectScript *code_body*.

Examples Using SQL Code

The following example creates a simple query, named PersonStateSP, exposed as a stored procedure. It declares no parameters and takes default values for *characteristics* and LANGUAGE:

```

WRITE !,"Creating a procedure"
&sql(CREATE PROCEDURE PersonStateSP() BEGIN
    SELECT Name,Home_State FROM Sample.Person ;
END)
IF SQLCODE=0 { WRITE !,"Created a procedure" }
ELSEIF SQLCODE=-361 { WRITE !,"Procedure already exists" }
ELSE { WRITE !,"SQL error: ",SQLCODE }

```

You can go to the Management Portal, select the **Classes** option, then select the SAMPLES namespace. There you will find the stored procedure created by the above example: User.procPersonStateSP.cls. From this display you can delete this procedure before rerunning the above program example. You can, of course, use **DROP PROCEDURE** to delete a procedure:

```

WRITE !,"Deleting a procedure"
&sql(DROP PROCEDURE SAMPLES.PersonStateSP)
IF SQLCODE=0 { WRITE !,"Deleted a procedure" }
ELSEIF SQLCODE=-362 { WRITE !,"Procedure did not exist" }
ELSE { WRITE !,"SQL error: ",SQLCODE }

```

The following example creates a procedure to update data. It uses **CREATE PROCEDURE** to generate the method UpdateSalary in the class Sample.Employee:

```

CREATE PROCEDURE UpdateSalary ( IN SSN VARCHAR(11), IN Salary INTEGER )
FOR Sample.Employee
BEGIN
    UPDATE Sample.Employee SET Salary = :Salary WHERE SSN = :SSN;
END

```

Examples Using ObjectScript Code

The following example creates the RandomLetterSP() stored procedure method that generates a random capital letter. You can then invoke this method as a function in a **SELECT** statement. A **DROP PROCEDURE** is provided to delete the RandomLetterSP() method.

```

CREATE PROCEDURE RandomLetterSP()
RETURNS INTEGER
LANGUAGE OBJECTSCRIPT
{
:Top
    SET x=$RANDOM(90)
    IF x<65 {GOTO Top}
    ELSE {QUIT $CHAR(x)}
}

SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH RandomLetterSP()

DROP PROCEDURE RandomLetterSP

```

The following **CREATE PROCEDURE** example uses ObjectScript calls to the **Execute()**, **Fetch()**, and **Close()** methods. Such procedures may also contain **FetchRows()** and **GetInfo()** method calls:

```

CREATE PROCEDURE GetTitle()
FOR Sample.Employee
RESULTS (ID %Integer)
CONTAINID 1
LANGUAGE OBJECTSCRIPT
Execute(INOUT qHandle %Binary)
{ QUIT 1 }
Fetch(INOUT qHandle %Binary, INOUT Row %List, INOUT AtEnd %Integer)
{ QUIT 1 }
Close(INOUT qHandle %Binary)
{ QUIT 1 }

```

The following **CREATE PROCEDURE** example uses an ObjectScript call to the %SQL.Statement result set class:

```

CREATE PROCEDURE Sample_Employee.GetTitle(
    INOUT Title VARCHAR(50) )
    RETURNS VARCHAR(30)
    FOR Sample.Employee
    LANGUAGE OBJECTSCRIPT
    {
    SET myquery="SELECT TOP 10 Name,Title FROM Sample.Employee"
    ZNSPACE "SAMPLES"
    SET tStatement = ##class(%SQL.Statement).%New()
    SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
    SET rset = tStatement.%Execute()
    DO rset.%Display()
    WRITE !,"End of data"
    }

```

If the ObjectScript code block fetches data into a local variable (for example, *Row*), you must conclude the code block with the line `SET Row= " "` to indicate an end-of-data condition.

The following example uses **CREATE PROCEDURE** with ObjectScript code that invokes Embedded SQL. It generates the method `GetTitle` in the class `Sample.Employee` and passes out the *Title* value as a parameter:

```

CREATE PROCEDURE Sample_Employee.GetTitle(
    IN SSN VARCHAR(11),
    INOUT Title VARCHAR(50) )
    RETURNS VARCHAR(30)
    FOR Sample.Employee
    LANGUAGE OBJECTSCRIPT
    {
        NEW SQLCODE,%ROWCOUNT
        &sql(SELECT Title INTO :Title FROM Sample.Employee
            WHERE SSN = :SSN)
        IF $GET(%sqlcontext)'= " " {
            SET %sqlcontext.%SQLCODE=SQLCODE
            SET %sqlcontext.%ROWCOUNT=%ROWCOUNT }
        QUIT
    }

```

It uses the `%sqlcontext` object, and sets its `%SQLCODE` and `%ROWCOUNT` properties using the corresponding SQL variables. Note the curly braces enclosing the ObjectScript code following the procedure's `LANGUAGE OBJECTSCRIPT` keyword. Within the ObjectScript code there is [Embedded SQL](#) code, marked by `&sql` and enclosed in parentheses.

See Also

- [SELECT](#)
- [CALL](#)
- [DROP PROCEDURE](#)
- [CREATE METHOD CREATE FUNCTION](#)
- [GRANT](#)
- “Defining and Using Stored Procedures” chapter in *Using Caché SQL*.
- “Querying the Database” chapter in *Using Caché SQL*

CREATE QUERY

Creates a query.

```
CREATE QUERY queryname(parameter_list) [characteristics]
  [ LANGUAGE SQL ]
  BEGIN
  code_body ;
  END

CREATE QUERY queryname(parameter_list) [characteristics]
  LANGUAGE OBJECTSCRIPT
  { code_body }
```

Arguments

<i>queryname</i>	The name of the query to be created in a stored procedure class. The <i>queryname</i> must be a valid identifier . A procedure name can be qualified (schema.procname), or unqualified (procname). An unqualified procedure name takes the system-wide default schema name . The <i>queryname</i> must be followed by parentheses, even if no parameters are specified.
<i>parameter_list</i>	<i>Optional</i> — A list of parameters to pass to the query. The parameter list is enclosed in parentheses, and parameters in the list are separated by commas. The parentheses are mandatory, even when no parameters are specified.
<i>characteristics</i>	<i>Optional</i> — One or more keywords specifying the characteristics of the query. Permitted keywords are RESULTS, CONTAINID, FOR, FINAL, PROCEDURE, SELECTMODE. Multiple characteristics are separated by whitespace (a space or line break). Characteristics can be specified in any order.
LANGUAGE OBJECTSCRIPT LANGUAGE SQL	<i>Optional</i> — A keyword clause specifying the programming language used for <i>code_body</i> . Specify either LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.
<i>code_body</i>	The program code for the query. SQL program code is prefaced with a BEGIN keyword and concludes with an END keyword. The <i>code_body</i> for a query consists of only one complete SQL statement (a SELECT statement). This SELECT statement ends with a semicolon (;). ObjectScript program code is enclosed in curly braces. ObjectScript code lines must be indented.

Description

The **CREATE QUERY** statement creates a query in a class. By default, a query named MySelect would be stored as User.queryMySelect or SQLUser.queryMySelect.

CREATE QUERY creates a query which may or may not be exposed as a stored procedure. To create a query that is exposed as a stored procedure, you must specify the PROCEDURE keyword as one of its *characteristics*. You can also use the [CREATE PROCEDURE](#) statement to create a query which is exposed as a stored procedure.

In order to create a query, you must have %CREATE_QUERY administrative privilege, as specified by the [GRANT](#) command. If you are attempting to create a query for an existing class with a defined owner, you must be logged in as the owner of the class. Otherwise, the operation fails with an SQLCODE -99 error.

Arguments

queryname

The name of the query to be created as a stored procedure. This name may be unqualified (StoreName) and take the [system-wide default schema name](#), or qualified by specifying the schema name (Patient.StoreName). You can use the `$$SYSTEM.SQL.DefaultSchema()` method to determine the current system-wide default schema name. The initial system-wide default schema name is `SQLUser` which corresponds to the class package name `User`.

Note that the FOR characteristic (described below) overrides the class name specified in *queryname*. If a method with this name already exists, the operation fails with an SQLCODE -361 error.

The name of the generated class is the package name corresponding to the schema name, followed by a dot, followed by “query”, followed by the specified *queryname*. For example, if the unqualified query name `RandomLetter` takes the initial default schema `SQLUser`, the resulting class name would be: `User.queryRandomLetter`. For further details, see [SQL to Class Name Transformations](#) in the “Defining and Using Stored Procedures” chapter of *Using Caché SQL*.

Caché SQL does not allow you to specify a *queryname* that differs only in letter case. Specifying a *queryname* that differs only in letter case from an existing query name results in an SQLCODE -400 error.

If the specified *queryname* already exists in the current namespace, the system generates an SQLCODE -361 error.

parameter-list

A list of parameter declarations for parameters used to pass values to the query. The parameter list is enclosed in parentheses, and parameter declarations in the list are separated by commas. The parentheses are mandatory, even if you specify no parameters.

Each parameter declaration in the list consists of (in order):

- An optional keyword specifying whether the parameter mode is IN (input value), OUT (output value), or INOUT (modify value). If omitted, the default parameter mode is IN.
- The parameter name. Parameter names are case-sensitive.
- The [data type](#) of the parameter.
- *Optional*: A default value for the parameter. You can specify the DEFAULT keyword followed by a default value; the DEFAULT keyword is optional. If no default is specified, the assumed default is NULL.

The following example creates a query exposed as a stored procedure with two input parameters, both of which have default values. The *topnum* input parameter specifies the optional DEFAULT keyword; the *minage* input parameter omits this keyword:

```
CREATE QUERY AgeQuery(IN topnum INT DEFAULT 10,IN minage INT 20)
  PROCEDURE
  BEGIN
  SELECT TOP :topnum Name,Age FROM Sample.Person
  WHERE Age > :minage ;
  END
```

The following are all valid **CALL** statements for this query: `CALL AgeQuery(6,65)`; `CALL AgeQuery(6)`; `CALL AgeQuery(,65)`; `CALL AgeQuery()`.

characteristics

The available *characteristics* keywords are as follows:

Characteristics Keyword	Description
CONTAINID <i>integer</i>	Specifies which field, if any, returns the ID. Set CONTAINID to the number of the column that returns the ID, or 0 if no column returns the ID. Caché does not validate that the named field actually contains the ID, so a user error here results in inconsistent data.
FOR <i>className</i>	Specifies the name of the class in which to create the method. If the class does not exist, it will be created. You can also specify a class name by qualifying the method name. The class name specified in the FOR clause overrides a class name specified by qualifying the method name.
FINAL	Specifies that subclasses cannot override the method. By default, methods are not final. The FINAL keyword is inherited by subclasses.
PROCEDURE	Specifies that the query is an SQL stored procedure. Stored procedures are inherited by subclasses. (This keyword can be abbreviated as PROC.)
RESULTS (<i>result_set</i>)	<p>Specifies the data fields in the order that they are returned by the query. If you specify a RESULTS clause, you must list all fields returned by the query as a comma-separated list enclosed in parentheses. Specifying fewer or more fields than are returned by the query results in a SQLCODE -76 cardinality mismatch error.</p> <p>For each field you specify a column name (which will be used as the column header) and a data type.</p> <p>If LANGUAGE SQL, you can omit the RESULTS clause. If you omit the RESULTS clause, the ROWSPEC is automatically generated during class compilation.</p>
SELECTMODE <i>mode</i>	Specifies the mode used to compile the query. The possible values are LOGICAL, ODBC, RUNTIME, and DISPLAY. The default is RUNTIME.

If you specify a method keyword (such as PRIVATE or RETURNS) that is not valid for a query, the system generates an SQLCODE -47 error. Specifying duplicate *characteristics* results in an SQLCODE -44 error.

The SELECTMODE clause specifies the mode in which data is returned. If the *mode* value is LOGICAL, then logical (internal storage) values are returned. For example, dates are returned in \$HOROLOG format. If the *mode* value is ODBC, logical-to-ODBC conversion is applied, and ODBC format values are returned. If the *mode* value is DISPLAY, logical-to-display conversion is applied, and display format values are returned. If the *mode* value is RUNTIME, the mode can be set (to LOGICAL, ODBC, or DISPLAY) at execution time by setting the %SQL.Statement class %SelectMode property, as described in “[Using Dynamic SQL](#)” chapter of *Using Caché SQL*. The RUNTIME mode default is LOGICAL. For further details on SelectMode options, refer to “[Data Display Options](#)” in the “Caché SQL Basics” chapter of *Using Caché SQL*. The value that you specify for SELECTMODE is added at the beginning of the ObjectScript class method code as: #SQL-Compile SELECT=*mode*. For further details, see [#SQLCompile Select](#) in the “ObjectScript Macros and the Macro Preprocessor” chapter of *Using Caché ObjectScript*.

The RESULTS clause specifies the results of a query. The SQL data type parameters in the RESULTS clause are translated into corresponding Caché data type parameters in the query’s ROWSPEC. For example, the RESULTS clause RESULTS (Code VARCHAR(15)) generates a ROWSPEC specification of ROWSPEC = “Code:%Library.String(MAXLEN=15)”.

LANGUAGE

A keyword clause specifying the language you are using for *code_body*. Permitted clauses are LANGUAGE OBJECTSCRIPT (for ObjectScript) or LANGUAGE SQL. If the LANGUAGE clause is omitted, SQL is the default.

If the LANGUAGE is SQL a class query of type %Library.SQLQuery is generated. If the LANGUAGE is OBJECTSCRIPT, a class query of type %Library.Query is generated.

code_body

The program code for the query to be created. You specify this code in either SQL or ObjectScript. The language used must match the LANGUAGE clause. However, code specified in ObjectScript can contain embedded SQL.

If the code you specify is SQL, it must consist of a single **SELECT** statement. The program code for a query in SQL is prefaced with a **BEGIN** keyword, followed by the program code (a **SELECT** statement). At the end of the program code, specify a semicolon (;) then an **END** keyword.

If the code you specify is OBJECTSCRIPT, it must contain calls to the **Execute()** and **Fetch()** class methods of the %Library.Query class provided by Caché, and may contain **Close()**, **FetchRows()**, and **GetInfo()** method calls. ObjectScript code is enclosed in curly braces. If **Execute()** or **Fetch()** are missing, an SQLCODE -46 error is generated upon compilation.

If the ObjectScript code block fetches data into a local variable (for example, *Row*), you must conclude the code block with the line `SET Row= " "` to indicate an end-of-data condition.

If the query is exposed as a stored procedure (by specifying the **PROCEDURE** keyword in *characteristics*), it uses a procedure context handler to pass the procedure context back and forth between the procedure and its caller.

When a stored procedure is called, an object of the class %Library.SQLProcContext is instantiated in the %sqlcontext variable. This is used to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server).

%sqlcontext consists of several properties, including an Error object, the SQLCODE error status, the SQL row count, and an error message. The following example shows the values used to set several of these:

```
SET %sqlcontext.%SQLCODE=SQLCODE
SET %sqlcontext.%ROWCOUNT=%ROWCOUNT
SET %sqlcontext.%Message=%msg
```

The values of SQLCODE and %ROWCOUNT are automatically set by the execution of an SQL statement. The %sqlcontext object is reset before each execution.

Alternatively, an error context can be established by instantiating a %SYSTEM.Error object and setting it as %sqlcontext.Error.

Caché uses the code you supply to generate the actual code of the query.

Examples

The following embedded SQL example creates a query named DocTestPersonState. It declares no parameters, sets the **SELECTMODE** *characteristic*, and takes the default (SQL) for LANGUAGE:

```
&sql(CREATE QUERY DocTestPersonState() SELECTMODE RUNTIME
      BEGIN
        SELECT Name,Home_State FROM Sample.Person ;
      END)
IF SQLCODE=0 { WRITE !,"Created a query" }
ELSEIF SQLCODE=-361 { WRITE !,"Query exists: ",%msg }
ELSE { WRITE !,"CREATE QUERY error: ",SQLCODE }
```

You can go to the Management Portal, select the **Classes** option, then select the SAMPLES namespace. There you will find the query created by the above example: User.queryDocTestPersonState.cls. From this display you can delete this query before rerunning the above program example. You can, of course, use **DROP QUERY** to delete created queries.

The following Embedded SQL example creates a method-based query named DocTestSQLCODEList which fetches a list of SQLCODEs and their descriptions. It sets a **RESULTS** result set characteristic, sets LANGUAGE as ObjectScript, and calls the **Execute()**, **Fetch()**, and **Close()** methods:

```
&sql(CREATE QUERY DocTestSQLCODEList()
      RESULTS (SQLCODE SMALLINT,Description VARCHAR(100))
      PROCEDURE
      LANGUAGE OBJECTSCRIPT
```

```

Execute(INOUT QHandle BINARY(255))
{
  SET QHandle=1,%i(QHandle)="
  QUIT ##lit($$OK)
}
Fetch(INOUT QHandle BINARY(255), INOUT Row %List, INOUT AtEnd INT)
{
  SET AtEnd=0,Row=""
  SET %i(QHandle)=$o(^%qCacheSQL("SQLCODE",%i(QHandle)))
  IF %i(QHandle)=" {SET AtEnd=1 QUIT ##lit($$OK) }
  SET Row=$lb(%i(QHandle),^%qCacheSQL("SQLCODE",%i(QHandle),1,1))
  QUIT ##lit($$OK)
}
Close(INOUT QHandle BINARY(255))
{
  KILL %i(QHandle)
  QUIT ##lit($$OK)
}
)
IF SQLCODE=0 { WRITE !,"Created a query" }
ELSEIF SQLCODE=-361 { WRITE !,"Query exists: ",%msg }
ELSE { WRITE !,"CREATE QUERY error: ",SQLCODE }

```

You can go to the Management Portal, select the **Classes** option, then select the SAMPLES namespace. There you will find the query created by the above example: User.queryDocTestSQLCODEList.cls. From this display you can delete this query before rerunning the above program example. You can, of course, use **DROP QUERY** to delete created queries.

The following Dynamic SQL example creates a query named DocTest, then executes this query using the **%PrepareClassQuery()** method of the %SQL.Statement class:

```

/* Creating the Query */
SET myquery=4
  SET myquery(1)="CREATE QUERY DocTest() SELECTMODE RUNTIME "
  SET myquery(2)="BEGIN "
  SET myquery(3)="SELECT TOP 5 Name,Home_State FROM Sample.Person ; "
  SET myquery(4)="END"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF SQLCODE=0 { WRITE !,"Created a query",! }
ELSEIF SQLCODE=-361 { WRITE !,"Query exists: ",%msg }
ELSE { WRITE !,"CREATE QUERY error: ",SQLCODE }
/* Calling the Query */
WRITE !,"Calling a class query",!
SET cqStatus = tStatement.%PrepareClassQuery("User.queryDocTest","DocTest")
  IF cqStatus'=1 {WRITE "%PrepareClassQuery failed:" DO $System.Status.DisplayError(cqStatus)}
SET rset = tStatement.%Execute()
WRITE "Query data",!,!
WHILE rset.%Next() {
  DO rset.%Print() }
WRITE !,"End of data"
/* Deleting the Query */
&sql(DROP QUERY DocTest)
IF SQLCODE=0 { WRITE !,"Deleted the query" }

```

For further details, refer to the [Dynamic SQL](#) chapter of *Using Caché SQL*.

See Also

- [SELECT](#)
- [CALL](#)
- [DROP QUERY](#)
- [CREATE PROCEDURE](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*
- “[Defining and Using Stored Procedures](#)” chapter in *Using Caché SQL*

CREATE ROLE

Creates a role.

```
CREATE ROLE role-name
```

Arguments

<i>role-name</i>	The name of the role to be created, which is an identifier . Role names are not case-sensitive. For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> .
------------------	---

Description

The **CREATE ROLE** command creates a role. A role is a named set of privileges that may be assigned to multiple users. A role may be assigned to multiple users, and a user may be assigned multiple roles. A role is available system-wide, it is not limited to a specific namespace.

A *role-name* can be any valid identifier of up to 64 characters. A role name must follow [identifier](#) naming conventions, with the following restriction. If the role name is a [delimited identifier](#) enclosed in quotation marks, it cannot contain a comma (,) or a colon (:). A delimited identifier can be an SQL reserved word. A role name can contain Unicode characters. Role names are not case-sensitive.

When initially created, a role is just a name; it has no privileges. To add privileges to a role, use the [GRANT](#) command. You can also use the **GRANT** command to assign one or more roles to a role. This permits you to create a hierarchy of roles.

If you invoke **CREATE ROLE** to create a role that already exists, SQL issues an SQLCODE -118 error. You can determine if a role already exists by invoking the `$$SYSTEM.SQL.RoleExists()` method:

```
WRITE $$SYSTEM.SQL.RoleExists("%All"), !
WRITE $$SYSTEM.SQL.RoleExists("Madmen")
```

This method returns 1 if the specified role exists, and 0 if the role does not exist. Role names are not case-sensitive.

To delete a role, use the **DROP ROLE** command.

Privileges

The **CREATE ROLE** command is a privileged operation. Before using **CREATE ROLE** in embedded SQL, it is necessary to be logged in as a user with `%Admin_Secure:USE` privilege. Failing to do so results in an SQLCODE -99 error (Privilege Violation). Use the `$$SYSTEM.Security.Login()` method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login(username,password)
&sql( )
```

You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login()` method. For further information, refer to `%SYSTEM.Security` in the *InterSystems Class Reference*.

Examples

The following examples attempt to create a role named BkUser. The user “FRED” in the first example does not have create role privileges. The user “_SYSTEM” in the second example does have create role privileges.

```
DO $$SYSTEM.Security.Login("FRED", "FredsPassword")
&sql(CREATE ROLE BkUser)
IF SQLCODE=-99 {
  WRITE !,"You don't have CREATE ROLE privileges" }
ELSEIF SQLCODE=-118 {
  WRITE !,"The role already exists" }
ELSE {
  WRITE !,"Created a role. Error code is: ",SQLCODE }
```

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
Main
&sql(CREATE ROLE BkUser)
IF SQLCODE=-99 {
  WRITE !,"You don't have CREATE ROLE privileges" }
ELSEIF SQLCODE=-118 {
  WRITE !,"The role already exists" }
ELSE {
  WRITE !,"Created a role. Error code is: ",SQLCODE }
Cleanup
SET toggle=$RANDOM(2)
IF toggle=0 {
  &sql(DROP ROLE BkUser)
  WRITE !,"DROP USER error code: ",SQLCODE
}
ELSE {
  WRITE !,"No drop this time"
  QUIT
}
```

(The **\$RANDOM** toggle is provided so that you can execute this example program repeatedly.)

See Also

- SQL statements: [DROP ROLE CREATE USER DROP USER GRANT REVOKE %CHECKPRIV](#)
- “Users, Roles, and Privileges” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*
- ObjectScript: [\\$ROLES](#) and [\\$USERNAME](#) special variables

CREATE TABLE

Creates a table definition.

```
CREATE [GLOBAL TEMPORARY] TABLE





```

This synopsis does not include keywords that are parsed for compatibility only, but perform no operation. These supported no-op keywords are listed in a separate section below.

Arguments

GLOBAL TEMPORARY	<i>Optional</i> — This keyword clause creates the table as a temporary table.
<i>table</i>	The name of the table to be created, specified as a valid identifier . A table name can be qualified (schema.table), or unqualified (table). An unqualified table name takes the system-wide default schema name .
<i>table-element</i>	<p>A comma-separated list of one or more field definitions or keyword phrases.</p> <p>Each field definition consists of (at minimum) a field name (specified as a valid identifier) followed by a data type.</p> <p>A keyword phrase can consist of just a keyword (%PUBLI-CROWID), a keyword followed by literal, or a keyword (%CLASSPARAMETER) followed by a name and associated literal.</p>
COLLATE <i>sqlcollation</i>	<i>Optional</i> — Specify one of the following SQL collation types: %EXACT, %MINUS, %PLUS, %SPACE, %SQLSTRING, %SQLUPPER, %TRUNCATE, or %MVR. The default is the namespace default collation (%SQLUPPER, unless changed). The %ALPHAUP, %STRING, and %UPPER collation types are deprecated and should not be used. %SQLSTRING, %SQLUPPER, %STRING, and %TRUNCATE may be specified with an optional maximum length truncation argument, an integer enclosed in parentheses. The percent sign (%) prefix to these collation parameter keywords is optional. The COLLATE keyword is optional. For further details refer to Table Field/Property Definition Collation in the “Collation” chapter of <i>Using Caché SQL</i> .
<i>uname</i> <i>pkname</i> <i>fkname</i>	<i>Optional</i> — The name of a constraint, specified as a valid identifier . This optional constraint name is used in ALTER TABLE to identify a defined constraint.
<i>field-commalist</i>	A field name or a comma-separated list of field names in any order. Used to define a unique, primary key, or foreign key constraint. All field names specified for a constraint must also be defined in the field definition . Must be enclosed in parentheses.
<i>reffield-commalist</i>	<i>Optional</i> — A field name or a comma-separated list of existing field names defined in the referenced table specified in the foreign key constraint. If specified, must be enclosed in parentheses. If omitted, a default value is taken, as described in Defining Foreign Keys .

Description

The **CREATE TABLE** command creates a table definition of the structure specified. Caché automatically creates a [persistent class](#) corresponding to this table definition, with properties corresponding to the field definitions. **CREATE TABLE** defines the corresponding class as [DdlAllowed](#). It does not specify an explicit [StorageStrategy](#) in the corresponding class definition; it uses the [default storage %CacheStorage](#). By default, **CREATE TABLE** specifies the [Final](#) class keyword in the corresponding class definition, indicating that it cannot have subclasses. (You can change this default using either the [SetDDLFinal\(\)](#) method, or the corresponding Management Portal [General SQL Settings: DDL tab](#) option.)

This reference page describes the following **CREATE TABLE** considerations:

- [Security and Privileges](#)
- [Creating a Table and then Inserting Data](#)
- [Specifying a Table Name](#)
- [Defining a Temporary Table](#)
- [%DESCRIPTION, %FILE, %EXTENTSIZE, %ROUTINE keywords](#)
- [%CLASSPARAMETER keyword](#)
- [Table Options Supported for Compatibility](#)
- [Defining Fields](#)
- [Defining Field Data Constraints](#)
- [Defining the Unique Fields Constraint](#)
- [The RowID Field, %PUBLICROWID, and Bitmap Extent Index](#)
- [Defining an IDENTITY Field](#)
- [Defining ROWVERSION and SERIAL Fields](#)
- [Defining the Primary Key](#)
- [Defining Foreign Keys](#)

SQL Security and Privileges

The **CREATE TABLE** command is a privileged operation. Prior to using **CREATE TABLE** it is necessary for your process to have [%CREATE_TABLE](#) privileges. Failing to do so results in an `SQLCODE -99` error (Privilege Violation). You can use the [GRANT](#) command to assign [%CREATE_TABLE](#) privileges to a user or role, if you hold appropriate granting privileges.

This privileges requirement is configurable, using either of the following:

- Use the `$$SYSTEM.SQL.SetSQLSecurity()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays an `SQL Security ON:` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **SQL Security Enabled**.

The default is “Yes” (1). When “Yes”, a user can only perform actions on a table or view for which that user has been granted privilege. This is the recommended setting for this option.

If this option is set to “No” (0), SQL Security is disabled for any new process started after changing this setting. This means privilege-based table/view security is suppressed. You can create a table without specifying a user. In this case, Dynamic SQL assigns “_SYSTEM” as user, and Embedded SQL assigns "" (the empty string) as user. Any user can perform actions on a table or view even if that user has no privileges to do so.

Embedded SQL does not use SQL privileges. In Embedded SQL, you can use the `$$SYSTEM.Security.Login()` method to log in as a user with appropriate privileges. You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login()` method. For further information, refer to `%SYSTEM.Security` in the *InterSystems Class Reference*.

The following embedded SQL example creates the Employee table:

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
NEW SQLCODE, %msg
&sql(CREATE TABLE Employee (
  EMPNUM      INT NOT NULL,
  NAMELAST   CHAR(30) NOT NULL,
  NAMEFIRST  CHAR(30) NOT NULL,
  STARTDATE  TIMESTAMP,
  SALARY     MONEY,
  ACCRUEDVACATION INT,
  ACCRUEDSICKLEAVE INT,
  CONSTRAINT EMPLOYEEPK PRIMARY KEY (EMPNUM))
)
IF SQLCODE=0 {WRITE !, "Table created"}
ELSE {WRITE !, "SQLCODE=", SQLCODE, ": ", %msg }
```

This table, named Employee, has a number of defined fields. The EMPNUM field (containing the employee's company ID number) is an integer value that cannot be NULL; additionally, it is declared as a primary key for the table. The employee's last and first names each have a field, both of which are character strings with a maximum length of 30, that cannot be NULL. Additionally, there are fields for the employee's start date, accrued vacation time, and accrued sick time (which use the `TIMESTAMP` and `INT` data types).

Use the following program to delete the table created in the previous example:

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
NEW SQLCODE, %msg
&sql(DROP TABLE Employee)
IF SQLCODE=0 {WRITE !, "Table deleted"}
ELSE {WRITE !, "SQLCODE=", SQLCODE, ": ", %msg }
```

CREATE TABLE and INSERT

Embedded SQL is compiled SQL. In [Embedded SQL](#) you cannot both create a table and insert data into that table in the same program. The reason is as follows: Table creation is performed at runtime. However, the **INSERT** statement needs to verify the existence of the table at compile time. A **SELECT** statement needs to verify the existence of its table(s) at compile time, and thus has the same restriction.

A compiled program can freely combine **CREATE TABLE** statements with DML statements (such as **INSERT** and **SELECT**) that refer to *other* already-existing tables.

You can circumvent this restriction by directing the preprocessor to handle an Embedded SQL program as Deferred SQL. This is done using the `#SQLCompile Mode=Deferred` macro preprocessor directive, as described in the [Preprocessor Directives Reference](#) section of *Using Caché ObjectScript*.

This restriction does not apply to [Dynamic SQL](#), which is parsed at runtime.

You can create a table from an existing table definition and insert data from the existing table in a single operation using the `$$SYSTEM.SQL.QueryToTable()` method.

Table Name

A table name can be qualified or unqualified.

- An unqualified table name has the following syntax: `tablename`; it omits *schema* (and the period `.` character). An unqualified table name takes the [system-wide default schema name](#). The initial system-wide default schema name is `SQLUser`, which corresponds to the default class package name `User`. Schema search path values are ignored.

The [system-wide default schema name can be configured](#).

To determine the current system-wide default schema name, use the `$$SYSTEM.SQL.DefaultSchema()` method.

- A qualified table name has the following syntax: `schema.tablename`. It can specify either an existing schema name or a new schema name. Specifying an existing schema name places the table within that schema. Specifying a new schema name creates that schema (and associated class package) and places the table within that schema.

Table names and schema names follow [SQL identifier](#) naming conventions, subject to additional constraints on the use of non-alphanumeric characters, uniqueness, and maximum length. Names beginning with a % character are reserved for system use. By default, schema names and table names are simple identifiers, and are not case-sensitive.

Caché uses the table name to generate a corresponding class name. Caché uses the schema name is used to generate a corresponding class package name. A class name contains only alphanumeric characters (letters and numbers) and must be unique within the first 96 characters. To generate a class name, Caché first strips out symbol (non-alphanumeric) characters from the table name, and then generates a unique class name, imposing uniqueness and maximum length restrictions. To generate a package name, Caché either strips out or performs special processing of symbol (non-alphanumeric) characters in the schema name. Caché then generates a unique package name, imposing uniqueness and maximum length restrictions. For further details on how package and class names are generated from schema and table names, refer to [Table Names and Schema Names](#) in the “Defining Tables” chapter of *Using Caché SQL*.

You can use the same name for a schema and a table. You cannot use the same name for a table and a view in the same schema.

A schema name is not case-sensitive; the corresponding class package name is case-sensitive. If you specify a schema name that differs only in case from an existing class package name, and the package definition is empty (contains no class definitions) Caché reconciles the two names by changing the case of the class package name. For further details on schema names, refer to [Table Names and Schema Names](#) in the “Defining Tables” chapter of *Using Caché SQL*.

Caché supports 16-bit (wide) characters for table and field names on Unicode systems. For most locales, accented letters can be used for table names and the accent marks are included in the generated class name. The following example performs validation tests on an SQL table name:

```
TableNameValidation
SET tname="MyTestTableName"
SET x=$SYSTEM.SQL.IsValidRegularIdentifier(tname)
IF x=0 {IF $LENGTH(tname)>200
    {WRITE "Tablename is too long" QUIT}
ELSEIF $SYSTEM.SQL.IsReservedWord(tname)
    {WRITE "Tablename is reserved word" QUIT}
ELSE {
    WRITE "Tablename contains invalid characters",!
    SET nls=##class(%SYS.NLS.Locale).%New()
    IF nls.Language [ "Japanese" {
        WRITE "Japanese locale cannot use accented letters"
        QUIT }
    QUIT }
}
ELSE { WRITE tname," is a valid table name"}
```

Note: The Japanese locale does not support accented letter characters in identifiers. Japanese identifiers may contain (in addition to Japanese characters) the Latin letter characters A-Z and a-z (65–90 and 97–122), the underscore character (95), and the Greek capital letter characters (913–929 and 931–937). The `nls.Language` test uses `[` (the Contains operator) rather than `=` because there are different Japanese locales for different operating system platforms.

Existing Table

To determine if a table already exists in the current namespace, use `$SYSTEM.SQL.TableExists()`.

What happens when you try to create a table that has the same name as an existing table depends on a configuration setting. By default, Caché rejects an attempt to create a table with the name of an existing table and issues an `SQLCODE -201` error. This is configurable as follows:

- The `$SYSTEM.SQL.SetDDLNo201()` method call. To determine the current setting, call `$SYSTEM.SQL.CurrentSettings()`, which displays a `Suppress SQLCODE=-201 Errors` setting.

- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow DDL CREATE TABLE or CREATE VIEW for Existing Table**.

The default is “No” (0). This is the recommended setting for this option. If this option is set to “Yes” (1), Caché deletes the class definition associated with the table and then recreates it. This is much the same as performing a **DROP TABLE**, deleting the existing table and then performing the **CREATE TABLE**. In this case, it is strongly recommended that the **Does DDL DROP TABLE Delete the Table's Data** SQL configuration option be set to “Yes” (the default).

GLOBAL TEMPORARY Table

Specifying the GLOBAL TEMPORARY keyword defines the table as a global temporary table. The table definition is global (available to all processes); the table data is temporary (persists for the duration of the process). The corresponding class definition contains an additional Class parameter `SQLTABLETYPE="GLOBAL TEMPORARY"`. Like standard Caché tables, the `ClassType=persistent`, and the class includes the **Final** keyword, indicating that it cannot have subclasses.

Regardless of which process creates a temporary table, the owner of the temporary table is automatically set to `_PUBLIC`. This means that all users can access a cached temporary table definition. For example, if a stored procedure creates a temporary table, the table definition can be accessed by any user that is permitted to invoke the stored procedure. This applies only to the temporary table definition; the temporary table data is specific to the invocation, and therefore can only be accessed by the current user process.

The table definition of a global temporary table is the same as a base table. A global temporary table must have a unique name; attempting to give it the same name as an existing base table results in an `SQLCODE -201` error. The table persists until it is explicitly deleted (using **DROP TABLE**). You can alter the table definition using **ALTER TABLE**.

The table data (including Stream data) and indices in a global temporary table are temporary. They are stored in process-private globals. This means that this data is only available to the process that created the global temporary table, and this data is deleted when the process terminates.

The following embedded SQL example creates a global temporary table:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
NEW SQLCODE, %msg
&sql(CREATE GLOBAL TEMPORARY TABLE TempEmp (
    EMPNUM      INT NOT NULL,
    NAMELAST    CHAR(30) NOT NULL,
    NAMEFIRST   CHAR(30) NOT NULL,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EMPNUM))
)
IF SQLCODE=0 {WRITE !, "Table created"}
ELSE {WRITE !, "SQLCODE=", SQLCODE, ": ", %msg }
```

%DESCRIPTION, %FILE, %EXTENTSIZE / %NUMROWS, %ROUTINE

These optional keyword phrases can be specified anywhere in the comma-separated list of table elements.

Caché SQL provides a `%DESCRIPTION` keyword, which you can use to provide a description for documenting a table or a field. `%DESCRIPTION` is followed by text string enclosed in single quotes. This text can be of any length, and can contain any characters, including blank spaces. (A single quote character within a description is represented by two single quotes. For example: 'Joe' 's Table'.) A table can have a `%DESCRIPTION`. Each field of a table can have its own `%DESCRIPTION`, specified after the data type. If you specify more than one table-wide `%DESCRIPTION` for a table, Caché issues an `SQLCODE -82` error. If you specify more than one `%DESCRIPTION` for a field, the system retains only the last `%DESCRIPTION` specified. In Studio, a description appears prefaced by three slashes on the line immediately before the corresponding table (Class) or field (Property). For example:

```
/// Joe's Table
```

Caché SQL provides a `%FILE` keyword, which is used to provide a file name for documenting a table. `%FILE` is followed by text string enclosed in single quotes. A table definition can have only one `%FILE` keyword; specifying multiples generates an `SQLCODE -83` error.

Caché SQL provides the optional `%EXTENTSIZE` and `%NUMROWS` keywords, which are used to store an integer recording the anticipated number of rows in this table. These two keywords are synonymous; `%EXTENTSIZE` is the preferred Caché term. When a table is being created to hold a known number of rows of data, especially if the initial number of rows is not likely to change subsequently (such as a table of states and provinces), setting `%EXTENTSIZE` can save space and improve performance. If not specified, the default initial allocation is 100,000 for a standard table, 50 for a temporary table. A table definition can have only one `%EXTENTSIZE` or `%NUMROWS` keyword; specifying multiples results in an SQLCODE -84 error. Once the table is populated with data, this `%EXTENTSIZE` value can be changed to the actual number of rows by running [Tune Table](#). For further details, see [“Optimizing Tables”](#).

Caché SQL provides a `%ROUTINE` keyword, which allows you to specify the routine name prefix for routines generated for this base table. `%ROUTINE` is followed by text string enclosed in single quotes. For example, `%ROUTINE 'myname'`, generates code in routines named `myname1`, `myname2`, and so forth. You cannot call a [user-defined \(“extrinsic”\) function](#) from a `%ROUTINE`. A table definition can have only one `%ROUTINE` keyword; specifying multiples results in an SQLCODE -85 error. In Studio, the routine name prefix appears as the `SqlRoutinePrefix` value.

%CLASSPARAMETER Keyword

The optional `%CLASSPARAMETER` keyword enables you to define a class parameter as part of the **CREATE TABLE** command. A class parameter is always defined as a constant value. You can specify multiple `%CLASSPARAMETER` keyword clauses, defining one class parameter per clause. Like all table keyword clauses, `%CLASSPARAMETER` can be specified anywhere in the comma-separated list of table elements; multiple `%CLASSPARAMETER` clauses are separated by commas.

The `%CLASSPARAMETER` keyword is followed by the class parameter name, an optional equal sign, and the literal value (a string or number) to assign to that class parameter. The following example defines two class parameters; the first `%CLASSPARAMETER` clause uses an equal sign, the second omits the equal sign:

```
CREATE TABLE OurEmployees (
    %CLASSPARAMETER DEFAULTGLOBAL = '^EMPLOYEE',
    %CLASSPARAMETER MANAGEDEXTENT 0,
    EMPNUM      INT NOT NULL,
    NAMELAST   CHAR(30) NOT NULL,
    NAMEFIRST  CHAR(30) NOT NULL,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EMPNUM))
```

Some of the class parameters currently in use are: *DEFAULTGLOBAL*, *DSINTERVAL*, *DSTIME*, *EXTENTQUERYSPEC*, *EXTENTSIZE*, *GUIDENABLED*, *IDENTIFIEDBY*, *MANAGEDEXTENT*, *READONLY*, *ROWLEVELSECURITY*, *SQLPREVENTFULLSCAN*, *USEEXTENTSET*, *VERSIONCLIENTNAME*, *VERSIONPROPERTY*. Refer to the `%Library.Persistent` class for descriptions of these class parameters.

The user can specify additional class parameters as needed. For further details refer to [Class Parameters in Using Caché Objects](#).

Options Supported for Compatibility Only

Caché SQL accepts the following **CREATE TABLE** options for parsing purposes only, to aid in the conversion of existing SQL code to Caché SQL. These options do not provide any actual functionality.

```
{ON | IN} dbspace-name
LOCK MODE [ROW | PAGE]
[CLUSTERED | NONCLUSTERED]
WITH FILLFACTOR = literal
MATCH [FULL | PARTIAL]
CHARACTER SET identifier
COLLATE identifier /* But note use of COLLATE keyword, described below */
NOT FOR REPLICATION
```

Field Definition

Following the table name, a set of parentheses contains the definitions of all of the fields (columns) of the table. Definitions of fields are separated by commas. By convention, each field definition is usually presented on a separate line and indentation is used; this is recommended, but not required. After the last field is defined, remember to provide a closing parenthesis for the field definition.

The parts of a field definition are separated by blank spaces. The field name is listed first, followed by its data characteristics. The data characteristics of a field are presented in the following sequence: the data type, the (optional) data size, then the (optional) data constraints. You can then append an optional field %DESCRIPTION to document the field.

Rather than defining a field, a field definition can reference an existing [embedded serial object](#) that defines multiple fields (properties). The field name is followed by the package and class name of the serial object. For example, `Office Sample.Address`. Do not specify a data type or data constraints; you can specify a %DESCRIPTION. You cannot create an embedded serial object using **CREATE TABLE**.

Note: Caché recommends that you avoid creating tables with over 400 columns. Redesign your database so that either: these columns become rows; the columns are divided among several related tables; or the data is stored in fewer columns as character streams or bit streams.

Field Name

Field names follow [identifier](#) conventions, with the same naming restrictions as table names. Field names beginning with a % character should be avoided (field names beginning with %z or %Z are permitted). A field name should not exceed 128 characters. By default, field names are simple identifiers. They are not case-sensitive. Attempting to create a field name that differs only in letter case from another field in the same table generates an SQLCODE -306 error. For further details see the “Identifiers” chapter of *Using Caché SQL*.

Caché uses the field name to generate a corresponding class property name. A property name contains only alphanumeric characters (letters and numbers) and is a maximum of 96 characters in length. To generate this property name, Caché first strips punctuation characters from the field name, and then generates a unique identifier of 96 (or less) characters. Caché substitutes an integer (beginning with 0) for the final character of a field name when this is needed to create a unique property name.

The following example shows how Caché handles field names that differ only in punctuation. The corresponding class properties for these fields are named PatNum, PatNu0, and PatNu1:

```
CREATE TABLE MyPatients (
  _PatNum VARCHAR(16),
  %Pat@Num INTEGER,
  Pat_Num VARCHAR(30),
  CONSTRAINT Patient_PK PRIMARY KEY (_PatNum))
```

The field name, as specified in **CREATE TABLE**, is shown in the class property as the [SqlFieldName](#) keyword value.

During a dynamic **SELECT** operation, Caché may generate property name aliases to facilitate common letter case variants. For example, given the field name Home_Street, Caché might assign the property name aliases home_street, HOME_STREET, and HomeStreet. Caché does not assign an alias if that name would conflict with the name of another field name, or with an alias assigned to another field name.

Data Types

Caché SQL supports most standard SQL [data types](#). A complete list of supported data types is provided in the [Data Types](#) section of this reference.

CREATE TABLE allows you to specify the same data type in several ways: VARCHAR(24), CHARACTER VARYING(24), %Library.String(MAXLEN=24), and %String(MAXLEN=24) all specify the same data type. Note however, that the default MAXLEN may differ: VARCHAR() and CHARACTER VARYING() default to MAXLEN=1; %Library.String and %String default to MAXLEN=50.

Caché maps these standard SQL data types to Caché data types by providing an SQL.SystemDataTypes mapping table and an SQL.UserDataTypes mapping table. SQL.UserDataTypes can be added to by the user to include additional user-defined data types.

To view and modify the current data type mappings, go to the Management Portal, select **[System] > [Configuration] > [System-defined DDL Mappings]**. To create additional data type mappings, go to the Management Portal, select **[System] > [Configuration] > [User-defined DDL Mappings]**.

If you specify a data type in SQL for which no corresponding Caché data type exists, the SQL data type name is used as the data type for the corresponding class property. You must create this user-defined Caché data type before DDL runtime (SQLExecute).

You may also override data type mappings for a single parameter value. For instance, suppose you didn't want VARCHAR(100) to map to the supplied standard mapping %String(MAXLEN=100). You could override this by adding a DDL data type of 'VARCHAR(100)' to the table and then specify its corresponding Caché type. For example:

```
VARCHAR(100) maps to MyString100(MAXLEN=100)
```

Data Size

Following a data type, you can present the permissible data size in parentheses. Whitespace between the data type name and data size parentheses is permitted, but not required.

For a string, data size represents the maximum number of characters. For example:

```
ProductName VARCHAR (64)
```

For a numeric that permits fractional numbers, this is represented as a pair of integers (p,s). The first integer (*p*) is the data type precision, but it is *not* identical to numerical precision (the number of digits in the number). This is because the underlying Caché data type classes do not have a precision, but instead use this number to calculate the MAXVAL and MINVAL; The second integer (*s*) is the scale, which specifies the maximum number of decimal digits. For example:

```
UnitPrice NUMERIC(6,2) /* maximum value 9999.99 */
```

To determine the maximum and minimum permissible values for a field, use the following ObjectScript functions:

```
WRITE $$maxval^%apiSQL(6,2),!
WRITE $$minval^%apiSQL(6,2)
```

Note that because p is not a digit count, it can be smaller than the scale s value:

```
FOR i=0:1:6 {
  WRITE "Max for (" ,i," ,2)=",$$maxval^%apiSQL(i,2),!}
```

For further details, refer to the [Data Types](#) reference page in this manual.

Field Data Constraints

Data constraints govern what values are permitted for a field, what the default value is for a field, and what type of collation is used for data values. All of these data constraints are optional. Multiple data constraints can be specified in any order, separated by a blank space. For further details, see [field-constraint](#).

NULL and NOT NULL

The NOT NULL data constraint keyword specifies that this field does not accept a null value; in other words, every record must have a specified value for this field. NULL and empty string (") are different values in Caché. You can input an empty string into a field that accepts character strings, even if that field is defined with a NOT NULL restriction. You cannot input an empty string into a numeric field. For further details, refer to the [NULL](#) section of the “Language Elements” chapter of *Using Caché SQL*.

The NULL data constraint keyword explicitly specifies that this field can accept a null value; this is the default definition for a field.

UNIQUE

The UNIQUE data constraint specifies that this field accepts only unique values. Thus, no two records can contain the same value for this field. The SQL [empty string](#) (") is considered to be a data value, so with the UNIQUE data constraint applied, no two records can contain an empty string value for this field. A NULL is not considered to be a data value, so the UNIQUE data constraint does not apply to multiple NULLs. To restrict use of NULL for a field, use the NOT NULL keyword constraint.

- The UNIQUE data constraint requires that all of the values for the specified field be unique values.
- The [UNIQUE fields constraint](#) (which uses the CONSTRAINT keyword) requires that all of the values for a specified group of fields when concatenated together result in a unique value. None of the individual fields are required to be limited to unique values.

Refer to the [Constraints option of Catalog Details](#) for ways to list the fields of a table that are defined with a unique constraint.

DEFAULT

The DEFAULT data constraint specifies the default data value that Caché automatically provides for this field during an INSERT operation if the INSERT does not supply a data value for this field. If the INSERT operation supplies NULL for the field data value, the NULL is taken rather than the default data value. It is therefore common to specify both the DEFAULT and the NOT NULL data constraints for the same field.

The DEFAULT value can be supplied as a literal value or as a keyword option. A string supplied as a literal default value must be enclosed in single quotes. A numeric default value does not require single quotes. For example:

```
CREATE TABLE membertest
(MemberId INT NOT NULL,
Membership_status CHAR(13) DEFAULT 'M',
Membership_term INT DEFAULT 2)
```

The DEFAULT value is not validated when creating a table. When defined, a DEFAULT value can ignore data type, data length, and data constraint restrictions. However, when using INSERT to supply data to the table, the DEFAULT value is constrained; it is not limited by data type and data length restrictions, but *is* limited by data constraint restrictions. For example, a field defined `Ordernum INT UNIQUE DEFAULT 'No Number'` can take the default once, ignoring the INT data type restriction, but cannot take the default a second time, as this would violate the UNIQUE field data constraint.

If no DEFAULT is specified, the implied default is NULL. If a field has a NOT NULL data constraint, you must specify a value for that field, either explicitly or by DEFAULT. Do not use the SQL [zero-length string](#) (empty string) as a NOT NULL default value. Refer to [NULL](#) section of the “Language Elements” chapter of *Using Caché SQL* for further details on NULL and the empty string.

DEFAULT Keywords

The DEFAULT data constraint can accept a keyword option to define its value. The following options are supported: NULL, USER, CURRENT_USER, SESSION_USER, SYSTEM_USER, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, SYSDATE, and OBJECTSCRIPT.

The USER, CURRENT_USER, and SESSION_USER default keywords set the field value to the ObjectScript [\\$USERNAME](#) special variable, as described in the *Caché ObjectScript Reference*.

The [CURRENT_DATE](#), [CURRENT_TIME](#), [CURRENT_TIMESTAMP](#), [GETDATE](#), [GETUTCDATE](#), and [SYSDATE](#) SQL functions can also be used as DEFAULT values. They are described in their respective reference pages. You can specify a timestamp function with or without a precision value when used as a DEFAULT value. **CURRENT_TIME** cannot take a precision value when used as a DEFAULT value.

[CURRENT_TIMESTAMP](#), [GETDATE](#), [GETUTCDATE](#), and [SYSDATE](#) can be specified as a default for a **%Library.TimeStamp** field (data type [TIMESTAMP](#) or [DATETIME](#)). Caché converts the date value to the appropriate format for the data type.

```
CREATE TABLE mytest
(TestId INT NOT NULL,
CREATE_DATE DATE DEFAULT CURRENT_TIMESTAMP(2),
WORK_START DATE DEFAULT SYSDATE)
```

You can use the [TO_DATE](#) function as the DEFAULT data constraint for data type DATE. You can use the [TO_TIMESTAMP](#) function as the DEFAULT data constraint for data type [TIMESTAMP](#).

The [OBJECTSCRIPT literal](#) keyword phrase enables you to generate a default value by providing a quoted string containing ObjectScript code, as shown in the following example:

```
CREATE TABLE mytest
(TestId INT NOT NULL,
CREATE_DATE DATE DEFAULT OBJECTSCRIPT '+$SHOROLOG' NOT NULL,
LOGNUM NUMBER(12,0) DEFAULT OBJECTSCRIPT '$INCREMENT(^LogNumber)')
```

See the [Caché ObjectScript Reference](#) for further information.

COMPUTECODE

The COMPUTECODE data constraint specifies ObjectScript code to compute a default data value for this field. The ObjectScript code is specified within curly braces. Within the ObjectScript code, SQL field names can be specified with curly brace delimiters. The ObjectScript code can consist of multiple lines of code. It can contain [Embedded SQL](#). Whitespace and line returns are permitted before or after the ObjectScript code curly brace delimiters.

COMPUTECODE specifies the [SqlComputeCode](#) field name and computation for its value. When you specify a computed field name, either in COMPUTECODE or in the [SqlComputeCode](#) class property, you must specify the SQL field name, not the corresponding generated table property name. The [SqlComputeCode](#) property keyword is described in the *Caché Class Definition Reference*.

A default data value supplied by compute code must be in Logical (internal storage) mode. Embedded SQL in compute code is automatically compiled and run in Logical mode.

The following example defines the Birthday COMPUTECODE field. It use ObjectScript code to compute its default value from the DOB field value:

```
CREATE TABLE MyStudents (
  Name VARCHAR(16) NOT NULL,
  DOB DATE,
  Birthday VARCHAR(10) COMPUTECODE {SET {Birthday}=$PIECE($ZDATE({DOB},9),",",")},
  Grade INT
)
```

The COMPUTECODE can contain the pseudo-field reference variables {%%CLASSNAME}, {%%CLASSNAMEQ}, {%%OPERATION}, {%%TABLENAME}, and {%%ID}. These pseudo-fields are translated into a specific value at class compilation time. All of these pseudo-field keywords are not case-sensitive.

The COMPUTECODE value is a default; it is only returned if you did not supply a value to the field. The COMPUTECODE value is not limited by data type restrictions. The COMPUTECODE value *is* limited by the UNIQUE data constraint and other data constraint restrictions. If you specify both a DEFAULT and a COMPUTECODE, the DEFAULT is always taken.

COMPUTECODE can optionally take a COMPUTEONCHANGE, CALCULATED, or TRANSIENT keyword. The following keyword combination behaviors are supported:

- COMPUTECODE: value is computed and stored upon INSERT, value is not changed upon UPDATE.
- COMPUTECODE with COMPUTEONCHANGE: value is computed and stored upon INSERT, is recomputed and stored upon UPDATE.
- COMPUTECODE with DEFAULT, and COMPUTEONCHANGE: default value is stored upon INSERT, value is computed and stored upon UPDATE.
- COMPUTECODE with CALCULATED or TRANSIENT: no value is stored; value is computed when queried.

If there is an error in the ObjectScript COMPUTECODE code, SQL does not detect this error until the code is executed for the first time. Therefore, if the value is first computed upon insert, the INSERT operation fails with an SQLCODE -415 error; if the value is first computed upon update, the UPDATE operation fails with an SQLCODE -415 error; if the value is first computed when queried, the SELECT operation fails with an SQLCODE -400 error.

A COMPUTECODE stored value can be [indexed](#). The application developer is responsible for making sure that computed field stored values are validated and normalized (numbers in [canonical form](#)), based on their data type, especially if you define (or intend to define) an index for the computed field.

COMPUTEONCHANGE

By itself, COMPUTECODE causes a field value to be computed and stored in the database during INSERT; this value remains unchanged by subsequent operations. By default, subsequent UPDATE or trigger code operations do not change the computed value. Specifying the COMPUTEONCHANGE keyword causes subsequent UPDATE or trigger code operations to recompute and replace this stored value.

If you use the COMPUTEONCHANGE clause to specify a field or comma-separated list of fields, any change to the value of one of these fields causes Caché to recompute the COMPUTECODE field value.

If a field specified in COMPUTEONCHANGE is not part of the table specification, an SQLCODE -31 is generated.

In the following example, Birthday is computed upon insert based on the DateOfBirth value. Birthday is recomputed when DateOfBirth is updated:

```
CREATE TABLE SQLUser.MyStudents (
  Name VARCHAR(16) NOT NULL,
  DateOfBirth DATE,
  Birthday VARCHAR(40) COMPUTECODE {
    SET {Birthday}=$PIECE($ZDATE({DOB},9),",",")
    _" changed: "_$ZTIMESTAMP }
  COMPUTEONCHANGE (DOB)
)
```

Note: An UPDATE to a DateOfBirth value that specifies the existing DateOfBirth value does not recompute the Birthday field value.

COMPUTEONCHANGE defines the [SqlComputeOnChange](#) keyword with the %%UPDATE value for the class property corresponding to the field definition. This property value is initially computed as part of the INSERT operation, and recomputed during an UPDATE operation. For a corresponding Persistent Class definition, refer to [Defining a Table by Creating a Persistent Class](#) in the “Defining Tables” chapter of *Using Caché SQL*.

CALCULATED and TRANSIENT

Specifying the CALCULATED or TRANSIENT keyword specifies that the COMPUTECODE field value is not saved in the database; it is calculated as part of each query operation that accesses it. This reduces the size of the data storage, but may slow query performance. Because these keywords cause Caché to not store the COMPUTECODE field value, these keywords and the COMPUTEONCHANGE keyword are mutually exclusive. The following is an example of a CALCULATED field:

```
CREATE TABLE MyStudents (
  Name VARCHAR(16) NOT NULL,
  DOB DATE,
  Days2Birthday INT COMPUTECODE{SET {Days2Birthday}=$ZD({DOB},14)-$ZD($H,14)} CALCULATED
)
```

CALCULATED defines the [Calculated](#) boolean keyword for the class property corresponding to the field definition. TRANSIENT defines the [Transient](#) boolean keyword for the class property corresponding to the field definition. These property keywords are described in the *Caché Class Definition Reference*.

CALCULATED and TRANSIENT provide nearly identical behavior, with the following differences. TRANSIENT means that Caché does not store the property. CALCULATED means that Caché does not allocate any instance memory for the property. Thus when CALCULATED is specified, TRANSIENT is implicitly set.

TRANSIENT properties cannot be indexed. CALCULATED properties cannot be indexed unless the property is also [SQLComputed](#).

Collation Parameters

The optional collation parameters specify what type of [string collation](#) to use when sorting values for a field. Caché SQL supports ten types of collation. If no collation is specified, the default is %SQLUPPER collation, which is not case-sensitive.

It is recommended that you specify the optional keyword COLLATE before the collation parameter for programming clarity, but this keyword is not required. The percent sign (%) prefix to the various collation parameter keywords is optional.

%EXACT collation follows the ANSI (or Unicode) character collation sequence. This provides case-sensitive string collation and recognizes leading and trailing blanks and tab characters.

The %MVR collation treats a string as a group of substrings. Numeric substrings are sorted in signed numeric collation sequence; non-numeric substrings are sorted in case-sensitive string collation sequence. %MVR is provided for compatibility with MultiValue database systems.

The %ALPHAUP, %SQLUPPER, %STRING, and %UPPER collations convert all letters to uppercase for the purpose of collation. Note that the %ALPHAUP, %STRING, and %UPPER collations are deprecated; %SQLUPPER is the preferred collation for this type of string collation. For further details on not case-sensitive collation, refer to the [%SQLUPPER](#) function.

The %SPACE, %SQLUPPER, and %STRING collations append a blank space to the data. This forces string collation of NULL and numeric values.

The %SQLSTRING, %SQLUPPER, and %STRING collations provide an optional *maxlen* parameter, which must be enclosed in parentheses. *maxlen* is a truncation integer that specifies the maximum number of characters to consider when performing collation. This parameter is useful when creating indices with fields containing large data values.

The %PLUS and %MINUS collations handle NULL as a zero (0) value.

Caché SQL provides functions for most of these collation types. Refer to the [%EXACT](#), [%MVR](#), [%ALPHAUP](#), [%SQLSTRING](#), [%SQLUPPER](#), [%STRING](#), and [%UPPER](#) functions for further details.

ObjectScript provides the **Collation()** method of the %SYSTEM.Util class for data collation conversion.

Note: To change the namespace default collation from %SQLUPPER (which is not case-sensitive) to another collation type, such as %SQLSTRING (which is case-sensitive), use the following command:

```
WRITE $$SetEnvironment^%apiOBJ("collation", "%Library.String", "SQLSTRING")
```

After issuing this command, you must purge indexes, recompile all classes, then rebuild indexes. Do not rebuild indices while the table's data is being accessed by other users. Doing so may result in inaccurate query results.

%DESCRIPTION

You can provide a description text for a field. This option follows the same conventions as providing a description text for a table. It is described with the other table elements, above.

Unique Fields Constraint

The unique fields constraint imposes a unique value constraint on the combined values of multiple fields. It has the following syntax:

```
CONSTRAINT uname UNIQUE (f1, f2)
```

This constraint specifies that the combination of values of fields *f1* and *f2* must always be unique, even though either of these fields by itself may take non-unique values. You can specify one, two, or more than two fields for this constraint.

All of the fields specified in this constraint must be defined in the field definition. If you specify a field in this constraint that does not also appear in the field definitions, an SQLCODE -86 error is generated. The specified fields should be defined as NOT NULL. None of the specified fields should be defined as **UNIQUE**, as this would make specifying this constraint meaningless.

Fields may be specified in any order. The field order dictates the field order for the corresponding index definition. Duplicate field names are permitted. Although you may specify a single field name in the UNIQUE fields constraint, this would be functionally identical to specify the UNIQUE data constraint to that field. A single-field constraint does provide a constraint name for future use.

You may specify multiple unique fields constraint statements in a table definition. Constraint statements can be specified anywhere in the field definition; by convention they are commonly placed at the end of the list of defined fields.

Refer to the [Constraints option of Catalog Details](#) for ways to list the fields of a table that are defined with a unique constraint.

The Constraint Name

The CONSTRAINT keyword and the unique fields constraint name are optional. The following are functionally equivalent:

```
CONSTRAINT myuniquefields UNIQUE (name, dateofbirth)
UNIQUE (name, dateofbirth)
```

The constraint name can be any valid [identifier](#). The constraint name uniquely identifies the constraint, and is also used to derive the corresponding index name. Specifying CONSTRAINT *name* is recommended; this constraint name is required when using the [ALTER TABLE](#) command to drop a constraint from the table definition.

ALTER TABLE cannot drop a column that is listed in CONSTRAINT UNIQUE. Attempting to do so generates an SQLCODE -322 error.

RowID Record Identifier

In SQL, every record is identified by a unique integer value, known as the RowID. In Caché SQL you do not need to specify a RowID field. When you create a table and specify the desired data fields, a RowID field is automatically created. This RowID is used internally, but is not mapped to a class property. By default, its existence is only visible when a class

is projected to an SQL table. In this projected SQL table, an additional RowID field appears. By default, this field is named "ID" and is assigned to column 1. For further details on the RowID, refer to [RowID Field](#) in the “Defining Tables” chapter in *Using Caché SQL*.

%PUBLICROWID

By default, the RowID is hidden and PRIVATE. Specifying the %PUBLICROWID keyword makes the RowID not hidden and public. If you specify the %PUBLICROWID keyword, the class corresponding to the table is defined with “Not Sql-RowIdPrivate”. This optional keyword can be specified anywhere in the comma-separated list of table elements. **ALTER TABLE** cannot be used to specify %PUBLICROWID.

If the RowID is public:

- RowID values are displayed by **SELECT ***.
- The RowID can be used as a [foreign key reference](#).
- If there is no defined primary key, the RowID is treated as an `Implicit PRIMARY KEY` constraint with the [Constraint Name](#) `RowIDField_As_PKey`.

Bitmap Extent Index

When you create a table using **CREATE TABLE**, by default Caché automatically defines a [bitmap extent index](#) for the corresponding class. The SQL MapName of the bitmap extent index is %%DDLBEIndex:

```
Index DDLBEIndex [ Extent, SqlName = "%%DDLBEIndex", Type = bitmap ];
```

This bitmap extent index is *not* created in any of the following circumstances:

- The table is defined as a [temporary table](#).
- The table defines an explicit [IDKEY index](#).
- The table contains a defined [IDENTITY field](#) that does not have `MINVAL=1`.
- A configuration setting overrides this default operation, either the `SetDDLDefineBitmapExtent()` method, or the corresponding Management Portal [General SQL Settings: DDL tab](#) option.

If, after creating a bitmap index, you invoke **CREATE BITMAPEXTENT INDEX** is run against a table where a bitmap extent index was automatically defined, the bitmap extent index previously defined is renamed to the name specified by the **CREATE BITMAPEXTENT INDEX** statement.

For DDL operations that automatically delete an existing bitmap extent index, refer to [ALTER TABLE](#).

For further details refer to “[Bitmap Extent Index](#)” in the “Defining and Building Indices” chapter of *Caché SQL Optimization Guide*.

IDENTITY Field

Caché SQL automatically creates a RowID field for each table, which contains a system-generated integer that serves as a unique record id (see section below). The optional **IDENTITY** keyword allows you to define a named field with the same properties as a RowID record id field. An **IDENTITY** field behaves as a single-field **IDKEY** index, whose value is a unique system-generated integer.

Just as with any system-generated ID field, an **IDENTITY** field has the following characteristics:

- You can only define one field per table as an **IDENTITY** field. Attempting to define more than one **IDENTITY** field for a table generates an `SQLCODE -308` error.
- The data type of an **IDENTITY** field must be an integer data type. If you do not specify a data type, its data type is automatically defined as `INTEGER`. You can specify any integer data type, such as `SMALLINT` or `BIGINT`. Any specified field constraints, such as `NOT NULL` or `UNIQUE` are accepted, but ignored.

- Data values are system-generated. They consist of unique, nonzero, positive integers.
- By default, **IDENTITY** field data values cannot be user-specified. By default, an **INSERT** statement does not, and can not, specify an **IDENTITY** field value. Attempting to do so generates an **SQLCODE -111** error. To determine whether an **IDENTITY** field value can be specified, call the **GetIdentityInsert()** method of the **%SYSTEM.SQL** class. To change this setting, call the **SetIdentityInsert()** method of the **%SYSTEM.SQL** class. For further details, refer to the **INSERT** statement.
- **IDENTITY** field data values cannot be modified in an **UPDATE** statement. Attempting to do so generates an **SQLCODE -107** error.
- The system automatically projects a primary key on the **IDENTITY** field to ODBC and JDBC. If a **CREATE TABLE** or **ALTER TABLE** statement defines a primary key constraint or a unique constraint on an **IDENTITY** field, or on a set of columns including an **IDENTITY** field, the constraint definition is ignored and no corresponding primary key or unique index definition is created.
- A **SELECT *** statement *does* return a table's **IDENTITY** field.

Following an **INSERT**, **UPDATE**, or **DELETE** operation, you can use the **LAST_IDENTITY** function to return the value of the **IDENTITY** field for the most-recently modified record. If no **IDENTITY** field is defined, **LAST_IDENTITY** returns the **RowID** value of the most-recently modified record.

The following two Embedded SQL programs create a table with an **IDENTITY** field and then insert a record into the table, generating an **IDENTITY** field value. Note that in Embedded SQL the **CREATE TABLE** and **INSERT** statements must be in separate programs:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(CREATE TABLE Employee (
EmpNum INT NOT NULL,
MyID IDENTITY NOT NULL,
Name CHAR(30) NOT NULL,
CONSTRAINT EMPLOYEEPK PRIMARY KEY (EmpNum)
)
IF SQLCODE'=0 {
WRITE !,"CREATE TABLE error is: ",SQLCODE }
ELSE {
WRITE !,"Table created" }

&sql(INSERT INTO Employee (EmpNum,Name)
SELECT ID,Name FROM SQLUser.Person WHERE Age >= '25')
IF SQLCODE'=0 {
WRITE !,"INSERT error is: ",SQLCODE }
ELSE {
WRITE !,"Record inserted into table" }
```

In this case, the primary key (**EmpNum**) is taken from the **ID** field of another table. Thus **EmpNum** values are unique integers, but (because of the **WHERE** clause) may contain gaps in their sequence. The **IDENTITY** field, **MyID**, assigns a user-visible unique sequential integer to each record.

ROWVERSION and SERIAL Fields

InterSystems SQL provides two integer counter field data types:

- **ROWVERSION** (**%Library.RowVersion**) counts inserts and updates to all **RowVersion** tables namespace-wide. Only inserts and updates in tables that contain a **ROWVERSION** field increment this integer counter. **ROWVERSION** values are unique and non-modifiable. This namespace-wide counter never resets.
- **SERIAL** (**%Library.Counter**) counts inserts to the table. By default, this field receives an automatically incremented integer. However a user can specify a value to this field. A user-specified value can increase the automatic counter increment starting point.

Defining a Primary Key

You can explicitly define a field (or combination of fields) as the primary record identifier by using the `PRIMARY KEY` clause. There are three syntactic forms for defining a primary key:

```
CREATE TABLE MyTable (Field1 INT PRIMARY KEY, Field2 INT)

CREATE TABLE MyTable (Field1 INT, Field2 INT, PRIMARY KEY (Field1))

CREATE TABLE MyTable (Field1 INT, Field2 INT, CONSTRAINT MyTablePK PRIMARY KEY (Field1))
```

The first syntax defines a field as the primary key; by designating it as the primary key, this field is by definition unique and not null. The second and third syntax can be used for a single field primary key but allow for a primary key consisting of more than one field. For example, `PRIMARY KEY (Field1, Field2)`. If you specify a single field, this field is by definition unique and not null. If you specify a comma-separated list of fields, each field is defined as not null but may contain duplicate values, so long as the combination of the field values is a unique value. The third syntax allows you to explicitly name your primary key; the first two syntax forms generate a primary key name as follows: table name + “PKey” + constraint count integer. For further details on generated primary key names, refer to [Constraints option of Catalog Details](#).

A primary key accepts only unique values and does not accept `NULL`. (The [primary key index](#) property is not automatically defined as Required; however, it effectively is required, since a `NULL` value cannot be filed or saved for a primary key field.) The [collation type](#) of a primary key is specified in the definition of the field itself.

When a class with a defined primary key is projected to SQL, the additional RowID field (default name "ID") appears; by default its values are identical to the values of the IDKEY index.

Refer to the [Constraints option of Catalog Details](#) for ways to list the fields of a table that are defined as the primary key.

When a class with a defined primary key is projected to SQL, the additional RowID field (default name "ID") appears; by default its values are identical to the values of the IDKEY index.

No Primary Key

In most cases, you should explicitly define a primary key. However, if a primary key is not designated, Caché attempts to use another field as the primary key for ODBC/JDBC projection, according to the following rules:

1. If there is an [IDKEY index](#) on a *single* field, report the IDKEY field as the SQLPrimaryKey field.
2. Else if the class is defined with `SqlRowIdPrivate=0` (the default), report the [RowID](#) field as the SQLPrimaryKey field.
3. Else if there is an IDKEY index, report the IDKEY fields as the SQLPrimaryKey fields.
4. Else do not report an SQLPrimaryKey.

Multiple Primary Keys

You can only define one primary key. What happens when you try to specify more than one primary key for a table is configuration-dependent. By default, Caché rejects an attempt to define a primary key when one already exists, or to define the same primary key twice, and issues an `SQLCODE -307` error. You can set this behavior as follows:

- The `$$SYSTEM.SQL.SetDDLNo307()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a `Suppress SQLCODE=-307 Errors` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow Create Primary Key Through DDL When Key Exists**.

The default is “No” (0). If this option is set to “No”, Caché issues an `SQLCODE -307` error when an attempt is made to add a primary key constraint to a table through DDL when a primary key constraint already exists for the table. The error is issued even if the second definition of the primary key is identical to the first definition.

For example, the following **CREATE TABLE** statement:

```
CREATE TABLE MyTable (f1 VARCHAR(16),
CONSTRAINT MyTablePK PRIMARY KEY (f1))
```

creates the primary key (if none exists). A subsequent **ALTER TABLE** statement:

```
ALTER TABLE MyTable ADD CONSTRAINT MyTablePK PRIMARY KEY (f1)
```

generates an SQLCODE -307 error.

If the **Allow Create Primary Key through DDL When Key Exists** option is set to “Yes” (1), Caché drops the existing primary key constraint and establishes the last-specified primary key as the table's primary key.

Defining Foreign Keys

A foreign key is a field that references another table; the value stored in the foreign key field is a value that uniquely identifies a record in the other table. The simplest form of this reference is shown in the following example, in which the foreign key explicitly references the primary key field CustID in the Customers table:

```
CREATE TABLE Orders (
  OrderID INT UNIQUE NOT NULL,
  OrderItem VARCHAR,
  OrderQuantity INT,
  CustomerNum INT,
  CONSTRAINT OrdersPK PRIMARY KEY (OrderID),
  CONSTRAINT CustomersFK FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID)
)
```

Most commonly, a foreign key references the primary key field of the other table. However, a foreign key can reference an IDKEY or an IDENTITY column. In every case, the foreign key reference must exist in the referenced table and must be defined as unique; the referenced field cannot contain duplicate values or NULL.

In a foreign key definition, you can specify:

- One field name: FOREIGN KEY (CustomerNum) REFERENCES Customers (CustID). The foreign key field (CustomerNum) and referenced field (CustID) may have different names (or the same name), but must have the same data type and field constraints.
- A comma-separated list of field names: FOREIGN KEY (CustomerNum, SalespersonNum) REFERENCES Customers (CustID, SalespID). The foreign key fields and referenced fields must correspond in number of fields and in order listed.
- An omitted field name: FOREIGN KEY (CustomerNum) REFERENCES Customers.

If you define a foreign key and omit the referenced field name, the foreign key defaults as follows:

1. The primary key field defined for the specified table.
2. If the specified table does not have a defined primary key, the foreign key defaults to the IDENTITY column defined for the specified table.
3. If the specified table has neither a defined primary key nor a defined IDENTITY column, the foreign key defaults to the RowID. This occurs only if the specified table defines the RowID as public; the specified table definition can do this explicitly, either by specifying the **%PUBLICROWID** keyword, or through the corresponding class definition with **SqlRowIdPrivate=0** (the default). If the specified table does not defines the RowID as public, Caché issues an SQLCODE -315 error. You must omit the referenced field name when defining a foreign key on the RowID; attempting to explicitly specify ID as the referenced field name results in an SQLCODE -316 error.

If none of these defaults apply, Caché issues an SQLCODE -315 error.

Refer to the [Constraints option of Catalog Details](#) for ways to list the fields of a table that are defined as foreign key fields and the generated Constraint Name for a foreign key.

In a class definition you can specify a Foreign Key that contains a field based on a parent table IDKEY property, as shown in the following example:

```
ForeignKey Claim(CheckWriterPost.Hmo,Id,Claim) References SQLUser.Claim.Claim(DBMSKeyIndex);
```

Because the parent field defined in a foreign key of a child has to be part of the parent class's **IDKEY index**, the only referential action supported for foreign keys of this type is **NO ACTION**.

- If a foreign key references a nonexistent table, Caché issues an **SQLCODE -310** error, with additional information provided in %msg.
- If a foreign key references a nonexistent field, Caché issues an **SQLCODE -316** error, with additional information provided in %msg.
- If a foreign key references a nonunique field, Caché issues an **SQLCODE -314** error, with additional information provided in %msg.

If the foreign key field references a single field, the two fields must have the same data type and field data constraints.

To define a **FOREIGN KEY**, the user must have **REFERENCES privilege** on the table being referenced or on the columns of the table being referenced. **REFERENCES** privilege is required if the **CREATE TABLE** is executed via **Dynamic SQL** or **xDBC**.

Referential Action Clause

If a table contains a foreign key, a change to one table has an effect on another table. To keep the data consistent, when you define a foreign key, you also define what effect a change to the record from which the foreign key data comes has on the foreign key value.

A Foreign Key definition may contain two referential action clauses:

```
ON DELETE ref-action
```

and

```
ON UPDATE ref-action
```

The **ON DELETE** clause defines the **DELETE** rule for the referenced table. When an attempt to delete a row from the referenced table is made, the **ON DELETE** clause defines what action should be taken for the row(s) in the referencing table.

The **ON UPDATE** clause defines the **UPDATE** rule for the referenced table. When an attempt to change (update) the primary key value of a row from the referenced table is made, the **ON UPDATE** clause defines what action should be taken for the row(s) in the referencing table.

Caché SQL supports the following Foreign Key referential actions:

- **NO ACTION**
- **SET DEFAULT**
- **SET NULL**
- **CASCADE**

NO ACTION — When a row is deleted or its key value updated in the referenced table, all referencing tables are checked to see if any row references the row being deleted or updated. If so, the delete or update fails. (This constraint does not apply if the foreign key references itself.) **NO ACTION** is the default.

SET NULL — When a row is deleted or its key value updated in the referenced table, all referencing tables are checked to see if any row references the row being deleted or updated. If so, the action causes the foreign key fields which reference the row being deleted or updated to be set to **NULL**. The foreign key field must allow **NULL** values.

SET DEFAULT — When a row is deleted or its key value updated in the referenced table, all referencing tables are checked to see if any row references the row being deleted or updated. If so, the action causes the foreign key fields which reference the row being deleted or updated to be set to the field's default value. If the foreign key field does not have a default value, it will be set to NULL. It is important to note that a row must exist in the referenced table which contains an entry for the default value.

CASCADE — When a row is deleted in the referenced table, all referencing tables are checked to see if any row references the row being deleted. If so, the delete causes rows whose foreign key fields which reference the row being deleted to be deleted as well.

When the key value of a row is updated in the referenced table, all referencing tables are checked to see if any row references the row being updated. If so, the update causes the foreign key fields which reference the row being updated to cascade the update to all referencing rows.

Your table definition should not have two foreign keys with different names that reference the same *identifier-commalist* field(s) and perform contradictory referential actions. In accordance with the ANSI standard, Caché SQL does not issue an error if you define two foreign keys that perform contradictory referential actions on the same field (for example, ON DELETE CASCADE and ON DELETE SET NULL). Instead, Caché SQL issues an error when a **DELETE** or **UPDATE** operation encounters these contradictory foreign key definitions.

Here is an embedded SQL example that issues a **CREATE TABLE** statement that uses both referential action clauses. Note that this example assumes a related table named Physician (with a primary key field of PhysNum) already exists.

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(CREATE TABLE Patient (
    PatNum VARCHAR(16),
    Name VARCHAR(30),
    DOB DATE,
    Primary_Physician VARCHAR(16) DEFAULT 'A10001982321',
    CONSTRAINT Patient_PK PRIMARY KEY (PatNum),
    CONSTRAINT Patient_Physician_FK FOREIGN KEY
        Primary_Physician REFERENCES Physician (PhysNum)
        ON UPDATE CASCADE
        ON DELETE SET NULL)
)
WRITE !, "SQL code: ", SQLCODE
```

For further information refer to the “[Using Foreign Keys](#)” chapter in *Using Caché SQL*.

Implicit Foreign Key

It is preferable to explicitly define all foreign keys. However, it is possible to project implicit foreign keys to ODBC/JDBC and the Management Portal.

If a foreign key is not explicitly defined, the rules for an implicit foreign key are as follows:

1. If there is an explicit foreign key defined, Caché reports this constraint.
2. Else, each reference column in the table is checked to see if the reference is to a table with an index that is a primary key and IDKEY. If so, Caché reports this reference as a foreign key constraint.
3. Else, if the reference field is the parent reference field and the referenced table reports the RowID field as the implicit primary key field, Caché reports this parent reference as a foreign key constraint.

If any of these implicit foreign key constraints are covered by an explicit foreign key definition, the implicit foreign key constraint is not defined.

Examples: Dynamic SQL and Embedded SQL

The following examples demonstrate a **CREATE TABLE** using Dynamic SQL and Embedded SQL. Note that in Dynamic SQL you can create a table and insert data into the table in the same program; in Embedded SQL you must use separate programs to create a table and insert data into that table.

The last program example deletes the table, so that you may run these examples repeatedly.

The following [Dynamic SQL](#) example creates the table `SQLUser.MyStudents`. Note that because `COMPUTECODE` is ObjectScript code, not SQL code, the ObjectScript `$PIECE` function uses double quote delimiters; because the line of code is itself a quoted string, the `$PIECE` delimiters must be escaped as literals by doubling them, as shown:

```
CreateStudentTable
ZNSPACE "Samples"
SET stuDDL=5
SET stuDDL(1)="CREATE TABLE SQLUser.MyStudents ("
SET stuDDL(2)="StudentName VARCHAR(32),StudentDOB DATE,"
SET stuDDL(3)="StudentAge INTEGER COMPUTECODE {SET {StudentAge}="
SET stuDDL(4)="$$PIECE(($PIECE($H,"","1)-{StudentDOB})/365,"."."1)} CALCULATED,"
SET stuDDL(5)="Q1Grade CHAR,Q2Grade CHAR,Q3Grade CHAR,FinalGrade VARCHAR(2))"
SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
SET qStatus = tStatement.%Prepare(.stuDDL)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rtn = tStatement.%Execute()
IF rtn.%SQLCODE=0 {WRITE !,"Table Create successful"}
ELSEIF rtn.%SQLCODE=-201 {WRITE "Table already exists, SQLCODE=",rtn.%SQLCODE,!}
ELSE {WRITE !,"table create failed, SQLCODE=",rtn.%SQLCODE,!
      WRITE rtn.%Message,! }
}
```

The following [Embedded SQL](#) example creates the table `SQLUser.MyStudents`:

```
ZNSPACE "Samples"
&sql(CREATE TABLE SQLUser.MyStudents (
  StudentName VARCHAR(32),StudentDOB DATE,
  StudentAge INTEGER COMPUTECODE {SET {StudentAge}=
  $PIECE(($PIECE($H,"","1)-{StudentDOB})/365,"."1)} CALCULATED,
  Q1Grade CHAR,Q2Grade CHAR,Q3Grade CHAR,FinalGrade VARCHAR(2))
)
IF SQLCODE=0 {WRITE !,"Created table" }
ELSEIF SQLCODE=-201 {WRITE !,"SQLCODE=",SQLCODE," ",%msg }
ELSE {WRITE !,"CREATE TABLE failed, SQLCODE=",SQLCODE }
```

The following example deletes the table created by the prior examples:

```
&sql(DROP TABLE SQLUser.MyStudents)
IF SQLCODE=0 {WRITE !,"Table deleted" }
ELSE {WRITE !,"SQLCODE=",SQLCODE," ",%msg }
```

See Also

- [ALTER TABLE, DROP TABLE](#)
- [SELECT, JOIN](#)
- [INSERT, UPDATE, INSERT OR UPDATE](#)
- “[Defining Tables](#)” chapter in *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

CREATE TRIGGER

Creates a trigger.

```
CREATE TRIGGER trigname {BEFORE | AFTER} event
  [ORDER integer]
  ON table
  [REFERENCING {OLD | NEW} [ROW] [AS] alias]
  action
```

Arguments

<i>trigname</i>	The name of the trigger to be created, which is an identifier . A trigger name may be qualified or unqualified; if qualified, its schema name must match the table's schema name. For further details see the "Identifiers" chapter of <i>Using Caché SQL</i> .
BEFORE <i>event</i> AFTER <i>event</i>	The time (BEFORE or AFTER) and type of trigger event. Available <i>event</i> options are INSERT, DELETE, UPDATE, and UPDATE OF. The UPDATE OF clause is followed by a column name or a comma-separated list of column names. The UPDATE OF clause can only be specified when LANGUAGE is SQL.
ORDER <i>integer</i>	<i>Optional</i> — The order in which triggers should be executed when there are multiple triggers for a table with the same time and event. If order is omitted, a trigger is assigned an order of 0.
ON <i>table</i>	The table the trigger is created for. A table name may be qualified or unqualified ; if qualified, the trigger must reside in the same schema as the table.
REFERENCING OLD ROW AS <i>alias</i> REFERENCING NEW ROW AS <i>alias</i>	<i>Optional</i> — A REFERENCING clause can only be used when LANGUAGE is SQL. A REFERENCING clause allows you to specify an alias that you can use to reference a column. REFERENCING OLD ROW allows you reference the old value of a column during an UPDATE or DELETE trigger. REFERENCING NEW ROW allows you to reference the new value of a column during an INSERT or UPDATE trigger. The ROW AS keywords are optional. For an UPDATE, you can specify both OLD and NEW in the same REFERENCING clause, as follows: REFERENCING OLD <i>oldalias</i> NEW <i>newalias</i> .

<i>action</i>	The program code for the trigger. The <i>action</i> argument can contain various optional keyword clauses, including (in order): a FOR EACH ROW clause; a WHEN clause with a predicate condition governing execution of the triggered action; and a LANGUAGE clause which specifies either LANGUAGE SQL or LANGUAGE OBJECTSCRIPT. If the LANGUAGE clause is omitted, SQL is the default. Following these clauses, you specify one or more lines of code specifying the action to perform when the trigger is executed.
---------------	--

Description

The **CREATE TRIGGER** command defines a trigger, a block of code to be executed when data in a specific table is modified. A trigger is executed (“fired”) when a specific triggering event occurs, such as a new row being inserted into a specified table. A triggering event may be an **INSERT**, **DELETE**, or **UPDATE** command. A trigger executes user-specified trigger code. You can specify that the trigger should execute this code before or after the execution of the triggering event. A trigger is specific to a specified table.

A single-event trigger is triggered by a specified **INSERT**, **DELETE**, or **UPDATE** operation. A multiple-event trigger is defined to execute when any one of the specified events occurs on the specified table. You can define an INSERT/UPDATE, an UPDATE/DELETE, or an INSERT/UPDATE/DELETE multiple-event trigger.

Trigger *action* code cannot modify data in the triggering record. For example, if an update to the data in a table column fires a trigger, that trigger’s code block cannot insert, update, or delete data in any of the records of that table.

Privileges and Locking

The **CREATE TRIGGER** command is a privileged operation. Before using **CREATE TRIGGER** it is necessary for your process to have %CREATE_TRIGGER privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can use the **GRANT** command to assign %CREATE_TRIGGER privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the `$$SYSTEM.Security.Login()` method to log in as a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login( "_SYSTEM", "SYS" )
&sql(      )
```

You must have the %Service_Login:Use privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

CREATE TRIGGER cannot be used on a [table created by defining a persistent class](#), unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg DDL not enabled for class 'Schema.tablename'.

The **CREATE TRIGGER** statement acquires a table-level lock on *table*. This prevents other processes from modifying the table’s data. This lock is automatically released at the conclusion of the **CREATE TRIGGER** operation.

Other Ways of Defining Triggers

You can define an SQL trigger as a Caché Object by including the `FOREACH = ROW/OBJECT` statement. The following is an example of an Object trigger:

```
Trigger SQLJournal [ CodeMode = objectgenerator, Event = INSERT/UPDATE, ForEach = ROW/OBJECT, Time = AFTER ]
{ /* ObjectScript trigger code
   that updates a journal file
   after a row is inserted or updated. */
}
```

Note: Caché SQL triggers are wholly separate from Caché class methods such as `%OnBeforeSave()`, `%OnAfterSave()`, and `%AddToSaveSet()` that may be invoked when a `%Save()` method is issued, and both of these are wholly separate from [Caché MultiValue triggers](#). A data modification operation in one realm can only fire the triggers defined for that realm. For example, a Caché SQL data modification operation cannot fire a MultiValue trigger. A Caché MultiValue data modification operation cannot fire a Caché SQL trigger.

Arguments

trigname

A trigger name follows the same [identifier](#) requirements as a table name, but not the same uniqueness requirements. A trigger name must be unique for a table within a schema. Thus, triggers referencing different tables in a schema may have the same name. A trigger and its associated table must reside in the same schema. You cannot use the same name for a trigger and a table in the same schema. Violating trigger naming conventions results in an `SQLCODE -400` error at **CREATE TRIGGER** execution time.

A trigger name may be unqualified or qualified. A qualified trigger name has the form:

```
schema.trigger
```

If the trigger name is unqualified, the trigger schema name defaults to the same schema as the specified table schema. If the table name is unqualified, the table schema name defaults to the same schema as the specified trigger schema. If both are unqualified, the [system-wide default schema name](#) is used; schema search paths are not used. If both are qualified, the trigger schema name must be the same as the table schema name. A schema name mismatch results in an `SQLCODE -366` error; this should only occur when both the trigger name and the table name are qualified and they specify different schema names.

Trigger names follow [identifier](#) conventions, subject to the restrictions below. By default, trigger names are simple identifiers. A trigger name should not exceed 128 characters. Trigger names are not case-sensitive.

Caché uses *trigname* to generate a corresponding trigger name in the Caché class. The corresponding class trigger name contains only alphanumeric characters (letters and numbers) and is a maximum of 96 characters in length. To generate this Caché identifier name, Caché first strips punctuation characters from the trigger name, and then generates a unique identifier of 96 (or less) characters, substituting a number for the 96th character when needed to create a unique name. This name generation imposes the following restrictions on the naming of triggers:

- A trigger name must include at least one letter. Either the first character of the trigger name or the first character after initial punctuation characters must be a letter.
- Caché supports 16-bit (wide) characters for trigger names on Unicode systems. A character is a valid letter if it passes the `$ZNAME` test.
- Because names generated for a Caché class do not include punctuation characters, it is not advisable (though possible) to create trigger names that differ only in their punctuation characters.
- A trigger name may be much longer than 96 characters, but trigger names that differ in their first 96 alphanumeric characters are much easier to work with.

event

A trigger specified as `INSERT` is executed when a row is inserted into the specified table. A trigger specified as `DELETE` is executed when a row is deleted from the specified table. A trigger specified as `UPDATE` is executed when a row is updated in the specified table.

A trigger specified as `UPDATE OF` is executed only when one or more of the specified columns is updated in a row in the specified table. Column names are specified as a comma-separated list. Column names can be specified in any order, but

duplicate column names are not permitted; this results in an SQLCODE -58 error at compile time. The UPDATE OF clause is only valid if the trigger code LANGUAGE is SQL (the default).

The time that the trigger is fired is specified by the BEFORE or AFTER keyword; these keywords specify that the trigger operation should occur either before or after Caché executes the triggering *event*. A BEFORE trigger is executed before performing the specified *event*, but *after* verifying the *event*. For example, Caché only executes a BEFORE DELETE trigger if the **DELETE** statement is valid for the specified row(s), and the process has the necessary privileges to perform the **DELETE**, including any foreign key referential integrity checks. If the process cannot perform the specified *event*, Caché issues an error code for the *event*; it does not execute the BEFORE trigger.

The following are examples of four of the eight possible *event* types:

```
CREATE TRIGGER TrigBI BEFORE INSERT ON Sample.Person
    INSERT INTO TLog VALUES ('before insert');

CREATE TRIGGER TrigAU AFTER UPDATE ON Sample.Person
    INSERT INTO TLog VALUES ('after update');

CREATE TRIGGER TrigBUOF BEFORE UPDATE OF Home_City,Home_State ON Sample.Person
    INSERT INTO TLog VALUES ('before address update');

CREATE TRIGGER TrigAD AFTER DELETE ON Sample.Person
    INSERT INTO TLog VALUES ('after delete');
```

ORDER

The ORDER clause determines the order in which triggers are executed when there are multiple triggers for the same table with the same time and event. For example, two AFTER DELETE triggers. The trigger with the lowest ORDER integer is executed first, then the next higher integer, and so on. If the ORDER clause is not specified, a trigger is created with an assigned ORDER number of 0 (zero). Thus, triggers with no ORDER clause are always executed before triggers with ORDER clauses.

You can assign the same order value to multiple triggers. You can also create multiple triggers with an (implicit or explicit) order of 0. Multiple triggers with the same time, event, and order are executed together in random order.

Triggers are executed in the sequence: time > order > event. Thus if you have a BEFORE INSERT trigger and a BEFORE INSERT/UPDATE trigger, the trigger with the lowest ORDER value would be executed first. If you have a BEFORE INSERT trigger and a BEFORE INSERT/UPDATE trigger with the same ORDER value, the INSERT is executed before the INSERT/UPDATE. This is because — time and order being the same — a single-event trigger is always executed before a multi-event trigger. If two (or more) triggers have identical time, order, and event values, the order of execution is random.

The following examples show how ORDER numbers work. All of these **CREATE TRIGGER** statements create triggers that are executed by the same event:

```
CREATE TRIGGER TrigA BEFORE DELETE ON doctable
    INSERT INTO TLog VALUES ('doc deleted');
-- Assigned ORDER=0

CREATE TRIGGER TrigB BEFORE DELETE ORDER 4 ON doctable
    INSERT INTO TReport VALUES ('doc deleted')
-- Specified as ORDER=4

CREATE TRIGGER TrigC BEFORE DELETE ORDER 2 ON doctable
    INSERT INTO Ttemps VALUES ('doc deleted')
-- Specified as ORDER=2

CREATE TRIGGER TrigD BEFORE DELETE ON doctable
    INSERT INTO Tflags VALUES ('doc deleted')
-- Also assigned ORDER=0
```

These triggers will execute in the sequence: (TrigA, TrigD), TrigC, TrigB. Note that TrigA and TrigD have the same order number, and thus execute in random sequence.

REFERENCING

The REFERENCING clause can specify an alias for the old value of a row, the new value of a row, or both. The old value is the row value before the triggered action of an UPDATE or DELETE trigger. The new value is the row value after the triggered action of an UPDATE or INSERT trigger. For an UPDATE trigger, you can specify aliases for both the before and after row values, as follows:

```
REFERENCING OLD ROW AS oldalias NEW ROW AS newalias
```

The keywords ROW and AS are optional. Therefore, the same clause can also be specified as:

```
REFERENCING OLD oldalias NEW newalias
```

It is not meaningful to refer to an OLD value before an INSERT or a NEW value after a DELETE. Attempting to do so results in an SQLCODE -48 error at compile time.

A REFERENCING clause can only be used when the *action* program code is SQL. Specifying a REFERENCING clause with the LANGUAGE OBJECTSCRIPT clause results in an SQLCODE -49 error.

action

A triggered action consists of the following elements:

- An optional FOR EACH clause. The available values are FOR EACH ROW, FOR EACH ROW_AND_OBJECT, and FOR EACH STATEMENT. FOR EACH ROW means that this trigger is invoked by an SQL filing operation. FOR EACH ROW_AND_OBJECT means that this trigger is invoked by either an SQL filing operation or a Caché Objects filing operation. FOR EACH STATEMENT is provided for compatibility with [TSQL triggers](#).
- An optional WHEN clause, consisting of the WHEN keyword followed by a predicate condition (simple or complex) enclosed in parentheses. If the predicate condition evaluates to TRUE, the trigger is executed. A WHEN clause can only be used when LANGUAGE is SQL. The WHEN clause can reference *oldalias* or *newalias* values. For further details on predicate condition expressions and a list of available predicates, refer to the [Overview of Predicates](#) page in this document.
- An optional LANGUAGE clause, either LANGUAGE SQL or LANGUAGE OBJECTSCRIPT. The default is LANGUAGE SQL.
- User-written code that is executed when the trigger is executed.

SQL Trigger Code

If LANGUAGE SQL (the default), the triggered statement is an SQL procedure block, consisting of either one SQL procedure statement followed by a semicolon, or the keyword BEGIN followed by one or more SQL procedure statements, each followed by a semicolon, concluding with an END keyword.

A triggered action is atomic, it is either fully applied or not at all, and cannot contain **COMMIT** or **ROLLBACK** statements. The keyword BEGIN ATOMIC is synonymous with the keyword BEGIN.

If LANGUAGE SQL, the **CREATE TRIGGER** statement can optionally contain a REFERENCING clause, a WHEN clause, and/or an UPDATE OF clause. An UPDATE OF clause specifies that the trigger should only be executed when an **UPDATE** is performed on one or more of the columns specified for this trigger. A **CREATE TRIGGER** statement with LANGUAGE OBJECTSCRIPT cannot contain these clauses.

SQL trigger code is executed as embedded SQL. This means that Caché converts SQL trigger code to ObjectScript; therefore, if you use Studio to view the class definition corresponding to your SQL trigger code, the trigger definition in Studio will show Language Mode as Caché.

When executing SQL trigger code, the system automatically resets (NEWs) all variable used in the trigger code. After the execution of each SQL statement, Caché checks SQLCODE. If an error occurs, Caché sets the %ok variable to 0, aborting and rolling back both the trigger code operation(s) and the associated **INSERT**, **UPDATE**, or **DELETE**.

ObjectScript Trigger Code

If LANGUAGE OBJECTSCRIPT, the **CREATE TRIGGER** statement cannot contain a REFERENCING clause, a WHEN clause, or an UPDATE OF clause. Specifying these SQL-only clauses with LANGUAGE OBJECTSCRIPT results in compile-time SQLCODE errors -49, -57, or -50, respectively.

If LANGUAGE OBJECTSCRIPT, the triggered statement is a block of one or more ObjectScript statements, enclosed by curly braces.

Because the code for a trigger is not generated as a procedure, all local variables in a trigger are public variables. This means all variables in triggers should be explicitly declared with a **NEW** statement; this protects them from conflicting with variables in the code that invokes the trigger.

If trigger code contains Caché [Macro Preprocessor statements](#) (# commands, ## functions, or \$\$\$macro references) these statements are compiled *before* the **CREATE TRIGGER** DDL code itself.

You can issue an error from trigger code by setting the %ok variable to 0. This creates a runtime error that aborts execution of the trigger. Trigger code can also set the %msg variable to a string describing the cause of the runtime error.

As of version 2012.1, the system generates trigger code only once, even for a multiple-event trigger.

Field References and Pseudo-field References

Trigger code written in ObjectScript can contain field references, specified as `{fieldname}`, where *fieldname* specifies an existing field in the current table. No blank spaces are permitted within the curly braces.

You can follow the *fieldname* with *N (new), *O (old), or *C (compare) to specify how to handle an inserted, updated, or deleted field data value, as follows:

- `{fieldname*N}`
 - For **UPDATE**, returns the new field value after the specified change is made.
 - For **INSERT**, returns the value inserted.
 - For **DELETE**, returns the value of the field before the delete.
- `{fieldname*O}`
 - For **UPDATE**, returns the old field value before the specified change is made.
 - For **INSERT**, returns NULL.
 - For **DELETE**, returns the value of the field before the delete.
- `{fieldname*C}`
 - For **UPDATE**, returns 1 (TRUE) if the new value differs from the old value, otherwise returns 0 (FALSE).
 - For **INSERT**, returns 1 (TRUE) if the inserted value is non-NULL, otherwise returns 0 (FALSE).
 - For **DELETE**, returns 1 (TRUE) if the value being deleted is non-NULL, otherwise returns 0 (FALSE).

For **UPDATE**, **INSERT**, or **DELETE**, `{fieldname}` returns the same value as `{fieldname*N}`.

Line returns are not permitted within a statement that sets a field value. For further details, refer to the [SqlComputeCode](#) property keyword in the *Caché Class Definition Reference*.

You can use the `GetColumns()` method to list the field names defined for a table. For further details, refer to [Column Names and Numbers](#) in the “Defining Tables” chapter of *Using Caché SQL*.

Trigger code written in ObjectScript can also contain the pseudo-field reference variables `{%%CLASSNAME}`, `{%%CLASSNAMEQ}`, `{%%OPERATION}`, `{%%TABLENAME}`, and `{%%ID}`. The pseudo-fields are translated into a specific value at class compilation time. All of these pseudo-field keywords are not case-sensitive.

- `{%%CLASSNAME}` and `{%%CLASSNAMEQ}` both translate to the name of the class which projected the SQL table definition. `{%%CLASSNAME}` returns an unquoted string and `{%%CLASSNAMEQ}` returns a quoted string.
- `{%%OPERATION}` translates to a string literal, either INSERT, UPDATE, or DELETE, depending on the operation that invoked the trigger.
- `{%%TABLENAME}` translates to the [fully qualified name of the table](#), returned as a quoted string.
- `{%%ID}` translates to the [RowID name](#). This reference is useful when you do not know the name of the RowID field.

Referencing Stream Property

When a [Stream field/property](#) is referenced in a trigger definition, like `{StreamField}`, `{StreamField*O}`, or `{StreamField*N}`, the value of the `{StreamField}` reference is the stream's OID (object ID) value.

For a BEFORE INSERT or BEFORE UPDATE trigger, if a new value is specified by the INSERT/UPDATE/ObjectSave, the `{StreamField*N}` value will be either the OID of the temporary stream object, or the new literal stream value. For a BEFORE UPDATE trigger, if a new value is not specified for the stream field/property, `{StreamField*O}` and `{StreamField*N}` will both be the OID of the current field/property stream object.

Referencing SQLComputed Property

When a transient `SqlComputed` field/property (either "Calculated" or explicitly "Transient") is referenced in a trigger definition, `Get()/Set()` method overrides are not recognized by the trigger. Use [SQLCOMPUTED/SQLCOMPUTONCHANGE](#), rather than overriding the property's `Get()` or `Set()` method.

Using `Get()/Set()` method overrides can result in the following erroneous result: The `{property*O}` value is determined using SQL and does not use the overridden `Get()/Set()` methods. Because the property is not stored on disk, `{property*O}` uses the `SqlComputeCode` to "recreate" the old value. However, `{property*N}` uses the overridden `Get()/Set()` methods to access the property's value. As a result, there is a possibility for `{property*O}` and `{property*N}` to be different (and thus `{property*C}=1`) even though the property did not actually change.

Labels

Trigger code may contain [line labels](#) (tags). To specify a label in trigger code, prefix the label line with a colon to indicate that this line should begin in the first column. Caché strips out the colon and treats the remaining line as a label. However, because trigger code is generated outside the scope of any procedure blocks, every label must be unique throughout the class definition. Any other code compiled into the class's routine must not have the same label defined, including in other triggers, in non-procedure block methods, [SqlComputeCode](#), and other code.

Note: This use of a colon prefix for a label takes precedence over the use of a colon prefix for a [host variable](#) reference. To avoid this conflict, it is recommended that embedded SQL trigger code lines never begin with a host variable reference. If you must begin a trigger code line with a host variable reference, you can designate it as a host variable (and not a label) by doubling the colon prefix.

Method Calls

You can call class methods from within trigger code, because class methods do not depend on having an open object. You must use the `##class(classname).Method()` syntax to invoke a method. You cannot use the `..Method()` syntax, because this syntax requires a current open object.

You can pass the value of a field of the current row as an argument of the class method, but the class method itself cannot use field syntax.

Listing Existing Triggers

You can use the `INFORMATION.SCHEMA.TRIGGERS` class to list the currently defined triggers. This class lists for each trigger the name of the trigger, the associated schema and table name, and the trigger creation timestamp. For each trigger it lists the `EVENT_MANIPULATION` property (`INSERT`, `UPDATE`, `DELETE`, `INSERT/UPDATE`, `INSERT/UPDATE/DELETE`) and `ACTION_TIMING` property (`BEFORE`, `AFTER`). It also lists the `ACTION_STATEMENT`, which is the generated SQL trigger code.

Trigger Runtime Errors

A trigger and its invoking event execute as an atomic operation on a single row basis. That is:

- A failed `BEFORE` trigger is rolled back, the associated **INSERT**, **UPDATE**, or **DELETE** operation is not executed, and all locks on the row are released.
- A failed `AFTER` trigger is rolled back, the associated **INSERT**, **UPDATE**, or **DELETE** operation is rolled back, and all locks on the row are released.
- A failed **INSERT**, **UPDATE**, or **DELETE** operation is rolled back, the associated `BEFORE` trigger is rolled back, and all locks on the row are released.
- A failed **INSERT**, **UPDATE**, or **DELETE** operation is rolled back, the associated `AFTER` trigger is not executed, and all locks on the row are released.

Note that integrity is maintained for the current row operation only. Your application program must handle data integrity issues involving operation on multiple rows by using transaction processing statements.

Because a trigger is an atomic operation, you cannot code transaction statements, such as commits and rollbacks, within trigger code.

If an **INSERT**, **UPDATE**, or **DELETE** operation causes multiple triggers to execute, the failure of one trigger causes all remaining triggers to remain unexecuted.

When a database operation fails because of a fatal runtime error, Caché issues an `SQLCODE -415` error. When a trigger operation fails, Caché issues one of the `SQLCODE` error codes `-130` through `-135` indicating the type of trigger that failed. You can force a trigger to fail by setting the `%ok` variable to `0` in the trigger code. You can also set the `%msg` variable to a string containing a message to be returned upon trigger failure.

Examples

The following two Embedded SQL programs demonstrate `CREATE TRIGGER` with an ObjectScript `INSERT` trigger. The first example creates a table and an `INSERT` trigger for that table. The second program issues an `INSERT` against the table, executing the trigger which writes a message. It then drops the table so that this program can be run repeatedly:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(CREATE TABLE TestDummy (
    testnum    INT NOT NULL,
    firstword  CHAR (30) NOT NULL,
    lastword   CHAR (30) NOT NULL,
    CONSTRAINT TestDummyPK PRIMARY KEY (testnum)
)
WRITE !,"SQL table code is: ",SQLCODE
&sql(CREATE TRIGGER TrigTestDummy AFTER INSERT ON TestDummy
    LANGUAGE OBJECTSCRIPT
    {WRITE "I just fired the trigger" }
)
WRITE !,"SQL trigger code is: ",SQLCODE
```

```

NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO TestDummy (testnum,firstword,lastword) VALUES
(46639,'hello','goodbye'))
IF SQLCODE=0 {
WRITE !,"Insert succeeded"
WRITE !,"Row count=",%ROWCOUNT
WRITE !,"Row ID=",%ROWID }
ELSE {
WRITE !,"Insert failed, SQLCODE=","SQLCODE }
&sql(DROP TABLE TestDummy)

```

The following examples demonstrate CREATE TRIGGER with an SQL INSERT trigger. The first embedded SQL program creates a table, an INSERT trigger for that table, and a log table for the trigger's use. The second embedded SQL program issues an INSERT against the table, which invokes the trigger, which logs an entry in the log table. After displaying the log entry, the program drops both tables so that this program can be run repeatedly:

```

DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(CREATE TABLE TestDummy (
testnum INT NOT NULL,
firstword CHAR (30) NOT NULL,
lastword CHAR (30) NOT NULL,
CONSTRAINT TestDummyPK PRIMARY KEY (testnum)
)
)
WRITE !,"SQL table code is: ",SQLCODE
&sql(CREATE TABLE TestDummyLog (
entry CHAR (60) NOT NULL)
)
WRITE !,"SQL log table code is: ",SQLCODE
&sql(CREATE TRIGGER TrigTestDummy AFTER INSERT ON TestDummy
LANGUAGE SQL
BEGIN
INSERT INTO TestDummyLog (entry) VALUES
(CURRENT_TIMESTAMP||' INSERT to TestDummy');
END )
)
WRITE !,"SQL trigger code is: ",SQLCODE

NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO TestDummy (testnum,firstword,lastword) VALUES
(46639,'hello','goodbye'))
IF SQLCODE=0 {
WRITE !,"Insert succeeded"
WRITE !,"Row count=",%ROWCOUNT
WRITE !,"Row ID=",%ROWID }
ELSE {
WRITE !,"Insert failed, SQLCODE=","SQLCODE }
&sql(SELECT entry INTO :logitem FROM TestDummyLog)
WRITE !,"Log entry: ",logitem
&sql(DROP TABLE TestDummy)
&sql(DROP TABLE TestDummyLog)
WRITE !,"finished!"

```

The following example includes a WHEN clause that specifies that the *action* should only be performed when the predicate condition in parentheses is met:

```

CREATE TRIGGER Trigger_2 AFTER INSERT ON Table_1
WHEN (f1 %STARTSWITH 'A')
BEGIN
INSERT INTO Log_Table VALUES (new_row.Category);
END

```

See Also

- [DROP TRIGGER](#)
- [GRANT](#)
- “Using Triggers” chapter in *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

CREATE USER

Creates a user account.

```
CREATE USER user-name IDENTIFY BY password
CREATE USER user-name IDENTIFIED BY password
```

Arguments

<i>user-name</i>	The name of the user to be created. The name is an identifier with a maximum of 128 characters. It can contain Unicode letters. <i>user-name</i> is not case-sensitive. For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> .
<i>password</i>	The password for this user. A <i>password</i> must be at least 3 characters, and cannot exceed 32 characters. Passwords are case-sensitive. Passwords can contain Unicode characters.

Description

The **CREATE USER** command creates a user account with the specified password.

A *user-name* can be any valid [identifier](#) of up to 128 characters. The user name can be an SQL reserved word if it is a delimited identifier enclosed in quotation marks, and **Support Delimited Identifiers=YES**. User names are not case-sensitive.

The IDENTIFY BY and IDENTIFIED BY keywords are synonyms.

A *password* can be a numeric literal, an identifier, or a quoted string. A numeric literal or an identifier does not have to be enclosed in quotes. A quoted string is commonly used to include blanks in a password; a quoted password can contain any combination of characters, with the exception of the quote character itself. A numeric literal must consist of only the characters 0 through 9. An [identifier](#) must start with a letter (uppercase or lowercase) or a % (percent symbol); this can be followed by any combination of letters, numbers, or any of the following symbols: _ (underscore), & (ampersand), \$ (dollar sign), or @ (at sign).

Passwords are case-sensitive. A password must be at least three characters, and less than 33 characters, in length. Specifying a password that is too long or too short generates an SQLCODE -400 error, with a %msg value of “ERROR #845: Password does not match length or pattern requirements”.

You cannot use a host variable to specify a *user-name* or *password* value.

Creating a user does not create any roles or grant any roles to the user. Instead, the user is given permissions for the database they are logging into, and USE permission on the %SQL/Service service if the user holds at least one SQL privilege in the namespace. To assign privileges or roles to a user, use the **GRANT** command. To create roles, use the **CREATE ROLE** command.

If you invoke **CREATE USER** to create a user that already exists, SQL issues an SQLCODE -118 error, with a %msg value of “User named '*name*' already exists”. You can determine if a user already exists by invoking the **\$\$SYSTEM.SQL.UserExists()** method:

```
WRITE $$SYSTEM.SQL.UserExists("Admin"),!
WRITE $$SYSTEM.SQL.UserExists("BertieWooster")
```

This method returns 1 if the specified user exists, and 0 if the user does not exist. User names are not case-sensitive.

Privileges

The **CREATE USER** command is a privileged operation. Prior to using **CREATE USER** in embedded SQL, it is necessary to be logged in as a user with appropriate privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the **\$\$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql( /* SQL code here */ )
```

You must have the `%Service_Login:Use` privilege to invoke the `$SYSTEM.Security.Login` method. For further information, refer to `%SYSTEM.Security` in the *InterSystems Class Reference*.

Examples

The following embedded SQL example creates a new user named “BillTest” with a password of “Carl4SHK”. (The `$RANDOM` toggle is provided so that you can execute this example program repeatedly.)

```
Main
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
SET x=$SYSTEM.SQL.UserExists("BillTest")
IF x=0 {&sql(CREATE USER BillTest IDENTIFY BY Carl4SHK)
      IF SQLCODE '= 0 {WRITE "CREATE USER error: ",SQLCODE,!
                    QUIT}
      }
WRITE "User BillTest exists",!
Cleanup
SET toggle=$RANDOM(2)
IF toggle=0 {
  &sql(DROP USER BillTest)
  IF SQLCODE '= 0 {WRITE "DROP USER error: ",SQLCODE,!}
}
ELSE {WRITE !,"No drop this time",!}
WRITE "User BillTest exists? ",$SYSTEM.SQL.UserExists("BillTest"),!
QUIT
```

See Also

- SQL statements: [ALTER USER](#), [DROP USER](#), [GRANT](#), [REVOKE](#), [CREATE ROLE](#)
- “Users, Roles, and Privileges” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*
- ObjectScript: [\\$ROLES](#) and [\\$USERNAME](#) special variables

CREATE VIEW

Creates a view.

```
CREATE [OR REPLACE] VIEW view-name [(column-commalist)]
AS select-statement
[ WITH READ ONLY | WITH [level] CHECK OPTION ]
```

Arguments

<i>view-name</i>	The name for the view being created. A valid identifier , subject to the same additional naming restrictions as a table name . A view name can be qualified (schema.viewname), or unqualified (viewname). An unqualified view name takes the system-wide default schema name . Note that you cannot use the same name for a table and a view in the same schema.
<i>column-commalist</i>	<i>Optional</i> — The column names that compose the view, one or more valid identifiers . If specified, this list is enclosed in parentheses and items in the list are separated by commas.
AS <i>select-statement</i>	A SELECT statement that defines the view.
WITH READ ONLY	<i>Optional</i> — Specifies that no insert, update, or delete operations can be performed through this view upon the table on which the view is based. The default is to permit these operations through a view, subject to the constraints described below.
WITH <i>level</i> CHECK OPTION	<i>Optional</i> — Specifies how insert, update, or delete operations are performed through this view upon the table on which the view is based. The <i>level</i> can be the keywords LOCAL or CASCADED. If no <i>level</i> is specified, the WITH CHECK OPTION default is CASCADED.

Description

The **CREATE VIEW** command defines the content of a [view](#). The **SELECT** statement that defines the view can reference more than one table and can reference other views.

Privileges

To select from the objects referenced in the **SELECT** clause of a view being created, it is necessary to have the appropriate privileges:

- When creating a view using Dynamic SQL or xDBC, you must have **SELECT** privileges on all the columns selected from the underlying tables (or views) referenced by the view. If you do not have **SELECT** privilege for a specified table (or view) the **CREATE VIEW** command will not execute.

However, when compiling a class that projects a defined view, these **SELECT** privileges are not enforced on the columns selected from the underlying tables (or views) referenced by the view. For example, if you create a view using a privileged routine (that has these **SELECT** privileges), you can later compile the view class, because you are the owner of the view, regardless of whether you have **SELECT** privileges for the tables referenced by the view.

- To receive **SELECT** privilege WITH **GRANT OPTION** for a view, you must have **WITH GRANT OPTION** for every table (or view) referenced by the view.

- To receive INSERT, UPDATE, DELETE, or REFERENCES privilege for a view, you must have the same privilege for every table (or view) referenced by the view. To receive WITH GRANT OPTION for any of these privileges, you must hold the privilege WITH GRANT OPTION on the underlying tables.
- If the view is specified WITH READ ONLY, the view is not granted INSERT, UPDATE, or DELETE privileges, regardless of the privileges you hold for the underlying tables. If the view is later redefined as read/write, these privileges are added when the class projecting the view is recompiled.

You can determine if the current user has these table-level privileges by invoking the `%CHECKPRIV` command. You can determine if a specified user has these table-level privileges by invoking the `$$SYSTEM.SQL.CheckPriv()` method. For privilege assignment, refer to the `GRANT` command.

The creator (owner) of a view is granted the `%ALTER` privilege WITH GRANT OPTION when the view is compiled.

The **CREATE VIEW** command is a privileged operation. Prior to using **CREATE VIEW** it is necessary for your process to have `%CREATE_VIEW` privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can use the **GRANT** command to assign `%CREATE_VIEW` privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the `$$SYSTEM.Security.Login()` method to log in as a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login( "_SYSTEM", "SYS" )
&sql(      )
```

You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to `%SYSTEM.Security` in the *InterSystems Class Reference*.

`%CREATE_VIEW` privileges are assigned using the `GRANT` command, which requires you to assign this privilege to a user or role. This requirement is configurable:

- The `$$SYSTEM.SQL.SetSQLSecurity()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a SQL Security ON: setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **SQL Security Enabled**.

The default is “Yes” (1). When “Yes”, a user can only perform actions on a table or view for which that user has been granted privilege. This is the recommended setting for this option.

If this option is set to “No” (0), SQL Security is disabled for any new process started after changing this setting. This means privilege-based table/view security is suppressed. You can create a table without specifying a user. In this case, Dynamic SQL assigns “_SYSTEM” as user, and embedded SQL assigns "" (the empty string) as user. Any user can perform actions on a table or view even if that user has no privileges to do so.

View Naming Conventions

A view name has the same naming conventions as a table name, and shares the same name set. Therefore, you cannot use the same name for a table and a view in the same schema. Attempting to do so results in an SQLCODE -201 error. A class that projects a table definition and a view definition with the same name also generates an SQLCODE -201 error.

View names follow `identifier` conventions, subject to the restrictions below. By default, view names are simple identifiers. A view name should not exceed 128 characters. View names are not case-sensitive. For further details see the “Identifiers” chapter of *Using Caché SQL*.

Caché uses the view name to generate a corresponding class name. A class name contains only alphanumeric characters (letters and numbers) and must be unique within the first 96 characters. To generate this class name, Caché first strips punctuation characters from the view name, and then generates a identifier that is unique within the first 96 characters, substituting an integer (beginning with 0) for the final character when needed to create a unique class name. Caché generates a unique class name from a valid view name, but this name generation imposes the following restrictions on the naming of views:

- A view name must include at least one letter. Either the first character of the view name or the first character after initial punctuation characters must be a letter.
- Caché supports 16-bit (wide) characters for view names on Unicode systems. A character is a valid letter if it passes the `$ZNAME` test.
- If the first character of the view name is a punctuation character, the second character cannot be a number. This results in an `SQLCODE -400` error, with a `%msg` value of “ERROR #5053: Class name 'schema.name' is invalid” (without the punctuation character). For example, specifying the view name `%7A` generates the `%msg` “ERROR #5053: Class name 'User.7A' is invalid”.
- Because generated class names do not include punctuation characters, it is not advisable (though possible) to create a view name that differs from an existing view or table name only in its punctuation characters. In this case, Caché substitutes an integer (beginning with 0) for the final character of the name to create a unique class name.
- A view name may be much longer than 96 characters, but view names that differ in their first 96 alphanumeric characters are much easier to work with.

A view name can be qualified or unqualified.

A qualified view name (`schema.viewname`) can specify an existing schema or a new schema. If it specifies a new schema, the system creates that schema.

An unqualified view name (`viewname`) takes the [system-wide default schema name](#).

Existing View

To determine if a specified view already exists in the current namespace, use the `$$SYSTEM.SQL.ViewExists()` method.

What happens when you try to create a view that has the same name as an existing view depends on the optional `OR REPLACE` keyword and on the configuration setting.

With OR REPLACE

If you specify **CREATE OR REPLACE VIEW**, the existing view is replaced by the view definition specified in the **SELECT** clause and any specified `WITH READ ONLY` or `WITH CHECK OPTION`. This is the same as performing the corresponding [ALTER VIEW](#) statement. Any privileges that had been granted to the original view remain.

This keyword phrase provides no functionality not available through **ALTER VIEW**. It is provided for compatibility with Oracle SQL code.

Without OR REPLACE

If you specify **CREATE VIEW**, Caché, by default, rejects an attempt to create a view with the name of an existing view and issues an `SQLCODE -201` error. This behavior is configurable. Note that changing this behavior affects both **CREATE VIEW** and **CREATE TABLE**. You can configure this behavior as follows:

- The `$$SYSTEM.SQL.SetDDLNo201()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a `Suppress SQLCODE=-201 Errors` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow DDL CREATE TABLE or CREATE VIEW for Existing Table**.

The default is “No” (0). This is the recommended setting for this option. If this option is set to “Yes” (1), Caché deletes the class definition associated with the view and then recreates it. This is much the same as performing a **DROP VIEW** and then performing a **CREATE VIEW**.

Column Names

A view can optionally include a *column-commalist* list of column names, enclosed in parentheses. These column names, if specified, are the names used to access and display the data for the columns when using that view. If the list of column

names is omitted, the column names of the **SELECT** source table are used. If you omit the list of column names, you must also omit the parentheses.

If you specify the *column-commalist*, the following apply:

- A column name list must specify the enclosing parentheses, even when specifying a single field. You must separate multiple column names with commas. Whitespace and [comments](#) are permitted within a *column-commalist*.
- The number of column names must correspond to the number of fields specified in the **SELECT** statement. Mismatch between the number of view columns and query columns results in an **SQLCODE -142** error at compile time.
- The names of column names must be valid [identifiers](#). They may be different names than the **SELECT** field names, the same names as the **SELECT** field names, or a combination of both. The specified order of the view column names corresponds to the order of the **SELECT** field names. Because it is possible to assign a view column the name of an unrelated **SELECT** field, you must exercise caution when assigning view column names.

The following example shows a **CREATE VIEW** with matching lists of view columns and query columns:

```
CREATE VIEW MyView (ViewCol1, ViewCol2, ViewCol3) AS
  SELECT TableCol1, TableCol2, TableCol3
  FROM MyTable
```

Alternatively, you can use the **AS** keyword in the query to specify the view columns as query column / view column pairs, as shown in the following example:

```
CREATE VIEW MyView AS
  SELECT TableCol1 AS ViewCol1,
         TableCol2 AS ViewCol2,
         TableCol3 AS ViewCol3
  FROM MyTable
```

SELECT Clause Considerations

A view does not have to be a simple subset of the rows and columns of one particular table. A view can be created using more than one table or other views with a **SELECT** clause of any complexity. There are, however, a few restrictions on the **SELECT** clauses in a view definition. A **CREATE VIEW** statement:

- Can only include an **ORDER BY** clause if this clause is paired with a **TOP** clause. If you wish to include all of the rows in the view, you can use a **TOP ALL** clause. You can include a **TOP** clause without an **ORDER BY** clause. However, if you include an **ORDER BY** clause without a **TOP** clause, an **SQLCODE -143** error is generated. If you project an SQL view from a view class, the query of which contains an **ORDER BY** clause, the **ORDER BY** clause is ignored in the view projection.
- Cannot contain [host variables](#) or include the **INTO** keyword. If you attempt to reference a host variable in the **SELECT** clause, the system generates an **SQLCODE -148** error.
- Cannot reference a temporary table.
- May have a **GROUP BY** clause.
- May use functions.

CREATE VIEW can contain a **UNION** statement to select columns from the union of two tables. You can specify a **UNION** as shown in the following embedded SQL example:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(CREATE VIEW MyView (vname,vstate) AS
SELECT t1.name,t1.home_state
  FROM Sample.Person AS t1
UNION
SELECT t2.name,t2.office_state
  FROM Sample.Employee AS t2)
IF SQLCODE=0 { WRITE !,"Created view" }
ELSE { WRITE "CREATE VIEW error SQLCODE=",SQLCODE }
```

Note that an unqualified view name, such as in the above example, defaults to the [system-wide default schema name](#) (for example, the initial schema default `SQLUser.MyView`), even though the tables referenced by the view are in the Sample schema. Thus it is usually a good practice to always qualify a view name to ensure that it is stored with its associated table(s).

View ID: %vid

When data is accessed through a view, Caché assigns a sequential integer view ID (%vid) to each row returned by that view. Like table row ID numbers, these view row ID numbers are system-assigned, unique, non-zero, non-null, and non-modifiable. This %vid is usually invisible. Unlike a table row ID, it is not displayed when using asterisk syntax; it is only displayed when explicitly specified in the **SELECT**. The %vid can be used to further restrict the number of rows returned by a **SELECT** accessing a view. For further details on using %vid, refer to the [Defining and Using Views](#) chapter of *Using Caché SQL*.

Updating Through Views

A view can be used to update the tables on which the view is based. You can **INSERT** new rows through the view, **UPDATE** data in rows seen through the view, and **DELETE** rows seen through the view. **INSERT**, **UPDATE**, and **DELETE** statements can be issued for a view, if the **CREATE VIEW** statement specified this ability. To allow updating through a view, specify **WITH CHECK OPTION** (the default) when defining the view.

To prevent updating through a view, specify **WITH READ ONLY**. Attempting an **INSERT**, **UPDATE**, or **DELETE** through a view created **WITH READ ONLY** generates an `SQLCODE -35` error.

In order to update through a view, you must have the appropriate privileges for the table or view to be updated, as specified by the [GRANT](#) command.

Updating through views is subject to the following restrictions:

- The view cannot be a class query projected as a view.
- The view's class cannot contain the class parameter `READONLY=1`.
- The view's **SELECT** statement cannot contain a **DISTINCT**, **TOP**, **GROUP BY**, or **HAVING** clause, or be part of a **UNION**.
- The view's **SELECT** statement cannot contain a subquery.
- The view's **SELECT** statement can only list value expressions that are column references.
- The view's **SELECT** statement can have only one table reference; it cannot contain **FROM** clause **JOIN** syntax or [arrow syntax](#) in the *select-list* or **WHERE** clause. The table reference must specify either an updateable table or an updateable view.

The **WITH CHECK OPTION** clause causes an insert or update operation to validate the resulting row against the **WHERE** clause of the view definition. This ensures that the inserted or modified row is part of the derived view table. There are two available check options:

- **WITH LOCAL CHECK OPTION** — only the **WHERE** clause of the view specified in the **INSERT** or **UPDATE** statement is checked.
- **WITH CASCADED CHECK OPTION** — the **WHERE** clause of the view specified in the **INSERT** or **UPDATE** statement and all underlying views are checked. This overrides any **WITH LOCAL CHECK OPTION** clauses in these underlying views. **WITH CASCADED CHECK OPTION** is recommended for all updateable views.

If you specify **WITH CHECK OPTION**, the check option defaults to **CASCADED**. The keyword **CASCADE** is a synonym for **CASCADED**.

If an **INSERT** operation fails **WITH CHECK OPTION** validation (as defined above), Caché issues an `SQLCODE -136` error.

If an **UPDATE** operation fails **WITH CHECK OPTION** validation (as defined above), Caché issues an SQLCODE -137 error.

Examples

The following example creates a view named "CityPhoneBook" from the PhoneBook table:

```
CREATE VIEW CityPhoneBook AS
  SELECT Name FROM PhoneBook WHERE City='Boston'
```

The following example creates a view named "GuideHistory" from the Guides table. It lists all titles (from the Title column) and whether or not the person is retired:

```
CREATE VIEW GuideHistory AS
  SELECT Guides, Title, Retired, Date_Retired
  FROM Guides
```

The following Embedded SQL example creates the table MyTest, and then creates a view for this table, MyTestView, which selects one field from MyTest:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(DROP TABLE Sample.MyTest)
&sql(DROP VIEW Sample.MyTestView)
CreateTable
  &sql(CREATE TABLE Sample.MyTest (
    TestNum      INT NOT NULL,
    FirstWord    CHAR (30) NOT NULL,
    LastWord     CHAR (30) NOT NULL,
    CONSTRAINT MyTestPK PRIMARY KEY (TestNum))
  )
  IF SQLCODE=0 { WRITE !,"Created table" }
  ELSE { WRITE "CREATE TABLE error SQLCODE=",SQLCODE }
CreateView
  &sql(CREATE VIEW Sample.MyTestView AS
    SELECT FirstWord FROM Sample.MyTest
    WITH CASCADED CHECK OPTION)
  IF SQLCODE=0 { WRITE !,"Created view" }
  ELSE { WRITE "CREATE VIEW error SQLCODE=",SQLCODE }
```

The following Embedded SQL example creates a view MyTestView, which selects two fields from MyTest. The SELECT query for this view contains a TOP clause and an ORDER BY clause:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(DROP TABLE Sample.MyTest)
&sql(DROP VIEW Sample.MyTestView)
CreateTable
  &sql(CREATE TABLE Sample.MyTest (
    TestNum      INT NOT NULL,
    FirstWord    CHAR (30) NOT NULL,
    LastWord     CHAR (30) NOT NULL,
    CONSTRAINT MyTestPK PRIMARY KEY (TestNum))
  )
  IF SQLCODE=0 { WRITE !,"Created table" }
  ELSE { WRITE "CREATE TABLE error SQLCODE=",SQLCODE }
CreateView
  &sql(CREATE VIEW Sample.MyTestView AS
    SELECT TOP ALL FirstWord,LastWord FROM Sample.MyTest
    ORDER BY LastWord)
  IF SQLCODE=0 { WRITE !,"Created view" }
  ELSE { WRITE "CREATE VIEW error SQLCODE=",SQLCODE }
```

The following example creates a view named "StaffWorksDesign" from three tables (Proj, Staff, and Works). The columns Name, Cost, and Project provide the data.

```
CREATE VIEW StaffWorksDesign (Name,Cost,Project)
  AS SELECT EmpName,Hours*2*Grade,PName
  FROM Proj,Staff,Works
  WHERE Staff.EmpNum=Works.EmpNum
  AND Works.PNum=Proj.PNum AND PType='Design'
```

The following example creates a view named "v_3" by selecting from b.table2 and a.table1 using a **UNION**:

```
CREATE VIEW v_3(fvarchar)
  AS SELECT DISTINCT *
  FROM
    (SELECT fVARIABLE2 FROM b.table2
     UNION ALL
     SELECT fVARIABLE1 FROM a.table1)
```

See Also

- [ALTER VIEW](#)
- [DROP VIEW](#)
- [CREATE TABLE](#)
- [GRANT](#)
- [SELECT](#)
- “[Defining and Using Views](#)” chapter in *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DECLARE

Declares a cursor.

```
DECLARE cursor-name CURSOR FOR query
```

Arguments

<i>cursor-name</i>	The name of the cursor, which must begin with a letter and contain only letters and numbers. (Cursor names do not follow SQL identifier conventions). Cursor names are case-sensitive. They are subject to additional naming restrictions, as described below.
<i>query</i>	A standard SELECT statement that defines the result set of the cursor. This SELECT can include the %NOFPLAN keyword to specify that Caché should ignore the frozen plan (if any) for this query. This SELECT can include an ORDER BY clause, with or without a TOP clause. This SELECT can specify a table-valued function in the FROM clause.

Description

A **DECLARE** statement declares a [cursor](#) used in [cursor-based Embedded SQL](#). After declaring a cursor, you issue an **OPEN** statement to open the cursor and then a series of **FETCH** statements to retrieve individual records. The cursor defines the **SELECT** query that is used to select records for retrieval by these **FETCH** statements. You issue a **CLOSE** statement to close (but not delete) the cursor.

As an SQL statement, **DECLARE** is only supported from Embedded SQL. For Dynamic SQL, use instead either a simple **SELECT** statement (with no **INTO** clause), or a combination of Dynamic SQL and Embedded SQL. Equivalent operations are supported through ODBC using the ODBC API.

DECLARE declares a forward-only (non-scrollable) cursor. Fetch operations begin with the first record in the query result set and proceed sequentially through the result set records. A **FETCH** can only fetch a record once. The next **FETCH** fetches the next sequential record in the result set.

Because **DECLARE** is a declaration, not an executed statement, it does not set or kill the SQLCODE variable.

DECLARE does not support the [#SQLCompile Mode=Deferred](#) preprocessor directive. Attempting to use Deferred mode with a **DECLARE**, **OPEN**, **FETCH**, or **CLOSE** cursor statement generates a #5663 compilation error.

Cursor Names

A cursor name must be unique within the routine and the corresponding class. A cursor name may be of any length, but must be unique within the first 29 characters. Cursor names are case-sensitive. Attempting to declare two cursors with the same name results in an error code -52 during compilation.

Cursor names are not namespace-specific. You can **DECLARE** a cursor in one namespace, and **OPEN**, **FETCH**, or **CLOSE** this cursor when in another namespace. Note that SQL tables are namespace-specific, so the **FETCH** operation must be invoked in the same namespace as the table from which records are being fetched.

The first character of a cursor name must be a letter. The second and subsequent characters of a cursor name must be either a letter or a number. Unlike SQL [identifiers](#), punctuation characters are not permitted in cursor names.

You can use a delimiter characters (double quotes) to specify an SQL reserved word as a cursor name. A delimited cursor name is *not* an SQL delimited identifier; delimited cursor names are still case-sensitive and cannot contain punctuation characters. In most cases, an SQL reserved word should not be used as a cursor name.

Updating through a Cursor

You can perform record updates and deletes through a declared cursor using an **UPDATE** or **DELETE** statement with the **WHERE CURRENT OF** clause. In Caché SQL a cursor can always be used for **UPDATE** or **DELETE** operations if you have the appropriate privileges on the affected tables and columns; refer to the **GRANT** statement for assigning object privileges.

A **DECLARE** statement can specify a **FOR UPDATE** or **FOR READ ONLY** keyword clause following the query. These clauses are optional and perform no operation. They are provided as a way to document in the code that the process issuing the query has or does not have the needed update and delete object privileges.

Examples

The following Embedded SQL example uses **DECLARE** to define a cursor for a query that specifies two output host variables. The cursor is then opened, fetched repeatedly, and closed:

```
SET name="John Doe",state="##"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name, Home_State
    INTO :name,:state FROM Sample.Person
    WHERE Home_State %STARTSWITH 'A'
    FOR READ ONLY)
WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
QUIT:(SQLCODE'=0)
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
WRITE !,"AFTER: Name=",name," State=",state
```

The following Embedded SQL example uses **DECLARE** to define a cursor for a query that specifies both output host variables in the **INTO** clause and input host variables in the **WHERE** clause. The cursor is then opened, fetched repeatedly, and closed:

```
NEW SQLCODE,%ROWCOUNT,%ROWID
SET EmpZipLow="10000"
SET EmpZipHigh="19999"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name,Home_Zip
    INTO :name,:zip
    FROM Sample.Employee WHERE Home_Zip BETWEEN :EmpZipLow AND :EmpZipHigh)
&sql(OPEN EmpCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,name," ",zip }
&sql(CLOSE EmpCursor)
QUIT
```

The following Embedded SQL example uses a **table-valued function** as the **FROM** clause of the *query*:

```
ZNSPACE "Samples"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name INTO :name FROM Sample.SP_Sample_By_Name('A')
    FOR READ ONLY)
&sql(OPEN EmpCursor)
QUIT:(SQLCODE'=0)
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE "Name=",name,! }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
```

See Also

- [CLOSE, FETCH, OPEN, WHERE CURRENT OF](#)

- [SQL Cursors](#) in the “Using Embedded SQL” chapter of *Using Caché SQL*

DELETE

Removes rows from a table.

```
DELETE [%NOFPLAN] [restriction] [FROM] table-ref [[AS] t-alias]
      [FROM select-table1 [[AS] t-alias]
        {,select-table2 [[AS] t-alias}} ]
      [WHERE condition-expression]
```

```
DELETE [restriction] [FROM] table-ref [[AS] t-alias]
      [WHERE CURRENT OF cursor]
```

Arguments

%NOFPLAN	<i>Optional</i> — The %NOFPLAN keyword specifies that Caché will ignore the frozen plan (if any) for this operation and generate a new query plan. The frozen plan is retained, but not used. For further details, refer to Frozen Plans in <i>Caché SQL Optimization Guide</i> .
<i>restriction</i>	<i>Optional</i> — One or more of the following keywords, separated by spaces: %NOLOCK, %NOCHECK, %NOINDEX, %NOTRIGGER.
FROM <i>table-ref</i>	<p>The table from which you are deleting rows. This is <i>not</i> a FROM clause; it is a FROM keyword followed by a single table reference. (The FROM keyword is optional; the <i>table-ref</i> is mandatory.)</p> <p>A table name (or view name) can be qualified (schema.table), or unqualified (table). An unqualified name is matched to its schema using either a schema search path (if provided) or the system-wide default schema name.</p> <p>Rather than a table reference, you can specify a view through which table rows can be deleted, or specify a subquery enclosed in parentheses. Unlike the SELECT statement FROM clause, you cannot specify <i>optimize-option</i> keywords here. You cannot specify a table-valued function or JOIN syntax in this argument.</p>
FROM clause	<p><i>Optional</i>— A FROM clause, specified <i>after</i> the <i>table-ref</i>. This FROM can be used to specify a <i>select-table</i> table or tables used to select which rows are to be deleted.</p> <p>Multiple tables can be specified as a comma-separated list or associated with ANSI join keywords. Any combination of tables or views can be specified. If you specify a comma between two <i>select-tables</i> here, Caché performs a CROSS JOIN on the tables and retrieves data from the results table of the JOIN operation. If you specify ANSI join keywords between two <i>select-tables</i> here, Caché performs the specified join operation. For further details, refer to the JOIN page of this manual.</p> <p>You can optionally specify one or more <i>optimize-option</i> keywords to optimize query execution. The available options are: %ALLINDEX, %FIRSTTABLE <i>tablename</i>, %FULL, %INORDER, %IGNOR-EINDICES, %NOFLATTEN, %NOMERGE, %NOSVSO, %NOTOPOPT, %NOUNIONOROPT, and %STARTTABLE. See FROM clause for more details.</p>

AS <i>t-alias</i>	<i>Optional</i> — An alias for a table or view name . An alias must be a valid identifier . The AS keyword is optional.
WHERE <i>condition-expression</i>	<i>Optional</i> — Specifies one or more boolean predicates used to limit which rows are to be deleted. You can specify a WHERE clause or a WHERE CURRENT OF clause, but not both. If a WHERE clause (or a WHERE CURRENT OF clause) is not supplied, DELETE removes all the rows from the table. For further details, see WHERE .
WHERE CURRENT OF <i>cursor</i>	<i>Optional: Embedded SQL only</i> — Specifies that the DELETE operation deletes the record at the current position of cursor . You can specify a WHERE CURRENT OF clause or a WHERE clause, but not both. If a WHERE CURRENT OF clause (or a WHERE clause) is not supplied, DELETE removes all the rows from the table. For further details, see WHERE CURRENT OF .

Description

The **DELETE** command removes rows from a table that meet the specified conditions. You can delete rows from a table directly, delete through a view, or delete rows selected using a subquery. Deleting through a view is subject to requirements and restrictions, as described in [CREATE VIEW](#).

The **DELETE** operation sets the [%ROWCOUNT](#) local variable to the number of deleted rows, and the [%ROWID](#) local variable to the RowID value of the last row deleted. If no rows are deleted, [%ROWCOUNT=0](#) and [%ROWID](#) is undefined or remains set to its previous value.

You must specify a *table-ref*; the FROM keyword before the *table-ref* is optional. To delete *all* rows from a table, you can simply specify:

```
DELETE FROM tablename
```

or

```
DELETE tablename
```

This deletes all row data from the table, but does not reset the [RowID](#), [IDENTITY](#), and [SERIAL \(%Library.Counter\)](#) field counters. The [TRUNCATE TABLE](#) command both deletes all row data from a table and resets these counters. By default, `DELETE FROM tablename` pulls delete triggers; you can specify `DELETE %NOTRIGGER FROM tablename` to not pull delete triggers. **TRUNCATE TABLE** does not pull delete triggers.

More commonly, a **DELETE** specifies the deletion of a specific row (or rows) based on a *condition-expression*. By default, a **DELETE** operation goes through all of the rows of a table and deletes all rows that satisfy the *condition-expression*. If no rows satisfy the *condition-expression*, **DELETE** completes successfully and sets [SQLCODE=100](#) (No more data).

You can specify a **WHERE** clause or a **WHERE CURRENT OF** clause (but not both). If the **WHERE CURRENT OF** clause is used, the **DELETE** operation deletes the record at the current position of the cursor. For an example of **DELETE** using **WHERE CURRENT OF**, see “[Embedded SQL and Dynamic SQL Examples](#)” below. For details on positioned operations, see [WHERE CURRENT OF](#).

By default, **DELETE** is an all-or-nothing event: either all specified rows are deleted completely, or no deletion is performed. Caché sets the status variable [SQLCODE](#), indicating the success or failure of the **DELETE**.

To delete a row from a table:

- The table must exist in the current (or specified) namespace. If the specified table cannot be located, Caché issues an [SQLCODE -30](#) error.
- You must have [DELETE](#) privilege for the table. Failing to have this privilege results in an [SQLCODE -99](#) (Privilege Violation) error. You can determine if the current user has [DELETE](#) privilege by invoking the [%CHECKPRIV](#) command.

You can determine if a specified user has **DELETE** privilege by invoking the `$$SYSTEM.SQL.CheckPriv()` method. For privilege assignment, refer to the [GRANT](#) command.

- The table cannot be locked **IN EXCLUSIVE MODE** by another process. Attempting to delete a row from a locked table results in an **SQLCODE -110** error, with a %msg such as the following: Unable to acquire lock for **DELETE** of table 'Sample.Person' on row with RowID = '10'. Note that an **SQLCODE -110** error occurs only when the **DELETE** statement locates the first record to be deleted, then cannot lock it within the timeout period.
- If the **DELETE** command's **WHERE** clause specifies a non-existent field, an **SQLCODE -29** is issued. To list all of the field names defined for a specified table, refer to [Column Names and Numbers](#) in the “Defining Tables” chapter of *Using Caché SQL*. If the field exists but none of the field values fulfill the **DELETE** command's **WHERE** clause, no rows are affected and **SQLCODE 100** (end of data) is issued.
- The table cannot be defined as **READONLY**. Attempting to compile an **DELETE** that references a read-only table results in an **SQLCODE -115** error. Note that this error is now issued at compile time, rather than only occurring at execution time. See the description of **READONLY** objects in the [Other Options for Persistent Classes](#) chapter of *Using Caché Objects*.
- If deleting through a view, the view cannot be defined as **WITH READ ONLY**. Attempting to do so results in an **SQLCODE -35** error. See the [CREATE VIEW](#) command for further details. Similarly, if you are attempting to delete through a subquery, the subquery must be updateable; for example, the following subquery results in an **SQLCODE -35** error: `DELETE FROM (SELECT COUNT(*) FROM Sample.Person) AS x.`
- The row to delete must exist. Usually, attempting to delete a nonexistent row results in an **SQLCODE 100** (No more data) because the specified row could not be located. However, in rare cases, a row to be deleted may be located, but then is immediately deleted by another process; this situation results in an **SQLCODE -106** error.
- All of the rows specified for deletion must be available for deletion. By default, if one or more rows cannot be deleted the **DELETE** operation fails and no rows are deleted. If a row to be deleted has been locked by another concurrent process, **DELETE** issues an **SQLCODE -110** error. If deleting one of the specified rows would violate foreign key referential integrity (and **%NOCHECK** is not specified), the **DELETE** issues an **SQLCODE -124** error. This default behavior is modifiable, as described below.
- Certain **%SYS** namespace system-supplied facilities are protected against deletion. For example, `DELETE FROM Security.Users` cannot be used to delete **_SYSTEM**, **_PUBLIC** or **UnknownUser**. Attempting to do so results in an **SQLCODE -134** error.

Atomicity

By default, **DELETE**, **UPDATE**, **INSERT**, and **TRUNCATE TABLE** are atomic operations. A **DELETE** either completes successfully or the whole operation is rolled back. If any of the specified rows cannot be deleted, none of the specified rows are deleted and the database reverts to its state before issuing the **DELETE**.

You can modify this default for the current process within SQL by invoking [SET TRANSACTION %COMMITMODE](#). You can modify this default for the current process in ObjectScript by invoking the `SetAutoCommit()` method. The following options are available:

- **IMPLICIT** or 1 (autocommit on) — The default behavior, as described above. Each **DELETE** constitutes a separate transaction.
- **EXPLICIT** or 2 (autocommit off) — If no transaction is in progress, a **DELETE** automatically initiates a transaction, but you must explicitly **COMMIT** or **ROLLBACK** to end the transaction. In **EXPLICIT** mode the number of database operations per transaction is user-defined.
- **NONE** or 0 (no auto transaction) — No transaction is initiated when you invoke **DELETE**. A failed **DELETE** operation can leave the database in an inconsistent state, with some of the specified rows deleted and some not deleted. To provide

transaction support in this mode you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

You can determine the atomicity setting for the current process using the `GetAutoCommit()` method, as shown in the following ObjectScript example:

```
DO $SYSTEM.SQL.SetAutoCommit($RANDOM(3))
SET x=$SYSTEM.SQL.GetAutoCommit()
IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
ELSE { WRITE "Explicit commit or rollback required" }
```

FROM Syntax

A **DELETE** command can contain two **FROM** keywords that specify tables. These two uses of **FROM** are fundamentally different:

- **FROM** before *table-ref* specifies the table (or view) from which rows are to be deleted. It is a **FROM** keyword, not a **FROM** clause. Only one table may be specified. No join syntax or *optimize-option* keywords may be specified. The **FROM** keyword itself is optional; the *table-ref* is required.
- **FROM** after *table-ref* is an optional **FROM** clause that can be used to determine which rows should be deleted. It may specify one or more than one tables. It supports all of the **FROM** clause syntax available to a **SELECT** statement, including join syntax and *optimize-option* keywords. This **FROM** clause is commonly (but not always) used with a **WHERE** clause.

Thus any of the following are valid syntactical forms:

```
DELETE FROM table WHERE ...
DELETE table WHERE ...
DELETE FROM table FROM table2 WHERE ...
DELETE table FROM table2 WHERE ...
```

This syntax supports complex selection criteria in a manner compatible with Transact-SQL.

The following example shows how the two **FROM** keywords might be used. It deletes those records from the **Employees** table where the same **EmpId** is also found in the **Retirees** table:

```
DELETE FROM Employees AS Emp
        FROM Retirees AS Rt
        WHERE Emp.EmpId = Rt.EmpId
```

If the two **FROM** keywords make reference to the same table, these references may either be to the same table, or to a join of two instances of the table. This depends on how table aliases are used:

- If neither table reference has an alias, both reference the same table:

```
DELETE FROM table1 FROM table1,table2 /* join of 2 tables */
```

- If both table references have the same alias, both reference the same table:

```
DELETE FROM table1 AS x FROM table1 AS x,table2 /* join of 2 tables */
```

- If both table references have aliases, and the aliases are different, Caché performs a join of two instances of the table:

```
DELETE FROM table1 AS x FROM table1 AS y,table2 /* join of 3 tables */
```

- If the first table reference has an alias, and the second does not, Caché performs a join of two instances of the table:

```
DELETE FROM table1 AS x FROM table1,table2 /* join of 3 tables */
```

- If the first table reference does not have an alias, and the second has a single reference to the table with an alias, both reference the same table, and this table has the specified alias:

```
DELETE FROM table1 FROM table1 AS x,table2 /* join of 2 tables */
```

- If the first table reference does not have an alias, and the second has more than one reference to the table, Caché considers each aliased instance a separate table and performs a join on these tables:

```
DELETE FROM table1 FROM table1,table1 AS x,table2 /* join of 3 tables */
DELETE FROM table1 FROM table1 AS x,table1 AS y,table2 /* join of 4 tables */
```

Restriction Arguments

To use a *restriction* argument, you must have the corresponding *admin-privilege* for the current namespace. Refer to [GRANT](#) for further details.

Specifying *restriction* argument(s) restricts processing as follows:

- `%NOCHECK` — suppress referential integrity checking for foreign keys that reference the rows being deleted.
- `%NOLOCK` — suppress row locking of the row being deleted. This should only be used when a single user/process is updating the database.
- `%NOINDEX` — suppresses deleting index entries in all indices for the rows being deleted. This should be used with extreme caution, because it leaves orphaned values in the table indices.
- `%NOTRIGGER` — suppress the pulling of base table triggers that are otherwise pulled during **DELETE** processing.

You can specify multiple *restriction* arguments in any order. Multiple arguments are separated by spaces.

If you specify a *restriction* argument when deleting a parent record, the same *restriction* argument will be applied when deleting the corresponding child records.

Referential Integrity

If you do not specify `%NOCHECK`, Caché uses the system configuration setting to determine whether to perform foreign key referential integrity checking. You can set the system default as follows:

- The `$$SYSTEM.SQL.SetFileRefIntegrity()` method call.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Perform Referential Integrity Checks on Foreign Keys for INSERT, UPDATE, and DELETE**. The default is “Yes”. If you change this setting, any new process started after changing it will have the new setting.

During a **DELETE** operation, for every foreign key reference a shared lock is acquired on the corresponding row in the referenced table. This row is locked until the end of the transaction. This ensures that the referenced row is not changed before a potential rollback of the **DELETE**.

If a series of foreign key references are defined as **CASCADE**, a **DELETE** operation could potentially result in a circular reference. Caché prevents **DELETE** with **CASCADE** referential action from performing a circular reference loop recursion. Caché ends the cascade sequence when it returns to the original table.

If a **DELETE** operation with `%NOLOCK` is performed on a [foreign key field defined with CASCADE, SET NULL, or SET DEFAULT](#), the corresponding referential action changing the foreign key table is also performed with `%NOLOCK`.

Transaction Locking

If you do not specify `%NOLOCK`, the system automatically performs standard record locking on **INSERT**, **UPDATE**, and **DELETE** operations. Each affected record (row) is locked for the duration of the current transaction.

The default lock threshold is 1000 locks per table. This means that if you delete more than 1000 records from a table during a transaction, the lock threshold is reached and Caché automatically escalates the locking level from record locks to a table lock. This permits large-scale deletes during a transaction without overflowing the lock table.

Caché applies one of the two following lock escalation strategies:

- “E”-type lock escalation: Caché uses this type of lock escalation if the following are true: (1) the class uses `%CacheStorage` (you can determine this from the [Catalog Details](#) in the Management Portal SQL schema display). (2) the class either does not specify an IDKey index, or specifies a single-property IDKey index. “E”-type lock escalation is described in the [LOCK](#) command in the *Caché ObjectScript Reference*.
- Traditional SQL lock escalation: The most likely reason why a class would not use “E”-type lock escalation is the presence of a multi-property IDKey index. In this case, each `%Save` increments the lock counter. This means if you do 1001 saves of a single object within a transaction, Caché will attempt to escalate the lock.

For both lock escalation strategies, you can determine the current system-wide lock threshold value using the `$$SYSTEM.SQL.GetLockThreshold()` method. The default is 1000. This system-wide lock threshold value is configurable:

- Using the `$$SYSTEM.SQL.SetLockThreshold()` method.
- Using the Management Portal. Go to **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Lock Threshold**. The default is 1000 locks. If you change this setting, any new process started after changing it will have the new setting.

You must have USE permission on the `%Admin Manage Resource` to change the lock threshold. Caché immediately applies any change made to the lock threshold value to all current processes.

One potential consequence of automatic lock escalation is a deadlock situation that might occur when an attempt to escalate to a table lock conflicts with another process holding a record lock in that table. There are several possible strategies to avoid this: (1) increase the lock escalation threshold so that lock escalation is unlikely to occur within a transaction. (2) substantially lower the lock escalation threshold so that lock escalation occurs almost immediately, thus decreasing the opportunity for other processes to lock a record in the same table. (3) apply a table lock for the duration of the transaction and do not perform record locks. This can be done at the start of the transaction by specifying `LOCK TABLE`, then `UNLOCK TABLE` (without the `IMMEDIATE` keyword, so that the table lock persists until the end of the transaction), then perform deletes with the `%NOLOCK` option.

Automatic lock escalation is intended to prevent overflow of the lock table. However, if you perform such a large number of deletes that a `<LOCKTABLEFULL>` error occurs, **DELETE** issues an `SQLCODE -110` error.

For further details on transaction locking refer to [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

Examples

The following examples both delete all rows from the `TempEmployees` table. Note that the `FROM` keyword is optional:

```
DELETE FROM TempEmployees
```

```
DELETE TempEmployees
```

The following example deletes employee number 234 from the `Employees` table:

```
DELETE
  FROM Employees
  WHERE EmpId = 234
```

The following example deletes all rows from the `ActiveEmployees` table in which the `CurStatus` column is set to "Retired":

```
DELETE FROM ActiveEmployees
  WHERE CurStatus = 'Retired'
```

The following example deletes rows using a subquery:

```
DELETE FROM (SELECT Name, Age FROM Sample.Person WHERE Age > 65)
```

Embedded SQL and Dynamic SQL Examples

In the following set of program examples, the first program creates a table named `SQLUser.WordPairs` with three columns. The next program inserts six records. Subsequent programs delete all English records using [cursor-based Embedded SQL](#), and delete all French records using [Dynamic SQL](#). The final program displays the remaining records, then deletes the table.

```
CreateTable
  ZNSPACE "Samples"
  &sql(CREATE TABLE SQLUser.WordPairs (
    Lang      CHAR(2) NOT NULL,
    Firstword CHAR(30),
    Lastword  CHAR(30) )
  )
  IF SQLCODE=0 {
    WRITE !,"Table created" }
  ELSEIF SQLCODE=-201 {WRITE !,"Table already exists"  QUIT}
  ELSE {
    WRITE !,"CREATE TABLE failed. SQLCODE=",SQLCODE }

InsertSixRecords
#SQLCompile Path=Cinema,Sample
ZNSPACE "Samples"
&sql(INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
  ('En','hello','goodbye'))
IF SQLCODE = 0 { WRITE !,"1st record inserted" }
ELSE { WRITE !,"Insert failed, SQLCODE=",SQLCODE
  QUIT}
&sql(INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
  ('Fr','bonjour','au revoir'))
IF SQLCODE = 0 { WRITE !,"2nd record inserted" }
ELSE { WRITE !,"Insert failed, SQLCODE=",SQLCODE  QUIT}
&sql(INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
  ('It','pronto','ciao'))
IF SQLCODE = 0 { WRITE !,"3rd record inserted" }
ELSE { WRITE !,"Insert failed, SQLCODE=",SQLCODE  QUIT}
&sql(INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
  ('Fr','oui','non'))
IF SQLCODE = 0 { WRITE !,"4th record inserted" }
ELSE { WRITE !,"Insert failed, SQLCODE=",SQLCODE  QUIT}
&sql(INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
  ('En','howdy','see ya'))
IF SQLCODE = 0 { WRITE !,"5th record inserted" }
ELSE { WRITE !,"Insert failed, SQLCODE=",SQLCODE  QUIT}
&sql(INSERT INTO WordPairs (Lang,Firstword,Lastword) VALUES
  ('Es','hola','adios'))
IF SQLCODE = 0 { WRITE !,"6th record inserted",!!
  SET myquery = "SELECT %ID,* FROM SQLUser.WordPairs"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  DO rset.%Display()
  WRITE !,"End of data" }
ELSE { WRITE !,"Insert failed, SQLCODE=",SQLCODE }

EmbeddedSQLDeleteEnglish
#SQLCompile Path=Sample
NEW %ROWCOUNT,%ROWID
&sql(DECLARE WPCursor CURSOR FOR
  SELECT Lang FROM WordPairs
  WHERE Lang='En')
&sql(OPEN WPCursor)
  QUIT:(SQLCODE'=0)
FOR { &sql(FETCH WPCursor)
  QUIT:SQLCODE
  &sql(DELETE FROM WordPairs
  WHERE CURRENT OF WPCursor)
  IF SQLCODE=0 {
    WRITE !,"Delete succeeded"
    WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
  ELSE {
    WRITE !,"Delete failed, SQLCODE=",SQLCODE }
  }
&sql(CLOSE WPCursor)
```

```

DynamicSQLDeleteFrench
SET sqltext = "DELETE FROM WordPairs WHERE Lang=?"
SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
SET qStatus = tStatement.%Prepare(sqltext)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rtn = tStatement.%Execute("Fr")
IF rtn.%SQLCODE=0 {
WRITE !,"Delete succeeded"
WRITE !,"Row count=",rtn.%ROWCOUNT," RowID of last record=",rtn.%ROWID }
ELSE {
WRITE !,"Delete failed, SQLCODE=",rtn.%SQLCODE }

DisplayAndDeleteTable
ZNSPACE "Samples"
SET myquery = "SELECT %ID,* FROM SQLUser.WordPairs"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
&sql(DROP TABLE SQLUser.WordPairs)
IF SQLCODE=0 {
WRITE !!,"Table deleted"
QUIT }
ELSE {
WRITE !,"Table delete failed, SQLCODE=",SQLCODE }

```

See Also

- [FROM](#)
- [TRUNCATE TABLE](#)
- [INSERT UPDATE](#)
- [CREATE VIEW](#)
- [WHERE](#)
- [WHERE CURRENT OF](#)
- [“Modifying the Database” chapter in *Using Caché SQL*](#)
- [“Defining Tables” chapter in *Using Caché SQL*](#)
- [“Defining Views” chapter of *Using Caché SQL*](#)
- [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DISTINCT

A **SELECT** clause that specifies to return only distinct values.

```
SELECT [DISTINCT [BY (item {,item})] ] | [ALL]
      select-item {,select-item2}
```

Arguments

DISTINCT	<i>Optional</i> — Returns rows for which the combined <i>select-item</i> value(s) are unique.
DISTINCT BY (<i>item1</i> {, <i>item2</i> })	<i>Optional</i> — Returns <i>select-item</i> values for rows for which the BY (<i>item</i>) value(s) are unique.
ALL	<i>Optional</i> — Returns all rows in the result set. The default.

Description

The optional **DISTINCT** clause appears after the **SELECT** keyword and before the optional **TOP clause** and the first *select-item*.

The **DISTINCT** clause is applied to the result set of the **SELECT** statement. It limits the rows returned to those that contain a distinct (non-duplicate) value. If no **DISTINCT** clause is specified, the default is to display all the rows that fulfill the **SELECT** criteria. The **ALL** clause is the same as specifying no **DEFAULT** clause; if you specify **ALL**, **SELECT** returns all the rows in the table that fulfill the **SELECT** criteria.

The **DISTINCT** clause has two forms:

- **SELECT DISTINCT:** Returns one row for each unique combination of *select-item* values. You can specify one or more than one *select-items*. For example, the following query returns a row with `Home_State` and `Age` values for each unique combination of `Home_State` and `Age` values:

```
SELECT DISTINCT Home_State, Age FROM Sample.Person
```

- **SELECT DISTINCT BY (item):** Returns one row for each unique combination of *item* values. You can specify a single *item* or a comma-separated list of *items*. The specified *item* or *item* list must be enclosed in parentheses. Spaces may be specified or omitted between the **BY** keyword and the parentheses. The *select-item* list may, but does not have to, include the specified *item(s)*. For example, the following query returns a row with `Name` and `Age` values for each unique combination of `Home_State` and `Age` values:

```
SELECT DISTINCT BY (Home_State, Age) Name, Age FROM Sample.Person
```

An *item* is commonly a column name. However, it can be any valid *select-item* value, except an aggregate function or an asterisk. It cannot be a column name alias.

The **DISTINCT** clause is applied before the **TOP** clause. If both are specified, the **SELECT** returns only rows with unique values, the number of unique value rows specified in the **TOP** clause.

If the column specified in the **DISTINCT** clause has rows that are **NULL** (contain no value), **DISTINCT** returns one row with **NULL** as a distinct (unique) value, as shown in the following examples:

```
SELECT DISTINCT FavoriteColors FROM Sample.Person
```

```
SELECT DISTINCT BY (FavoriteColors) Name, FavoriteColors FROM Sample.Person
ORDER BY FavoriteColors
```

A DISTINCT clause is not meaningful in an [Embedded SQL](#) simple query, because in this type of Embedded SQL a **SELECT** always returns only one row of data. However, an Embedded SQL [cursor-based query](#) can return multiple rows of data; in a cursor-based query a DISTINCT clause returns only unique value rows.

DISTINCT and ORDER BY

The **ORDER BY** clause is applied before the DISTINCT clause. You can therefore use the combination of DISTINCT and ORDER BY to return values from specific rows.

For example, the following program returns one row for each distinct Home_State value. Because the rows have been ordered by Name value in ascending collation sequence, the row returned for each Home_State is the one with the highest Name value:

```
SELECT DISTINCT BY (Home_State) Home_State,Name FROM Sample.Person
ORDER BY Name
```

For further details and program examples, refer to the [ORDER BY clause](#) reference page.

DISTINCT and GROUP BY

DISTINCT and GROUP BY both group records by a specified field (or fields) and return one record for each unique value of that field. One significant difference between them is that DISTINCT calculates aggregate functions before grouping. GROUP BY calculates aggregate functions after grouping. This difference is shown in the following examples:

```
SELECT DISTINCT BY (ROUND(Age,-1)) Age,AVG(Age) AS AvgAge FROM Sample.Person
/* AVG(Age) returns average of all ages in table */
```

```
SELECT Age,AVG(Age) AS AvgAge FROM Sample.Person GROUP BY ROUND(Age,-1)
/* AVG(Age) returns an average age for each age group */
```

A DISTINCT clause can be specified with one or more aggregate function fields, though this is rarely meaningful because an aggregate function returns a single value. Thus the following example returns a single row:

```
SELECT DISTINCT BY (AVG(Age)) Name,Age,AVG(Age) FROM Sample.Person
```

CAUTION: If a DISTINCT clause with [aggregate functions](#) as the only *item* or *select-item* is used with a GROUP BY clause, the DISTINCT clause is ignored. The intended combination of DISTINCT, aggregate function, and GROUP BY can be achieved using a subquery. For further details and program examples, refer to the [GROUP BY clause](#) reference page.

Letter Case and DISTINCT Optimization

DISTINCT groups together string values based on the [collation type](#) defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

If the field/property collation type is SQLUPPER, grouped field values are returned in all uppercase letters. To group values by original letter case, or to display the returned values for a grouped field in their original letter case, use the **%EXACT** collation function. This is shown in the following examples, which assume that the Home_City field is defined with collation type SQLUPPER and contains the values ‘New York’ and ‘new york’:

```
SELECT DISTINCT BY (Home_City) Name,Home_City FROM Sample.Person
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in uppercase letters.
   Thus, 'NEW YORK' is returned. */
```

```
SELECT DISTINCT BY (Home_City) Name,%EXACT(Home_City) FROM Sample.Person
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in original letter case.
   Thus, 'New York' or 'new york' may be returned, but not both. */
```

```
SELECT DISTINCT BY (%EXACT(Home_City)) Name,Home_City FROM Sample.Person
/* groups together Home_City values by their original letter case
   returns the name of each grouped city in original letter case.
   Thus, both 'New York' and 'new york' are returned.
   Optimization is not used. */
```

You can optimize query performance for queries that contain a **DISTINCT** clause by using the Management Portal. Select **[System] > [Configuration] > [General SQL Settings]**, then view and edit the **DISTINCT Optimization Turned ON** option. (This optimization also works for the [GROUP BY clause](#).) The default is “Yes”. For further details, refer to [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.

You can also set this system-wide option to 1 or 0 with the **SetFastDistinct()** method:

```
WRITE $SYSTEM.SQL.SetFastDistinct(1)
```

This optimization takes advantage of indices for the selected field(s). It is therefore only meaningful if an index exists for one or more of the selected fields. It collates field values as they are stored in the index; alphabetic strings are returned in all uppercase letters. You can set this system-wide option, then override it for specific queries by using the **%EXACT** collation function to preserve letter case.

Other Uses of **DISTINCT**

- **Asterisk Syntax:** The syntax **DISTINCT *** is legal, but not meaningful, because all rows, by definition, contain some distinct unique identifier. The syntax **DISTINCT BY (*)** is not legal.
- **Subquery:** The use of a **DISTINCT** clause in a subquery is legal, but not meaningful, because a subquery returns a single value.
- **No Row Data Selected:** The **DISTINCT** clause can be used with a **SELECT** that does not access any table data. If the **SELECT** contains a **FROM** clause, specifying **DISTINCT** results in one row contain these non-table values; if you do not specify **DISTINCT** (or **TOP**) the **SELECT** results in as many rows with identical values as the number of rows in the **FROM** clause table. If the **SELECT** does not contain a **FROM** clause, **DISTINCT** is legal but not meaningful. See [FROM clause](#) for more details.
- **Aggregate Function:** A **DISTINCT** clause can be used within an [aggregate function](#) to select only distinct (unique) field values for inclusion in the aggregate. Unlike the **SELECT DISTINCT** clause, **DISTINCT** within an aggregate function does not include **NULL** as a distinct (unique) value. Note that the **MAX** and **MIN** aggregate functions parse **DISTINCT** clause syntax without error, but this syntax performs no operation.

DISTINCT and **%ROWID**

Specifying the **DISTINCT** keyword causes a [cursor-based Embedded SQL query](#) to not set the **%ROWID** variable. **%ROWID** is not set even when **DISTINCT** does not limit the rows returned. This is shown in the following example:

```
SET %ROWID=999
&sql(DECLARE EmpCursor CURSOR FOR
      SELECT DISTINCT Name, Home_State
      INTO :name,:state FROM Sample.Person
      WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
      QUIT:(SQLCODE'=0)
FOR { &sql(FETCH EmpCursor)
      QUIT:SQLCODE
      WRITE !,"RowID: ",%ROWID," row count: ",%ROWCOUNT
      WRITE " Name=",name," State=",state
}
&sql(CLOSE EmpCursor)
```

This change of query behavior only applies to cursor-based Embedded SQL **SELECT** queries. Dynamic SQL **SELECT** queries and non-cursor Embedded SQL **SELECT** queries never set **%ROWID**.

DISTINCT and Transaction Processing

Specifying the `DISTINCT` keyword causes a query to retrieve all current data, including data that has not yet been committed by the current transaction. The transaction's `READ COMMITTED` isolation mode parameter (if set) is ignored; all data is retrieved in `READ UNCOMMITTED` mode. For further details, refer to [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

Examples

The following query returns one row for each distinct `Home_State` value:

```
SELECT DISTINCT Home_State FROM Sample.Person
ORDER BY Home_State
```

The following query returns one row for each distinct `Home_State` value, but returns additional fields for that row. Which row is retrieved is not predictable:

```
SELECT DISTINCT BY (Home_State) %ID,Name,Home_State,Office_State FROM Sample.Person
ORDER BY Home_State
```

The following query returns one row for each distinct combination of `Home_State` and `Office_State` values. Depending on the data, it will either return more rows or the same number of rows as the previous example:

```
SELECT DISTINCT BY (Home_State,Office_State) %ID,Name,Home_State,Office_State FROM Sample.Person
ORDER BY Home_State,Office_State
```

The following query uses `DISTINCT BY` to return one row for each distinct `Name` length:

```
SELECT DISTINCT BY ($LENGTH(Name)) Name,$LENGTH(Name) AS lname
FROM Sample.Person
ORDER BY lname
```

The following query uses `DISTINCT BY` to return one row for each distinct first element of `FavoriteColors %List` values. It lists one distinct row with `FavoriteColors NULL`:

```
SELECT DISTINCT BY ($LIST(FavoriteColors,1)) Name,FavoriteColors,$LIST(FavoriteColors,1) AS FirstColor
FROM Sample.Person
```

The following query returns the first 20 distinct `Home_State` values retrieved from `Sample.Person` in ascending collation sequence order. The “top” rows reflect the `ORDER BY` clause sequencing of all of the rows in `Sample.Person`.

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person ORDER BY Home_State
```

The following query uses `DISTINCT` in both the main query and in a `WHERE` clause subquery. It returns the first 20 distinct `Home_State` values in `Sample.Person` that are also in `Sample.Employee`. If the subquery `DISTINCT` was not provided, it would retrieve the distinct `Home_State` values in `Sample.Person` that match a random selection of `Home_State` values in `Sample.Employee`:

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person
WHERE Home_State IN(SELECT DISTINCT TOP 20 Home_State FROM Sample.Employee)
ORDER BY Home_State
```

The following query returns the first 20 distinct `FavoriteColor` values. This reflects the `ORDER BY` clause sequencing of all of the rows in `Sample.Person`. The `FavoriteColors` field is known to have `NULLs`, so one distinct row with `FavoriteColors NULL` appears at the top of the collation sequence.

```
SELECT DISTINCT BY (FavoriteColors) TOP 20 FavoriteColors,Name FROM Sample.Person
ORDER BY FavoriteColors
```

Also note in the preceding example that because `FavoriteColors` is a list field, the collation sequence includes the element length byte. Thus distinct list values beginning with a three-letter element (`RED`) are listed before list values beginning with a four-letter element (`BLUE`).

See Also

- [SELECT](#) statement
- [GROUP BY](#) clause
- [ORDER BY](#) clause
- [TOP](#) clause
- [Aggregate Functions](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*
- “[Collation](#)” chapter in *Using Caché SQL*

DROP DATABASE

Deletes a database (namespace).

```
DROP DATABASE dbname [RETAIN_FILES]
```

Arguments

<i>dbname</i>	The name of the database (namespace) to be deleted.
RETAIN_FILES	<i>Optional</i> — If specified, the physical database files (CACHE.DAT files) will not be deleted. The default is to delete the .DAT files along with the namespace and the other database entities.

Description

The **DROP DATABASE** command deletes a namespace and its associated database.

The specified *dbname* is the name of the namespace and the directory that contains the corresponding database files. Specify *dbname* as an [identifier](#). Namespace names are not case-sensitive. If the specified *dbname* namespace does not exist, Caché issues an SQLCODE -340 error.

The **DROP DATABASE** command is a privileged operation. Prior to using **DROP DATABASE**, it is necessary to be logged in as a user with the %Admin_Manage resource. The user must also have READ permission on the resource for the routines and global's database definitions. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the `$$SYSTEM.Security.Login()` method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login( "_SYSTEM", "SYS" )
&sql( )
```

You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

DROP DATABASE cannot be used to drop a system namespace, regardless of privileges. Attempting to do so results in an SQLCODE -342 error.

DROP DATABASE cannot be used to drop the namespace that you are currently using or connected to. Attempting to do so results in an SQLCODE -344 error.

You can also delete a namespace using the Management Portal. Select **System Administration, Configuration, System Configuration, Namespaces** to list the existing namespaces. Click the **Delete** button for the namespace you wish to delete.

RETAIN_FILES

If you specify this option, the physical file structure is retained; the database and its associated namespace is removed. After performing this operation, a subsequent attempt to use *dbname* results in the following:

- **DROP DATABASE** without RETAIN_FILES *cannot* remove this physical file structure. Instead, it results in an SQLCODE -340 error (Database not found).
- **DROP DATABASE** with RETAIN_FILES also results in an SQLCODE -340 error (Database not found).
- **CREATE DATABASE** *cannot* create a new database with the same name. Instead, it results in an SQLCODE -341 error (Cannot create database file for database).
- Attempting to use this namespace results in a <NAMESPACE> error.

Example

The following example deletes a namespace and its associated database (in this case 'C:\InterSystems\Cache\mgr\DocTestDB'). It retains the physical database files:

```
CREATE DATABASE DocTestDB ON DIRECTORY 'C:\InterSystems\Cache142\mgr\DocTestDB'
```

```
DROP DATABASE DocTestDB RETAIN_FILES
```

See Also

- [CREATE DATABASE](#) command
- [USE DATABASE](#) command

DROP FUNCTION

Deletes a function.

```
DROP FUNCTION name [ FROM className ]
```

Arguments

<i>name</i>	The name of the function to be deleted. The name is an identifier . Do not specify the function's parameter parentheses. A <i>name</i> can be qualified (schema.name), or unqualified (name). An unqualified function name takes the system-wide default schema name , unless the FROM <i>className</i> clause is specified.
FROM <i>className</i>	<i>Optional</i> — If specified, the FROM <i>className</i> clause deletes the function from the given class. If the FROM clause is not specified, Caché searches all classes of the schema for the function, and deletes it. However, if no function of this name is found, or more than one function of this name is found, an error code is returned. If the deletion of the function results in an empty class, DROP FUNCTION deletes the class as well.

Description

The **DROP FUNCTION** command deletes a function. When you drop a function, Caché revokes it from all users and roles to whom it has been granted and removes it from the database.

In order to drop a function, you must have %DROP_FUNCTION administrative privilege, as specified by the [GRANT](#) command. Otherwise, the system generates an SQLCODE -99 error (Privilege Violation).

The following combinations of *name* and FROM *className* are supported. Note that the FROM clause specifies the class package name and function name, not the SQL names. In these examples, the [system-wide default schema name](#) is SQLUser, which corresponds to the User class package:

- DROP FUNCTION BonusCalc FROM funcBonusCalc: drops the function SQLUser.BonusCalc().
- DROP FUNCTION BonusCalc FROM User.funcBonusCalc: drops the function SQLUser.BonusCalc().
- DROP FUNCTION Test.BonusCalc FROM funcBonusCalc: drops the function SQLUser.BonusCalc().
- DROP FUNCTION BonusCalc FROM Employees.funcBonusCalc: drops the function Employees.BonusCalc().
- DROP FUNCTION Test.BonusCalc FROM Employees.funcBonusCalc: drops the function Employees.BonusCalc().

If the specified function does not exist, **DROP FUNCTION** generates an SQLCODE -362 error. If the specified class does not exist, **DROP FUNCTION** generates an SQLCODE -360 error. If the specified function could refer to two or more functions, **DROP FUNCTION** generates an SQLCODE -361 error; you must specify a *className* to resolve this ambiguity.

Examples

The following embedded SQL example attempts to delete myfunc from the class User.Employee. (Refer to **CREATE TABLE** for an example that creates class User.Employee.)

```
&sql(DROP FUNCTION myfunc FROM User.Employee)
IF SQLCODE=0 {
  WRITE !,"Function deleted" }
ELSEIF SQLCODE=-360 {
  WRITE !,"Nonexistent class: ",%msg }
ELSEIF SQLCODE=-362 {
  WRITE !,"Nonexistent function: ",%msg }
ELSE {WRITE !,"Unexpected Error code: ",SQLCODE}
```

See Also

- [CREATE FUNCTION](#)
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DROP INDEX

Removes an index.

```
DROP INDEX index-name [ON [TABLE] [schema-name.]table-name]
```

```
DROP INDEX [schema-name.]table-name.index-name
```

Arguments

<i>index-name</i>	The name of the index to be deleted. <i>index-name</i> may be a standard index or a bitmap index.
ON <i>table-name</i> or ON TABLE <i>table-name</i>	<i>Optional</i> — The name of the table associated with the index. You can specify the <i>table-name</i> using either syntax: The first syntax uses the ON clause; the TABLE keyword is optional. The second syntax uses the qualified name syntax <code>schema-name.table-name.index-name</code> . A <i>table-name</i> can be qualified (schema.table), or unqualified (table). An unqualified table name takes the system-wide default schema name . If you omit the <i>table-name</i> entirely, Caché deletes the first index found that matches <i>index-name</i> , as described below.

Description

A **DROP INDEX** statement deletes an index. **DROP INDEX** does not apply to indices created by defining PRIMARY KEY or UNIQUE constraints (created by using the PRIMARY KEY or UNIQUE options of either the **CREATE TABLE** or **ALTER TABLE** statements, respectively).

You may wish to delete an index for any of the following reasons:

- You intend to perform large numbers of **INSERT**, **UPDATE**, or **DELETE** operations on a table. Rather than accepting the performance overhead of having each of these operations write to the index, you can use the %NOINDEX option for the operation. Or, in certain cases, it may be preferable to delete the index, perform the bulk changes to the database, and then recreate the index and populate it.
- An index exists for a field or combination of fields that are not used for query operations. In this case, the performance overhead of maintaining the index may not be worthwhile.
- An index exists for a field or combination of fields that now contain large amounts of duplicate data. In this case, the minimal gain to query performance may not be worthwhile.

You cannot drop an IDKEY index when there is data in the table. Attempting to do so generates an SQLCODE -325 error.

Privileges and Locking

The **DROP INDEX** command is a privileged operation. Prior to using **DROP INDEX** it is necessary for your process to have either %ALTER_TABLE administrative privileges or the %ALTER privilege for the specified table. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has %ALTER privilege by invoking the %CHECKPRIV command. You can determine if a specified user has %ALTER privilege by invoking the \$SYSTEM.SQL.CheckPriv() method. You can use the **GRANT** command to assign these privileges, if you hold appropriate granting privileges.

DROP INDEX cannot be used on a [table created by defining a persistent class](#), unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg DDL not enabled for class 'Schema.tablename' >

The **DROP INDEX** statement acquires a table-level lock on *table-name*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **DROP INDEX** operation.

Index Name and Table Name

When specify an *index-name* to create an index, the system generates a corresponding class index name by stripping out any punctuation characters; it retains the *index-name* you specified in the class as the `SqlName` value for the index.

You can specify the table associated with the index using either **DROP INDEX** syntax form:

- `index-name ON TABLE` syntax: specifying the table name is optional. If omitted, Caché searches all of the classes in the namespace for the corresponding index.
- `table-name.index-name` syntax: specifying the table name is required.

In either syntax, the table name can be unqualified (table), or qualified (schema.table). If the schema name is omitted, the [system-wide default schema name](#) is used.

If **DROP INDEX** does not specify a table name, Caché searches through all indices for an index `SqlName` matching *index-name*, or an index name matching *index-name* for indices where an `SqlName` is not specified for the index. If Caché finds no matching indices in any class, an `SQLCODE -333` error is generated, indicating no such index exists. If Caché finds more than one matching index, **DROP INDEX** cannot determine which index to drop; it issues an `SQLCODE -334` error: “Index name is ambiguous. Index found in multiple tables.” Index names in Caché are not unique per namespace.

Nonexistent Index

If you try to delete a nonexistent index, **DROP INDEX** issues an `SQLCODE -333` error, by default. However, this default can be overridden system-wide by setting a configuration option as follows:

- The `$$SYSTEM.SQL.SetDDLNo333()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a `Suppress SQLCODE=-333 Errors` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow DDL DROP of Non-existent Index**.

The default is “No” (0). By default, Caché issues an `SQLCODE -333` error. This is the recommended setting for this option. Set this option to “Yes” (1) if you want a **DROP INDEX** for a nonexistent index to perform no operation and issue no error message. For further details, refer to [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.

Table Name

If you specify the optional *table-name*, it must correspond to an existing table.

- If the specified *table-name* does not exist, Caché issues an `SQLCODE -30` error and sets `%msg` to `Table 'SQLUser.tname' does not exist.`
- If the specified *table-name* exists but does not have an index named *index-name*, Caché issues an `SQLCODE -333` error and sets `%msg` to `Attempt to DROP INDEX 'MyIndex' on table SQLUSER.TNAME failed - index not found.`
- If the specified *table-name* is a view, Caché issues an `SQLCODE -333` error and sets `%msg` to `Attempt to DROP INDEX 'EmpSalaryIndex' on view SQLUSER.VNAME failed. Indices only supported for tables, not views.`

Examples

The first example creates a table named `Employee`, which is used in all of the examples in this section.

The following embedded SQL example creates an index named `"EmpSalaryIndex"` and later removes it. Note that here **DROP INDEX** does not specify the table associated with the index; it assumes that `"EmpSalaryIndex"` is a unique index name in this namespace.

```

&sql(CREATE TABLE Employee (
EMPNUM      INT NOT NULL,
NAMELAST    CHAR(30) NOT NULL,
NAMEFIRST   CHAR(30) NOT NULL,
STARTDATE   TIMESTAMP,
SALARY      MONEY,
ACCRUEDVACATION  INT,
ACCRUEDSICKLEAVE  INT,
CONSTRAINT EMPLOYEEPK PRIMARY KEY (EMPNUM))
)
WRITE !,"SQLCODE=",SQLCODE," Created a table"
&sql(CREATE INDEX EmpSalaryIndex
      ON TABLE Employee
      (NameLast,Salary))
WRITE !,"SQLCODE=",SQLCODE," Created an index"
/* use the index */
NEW SQLCODE,%msg
&sql(DROP INDEX EmpSalaryIndex)
WRITE !,"SQLCODE=",SQLCODE," Deleted an index"
WRITE !,"message",%msg

```

The following embedded SQL example specifies the table associated with the index to be dropped using an ON TABLE clause:

```

&sql(CREATE INDEX EmpVacaIndex
      ON TABLE Employee
      (NameLast,AccruedVacation))
WRITE !,"SQLCODE=",SQLCODE," Created an index"
/* use the index */
&sql(DROP INDEX EmpVacaIndex ON TABLE Employee)
WRITE !,"SQLCODE=",SQLCODE," Deleted an index"

```

The following embedded SQL example specifies the table associated with the index to be dropped using qualified name syntax:

```

&sql(CREATE INDEX EmpSickIndex
      ON TABLE Employee
      (NameLast,AccruedSickLeave))
WRITE !,"SQLCODE=",SQLCODE," Created an index"
/* use the index */
&sql(DROP INDEX Employee.EmpSickIndex)
WRITE !,"SQLCODE=",SQLCODE," Deleted an index"

```

The following command attempts to drop a nonexistent index. It generates an SQLCODE -333 error:

```
DROP INDEX PeopleIndex ON TABLE Employee
```

See Also

- [CREATE INDEX](#)
- “[Defining and Building Indices](#)” chapter in *Caché SQL Optimization Guide*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DROP METHOD

Deletes a method.

```
DROP METHOD name [ FROM className ]
```

Arguments

<i>name</i>	The name of the method to be deleted. The name is an identifier . Do not specify the method's parameter parentheses. A <i>name</i> can be qualified (schema.name), or unqualified (name). An unqualified method name takes the system-wide default schema name , unless the FROM <i>className</i> clause is specified.
FROM <i>className</i>	<i>Optional</i> — If specified, the FROM <i>className</i> clause deletes the method from the given class. If this clause is not specified, Caché searches all classes of the schema for the method, and deletes it. However, if no method of this name is found, or more than one method of this name is found, an error code is returned. If the deletion of the method results in an empty class, DROP METHOD deletes the class as well.

Description

The **DROP METHOD** command deletes a method. When you delete a method, Caché revokes it from all users and roles to whom it has been granted and removes it from the database.

In order to delete a method, you must have %DROP_METHOD administrative privilege, as specified by the [GRANT](#) command. If you are attempting to delete a method for a class with a defined owner, you must be logged in as the owner of the class. Otherwise, the system generates an SQLCODE -99 error (Privilege Violation).

The following combinations of *name* and FROM *className* are supported. Note that the FROM clause specifies the class package name and method name, not the SQL names. In these examples, the [system-wide default schema name](#) is SQLUser, which corresponds to the User class package:

- DROP METHOD BonusCalc FROM methBonusCalc: drops the method SQLUser.BonusCalc().
- DROP METHOD BonusCalc FROM User.methBonusCalc: drops the method SQLUser.BonusCalc().
- DROP METHOD Test.BonusCalc FROM methBonusCalc: drops the method SQLUser.BonusCalc().
- DROP METHOD BonusCalc FROM Employees.methBonusCalc: drops the method Employees.BonusCalc().
- DROP METHOD Test.BonusCalc FROM Employees.methBonusCalc: drops the method Employees.BonusCalc().

If the specified method does not exist, **DROP METHOD** generates an SQLCODE -362 error. If the specified *className* does not exist, **DROP METHOD** generates an SQLCODE -360 error. If the specified method could refer to two or more methods, **DROP METHOD** generates an SQLCODE -361 error; you must specify a *className* to resolve this ambiguity.

If a method has been defined with the PROCEDURE characteristic keyword, you can determine if it exists in the current namespace by invoking the \$SYSTEM.SQL.ProcedureExists() method. A method defined with the PROCEDURE keyword can be deleted either by **DROP METHOD** or **DROP PROCEDURE**.

Examples

The following embedded SQL example attempts to delete mymeth from the class User.Employee. (Refer to **CREATE TABLE** for an example that creates class User.Employee.)

```
&sql(DROP METHOD mymeth FROM User.Employee)
IF SQLCODE=0 {
  WRITE !,"Method deleted" }
ELSEIF SQLCODE=-360 {
  WRITE !,"Nonexistent class: ",%msg }
ELSEIF SQLCODE=-362 {
  WRITE !,"Nonexistent method: ",%msg }
ELSE {WRITE !,"Unexpected Error code: ",SQLCODE}
```

See Also

- [CREATE METHOD](#)
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DROP PROCEDURE

Deletes a procedure.

```
DROP PROCEDURE procname [ FROM className ]
DROP PROC procname [ FROM className ]
```

Arguments

<i>procname</i>	The name of the procedure to be deleted. The name is an identifier . Do not specify the procedure's parameter parentheses. A <i>name</i> can be qualified (schema.name), or unqualified (name). An unqualified procedure name takes the system-wide default schema name , unless the FROM <i>className</i> clause is specified.
FROM <i>className</i>	<i>Optional</i> — If specified, the FROM <i>className</i> clause deletes the procedure from the given class. If this clause is not specified, Caché searches all classes of the schema for the procedure, and deletes it. However, if no procedure of this name is found, or more than one procedure of this name is found, an error code is returned. If the deletion of the procedure results in an empty class, DROP PROCEDURE deletes the class as well.

Description

The **DROP PROCEDURE** command deletes a procedure in the current namespace. When you drop a procedure, Caché revokes it from all users and roles to whom it has been granted and removes it from the database.

In order to drop a procedure, you must have %DROP_PROCEDURE administrative privilege, as specified by the [GRANT](#) command. If you are attempting to delete a procedure for a class with a defined owner, you must be logged in as the owner of the class. Otherwise, the system generates an SQLCODE -99 error (Privilege Violation).

The *procname* is not case-sensitive. You must specify *procname* without parameter parentheses; specifying parameter parentheses results in an SQLCODE -25 error.

The following combinations of *procname* and FROM *className* are supported. Note that the FROM clause specifies the class package name and procedure name, not the SQL names. In these examples, the [system-wide default schema name](#) is SQLUser, which corresponds to the User class package:

- DROP PROCEDURE BonusCalc FROM procBonusCalc: drops the procedure SQLUser.BonusCalc().
- DROP PROCEDURE BonusCalc FROM User.procBonusCalc: drops the procedure SQLUser.BonusCalc().
- DROP PROCEDURE Test.BonusCalc FROM procBonusCalc: drops the procedure SQLUser.BonusCalc().
- DROP PROCEDURE BonusCalc FROM Employees.procBonusCalc: drops the procedure Employees.BonusCalc().
- DROP PROCEDURE Test.BonusCalc FROM Employees.procBonusCalc: drops the procedure Employees.BonusCalc().

If the specified procedure does not exist, **DROP PROCEDURE** generates an SQLCODE -362 error. If the specified class does not exist, **DROP PROCEDURE** generates an SQLCODE -360 error. If the specified procedure could refer to two or more procedures, **DROP PROCEDURE** generates an SQLCODE -361 error; you must specify a *className* to resolve this ambiguity.

To determine if a specified *procname* exists in the current namespace, use the `$$SYSTEM.SQL.ProcedureExists()` method. This method recognizes both procedures and methods defined with the PROCEDURE keyword. A method defined with the PROCEDURE keyword can be deleted using **DROP PROCEDURE**.

If you execute a **DROP PROCEDURE** for a procedure that is an ObjectScript class query procedure, Caché will also drop the methods related to the procedure, such as `myprocExecute()`, `myprocGetInfo()`, `myprocFetch()`, `myprocFetchRows()`, and `myprocClose()`.

Examples

The following embedded SQL example attempts to delete `myprocSP` from the class `User.Employee`. (Refer to **CREATE TABLE** for an example that creates class `User.Employee`.)

```
&sql(DROP PROCEDURE myprocSP FROM User.Employee)
IF SQLCODE=0 {
    WRITE !,"Procedure deleted" }
ELSEIF SQLCODE=-360 {
    WRITE !,"Nonexistent class: ",%msg }
ELSEIF SQLCODE=-362 {
    WRITE !,"Nonexistent procedure: ",%msg }
ELSE {WRITE !,"Unexpected Error code: ",SQLCODE}
```

See Also

- [CREATE PROCEDURE](#)
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DROP QUERY

Deletes a query.

```
DROP QUERY name [ FROM className ]
```

Arguments

<i>name</i>	The name of the query to be deleted. The name is an identifier . Do not specify the query's parameter parentheses. A <i>name</i> can be qualified (schema.name), or unqualified (name). An unqualified query name takes the system-wide default schema name , unless the FROM <i>className</i> clause is specified.
FROM <i>className</i>	<i>Optional</i> — If specified, the FROM <i>className</i> clause deletes the query from the given class. If this clause is not specified, Caché searches all classes of the schema for the query, and deletes it. However, if no query of this name is found, or more than one query of this name is found, an error code is returned. If the deletion of the query results in an empty class, DROP QUERY deletes the class as well.

Description

The **DROP QUERY** command deletes a query. When you drop a query, Caché revokes it from all users and roles to whom it has been granted and removes it from the database.

In order to drop a query, you must have %DROP_QUERY administrative privilege, as specified by the [GRANT](#) command. If you are attempting to delete a query for a class with a defined owner, you must be logged in as the owner of the class. Otherwise, the system generates an SQLCODE -99 error (Privilege Violation).

The following combinations of *name* and FROM *className* are supported. Note that the FROM clause specifies the class package name and query name, not the SQL names. In these examples, the [system-wide default schema name](#) is SQLUser, which corresponds to the User class package:

- DROP QUERY BonusCalc FROM queryBonusCalc: drops the query SQLUser.BonusCalc().
- DROP QUERY BonusCalc FROM User.queryBonusCalc: drops the query SQLUser.BonusCalc().
- DROP QUERY Test.BonusCalc FROM queryBonusCalc: drops the query SQLUser.BonusCalc().
- DROP QUERY BonusCalc FROM Employees.queryBonusCalc: drops the query Employees.BonusCalc().
- DROP QUERY Test.BonusCalc FROM Employees.queryBonusCalc: drops the query Employees.BonusCalc().

If the specified query does not exist, **DROP QUERY** generates an SQLCODE -362 error. If the specified class does not exist, **DROP QUERY** generates an SQLCODE -360 error. If the specified query could refer to two or more queries, **DROP QUERY** generates an SQLCODE -361 error; you must specify a *className* to resolve this ambiguity.

Examples

The following embedded SQL example attempts to delete myq from the class User.Employee. (Refer to **CREATE TABLE** for an example that creates class User.Employee.)

```
&sql(DROP QUERY myq FROM User.Employee)
IF SQLCODE=0 {
  WRITE !,"Query deleted" }
ELSEIF SQLCODE=-360 {
  WRITE !,"Nonexistent class: ",%msg }
ELSEIF SQLCODE=-362 {
  WRITE !,"Nonexistent query: ",%msg }
ELSE {WRITE !,"Unexpected Error code: ",SQLCODE}
```

See Also

- [CREATE QUERY](#)
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DROP ROLE

Deletes a role.

```
DROP ROLE role-name
```

Arguments

<i>role-name</i>	The name of the role to be deleted. The name is an identifier . Role names are not case-sensitive. For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> .
------------------	--

Description

The **DROP ROLE** statement deletes a role. When you drop a role, Caché revokes it from all users and roles to whom it has been granted and removes it from the database.

You can determine if a role exists by invoking the `$$SYSTEM.SQL.RoleExists()` method. If you attempt to drop a role that does not exist (or has already been dropped), **DROP ROLE** issues an SQLCODE -118 error.

Privileges

The **DROP ROLE** command is a privileged operation. Prior to using **DROP ROLE** in embedded SQL, it is necessary to fulfill at least one of the following requirements:

- You must have `%Admin_Secure:USE` privilege.
- You are the owner of the role.
- You were granted the role `WITH ADMIN OPTION`.

Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the `$$SYSTEM.Security.Login()` method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql( )
```

You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to `%SYSTEM.Security` in the *InterSystems Class Reference*.

Examples

The following embedded SQL example creates a role named BkUser and later deletes it:

```
DO $$SYSTEM.Security.Login("MyName", "SecretPassword")
&sql(CREATE ROLE BkName)
IF SQLCODE=-99 {
WRITE !,"You don't have CREATE ROLE privileges" }
ELSE { WRITE !,"Created a role" }
/* Use role */
&sql(DROP ROLE BkName)
IF SQLCODE=-99 {
WRITE !,"You don't have DROP ROLE privileges" }
ELSE { WRITE !,"Dropped the role" }
```

See Also

- SQL statements: [CREATE ROLE CREATE USER DROP USER GRANT REVOKE %CHECKPRIV](#)
- “Users, Roles, and Privileges” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

- ObjectScript: [\\$ROLES](#) and [\\$USERNAME](#) special variables

DROP TABLE

Deletes a table and (optionally) its data.

```
DROP TABLE table
  [RESTRICT | CASCADE] [%DELDATA | %NODELDATA]
```

Arguments

<i>table</i>	The name of the table to be deleted. The table name can be qualified (schema.table), or unqualified (table). An unqualified table name takes the system-wide default schema name . Schema search path values are not used.
RESTRICT CASCADE	<i>Optional</i> — RESTRICT only allows a table with no dependent views or integrity constraints to be deleted. CASCADE allow a table with dependent views or integrity constraints to be deleted; any referencing views or integrity constraints will also be deleted as part of the table deletion. (See restriction on CASCADE below.)
%DELDATA %NODELDATA	<i>Optional</i> — These keywords specify whether to delete data associated with a table when deleting the table. The default is to delete table data.

Description

The **DROP TABLE** command deletes a table and its corresponding persistent class definition. If the table is the last item in its schema, deleting the table also deletes the schema and its corresponding persistent class package.

By default, **DROP TABLE** deletes both the table definition and the [table's data](#) (if any exists). The %NODELDATA keyword allows you to specify deletion of the table definition but not the table's data.

In order to delete a table, the following conditions must be met:

- The table must exist in the current namespace. Attempting to delete a non-existent table generates an SQLCODE -30 error.
- The table definition must be modifiable. If the class that projects the table is defined without [DdlAllowed](#), attempting to delete the table generates an SQLCODE -300 error.
- The table must not be locked by another concurrent process. If the table is locked, **DROP TABLE** waits indefinitely for the lock to be released. If lock contention is a possibility, it is important that you [LOCK](#) the table IN EXCLUSIVE MODE before issuing a **DROP TABLE**.
- You must have the necessary privileges to delete the table. Attempting to delete a table without the necessary privileges generates an SQLCODE -99 error.

This statement can also be invoked using the `$$SYSTEM.SQL.DropTable()` method call:

```
$$SYSTEM.SQL.DropTable(tablename, deldata, SQLCODE, %msg)
```

Privileges

The **DROP TABLE** command is a privileged operation. Prior to using **DROP TABLE** it is necessary for your process to have either %DROP_TABLE administrative privilege or a DELETE object privilege for the specified table. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has DELETE privilege by invoking the [%CHECKPRIV](#) command. You can determine if a specified user has DELETE privilege by invoking the `$$SYSTEM.SQL.CheckPriv()` method. You can use the [GRANT](#) command to assign %DROP_TABLE privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the `$$SYSTEM.Security.Login()` method to log in as a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login( "_SYSTEM", "SYS" )
&sql( )
```

You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to `%SYSTEM.Security` in the *InterSystems Class Reference*.

DROP TABLE cannot be used on a table created by defining a persistent class, unless the table class definition includes `[DdlAllowed]`. Otherwise, the operation fails with an SQLCODE -300 error with the `%msg DDL not enabled for class 'Schema.tablename'>`

Existing Object Privileges

Deleting a table does not delete the object privileges for that table. For example, the privilege granted to a user to insert, update, or delete data on that table. This has the following two consequences:

- If a table is deleted, and then another table with the same name is created, users and roles will have the same privileges on the new table that they had on the old table.
- Once a table is deleted, it is not possible to revoke object privileges for that table.

For these reasons, it is generally recommended that you use the `REVOKE` command to revoke object privileges from a table before deleting the table.

Table Containing Data

By default, **DROP TABLE** deletes the table definition and deletes the table's data. This table data delete is an atomic operation; if **DROP TABLE** encounters data that cannot be deleted (for example, a row with a referential constraint) any data deletion already performed is automatically rolled back, with the result that no table data is deleted.

The deletion of data can be overridden on a per-table basis, or system-wide. When deleting a table, you can specify **DROP TABLE** with the `%NODELDATA` option to prevent the automatic deletion of the table's data. The default system configuration setting is to delete table data. If, however, the system-wide default is set to not delete table data, you can delete data on a per-table basis by specifying **DROP TABLE** with the `%DELDATA` option.

You can set the system-wide default for table data deletion as follows:

- The `$$SYSTEM.SQL.SetDDLDropTabDelData()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a `DROP TABLE Deletes data` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Does DDL DROP TABLE Delete the Table's Data**.

The default is “Yes” (1). This is the recommended setting for this option. Set this option to “No” (0) if you want **DROP TABLE** to not delete the table's data when it deletes the table definition.

You can use the `TRUNCATE TABLE` command to delete the table's data without deleting the table definition.

Lock Applied

The **DROP TABLE** statement acquires an exclusive table-level lock on *table*. This prevents other processes from modifying the table definition or the table data while table deletion is in process. This table-level lock is sufficient for deleting both the table definition and the table data; **DROP TABLE** does not acquire a lock on each row of the table data. This lock is automatically released at the end of the **DROP TABLE** operation.

Foreign Key Constraints

By default, you cannot drop a table if any foreign key constraints are defined on another table that references the table you are attempting to drop. You must drop all referencing foreign key constraints before dropping the table they reference.

Failing to delete these foreign key constraints before attempting a **DROP TABLE** operation results in an SQLCODE -320 error.

This default behavior is consistent with the RESTRICT keyword option. The CASCADE keyword option is not supported for foreign key constraints.

To change this default foreign key constraint behavior, refer to the COMPILEMODE=NOCHECK option of the [SET OPTION](#) command.

Associated Queries

Dropping a table automatically purges any related [cached queries](#) and purges query information as stored in %SYS.PTools.SQLQuery. Dropping a table automatically purges any [SQL runtime statistics \(SQL Stats\)](#) information for any related query.

Nonexistent Table

To determine if a specified table exists in the current namespace, use the `$$SYSTEM.SQL.TableExists()` method.

If you try to delete a nonexistent table, **DROP TABLE** issues an SQLCODE -30 error by default. However, this error-reporting behavior can be overridden by setting the system configuration as follows:

- The `$$SYSTEM.SQL.SetDDLNo30()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a `Suppress SQLCODE=-30 Errors:` setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow DDL DROP of Non-existent Table or View**.

The default is “No” (0). This is the recommended setting for this option. Set this option to “Yes” (1) if you want a **DROP TABLE** for nonexistent table to perform no operation and not issue an error message.

Examples

The following embedded SQL example creates a table named SQLUser.MyEmployees and later deletes it. This example specifies that any data associated with this table *not* be deleted when the table is deleted:

```
&sql(CREATE TABLE SQLUser.MyEmployees (
    NAMELAST    CHAR (30) NOT NULL,
    NAMEFIRST   CHAR (30) NOT NULL,
    STARTDATE   TIMESTAMP,
    SALARY      MONEY))
WRITE !,"Created a table"
/*
    &sql(SQL code populating SQLUser.MyEmployees table)
    &sql(SQL code using SQLUser.MyEmployees table)
*/
NEW SQLCODE, %msg
&sql(DROP TABLE SQLUser.MyEmployees %NODELDATA)
IF SQLCODE=0 {WRITE !,"Table deleted"}
ELSE {WRITE !,"SQLCODE=",SQLCODE," : ", %msg }
```

See Also

- [ALTER TABLE, CREATE TABLE, TRUNCATE TABLE](#)
- “[Defining Tables](#)” chapter in *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DROP TRIGGER

Deletes a trigger.

```
DROP TRIGGER name [ FROM table ]
```

Arguments

<i>name</i>	The name of the trigger to be deleted. A trigger name may be qualified or unqualified; if qualified, its schema name must match the table's schema name.
FROM <i>table</i>	<i>Optional</i> —The table the trigger is to be deleted from. If the FROM clause is specified, only the table is searched for the named trigger. If the FROM clause is not specified, the entire schema specified in <i>name</i> is searched for the named trigger.

Description

The **DROP TRIGGER** statement deletes a trigger.

Privileges and Locking

The **DROP TRIGGER** command is a privileged operation. Prior to using **DROP TRIGGER** it is necessary for your process to have %DROP_TRIGGER administrative privilege. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can use the **GRANT** command to assign %DROP_TRIGGER privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the `$$SYSTEM.Security.Login()` method to log in as a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(      )
```

You must have the `%Service_Login:Use` privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

DROP TRIGGER cannot be used on a [table created by defining a persistent class](#), unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg DDL not enabled for class 'Schema.tablename'.

The **DROP TRIGGER** statement acquires a table-level lock on *table*. This prevents other processes from modifying the table's data. This lock is automatically released at the conclusion of the **DROP TRIGGER** operation.

FROM Clause

A trigger and its table must reside in the same schema. If the trigger name is unqualified, the trigger schema name defaults to the same schema as the table schema, as specified in the FROM clause. If the trigger name is unqualified, and there is no FROM clause, or the table name is also unqualified, the trigger schema defaults to the [system-wide default schema name](#); schema search paths are not used. If both names are qualified, the trigger schema name must be the same as the table schema name. A schema name mismatch results in an SQLCODE -366 error; this should only occur when both the trigger name and the table name are qualified and they specify different schema names.

In Caché SQL, a trigger name must be unique within its schema for a specific table. Thus it is possible to have more than one trigger in a schema with the same name. The optional FROM clause is used to determine which trigger to delete:

- If no FROM clause is specified, and Caché locates a unique trigger in the schema that matches the specified name, Caché deletes the trigger.
- If a FROM clause is specified, and Caché locates a unique trigger in the schema that matches both the specified name and the FROM table name, Caché deletes the trigger.

- If no FROM clause is specified, and Caché locates more than one trigger that matches the specified name, Caché issues an SQLCODE -365 error.
- If Caché locates no trigger that matches the specified name, either for the table specified in the FROM clause or, if there is no FROM clause, for any table in the schema, Caché issues an SQLCODE -363 error.

Examples

The following example deletes a trigger named Trigger_1 associated with any table in the [system-wide default schema](#). (The initial default schema is SQLUser):

```
DROP TRIGGER Trigger_1
```

The following example deletes a trigger named Trigger_2 associated with any table in the A schema.

```
DROP TRIGGER A.Trigger_2
```

The following example deletes a trigger named Trigger_3 associated with the Patient table in the [system-wide default schema](#). If a trigger named Trigger_3 is found, but it is not associated with Patient, InterSystems IRIS issues an SQLCODE -363 error.

```
DROP TRIGGER Trigger_3 FROM Patient
```

The following examples all delete a trigger named Trigger_4 associated with the Patient table in the Test schema.

```
DROP TRIGGER Test.Trigger_4 FROM Patient
```

```
DROP TRIGGER Trigger_4 FROM Test.Patient
```

```
DROP TRIGGER Test.Trigger_4 FROM Test.Patient
```

See Also

- [CREATE TRIGGER](#)
- [GRANT](#)
- “[Using Triggers](#)” chapter in *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

DROP USER

Removes a user account.

```
DROP USER user-name
```

Arguments

<i>user-name</i>	The name of the user to be removed.
------------------	-------------------------------------

Description

The **DROP USER** command removes a user account. This user account was created and the *user-name* specified using **CREATE USER**. If the specified *user-name* does not correspond to an existing user account, Caché issues an SQLCODE -118 error. User names are not case-sensitive.

You can also delete a user by using the Management Portal. Select **System Administration, Security, Users** to list the existing users. On this table of user accounts you can click **Delete** for the user account you wish to delete.

Privileges

The **DROP USER** command is a privileged operation. Prior to using **DROP USER** in embedded SQL, it is necessary to be logged in as a user with appropriate privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the **\$\$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login( "_SYSTEM", "SYS" )
&sql( )
```

You must have the **%Service_Login:Use** privilege to invoke the **\$\$SYSTEM.Security.Login** method. For further information, refer to **%SYSTEM.Security** in the *InterSystems Class Reference*.

Examples

You can drop PSMITH by issuing the statement:

```
DROP USER psmith
```

See Also

- SQL statements: [CREATE USER ALTER USER GRANT REVOKE %CHECKPRIV](#)
- “Users, Roles, and Privileges” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*
- ObjectScript: [\\$ROLES](#) and [\\$USERNAME](#) special variables

DROP VIEW

Deletes a view.

```
DROP VIEW view-name [CASCADE | RESTRICT]
```

Arguments

<i>view-name</i>	The name of the view to be deleted. A view name can be qualified (schema.viewname), or unqualified (viewname). An unqualified view name takes the system-wide default schema name .
CASCADE RESTRICT	<i>Optional</i> — Specify the CASCADE keyword to drop any other view that references <i>view-name</i> . Specify RESTRICT to issue an SQLCODE -321 error if there is another view that references <i>view-name</i> . The default is RESTRICT.

Description

The **DROP VIEW** command removes a [view](#), but does not remove the underlying tables or data.

A drop view operation can also be invoked using the **DropView()** method call:

```
$SYSTEM.SQL.DropView(viewname, SQLCODE, %msg)
```

Privileges

The **DROP VIEW** command is a privileged operation. Prior to using **DROP VIEW** it is necessary for your process to have either %DROP_VIEW administrative privilege or a DELETE object privilege for the specified view. Failing to do so results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has DELETE privilege by invoking the %CHECKPRIV command. You can determine if a specified user has DELETE privilege by invoking the \$SYSTEM.SQL.CheckPriv() method. You can use the GRANT command to assign %DROP_VIEW privileges, if you hold appropriate granting privileges.

In embedded SQL, you can use the \$SYSTEM.Security.Login() method to log in as a user with appropriate privileges:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(      )
```

You must have the %Service_Login:Use privilege to invoke the \$SYSTEM.Security.Login method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

Nonexistent View

To determine if a specified view exists in the current namespace, use the \$SYSTEM.SQL.ViewExists() method.

If you try to delete a nonexistent view, **DROP VIEW** issues an SQLCODE -30 error by default. However, this error-reporting behavior can be overridden by setting the system configuration as follows:

- The \$SYSTEM.SQL.SetDDLNo30() method call. To determine the current setting, call \$SYSTEM.SQL.CurrentSettings(), which displays a Suppress SQLCODE=-30 Errors: setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Allow DDL DROP of Non-existent Table or View**.

The default is “No” (0). This is the recommended setting for this option. Set this option to “Yes” (1) if you want **DROP VIEW** and **DROP TABLE** for nonexistent views and tables to perform no operation and issue no error message.

VIEW Referenced by Other Views

If you try to delete a view referenced by other views in their queries, **DROP VIEW** issues an SQLCODE -321 error by default. This is the RESTRICT keyword behavior.

By specifying the CASCADE keyword, an attempt to delete a view referenced by other views in their queries succeeds. The **DROP VIEW** also deletes these other views. If Caché cannot perform all cascade view deletions (for example, due to an SQLCODE -300 error) no views are deleted.

Associated Queries

Dropping a view automatically purges any related [cached queries](#) and purges query information as stored in %SYS.PTools.SQLQuery. Dropping a view automatically purges any [SQL runtime statistics \(SQL Stats\)](#) information for any related query.

Examples

The following embedded SQL example creates a view named "CityAddressBook" and later deletes the view. Because it is specified with the RESTRICT keyword (the default), an SQLCODE -321 error is issued if the view is referenced by other views:

```
&sql(CREATE VIEW CityAddressBook AS
  SELECT Name,Home_Street FROM Sample.Person
  WHERE Home_City='Boston')
IF SQLCODE=0 { WRITE !,"View created" }
ELSE { WRITE !,"CREATE VIEW error: ",SQLCODE
      QUIT }
/* Use the view */
NEW SQLCODE,%msg
&sql(DROP VIEW CityAddressBook RESTRICT)
IF SQLCODE=0 { WRITE !,"View dropped" }
ELSEIF SQLCODE=-30 { WRITE !,"View non-existent",!,%msg }
ELSEIF SQLCODE=-321 { WRITE !,"View referenced by other views",!,%msg }
ELSE { WRITE !,"Unexpected DROP VIEW error: ",SQLCODE,!,%msg }
```

See Also

- [ALTER VIEW CREATE VIEW GRANT](#)
- “Views” chapter in *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

FETCH

Repositions a cursor, and retrieves data from it.

```
FETCH cursor-name [ INTO host-variable-list ]
```

Arguments

<i>cursor-name</i>	The name of a currently open cursor. The cursor name was specified in the DECLARE statement. Cursor names are case-sensitive.
INTO <i>host-variable-list</i>	<i>Optional</i> — Places data from the columns of a fetch into local variables. The <i>host-variable-list</i> specifies a host variable, or a comma-separated list of host variables, that are targets to contain data associated with the cursor. The INTO clause is optional. If it is not specified, the FETCH statement positions the cursor only.

Description

Within an embedded SQL application, a **FETCH** statement retrieves data from a [cursor](#). The required sequence of actions is: **DECLARE**, **OPEN**, **FETCH**, **CLOSE**. Attempting a **FETCH** on a cursor that is not open results in an SQLCODE - 102 error.

As an SQL statement, this is supported only from within embedded SQL. Equivalent operations are supported through ODBC using the ODBC API. For further details, refer to the [Embedded SQL](#) chapter in *Using Caché SQL*.

An **INTO** clause can be specified as a clause of the **DECLARE** statement, as a clause of the **FETCH** statement, or both. The **INTO** clause allows data from the columns of a fetch to be placed into local [host variables](#). Each host variable in the list, from left to right, is associated with the corresponding column in the cursor result set. The data type of each variable must either match or be a supported implicit conversion of the data type of the corresponding result set column. The number of variables must match the number of columns in the cursor select list.

The **FETCH** operation completes when the cursor advances to the end of the data. This sets SQLCODE=100 (No more data). It also sets the [%ROWCOUNT](#) variable to the number of fetched rows.

Note: The values returned by **INTO** clause host variables are only reliable while SQLCODE=0. If SQLCODE=100 (No more data) the host variable values should not be used.

The *cursor-name* is not namespace-specific. Changing the current namespace has no effect on use of a declared cursor. The only namespace consideration is that **FETCH** must occur in the namespace that contains the table(s) being queried.

FETCH does not support the [#SQLCompile Mode=Deferred](#) preprocessor directive. Attempting to use Deferred mode with a **DECLARE**, **OPEN**, **FETCH**, or **CLOSE** cursor statement generates a #5663 compilation error.

%ROWID

When a **FETCH** retrieves a row of an updateable cursor, it sets [%ROWID](#) to the RowID value of the fetched row. An updateable cursor is one in which the top FROM clause contains exactly one element, either a table name or an updateable view name.

This setting of [%ROWID](#) for each row retrieved is subject to the following conditions:

- The **DECLARE cursorname CURSOR** and **OPEN cursorname** statements do not initialize [%ROWID](#); the [%ROWID](#) value is unchanged from its prior value. The first successful **FETCH** sets [%ROWID](#). Each subsequent **FETCH** that retrieves a row resets [%ROWID](#) to the current RowID. **FETCH** sets [%ROWID](#) if it retrieves a row of an updateable cursor. If the cursor is not updateable, [%ROWID](#) remains unchanged. If no rows matched the query selection criteria,

FETCH does not change the prior the %ROWID value. Upon **CLOSE** or when **FETCH** issues an SQLCODE 100 (No Data, or No More Data), %ROWID contains the RowID of the last row retrieved.

- A cursor-based **SELECT** with a **DISTINCT** keyword or a **GROUP BY** clause does not set %ROWID. The %ROWID value is unchanged from its previous value (if any).
- A cursor-based **SELECT** that performs only [aggregate operations](#) does not set %ROWID. The %ROWID value is unchanged from its previous value (if any).

An Embedded SQL **SELECT** with no declared cursor does not set %ROWID. The %ROWID value is unchanged upon the completion of a simple **SELECT** statement.

FETCH for UPDATE or DELETE

You can use **FETCH** to retrieve a row for update or delete. The **UPDATE** or **DELETE** must specify the **WHERE CURRENT OF** clause. The **DECLARE** should specify the **FOR UPDATE** clause. The following example shows a cursor-based delete that deletes all selected rows:

```
ZN "Samples"
&sql(DECLARE MyCursor CURSOR FOR SELECT %ID,Status
      FROM Sample.Quality WHERE Status='Bad' FOR UPDATE)
&sql(OPEN MyCursor)
      QUIT:(SQLCODE'=0)
NEW %ROWCOUNT,%ROWID
FOR {&sql(FETCH MyCursor) QUIT:SQLCODE'=0
     &sql(DELETE FROM Sample.Quality WHERE CURRENT OF MyCursor) }
WRITE !,"Number of rows updated=",%ROWCOUNT
&sql(CLOSE MyCursor)
```

Examples

The following Embedded SQL example shows **FETCH** invoked by an argumentless **FOR** loop retrieving data from a cursor named EmpCursor. The **INTO** clause is specified in the **DECLARE** statement:

```
&sql(DECLARE EmpCursor CURSOR FOR
      SELECT Name, Home_State
      INTO :name,:state FROM Sample.Employee
      WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
      QUIT:(SQLCODE'=0)
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
      QUIT:SQLCODE'=0
      WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
      WRITE " Name=",name," State=",state,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)
```

The following Embedded SQL example shows **FETCH** invoked by an argumentless **FOR** loop retrieving data from a cursor named EmpCursor. The **INTO** clause is specified as part of the **FETCH** statement:

```
&sql(DECLARE EmpCursor CURSOR FOR
      SELECT Name,Home_State FROM Sample.Employee
      WHERE Home_State %STARTSWITH 'M')
&sql(OPEN EmpCursor)
      QUIT:(SQLCODE'=0)
FOR { &sql(FETCH EmpCursor INTO :name,:state)
      QUIT:SQLCODE'=0
      WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
      WRITE " Name=",name," State=",state,! }
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)
```

The following Embedded SQL example shows **FETCH** invoked using a **WHILE** loop:

```

&sql(DECLARE C1 CURSOR FOR
      SELECT Name,Home_State INTO :name,:state FROM Sample.Person
      WHERE Home_State %STARTSWITH 'M')
&sql(OPEN C1)
      QUIT:(SQLCODE'=0)
&sql(FETCH C1)
      WHILE (SQLCODE = 0) {
          WRITE "count: ",%ROWCOUNT," RowID: ",%ROWID,!
          WRITE " Name=",name," State=",state,!
          &sql(FETCH C1) }
      WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE C1)

```

The following Embedded SQL example shows **FETCH** retrieving aggregate function values. %ROWID is not set:

```

&sql(DECLARE PersonCursor CURSOR FOR
      SELECT COUNT(*),AVG(Age) FROM Sample.Person )
&sql(OPEN PersonCursor)
      QUIT:(SQLCODE'=0)
      NEW %ROWCOUNT
      FOR { &sql(FETCH PersonCursor INTO :cnt,:avg)
          QUIT:SQLCODE'=0
          WRITE %ROWCOUNT," Num People=",cnt," Average Age=",avg,! }
      WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE PersonCursor)

```

The following Embedded SQL example shows **FETCH** retrieving DISTINCT values. %ROWID is not set:

```

&sql(DECLARE EmpCursor CURSOR FOR
      SELECT DISTINCT Home_State FROM Sample.Employee
      WHERE Home_State %STARTSWITH 'M'
      ORDER BY Home_State )
&sql(OPEN EmpCursor)
      QUIT:(SQLCODE'=0)
      NEW %ROWCOUNT
      FOR { &sql(FETCH EmpCursor INTO :state)
          QUIT:SQLCODE'=0
          WRITE %ROWCOUNT," State=",state,! }
      WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE EmpCursor)

```

The following Embedded SQL example shows that a cursor persists across namespaces. This cursor is declared in **SAMPLES**, opened in **DOCBOOK**, fetched in **SAMPLES**, and closed in **USER**. Note that the **FETCH** must be executed in the namespace that contains the table being queried, **Sample.Employee**:

```

&sql(USE DATABASE "USER")
      WRITE $ZNSPACE,!
&sql(DECLARE NSCursor CURSOR FOR SELECT Name INTO :name FROM Sample.Employee)
&sql(USE DATABASE DOCBOOK)
      WRITE $ZNSPACE,!
&sql(OPEN NSCursor)
      QUIT:(SQLCODE'=0)
&sql(USE DATABASE SAMPLES)
      WRITE $ZNSPACE,!
      NEW SQLCODE,%ROWCOUNT,%ROWID
      FOR { &sql(FETCH NSCursor)
          QUIT:SQLCODE
          WRITE "Name=",name,! }
&sql(USE DATABASE "USER")
      WRITE $ZNSPACE,!
&sql(CLOSE NSCursor)
      WRITE "Close SQLCODE: ",SQLCODE,!

```

See Also

- [CLOSE, DECLARE, OPEN](#)
- [SQL Cursors](#) in the “Using Embedded SQL” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

FROM

A **SELECT** clause that specifies one or more tables to query.

```
SELECT ... FROM [optimize-option] table-ref [[AS] t-alias][,table-ref [[AS]
t-alias]][,...]
```

Arguments

<i>optimize-option</i>	<i>Optional</i> — A single keyword, or a series of keywords separated by spaces, that specify query optimization options (optimizer hints). The following keywords are supported: %ALLINDEX , %FIRSTTABLE <i>tablename</i> , %FULL , %IGNOREINDEX <i>name</i> , %INORDER , %NOFLATTEN , %NOMERGE , %NOREDUCE , %NOSVSO , %NOTOPOPT , %NOUNIONOROPT , %PARALLEL , and %STARTTABLE .
<i>table-ref</i>	One or more tables , views , table-valued functions , or subqueries from which data is being retrieved, specified as a comma-separated list or with JOIN syntax. Some restrictions apply on using views with JOIN syntax. You can specify a subquery, enclosed in parentheses.
AS <i>t-alias</i>	<i>Optional</i> — An alias for the table name. Must be a valid identifier . For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> . Can be specified with or without the optional AS keyword.

Description

The FROM clause specifies one or more tables (or views, or subqueries) from which data is queried within a **SELECT** statement. If no table data is being queried, the FROM clause is optional, as described below.

Multiple tables are specified as a comma-separated list, or a list separated by other JOIN syntax. Each table name can optionally be supplied an alias.

Table name aliases are used when specifying field names for multiple tables in the **SELECT** statement. If two (or more) tables are specified in the FROM clause, you indicate which table’s field you want by specifying `tablename.fieldname` for each field in the **SELECT** *select-item* clause. Because table names are often long names, a short table name alias is useful in this context (`t-alias.fieldname`).

The following example show the use of table name aliases:

```
SELECT e.Name,c.Name
FROM Sample.Company AS c,Sample.Employee AS e
```

The AS keyword can be omitted. It is provided for compatibility and clarity.

Supplying a Schema Name to a Table Reference

A *table-ref* name is either qualified (`schema.tablename`) or unqualified (`tablename`). The schema name for an unqualified table name (or view name) is supplied using a schema search path or the system-wide default schema name:

1. If a [schema search path](#) is provided, Caché searches the specified schemas for a matching table name.
2. If a schema search path is not provided, or the schema search path does not produce a match, the [system-wide default schema name](#) is used.

Table Joins

When you specify multiple table names in a FROM clause, Caché SQL performs join operations on those tables. The type of join performed is specified by a join keyword phrase or symbol between each pair of table names. When two table names are separated by a comma, a cross join is performed. For further details on the different types of joins and their syntax, refer to [JOIN](#).

The sequence in which joins are performed is automatically determined by the SQL query optimizer and is not based on the sequence that the tables are listed in the query. If desired, you can control the sequence in which joins are performed by specifying a query optimization option.

The following three **SELECT** statements show the row counts for two individual tables, and the row count for a **SELECT** specifying both tables. This latter results in a much larger table, a Cartesian product, where every row in the first table is matched with every row of the second table, an operation known as a [Cross Join](#).

```
SELECT COUNT(*)
FROM Sample.Company
```

```
SELECT COUNT(*)
FROM Sample.Vendor
```

```
SELECT COUNT(*)
FROM Sample.Company, Sample.Vendor
```

You can perform the same operation using explicit **CROSS JOIN** syntax:

```
SELECT COUNT(*)
FROM Sample.Company CROSS JOIN Sample.Vendor
```

In most cases, the extensive data duplication of a cross join is not desirable, and some other type of join is preferable.

If you specify a [WHERE clause](#) in the **SELECT** statement, the cross join is performed, then the WHERE clause predicate(s) determine the result set. This is equivalent to performing an **INNER JOIN** with an **ON** clause. Thus the following two examples return identical results:

```
SELECT p.Name, p.Home_State, em.Name, em.Office_State
FROM Sample.Person AS p, Sample.Employee AS em
WHERE p.Name %STARTSWITH 'E' AND em.Name %STARTSWITH 'E'
```

```
SELECT p.Name, p.Home_State, em.Name, em.Office_State
FROM Sample.Person AS p INNER JOIN Sample.Employee AS em
ON p.Name %STARTSWITH 'E' AND em.Name %STARTSWITH 'E'
```

You can specify explicit join syntax (rather than using commas) in the FROM *table-ref* list to perform other types of join operations. For further details, refer to [JOIN](#).

Query Optimization Options

By default, the Caché SQL query optimizer uses sophisticated and flexible algorithms to optimize the performance of complex queries involving join operations and/or multiple indexes. In most cases, these defaults provide optimal performance. However, in infrequent cases, you may wish to give “hints” to the query optimizer, specifying one or more aspects of query optimization. For this reason, Caché SQL provides *optimize-option* keywords in the FROM clause. You can specify multiple optimization keywords in any order, separated by blank spaces. For further details, refer to [Optimizing SQL Queries](#) in the *Caché SQL Optimization Guide*.

You can use *optimize-option* FROM clause keywords in a simple **SELECT** statement, in a **CREATE VIEW** view definition **SELECT** statement, or in a subquery **SELECT** statement within the FROM clause.

Some query optimization options can be configured system-wide from the Management Portal: [%ALLINDEX](#), [%NOFLATTEN](#), [%NOMERGE](#), [%NOSVSO](#), [%NOTOPOPT](#), and [%NOUNIONOROPT](#). From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **General SQL Settings**. The **Optimization** tab displays these options.

By default, none of these system-wide options are set; they should only be set with extreme caution in rare and specific circumstances. To determine the current system-wide settings, call `$$SYSTEM.SQL.CurrentSettings()`.

%ALLINDEX

This optional keyword specifies that all indexes that provide any benefit are used for the first table in the query join order. This keyword should only be used when there are multiple defined indexes. The optimizer default is to use only those indexes that the optimizer judges to be most beneficial. By default, this includes all efficient equality indexes, and selected indexes of other types. `%ALLINDEX` uses all possibly beneficial indexes of all types. Testing all indexes has a larger overhead, but under some circumstances it may provide better performance than the default optimization. This option is especially helpful when using multiple range condition indexes and inefficient equality condition indexes. In these circumstances, accurate index selectivity may not be available to the query optimizer. `%ALLINDEX` can be used with `%IGNOREINDEX` to include/exclude specific indexes. Generally, `%ALLINDEX` should not be used with a `TOP` clause query.

You can use `%STARTTABLE` with `%ALLINDEX` to specify which table the `%ALLINDEX` applies to.

You can specify exceptions to `%ALLINDEX` for specific conditions with the `%NOINDEX` condition-level hint. The `%NOINDEX` hint is placed in front of each query selection [condition](#) for which no index should be used. For example, `WHERE %NOINDEX hiredate < ?`. This is most commonly used when the overwhelming majority of the data is not excluded by the condition. With a less-than (<) or greater-than (>) condition, use of the `%NOINDEX` condition-level hint is often beneficial. With an equality condition, use of the `%NOINDEX` condition-level hint provides no benefit. With a [join condition](#), `%NOINDEX` is not supported for `=*` and `*=` `WHERE` clause outer joins; `%NOINDEX` is supported for `ON` clause joins. For further details, refer to “[Using Indices](#)” in the “[Optimizing Query Performance](#)” chapter in *Caché SQL Optimization Guide*.

%FIRSTTABLE

```
%FIRSTTABLE tablename
```

This optional keyword specifies that the query optimizer should start to performs joins with the specified *tablename*. The *tablename* names a table that is specified later in the join sequence. The join order for the remaining tables is left to the query optimizer. This hint is functionally identical to `%STARTTABLE`, but provides you with the flexibility to specify the join table sequence in any order.

The *tablename* must be a simple identifier, either a table alias or an unqualified table name. A qualified table name (schema.table) cannot be used. If the query specifies a table alias, the table alias must be used as *tablename*. For example:

```
FROM %FIRSTTABLE P Sample.Employee AS E JOIN Sample.Person AS P ON E.Name = P.Name
```

`%FIRSTTABLE` and `%STARTTABLE` both enable you to specify the initial table to use for join operations. `%INORDER` enables you to specify the order of all tables used for join operations. These three keywords are mutually exclusive; specify one and one only. If these keywords are not used the query optimizer performs joins on tables in the sequence it considers optimal, regardless of the sequence in which the tables are listed.

You cannot use `%FIRSTTABLE` or `%STARTTABLE` to begin the join order with the right-hand side of a `LEFT OUTER JOIN` (or the left-hand side of a `RIGHT OUTER JOIN`). Attempting to do so results in an `SQLCODE -34` error: “Optimizer failed to find a usable join order”.

For further details, refer to the `%STARTTABLE` query optimization option.

%FULL

This optional keyword specifies that the compiler optimizer examines all alternative join sequences to maximize access performance. For example, when creating a stored procedure, the increased compile time may be worthwhile to provide for more optimized access. The default optimization is to not examine less likely join sequences when there are many tables in the `FROM` clause. `%FULL` overrides this default behavior.

You might specify both the `%INORDER` and the `%FULL` keywords when the `FROM` clause includes tables accessed with [arrow syntax](#), which lead to tables whose order is unconstrained.

%IGNOREINDEX

This optional keyword specifies that the query optimizer ignore the specified index or list of indices. (The deprecated synonym `%IGNOREINDICES` is supported for backwards compatibility.)

Following this keyword you specify one or more index names. Multiple index names must be separated by commas. You can specify an index name using either of the following formats:

```
%IGNOREINDEX [[schemaname.]tablename.]indexname [,...] %IGNOREINDEX
[[schemaname.]tablename.]* [,...]
```

The *schemaname* and *tablename* are optional. If omitted, the current default schema and the table name specified as `FROM table-ref` are used. The asterisk (*) wildcard specifies all of the index names for the specified table. You can specify index names in any order. Caché SQL does not validate the index names you specify (or their *schemaname* and *tablename*); a nonexistent or duplicate index name is simply ignored.

By using this optimization constraint, you can cause the query optimizer to not use an index that is not optimal for a specific query. By specifying all index names but one, you can, in effect, force the query optimizer to use the remaining index.

You can also ignore a specific index for a specific condition expression by prefacing the condition with the `%NOINDEX` keyword. For further details, refer to “[Using Indices](#)” in the “[Optimizing Query Performance](#)” chapter in the *Caché SQL Optimization Guide*.

%INORDER

This optional keyword specifies that the query optimizer performs joins in the order that the tables are listed in the `FROM` clause. This minimizes compile time. The join order of tables referenced with arrow syntax is unrestricted (for information on using arrow syntax, refer to [Implicit Joins](#) in *Using Caché SQL*). Flattening of subqueries and index usage are unaffected.

`%INORDER` cannot be used with a `CROSS JOIN` or a `RIGHT OUTER JOIN`. If the table order specified is inconsistent with the requirements of an outer join, an `SQLCODE -34` error is generated: “Optimizer failed to find a usable join order.” To avoid this, it is recommended that `%INORDER`, when used with outer joins, only be used with ANSI-style left outer joins or full outer joins.

Views and table subqueries are processed in the order that they are specified in the `FROM` clause.

- Streamed View: `%INORDER` has no effect on the order of processing of tables within the view.
- Merged View: `%INORDER` causes the view tables to be processed in the view’s `FROM` clause order, at the point of reference to the view.

Compare this keyword with `%FIRSTTABLE` and `%STARTTABLE`, both of which specify only the initial join table, rather than the full join order. See `%STARTTABLE` for a table of merge behaviors with different join order optimizations.

The `%INORDER` and `%PARALLEL` optimizations cannot be used together; if both are specified, `%PARALLEL` is ignored.

%NOFLATTEN

This optional keyword is specified in the `FROM` clause of a quantified subquery — a subquery that returns a boolean value. It specifies that the compiler optimizer should inhibit subquery flattening. This optimization option disables “flattening” (the default), which optimizes a query containing a quantified subquery by effectively integrating the subquery into the query: adding the tables of the subquery to the `FROM` clause of the query and converting conditions in the subquery to joins or restrictions in the query’s `WHERE` clause.

The following are examples of quantified subqueries using `%NOFLATTEN`:

```
SELECT Name,Home_Zip FROM Sample.Person WHERE Home_Zip IN
      (SELECT Office_Zip FROM %NOFLATTEN Sample.Employee)
```

```
SELECT Name, (SELECT Name FROM Sample.Company WHERE EXISTS
              (SELECT * FROM %NOFLATTEN Sample.Company WHERE Revenue > 50000000))
FROM Sample.Person
```

The %INORDER and %STARTTABLE optimizations implicitly specify %NOFLATTEN.

%NOMERGE

This optional keyword is specified in the FROM clause of a subquery. It specifies that the compiler optimizer should inhibit the conversion of a subquery to a view. This optimization option disables the optimizing of a query containing a subquery by adding the subquery to the FROM clause of the query as an in-line view; comparisons from the subquery to fields of the query are moved to the query's WHERE clause as joins.

%NOREDUCE

This optional keyword is specified in the FROM clause of a streamed subquery — a subquery that returns a result set of rows, a [subquery in the enclosing query's FROM clause](#). It specifies that the compiler optimizer should inhibit the merging of the subquery (or view) into the containing query.

In the following example, the query optimizer would normally “reduce” this query by performing a Cartesian product join of Sample.Person with the subquery. The %NOREDUCE optimization option prevents this. Caché instead builds a temporary index on gname and performs the join on this temporary index:

```
SELECT * FROM Sample.Person AS p,
        (SELECT Name||'goo' AS gname FROM %NOREDUCE Sample.Employee) AS e
WHERE p.name||'goo' = e.gname
```

%NOSVSO

This optional keyword is specified in the FROM clause of a quantified subquery — a subquery that returns a boolean value. It specifies that the compiler optimizer should inhibit Set-Valued Subquery Optimization (SVSO).

In most cases, Set-Valued Subquery Optimization improves the performance of [\[NOT\] EXISTS](#) and [\[NOT\] IN](#) subqueries, especially with subqueries with only one, separable correlating condition. It does this by populating a temporary index with the data values that fulfill the condition. Rather than repeatedly executing the subquery, Caché looks up these values in the temporary index. For example, SVSO optimizes NOT EXISTS (SELECT P.num FROM Products P WHERE S.num=P.num AND P.color='Pink') by creating a temporary index for P.num.

SVSO optimizes subqueries where the [ALL](#) or [ANY](#) keyword is used with a relative operator (>, >=, <, or <=) and a subquery, such as ...WHERE S.num > ALL (SELECT P.num ...). It does this by replacing the subquery expression *sqbExpr* (P.num in this example) with MIN(*sqbExpr*) or MAX(*sqbExpr*), as appropriate. This supports fast computation when there is an index on *sqbExpr*.

The %INORDER and %STARTTABLE optimizations do not inhibit Set-Valued Subquery Optimization.

%NOTOPOPT

This optional keyword is specified when using a [TOP](#) clause with an [ORDER BY](#) clause. By default, TOP with ORDER BY optimizes for fastest time-to-first-row. Specifying %NOTOPOPT (no TOP optimization) instead optimizes the query for fastest retrieval of the complete result set.

%NOUNIONOROPT

This optional keyword is specified in the FROM clause of a query or subquery. It disables the automatic optimizations provided for multiple OR conditions and for subqueries against a [UNION](#) query expression. These automatic optimizations transform multiple OR conditions to UNION subqueries, or UNION subqueries to OR conditions, where deemed appropriate. These UNION/OR transformations allow EXISTS and other low-level predicates to migrate to top-level conditions where they are available to Caché query optimizer indexing. These default transformations are desirable in most situations.

However, in some situations these UNION/OR transformations impose a significant overhead burden. %NOUNIONOROPT disables these automatic UNION/OR transformations for all conditions in the WHERE clause associated with this FROM

clause. Thus, in a complex query, you can disable these automatic UNION/OR optimizations for one subquery while allowing them in other subqueries.

The **UNION %PARALLEL** keyword disables automatic UNION-to-OR optimizations.

The %INORDER and %STARTTABLE optimizations inhibit OR-to-UNION optimizations. The %INORDER and %STARTTABLE optimizations do not inhibit UNION-to-OR optimizations.

%PARALLEL

This optional keyword is specified in the FROM clause of a query. It suggests that Caché perform parallel processing of the query, using multiple processors (if applicable). This can significantly improve performance of some queries that uses one or more **COUNT**, **SUM**, **AVG**, **MAX**, or **MIN** aggregate functions, and/or a **GROUP BY** clause, as well as many other types of queries. These are commonly queries that process a large quantity of data and return a small result set. For example, `SELECT AVG(SaleAmt) FROM %PARALLEL User.AllSales GROUP BY Region` would likely use parallel processing.

A query that specifies both individual fields and an aggregate function and does not include a **GROUP BY** clause cannot perform parallel processing. For example, `SELECT Name,AVG(Age) FROM %PARALLEL Sample.Person` does not perform parallel processing, but `SELECT Name,AVG(Age) FROM %PARALLEL Sample.Person GROUP BY Home_State` does perform parallel processing.

%PARALLEL is intended for **SELECT** queries and their subqueries. An **INSERT** command subquery cannot use %PARALLEL.

Specifying %PARALLEL may degrade performance for some queries. Running a query with %PARALLEL on a system with multiple concurrent users may result in degraded overall performance.

Note: A query that specifies %PARALLEL must be run in a database that is read/write, not readonly. Otherwise, a <PROTECT> error may occur.

Regardless of the presence of the %PARALLEL keyword in the FROM clause, some queries may use linear processing, not parallel processing: some queries do not support parallel processing; some queries, when optimized, may be found to not benefit from parallel processing. You can determine if and how Caché has partitioned a query for parallel processing using [Show Plan](#).

For further details, refer to [Parallel Query Processing](#) in the “Optimizing Query Performance” chapter of the *Caché SQL Optimization Guide*.

%STARTTABLE

This optional keyword specifies that the query optimizer should start to performs joins with the first table listed in the FROM clause. The join order for the remaining tables is left to the query optimizer. Compare this keyword with %INORDER, which specifies the complete join order.

%STARTTABLE cannot be used with a **CROSS JOIN** or a **RIGHT OUTER JOIN**. You cannot use %STARTTABLE (or %FIRSTTABLE) to begin the join order with the right-hand side of a **LEFT OUTER JOIN** (or the left-hand side of a **RIGHT OUTER JOIN**). If the start table specified is inconsistent with the requirements of an outer join, an SQLCODE - 34 error is generated: “Optimizer failed to find a usable join order.” To avoid this, it is recommended that %STARTTABLE, when used with outer joins, only be used with ANSI-style left outer joins or full outer joins.

The following table shows the merge behavior when combining a superquery parent and an in-line view with %INORDER and %STARTTABLE optimizations:

	<i>Superquery with no join optimizer</i>	<i>Superquery with %STARTTABLE</i>	<i>Superquery with %INORDER</i>
<i>View with no join optimizer</i>	merge view if possible	If the view is the superquery start: don't merge. Otherwise, merge view if possible.	merge if possible; view's underlying tables are unordered.
<i>View with %STARTTABLE</i>	don't merge	If the view is the superquery start: merge, if possible. View's start table becomes superquery's start table. Otherwise, don't merge.	don't merge
<i>View with %INORDER</i>	don't merge	don't merge	If the view is not controlled by the %INORDER: don't merge. Otherwise, merge view if possible; view's order becomes substituted into superquery join order.

The %FIRSTTABLE hint is functionally identical to %STARTTABLE, but provides you with the flexibility to specify the join table sequence in any order.

Table-Valued Functions in the FROM Clause

A table-valued function is a class query that is projected as a stored procedure and returns a single result set. A table-valued function is any class query which has `SqlProc TRUE`. A class query used as a table-valued function must be compiled in either LOGICAL or RUNTIME mode. When used as a table-valued function and compiled in RUNTIME mode, the table-valued function query will be called in LOGICAL mode.

A table-valued function follows the same naming conventions as a stored procedure name for a class query. Parameter parentheses are mandatory; the parentheses may be empty, enclose a literal or a host variable, or a comma-separated list of literals and host variables. If you specify no parameters (empty parentheses or the null string), the table-valued function returns all data rows.

To issue a query using a table-valued function, the user must hold the EXECUTE privilege on the stored procedure that defines the table-valued function. The user must also have SELECT privileges on the tables or views accessed by the table-valued function query.

In the following example, the class query `Sample.Person.ByName` is projected as a stored procedure and can thus be used as a table-valued function:

```
SELECT Name, DOB FROM Sample.SP_Sample_By_Name('A')
```

The following Dynamic SQL example specifies the same table-valued function. It uses the `%Execute()` method to supply parameter values to the ? input parameter:

```

ZNSPACE "SAMPLES"
SET myquery="SELECT Name,DOB FROM Sample.SP_Sample_By_Name(?)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("A")
DO rset.%Display()
WRITE !,"End of A data",!!
SET rset = tStatement.%Execute("B")
DO rset.%Display()
WRITE !,"End of B data"

```

A table-valued function can only be used in the FROM clause of either a **SELECT** statement or a **DECLARE** statement. A table-valued function name can be qualified with a schema name or unqualified (without a schema name); an unqualified name uses the default schema. In a **SELECT** statement FROM clause, a table-valued function can be used wherever a table name can be used. It can be used in a view or a subquery, and can be joined to other *table-ref* items using a comma-separated list or explicit **JOIN** syntax.

A table-valued function cannot be directly used in an **INSERT**, **UPDATE**, or **DELETE** statement. You can, however, specify a subquery for these commands that specifies a table-valued function.

Caché SQL does not define the **EXTENTS** or **SELECTIVITY** for table-valued function columns.

Subqueries in the FROM Clause

You can specify a subquery in the FROM clause. This is known as a streamed subquery. The subquery is treated the same as a table, including its use in JOIN syntax and the optional assignment of an alias using the AS keyword. A FROM clause can contain multiple tables, views, and subqueries in any combination, subject to the restrictions of the JOIN syntax, as described in [JOIN](#).

A subquery is enclosed in parentheses. The following example shows a subquery in a FROM clause:

```

SELECT name,region
FROM (SELECT t1.name,t1.state,t2.region
      FROM Employees AS t1 LEFT OUTER JOIN Regions AS t2
      ON t1.state=t2.state)
GROUP BY region

```

A subquery can specify a **TOP clause**. A subquery can contain an **ORDER BY clause** when paired with a TOP clause.

A subquery can use **SELECT *** syntax, subject to the following restriction: because a FROM clause results in a value expression, a subquery containing **SELECT *** must yield only one column.

A join within a subquery cannot be a **NATURAL** join or take a **USING** clause.

FROM Subqueries and %VID

When a FROM subquery is invoked, it returns a **%VID** for each subquery row returned. A **%VID** is an integer counter field; its values are system-assigned, unique, non-null, non-zero, and non-modifiable. The **%VID** is only returned when explicitly specified. It is returned as data type **INTEGER**. Because **%VID** values are sequential integers, they are far more meaningful if the subquery returns ordered data; a subquery can only use an **ORDER BY** clause when it is paired with a **TOP** clause.

Because the **%VID** is a sequential integer, it can be used to determine the ranking of items in a subquery with an **ORDER BY** clause. In the following example, the 10 newest records are listed in Name order, but their timestamp ranking is easily seen using the **%VID** values:

```

SELECT Name,%VID,TimeStamp FROM
  (SELECT TOP 10 * FROM MyTable ORDER BY TimeStamp DESC)
ORDER BY Name

```

One common use of the **%VID** is to “window” the result set, dividing execution into sequential subsets that fit the number of lines available in a display window. For example, display 20 records, then wait for the user to press Enter, then display the next 20 records.

The following example uses %VID to “window” the results into subsets of 10 records:

```
ZNSPACE "SAMPLES"
SET myq=4
SET myq(1)="SELECT %VID,* "
SET myq(2)="FROM (SELECT TOP 60 Name,Age FROM Sample.Person "
SET myq(3)="WHERE Age > 55 ORDER BY Name) "
SET myq(4)="WHERE %VID BETWEEN ? AND ?"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myq)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
FOR i=1:10:60 {
SET rset = tStatement.%Execute(i,i+9)
WHILE rset.%Next() {
DO rset.%Print() }
WRITE !!
}
WRITE "End of data"
```

For details on using %VID, refer to the [Defining and Using Views](#) chapter of *Using Caché SQL*.

Optional FROM Clause

If no table data is referenced (directly or indirectly) by the **SELECT** item list, the FROM clause is optional. This kind of **SELECT** may be used to return data from functions, operator expressions, constants, or host variables. For a query that references no table data:

- If the FROM clause is omitted, a maximum of one row of data is returned, regardless of the TOP keyword value; TOP 0 returns no data. The **DISTINCT** clause is ignored. No privileges are required.
- If the FROM clause is specified, it must specify an existing table in the current namespace. You must have SELECT privilege for that table, even though the table is not referenced. The number of identical rows of data returned is equal to the number of rows in the specified table, unless you specify a TOP or DISTINCT clause, or limit it with a WHERE or HAVING clause. Specifying a **DISTINCT** clause limits the output to a single row of data. The TOP keyword limits the output to the number of rows specified by the TOP value; TOP 0 returns no data.

With or without a FROM clause, subsequent clauses (WHERE, GROUP BY, HAVING or ORDER BY) may be specified. A WHERE or HAVING clause may be used to determine whether or not to return results, or how many identical rows of results to return. These clauses may reference a table, even if no FROM clause is specified. A GROUP BY or ORDER BY clause may be specified, but these clauses are not meaningful.

The following are examples of **SELECT** statements that reference no table data. Both examples return one row of information.

The following example omits the FROM clause. The **DISTINCT** keyword is not needed, but may be specified. No **SELECT** clauses are permitted.

```
SELECT 3+4 AS Arith,
       {fn NOW} AS NowDateTime,
       {fn DAYNAME({fn NOW})} AS NowDayName,
       UPPER('MixEd cAsE EXPreSSion') AS UpCase,
       {fn PI} AS PiConstant
```

The following example includes a FROM clause. The **DISTINCT** keyword is used to return a single row of data. The FROM clause table reference must be a valid table. The **ORDER BY** clause is permitted here, but is meaningless. Note that the **ORDER BY** clause must specify a valid select item alias:

```
SELECT DISTINCT 3+4 AS Arith,
               {fn NOW} AS NowDateTime,
               {fn DAYNAME({fn NOW})} AS NowDayName,
               UPPER('MixEd cAsE EXPreSSion') AS UpCase,
               {fn PI} AS PiConstant
FROM Sample.Person
ORDER BY NowDateTime
```

The following examples both use a WHERE clause to determine whether or not to return results. The first includes a FROM clause and uses the **DISTINCT** keyword is to return a single row of data. The second omits the FROM clause, and therefore

returns at most a single row of data. In both cases, the WHERE clause table reference must be a valid table for which you have SELECT privilege:

```
SELECT DISTINCT
    {fn NOW} AS DataOKDate
FROM Sample.Person
WHERE FOR SOME (Sample.Person)(Name %STARTSWITH 'A')
```

```
SELECT {fn NOW} AS DataOKDate
WHERE FOR SOME (Sample.Person)(Name %STARTSWITH 'A')
```

See Also

- [SELECT](#)
- [JOIN](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*
- “[Defining Tables](#)” chapter in *Using Caché SQL*
- “[Optimizing SQL Queries](#)” in the *Caché SQL Optimization Guide*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

GRANT

Grants privileges to a user or role.

```
GRANT admin-privilege TO grantee [WITH ADMIN OPTION]
GRANT role TO grantee [WITH ADMIN OPTION]

GRANT object-privilege ON object-list TO grantee [WITH GRANT OPTION]
GRANT SELECT ON CUBE[S] object-list TO grantee [WITH GRANT OPTION]
GRANT column-privilege (column-list) ON table TO grantee [WITH GRANT OPTION]
```

Arguments

<i>grantee</i>	A comma-separated list of one or more users or roles. Valid values are a list of users, a list of roles, "*", or _PUBLIC. The asterisk (*) specifies all currently defined users who do not have the %All role. The _PUBLIC keyword specifies all currently defined and yet-to-be-defined users.
<i>admin-privilege</i>	An administrative-level privilege or a comma-separated list of administrative-level privileges being granted. The list may consist of one or more of the following in any order: %CREATE_METHOD, %DROP_METHOD, %CREATE_FUNCTION, %DROP_FUNCTION, %CREATE_PROCEDURE, %DROP_PROCEDURE, %CREATE_QUERY, %DROP_QUERY, %CREATE_TABLE, %ALTER_TABLE, %DROP_TABLE, %CREATE_VIEW, %ALTER_VIEW, %DROP_VIEW, %CREATE_TRIGGER, %DROP_TRIGGER %DB_OBJECT_DEFINITION, which grants all 16 of the above privileges. %NOCHECK, %NOINDEX, %NOLOCK, %NOTRIGGER privileges for INSERT, UPDATE, and DELETE operations.
<i>role</i>	A role or comma-separated list of roles whose privileges are being granted.
<i>object-privilege</i>	A basic-level privilege or comma-separated list of basic-level privileges being granted. The list may consist of one or more of the following: %ALTER, %DELETE, %SELECT, %INSERT, %UPDATE, %EXECUTE, and %REFERENCES. You can confer all table and view privileges using either "ALL [PRIVILEGES]" or "*" as the argument value. Note that you can only grant SELECT privilege to CUBES.
<i>object-list</i>	A comma-separated list of one or more tables , views , stored procedures, or cubes for which the <i>object-privilege</i> (s) are being granted. You can use the SCHEMA keyword to specify granting the <i>object-privilege</i> to all objects in the specified schema. You can use "*" to specify granting the <i>object-privilege</i> to all tables, or to all non-hidden Stored Procedures, in the current namespace. Note that a cubes <i>object-list</i> requires the CUBE (or CUBES) keyword, and can only be granted SELECT privilege.
<i>column-privilege</i>	A basic-level privilege being granted to one or more listed columns. Available options are SELECT, INSERT, UPDATE, and REFERENCES.
<i>column-list</i>	A list of one or more column names, separated by commas and enclosed in parentheses.
<i>table</i>	The name of the table or view that contains the <i>column-list</i> columns.

Description

The **GRANT** command gives privileges to do specified tasks on specified tables, views, columns, or other entities to one or more specified users or roles. You can do the following basic operations:

- Grant a privilege to a user.
- Grant a privilege to a role.
- Grant a role to a user.
- Grant a role to a role, creating a hierarchy of roles.

If you grant a privilege to a user, the user can immediately exercise the privilege. If you grant a privilege to a role, users who have been granted the role can immediately exercise the privilege. If you revoke a privilege, the user immediately loses the privilege. A privilege is effectively granted to a user only once. Multiple users can grant the same privilege to a user multiple times, but a single **REVOKE** removes the privilege.

Privileges are granted on a per-namespace basis.

SQL privileges are only enforced through ODBC, JDBC, and Dynamic SQL (`%SQL.Statement` and `%Library.ResultSet`).

Because **GRANT** prepares and executes quickly, and is generally run only once, Caché does not create a cached query for **GRANT** in ODBC, JDBC, or Dynamic SQL. The expansion of * is performed when the **GRANT** command is executed.

GRANT admin-privilege

SQL administrative (admin) privileges apply to users or roles. Any privilege that is not tied to any particular object (and thus is a general right for that user or role) is considered an admin privilege. These privileges are granted on a per-namespace basis for the current namespace.

The `%DB_OBJECT_DEFINITION` privilege grants all 16 of the data definition privileges. It does not grant `%NOCHECK`, `%NOINDEX`, `%NOLOCK`, and `%NOTRIGGER` privileges, which must be granted explicitly.

The `%NOCHECK`, `%NOINDEX`, `%NOLOCK`, and `%NOTRIGGER` privileges grant use of these options in the *restriction* clause of an `INSERT`, `UPDATE`, `INSERT OR UPDATE`, or `DELETE` statement. They have no effect on the use of the `%NOINDEX` keyword as a preface to a predicate condition. Because `TRUNCATE TABLE` performs a delete of all of the rows from a table with `%NOTRIGGER` behavior, you must have `%NOTRIGGER` privilege in order to run **TRUNCATE TABLE**. You must have the appropriate `%NOCHECK`, `%NOINDEX`, `%NOLOCK`, or `%NOTRIGGER` privilege to use that *restriction* when preparing an `INSERT`, `UPDATE`, `INSERT OR UPDATE`, or `DELETE` statement.

If the specified admin privilege is not a valid privilege name (for example, due to a spelling error), Caché completes successfully, issuing an SQLCODE 100 (reached end of data); Caché does not check if the specified user (or role) exists. If the specified admin privilege is valid, but the specified user (or role) does not exist, Caché issues an SQLCODE -118 error.

GRANT role

This form of **GRANT** assigns a user to a specified role. You can also assign a role to another role. If the specified role that receives the assignment does not exist, Caché issues an SQLCODE 100 (reached end of data). If the specified user (or role) that is assigned to a role does not exist, Caché issues an SQLCODE -118 error. If you are not the SuperUser, and you are attempting to grant a role that you don't own and don't have ADMIN OPTION for, Caché issues an SQLCODE -112 error.

Roles are created using the `CREATE ROLE` statement. If the role name is a *delimited identifier*, you must enclose it in quotation marks when assigning to it.

Roles can be granted or revoked via either the SQL **GRANT** and **REVOKE** commands, or via Caché System Security:

- Go to the Management Portal, select **[System] > [Security Management] > [Users]**, select **Edit** for the desired user, then select the **Roles** tab to assign (or unassign) the user to one or more roles.

- Go to the Management Portal, select **[System] > [Security Management] > [Roles]**, select **Edit** for the desired role, then select the **Assigned To** tab to assign (or unassign) the role to one or more roles. Note that the ObjectScript `$ROLES` special variable does not display roles granted to roles.

GRANT object-privilege

Object privileges give a user or role some right to a particular object. You grant an *object-privilege* ON an *object-list* TO a *grantee*. An *object-list* can specify one or more tables, views, stored procedures, or cubes in the current namespace. By using comma-separated lists, a single **GRANT** statement can grant multiple object privileges on multiple objects to multiple users and/or roles.

The following are the available *object-privilege* values:

- The `%ALTER` and `DELETE` privileges grant access to table or view definitions.
- The `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `REFERENCES` privileges grant access to table data.
- The `EXECUTE` privilege grants access to stored procedures. This privilege is required to execute a stored procedure or to call a user-defined SQL function in a query. For example, `SELECT Field1, MyFunc() FROM SQLUser.MyTable` requires `SELECT` privilege on `SQLUser.MyTable` and `EXECUTE` privilege on the `SQLUser.MyFunc` procedure.
- The `ALL PRIVILEGES` privilege grants all table and view privileges; it does not grant the `EXECUTE` privilege.

You can use the asterisk (*) wildcard as the *object-list* value to grant the *object-privilege* to all of the objects in the current namespace. For example, `GRANT SELECT ON * TO Deborah` grants this user `SELECT` privilege for all tables and views. `GRANT EXECUTE ON * TO Deborah` grants this user `EXECUTE` privilege for all non-hidden Stored Procedures.

You can use `SCHEMA schema-name` as the *object-list* value to grant the *object-privilege* to all of the tables, views, and stored procedures in the named schema, in the current namespace. For example, `GRANT SELECT ON SCHEMA Sample TO Deborah` grants this user `SELECT` privilege for all objects in the `Sample` schema. This includes all objects that will be defined in this schema in the future. You can specify multiple schemas as a comma-separated list; for example, `GRANT SELECT ON SCHEMA Sample, Cinema TO Deborah` grants `SELECT` privilege for all objects in both the `Sample` and the `Cinema` schemas.

Cubes are SQL identifiers that are not qualified by a schema name. To specify a cubes *object-list*, you must specify the `CUBE` (or `CUBES`) keyword. You can only grant `SELECT` privilege to a cube.

The following example demonstrates the granting of the `SELECT` and `UPDATE` privileges to a specific user for a specific table:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
CreateUser
SET x=$SYSTEM.SQL.UserExists("DeborahTest")
IF x=0 {&sql(CREATE USER DeborahTest IDENTIFY BY birdpw)
      IF SQLCODE != 0 {WRITE "CREATE USER error: ", SQLCODE, !
                     QUIT}
      }
ELSE {WRITE "User DeborahTest exists, not changing privileges", !
      QUIT }
GrantPrivsToUser
&sql(GRANT SELECT, UPDATE ON SQLUSER.T1 TO DeborahTest)
WRITE !, "GRANT error code: ", SQLCODE
DropUser
&sql(DROP USER DeborahTest)
IF SQLCODE != 0 {WRITE "DROP USER error: ", SQLCODE, !}
```

Privileges can only be granted explicitly to a table, view, or stored procedure that already exists. If the specified object does not exist, Caché issues an `SQLCODE -30` error. You can, however, grant privileges to a schema, which grant privileges both to all existing objects in that schema and to all future objects in that schema that did not exist when the privilege was granted.

If the owner of a table is `_PUBLIC`, users do not need to be granted object privileges to access the table.

If the specified user does not exist, Caché issues an SQLCODE -118 error. If the specified object privilege has already been granted, Caché issues an SQLCODE 100 (reached end of data).

Object privileges can be granted or revoked by any of the following:

- The **GRANT** and **REVOKE** commands.
- The %SYSTEM.SQL **GrantObjPriv()** and **RevokeObjPriv()** methods. These methods issue an SQLCODE -118 if any of the provided users or roles are invalid.
- Via Caché System Security. Go to the Management Portal, select **[System] > [Security Management] > [Users]** (or **[System] > [Security Management] > [Roles]**) select **Edit** for the desired user or role, then select the **SQL Tables** or **SQL Views** tab. Select the desired **Namespace** from the drop-down list. Then select the **Add Tables** or **Add Views** button. In the displayed window, choose a schema, select one or more tables, and assign privileges.

You can determine if the current user has a specified object privilege by invoking the %CHECKPRIV command. You can determine if a specified user has a specified table-level object privilege by invoking the \$SYSTEM.SQL.CheckPriv() method, as shown in the following example:

```
WRITE "SELECT privilege? ", $SYSTEM.SQL.CheckPriv("DeborahTest", "1,SQLUSER.TestT1", "s"), !
WRITE "UPDATE privilege? ", $SYSTEM.SQL.CheckPriv("DeborahTest", "1,SQLUSER.TestT1", "u"), !
WRITE "DELETE privilege? ", $SYSTEM.SQL.CheckPriv("DeborahTest", "1,SQLUSER.TestT1", "d"), !
```

Object Owner Privileges

The owner of a table, view, or procedure always has all SQL privileges implicitly on the SQL object. The owner of the object has privileges on the object in all namespaces to which the object is mapped. Prior to Caché 2013.1, the owner of the object would only have privileges on the object in the namespace the class was compile in. You must recompile a pre-2013.1 object in order to have privileges on the object in all namespaces.

GRANT column-privilege

Column privileges give a user or role a specified privilege to a specified list of columns on a specified table or view. This permits you to allow access to some table columns and not to other columns of the same table. This gives more specific access control than the GRANT object-privilege option, which defines privileges for an entire table or view. When granting privileges to a grantee, you should grant either table-level privilege or column-level privileges for a table, but not both. The SELECT, INSERT, UPDATE, and REFERENCES privileges can be used to grant access to data in individual columns.

A user having a SELECT, INSERT, UPDATE, or REFERENCES *object-privilege* on a table WITH GRANT OPTION can grant to other users a *column-privilege* of the same type for columns of that table.

You can specify a single column, or a comma-separated list of columns. The *column-list* must be enclosed in parentheses. Column names can be specified in any order, and duplication is permitted. Granting a column privilege to a column that already has that privilege has no effect.

The following example grants the UPDATE privilege for two columns:

```
GRANT UPDATE(Name,FavoriteColors) ON Sample.Person TO Deborah
```

You can grant column privileges on a table or a view. You can grant column privileges to any type of *grantee*, including a list of users, a list of roles, *, and _PUBLIC. However, you cannot use the asterisk (*) wildcard for privileges, field names, or table names.

If a user inserts a new record into a table, data is inserted into only those fields for which column privileges have been granted. All other data columns are set to either the defined column default value, or to NULL if there is no defined default value. You cannot grant column-level INSERT or UPDATE privileges to the RowID and Identity columns. Upon INSERT, Caché SQL automatically provides a RowID and (if needed) an Identity column value.

Column-level privileges can be granted or revoked via either the SQL **GRANT** and **REVOKE** commands, or via Caché System Security. Go to the Management Portal, select **[System] > [Security Management] > [Users]** (or **[System] > [Security Management] > [Roles]**), select **Edit** for the desired user or role, then select the **SQL Tables** or **SQL Views** tab. Select the

desired **Namespace** from the drop-down list. Then select the **Add Columns** button. In the displayed window, choose a schema, choose a table, select one or more columns, and assign privileges.

Granting Multiple Privileges

You can use a single **GRANT** statement to specify the following combinations of privileges:

- One or more roles.
- One or more table-level privileges and one or more column-level privileges. To specify multiple table-level and column-level privileges, the privilege must immediately precede a *column-list* to grant a column-level privilege. Otherwise, it grants a table-level privilege.
- One or more admin-privileges. You cannot include admin-privileges and role names or object privileges in the same **GRANT** statement. Attempting to do so results in an SQLCODE -1 error.

The following example grants Deborah table-level SELECT and UPDATE privileges, and column-level INSERT privileges:

```
GRANT SELECT,UPDATE,INSERT(Name,FavoriteColors) ON Sample.Person TO Deborah
```

The following example grants Deborah column-level SELECT, INSERT, and UPDATE privileges:

```
GRANT SELECT(Name,FavoriteColors),INSERT(Name,FavoriteColors),UPDATE(FavoriteColors) ON Sample.Person TO Deborah
```

The WITH GRANT OPTION Clause

The owner of an object automatically holds all privileges on that object. The **GRANT** statement's TO clause specifies the users or roles to whom access is being granted. After using the TO option to specify the grantee, you may optionally specify the WITH GRANT OPTION keyword clause to allow the grantee(s) to also be able to grant the same privileges to other users. You can use the WITH GRANT OPTION keyword clause with object privileges or column privileges. The **REVOKE** command with CASCADE can be used to undo this cascading series of granted privileges.

For instance, you can give the user Chris %ALTER, SELECT, and INSERT privileges on the EMPLOYEES table with the following command:

```
GRANT %ALTER, SELECT, INSERT
      ON EMPLOYEES
      TO Chris
```

To also give Chris the ability to give these privileges to other users, the GRANT command includes the WITH GRANT OPTION clause:

```
GRANT %ALTER, SELECT, INSERT
      ON EMPLOYEES
      TO Chris WITH GRANT OPTION
```

You can find out the results of a GRANT statement using the %SQLCatalogPriv.SQLUsers() method call.

Granting privileges to a schema WITH GRANT OPTION allow the grantee(s) to be able to grant the same schema privileges to other users. However, it does not allow the grantee to grant a privilege on a specified object within that schema, unless the user has been explicitly granted the privilege on that particular object WITH GRANT OPTION. This is shown in the following example:

- UserA and UserB start with no privileges.
- You grant UserA SELECT privilege on schema Sample WITH GRANT OPTION.
- UserA can grant SELECT privilege on schema Sample to UserB.
- UserA *cannot* grant SELECT privilege on table Sample.Person to UserB.

The WITH ADMIN OPTION Clause

The WITH ADMIN OPTION clause grants the *grantee* the right to grant the same privileges it received to others. To grant a system privilege, you must have been granted the system privilege WITH ADMIN OPTION.

You may grant a role if either the role has been granted to you WITH ADMIN OPTION, or if you have the %Admin_Secure:"U" resource.

A grant WITH ADMIN OPTION supersedes a previous grant of the same privilege(s) without this option. Thus, if you grant a user a privilege without WITH ADMIN OPTION, and then grant the same privilege to the user WITH ADMIN OPTION, the user has the WITH ADMIN OPTION rights. However, a grant without the WITH ADMIN OPTION *does not* supersede a previous grant of the same privilege(s) with this option. To remove WITH ADMIN OPTION rights from a privilege, you must revoke the privilege and then re-grant the privilege without this clause.

Exporting Privileges

You can export privileges using the `$$SYSTEM.SQL.Export()` method. When you specify a table in this method, Caché exports both all table-level privileges and all column-level privileges granted for that table. For further details, refer to the *InterSystems Class Reference*.

Obsolete Privileges

At Caché Version 5.1 and all subsequent versions, **GRANT** no longer supports the following general administrative privileges: %GRANT_ANY_PRIVILEGE, %CREATE_USER, %ALTER_USER, %DROP_USER, %CREATE_ROLE, %GRANT_ANY_ROLE, %DROP_ANY_ROLE. Control of these privileges is handled at the system level, rather than through SQL. These SQL privileges were available in prior versions of Caché, and may appear in existing code. An attempt to grant one of these to a user may execute, but it results not in the granting of a privilege, but in the granting of a role having this name.

Caché Security

Before using **GRANT** in embedded SQL, it is necessary to be logged in as a user with appropriate privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation). Use the `$$SYSTEM.Security.Login()` method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql( )
```

You must have the %Service_Login:Use privilege to invoke the `$$SYSTEM.Security.Login` method. For further information, refer to %SYSTEM.Security in the *InterSystems Class Reference*.

Enforcement of Privileges

SQL privileges are only enforced through ODBC, JDBC, and Dynamic SQL (`%SQL.Statement` and `%Library.ResultSet`).

The enforcement of privileges depends upon the setting of the **SQL Security Enabled** configuration option, as follows:

- The `$$SYSTEM.SQL.SetSQLSecurity()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`, which displays a SQL Security ON: setting.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **SQL Security Enabled**.

The default is “Yes” (1). When “Yes”, a user can only perform actions for which that user has been granted privilege. This is the recommended setting for this option.

If this option is set to “No” (0), SQL Security is disabled for any new process started after changing this setting. This means privilege-based table/view security is suppressed. You can create a table without specifying a user. In this case, the Management Portal assigns “_SYSTEM” as user, and embedded SQL assigns "" (the empty string) as user. Any user can

perform actions on a table or view even if that user has no privileges to do so. For further details, refer to [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.

Examples

The following example creates a user, creates a role, and then assigns the role to the user. If the user or role already exists, it issues SQLCODE -118 error. If the assignment of the privilege or the role has already been done, no error is issued (SQLCODE = 0).

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
CreateUser
SET x=$SYSTEM.SQL.UserExists("MarthaTest")
IF x=0 {&sql(CREATE USER MarthaTest IDENTIFY BY birdpw)
      IF SQLCODE != 0 {WRITE "CREATE USER error: ",SQLCODE,!
                    QUIT}
      }
ELSE {WRITE "User MarthaTest exists, not changing its roles",!
      QUIT }
CreateRoleAndGrant
&sql(CREATE ROLE workerbee)
WRITE !,"CREATE ROLE error code: ",SQLCODE
&sql(GRANT %CREATE_TABLE TO workerbee)
WRITE !,"GRANT privilege error code: ",SQLCODE
&sql(GRANT workerbee TO MarthaTest)
WRITE !,"GRANT role error code: ",SQLCODE
```

The following example shows the assignment of multiple privileges. It creates a user and creates two roles. A single **GRANT** statement assigns these roles and a list of admin-privileges to the user. If the user or a role already exists, it issues SQLCODE -118 error. If the assignment of a privilege or a role has already been done, no error is issued (SQLCODE = 0).

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
CreateUser
SET x=$SYSTEM.SQL.UserExists("NoahTest")
IF x=0 {&sql(CREATE USER NoahTest IDENTIFY BY birdpw)
      IF SQLCODE != 0 {WRITE "CREATE USER error: ",SQLCODE,!
                    QUIT}
      }
ELSE {WRITE "User NoahTest exists, not changing its roles",!
      QUIT }
Create2RolesAndGrant
&sql(CREATE ROLE workerbee)
WRITE !,"CREATE ROLE 1 error code: ",SQLCODE
&sql(CREATE ROLE drone)
WRITE !,"CREATE ROLE 2 error code: ",SQLCODE
&sql(GRANT workerbee,drone,%CREATE_TABLE,%DROP_TABLE TO NoahTest)
WRITE !,"GRANT roles & privileges error code: ",SQLCODE
```

The following example grants all seven basic privileges ON all tables in the current namespace TO all currently defined users who do not have the %All role:

```
DO $SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql(GRANT * ON * TO *)
```

See Also

- [%CHECKPRIV REVOKE](#)
- [SELECT INSERT DELETE UPDATE](#)
- [CREATE USER CREATE ROLE](#)
- “Users, Roles, and Privileges” chapter of *Using Caché SQL*
- [CREATE FUNCTION CREATE METHOD CREATE PROCEDURE CREATE QUERY](#)
- [CREATE TABLE CREATE VIEW CREATE TRIGGER](#)
- [SQLCODE error messages](#) listed in the *Caché Error Reference*
- ObjectScript: [\\$ROLES](#) and [\\$USERNAME](#) special variables

GROUP BY

A **SELECT** clause that groups the resulting rows of a query according to one or more columns.

```
SELECT ...
GROUP BY field
```

Arguments

<i>field</i>	One or more fields from which data is being retrieved. Either a single field name or a comma-separated list of field names.
--------------	---

Description

GROUP BY is a clause of the **SELECT** statement. The optional **GROUP BY** clause appears after the **FROM** clause and the optional **WHERE** clause, and before the optional **HAVING** and **ORDER BY** clauses.

The **GROUP BY** clause takes the resulting rows of a query and breaks them up into individual groups according to one or more database columns. When you use **SELECT** in conjunction with **GROUP BY**, one row is retrieved for each distinct value of the **GROUP BY** fields. **GROUP BY** treats fields with **NULL** (no specified value) as a separate distinct value group.

The **GROUP BY** clause is conceptually similar to the Caché extension **%FOREACH**, but **GROUP BY** operates on an entire query, while **%FOREACH** allows selection of aggregates on sub-populations without restricting the entire query population.

Specifying a Field

The simplest form of a **GROUP BY** clause specifies a single field, such as `GROUP BY Home_State`. The field must be specified by its column name. You cannot specify a field by column alias or by column number. You cannot specify an aggregate field; attempting to do so generates an **SQLCODE -19** error.

Valid field values include the following: A column name (`GROUP BY Home_State`); an **%ID** (which returns all rows); a scalar function specifying a column name (`GROUP BY ROUND(Age, -1)`); a **collation function** specifying a column name (`GROUP BY %EXACT(Home_City)`).

A **GROUP BY** clause can use the arrow syntax (`->`) operator to specify a field in a table that is not the base table. For example: `GROUP BY Company->Name`. For further details, refer to [Implicit Joins](#) in *Using Caché SQL*.

Specifying a literal in a **GROUP BY** clause returns 1 row; which row is returned is indeterminate. Thus, specifying 7, 'Chicago', "", 0, or **NULL** all return 1 row.

Aggregate Functions with GROUP BY and DISTINCT BY

The **GROUP BY** clause is applied before **aggregate functions** are calculated. In the following example, the **COUNT** aggregate function counts the number of rows in each **GROUP BY** group:

```
SELECT Home_State, COUNT(Home_State)
FROM Sample.Person
GROUP BY Home_State
```

The **DISTINCT BY** clause is applied after aggregate functions are calculated. In the following example, the **COUNT** aggregate function counts the number of rows in the entire table:

```
SELECT DISTINCT BY(Home_State) Home_State, COUNT(Home_State)
FROM Sample.Person
```

In order to calculate an aggregate function for the entire table, rather than a **GROUP BY** group, you can specify a *select-item* subquery:

```
SELECT Home_State, (SELECT COUNT(Home_State) FROM Sample.Person)
FROM Sample.Person
GROUP BY Home_State
```

A **GROUP BY** clause should not be used with a **DISTINCT** clause when the select list consists of an aggregate field. For example, the following query is *intended* to return the distinct numbers of people who share the same `Home_State`:

```
/* This query DOES NOT apply the DISTINCT keyword */
/* It is provided as a cautionary example */
SELECT DISTINCT COUNT(*) AS mynum
FROM Sample.Person
GROUP BY Home_State
ORDER BY mynum
```

This query did not return the expected results because it did not apply the **DISTINCT** keyword. To apply both a **DISTINCT** aggregate and a **GROUP BY** clause, use a subquery as shown in the following example:

```
SELECT DISTINCT *
FROM (SELECT COUNT(*) AS mynum
      FROM Sample.Person
      GROUP BY Home_State) AS Sub
ORDER BY Sub.mynum
```

This example successfully returns the distinct numbers of people who share the same `Home_State`. For instance, if any `Home_State` is shared by 8 people, the query returns an 8.

If an aggregate function does not apply to any data in the table, it returns `%ROWCOUNT=1` with a `NULL` (or 0) value for the aggregate. For example:

```
SELECT AVG(Age) FROM Sample.Person WHERE Name %STARTSWITH 'ZZZZ'
```

However, if this type of query contains a **GROUP BY** clause, it returns `%ROWCOUNT=0`.

```
SELECT AVG(Age) FROM Sample.Person WHERE Name %STARTSWITH 'ZZZZ' GROUP BY Home_State
```

Collation, Letter Case, and Optimization

This section describes how **GROUP BY** handles data values that differ only in letter case.

- Group Lettercase Variants Together (return uppercase):

By default, **GROUP BY** groups together string values based on the **collation** specified for the *field* when it was created. Caché has a **default string collation**, which can be set for each namespace; the initial string collation default for all namespaces is `SQLUPPER`. Therefore, commonly, **GROUP BY** collation is not case-sensitive unless otherwise specified.

GROUP BY groups together the values of a field with `SQLUPPER` collation based on their uppercase letter collation. Field values that differ only in letter case are grouped together. Grouped field values are returned in all uppercase letters. This has the performance advantage of allowing **GROUP BY** to use the index for the field, rather than accessing the actual field values. It has the consequence that the **GROUP BY** field value is returned in all uppercase letters, even if none of the actual data values are in all uppercase letters.

- Group Lettercase Variants Together (return actual lettercase):

GROUP BY can group together values that differ in lettercase and return grouped field values with an actual field lettercase value (randomly selecting). This has the advantage that the returned value is an actual value, showing the lettercase of at least one value in the data. It has the performance disadvantage of not being able to use the field's index. You can specify this for an individual query by applying the **%EXACT** collation function to the select-item field. You can configure this behavior for all queries that contain a **GROUP BY** clause by using the Management Portal. Select **System Administration, Configuration, SQL and Object Settings, General SQL Settings**. From the **Optimization** tab, clear the **DISTINCT Optimization Turned ON** check box. By default, this check box is selected, which causes grouping of alphabetic values by their uppercase letter collation.

- Do Not Group Lettercase Variants Together (return actual lettercase):

GROUP BY can perform case-sensitive grouping of values by applying the [%EXACT](#) collation function to the **GROUP BY** field. This has the advantage of returning every lettercase variant as a separate group. It has the performance disadvantage of not being able to use the field's index.

The following examples show these behaviors. These examples assume that Sample.Person contains records with a Home_City field with SQLUPPER collation and values of 'New York' and 'new york':

```
SELECT Home_City FROM Sample.Person GROUP BY Home_City
/* groups together Home_City values by their uppercase letter values
   returns the name of each grouped city in uppercase letters.
   Thus, 'NEW YORK' is returned. */

SELECT %EXACT(Home_City) FROM Sample.Person GROUP BY Home_City
/* groups together Home_City values by their uppercase letter values
   returns the name of a grouped city in original letter case.
   Thus, 'New York' or 'new york' may be returned, but not both. */

SELECT Home_City FROM Sample.Person GROUP BY %EXACT(Home_City)
/* groups together Home_City values by their original letter case
   returns the name of each grouped city in original letter case.
   Thus, both 'New York' and 'new york' are returned as separate groups. */
```

You can configure optimization for queries that contain a GROUP BY clause using the Management Portal. Select **System Administration, Configuration, SQL and Object Settings, General SQL Settings**. From the **Optimization** tab view and set the **DISTINCT Optimization Turned ON** check box option. (This optimization also works for the [DISTINCT](#) clause.) The default is selected (“Yes”). When this optimization is in effect, grouping of alphabetic values is done using their uppercase letter collation. For further details, refer to [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.

You can also set this system-wide option to 1 or 0 with the `$$SYSTEM.SQL.SetFastDistinct()` method:

```
WRITE $$SYSTEM.SQL.SetFastDistinct(1)
```

This optimization takes advantage of indices for the selected field(s). It is therefore only meaningful if an index exists for one or more of the selected fields. It collates field values as they are stored in the index; alphabetic strings are returned in all uppercase letters. You can set this system-wide option, then override it for specific queries by using the [%EXACT](#) collation function to preserve letter case.

[%ROWID](#)

Specifying a GROUP BY clause causes a [cursor-based Embedded SQL query](#) to not set the [%ROWID](#) variable. [%ROWID](#) is not set even when GROUP BY does not limit the rows returned. This is shown in the following example:

```
SET %ROWID=999
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name, Home_State
    INTO :name,:state FROM Sample.Person
    WHERE Home_State %STARTSWITH 'M'
    GROUP BY Home_State)
&sql(OPEN EmpCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"RowID: ",%ROWID," row count: ",%ROWCOUNT
    WRITE " Name=",name," State=",state
}
&sql(CLOSE EmpCursor)
```

This change of query behavior only applies to cursor-based Embedded SQL **SELECT** queries. Dynamic SQL **SELECT** queries and non-cursor Embedded SQL **SELECT** queries never set [%ROWID](#).

[Transaction Committed Changes](#)

A query containing a **GROUP BY** clause does not support READ COMMITTED isolation level. In a transaction defined as READ COMMITTED, a **SELECT** statement without a **GROUP BY** clause returns only data modifications that have

been committed; in other words, it returns the state of the data before the current transaction. A **SELECT** statement with a **GROUP BY** clause returns all data modifications made, whether or not they have been committed.

Example

The following example groups names by their initial letter. It returns the initial letter, the count of names sharing that initial letter, and an example of a one of the name values. Names are grouped using their `SQLUPPER` collation, regardless of the letter case of the actual values. Note that the `Name` select-item contains the uppercase initial letter; `%EXACT` collation is used to display an actual name value:

```
SELECT Name AS Initial,COUNT(Name) AS SameInitial,%EXACT(Name) AS Example
FROM Sample.Person GROUP BY %SQLUPPER(Name,2)
```

See Also

- [SELECT](#)
- [DISTINCT](#) clause
- [JOIN](#)

HAVING

A **SELECT** clause that specifies one or more restrictive conditions.

```
SELECT field
FROM table
GROUP BY field
HAVING condition-expression

SELECT aggregatefunc(field %AFTERHAVING)
FROM table
[GROUP BY field]
HAVING condition-expression
```

Arguments

<i>condition-expression</i>	An expression consisting of one or more boolean predicates governing which data values are to be retrieved.
-----------------------------	---

Description

The optional **HAVING** clause appears after the **FROM** clause and the optional **WHERE** and **GROUP BY** clauses, and before the optional **ORDER BY** clause.

The **HAVING** clause of a **SELECT** statement qualifies or disqualifies specific rows from the query selection. The rows that qualify are those for which the *condition-expression* is true. The *condition-expression* is a series of logical tests (predicates) which can be linked by the AND and OR logical operators. For further details, see the **WHERE** clause.

The **HAVING** clause is like a **WHERE** clause that can operate on groups, rather than on the full data set. Thus, in most cases, the **HAVING** clause is used either with an **aggregate function** using the **%AFTERHAVING** keyword, or in combination with a **GROUP BY** clause, or both.

A **HAVING** clause *condition-expression* can also specify an **aggregate function**. A **WHERE** clause *condition-expression* cannot specify an aggregate function. This is shown in the following example:

```
SELECT Name, Age, AVG(Age) AS AvgAge
FROM Sample.Person
HAVING Age > AVG(Age)
ORDER BY Age
```

A **HAVING** clause often serves to compare aggregates of sub-populations against aggregates for an entire population.

Aggregate Functions in the select-item List

The **HAVING** clause selects which rows to return. By default, this row selection does not determine the value of aggregate functions in the *select-item* list. This is because the **HAVING** clause is parsed after aggregate functions in the *select-item* list.

In the following example, only those rows with `Age > 65` are returned. But the `AVG(Age)` is calculated based on all rows, not just those selected by the **HAVING** clause:

```
SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person
HAVING Age > 65
ORDER BY Age
```

Compare this to a **WHERE** clause, which selects both which rows to return and which row values to include in aggregate functions in the *select-item* list:

```
SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person
WHERE Age > 65
ORDER BY Age
```

A **HAVING** clause can be used in a query that *only* returns aggregate values:

- **Aggregate Threshold:** The **HAVING** clause uses an aggregate threshold to determine whether to return 1 row (containing the query aggregate values) or 0 rows. Thus you can use a **HAVING** clause to only return an aggregate calculation when an aggregate threshold is achieved. The following example only returns an average of the *Age* values for all rows in the table when there are at least 100 rows in the table. If there are less than 100 rows, the average of the *Age* values for all rows might not be deemed meaningful, and therefore should not be returned:

```
SELECT AVG(Age) FROM Sample.Person HAVING COUNT(*)>99
```

- **Multiple Rows:** A **HAVING** clause with an aggregate function and no **GROUP BY** clause returns the number of rows that fulfill the **HAVING** clause condition. The aggregate function value is calculated based on all of the rows in the table:

```
SELECT AVG(Age) FROM Sample.Person HAVING %ID<10
```

This is in contrast to a **WHERE** clause with an aggregate function, which returns one row. The aggregate function value is calculated based on rows that fulfill the **WHERE** clause condition:

```
SELECT AVG(Age) FROM Sample.Person WHERE %ID<10
```

%AFTERHAVING

The **%AFTERHAVING** keyword can be used with an aggregate function in the *select-item* list to specify that the aggregate operation is to be performed after the **HAVING** clause condition is applied.

```
SELECT Name, Age, AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
HAVING Age > 40 AND Age < 65
ORDER BY Age
```

The **%AFTERHAVING** keyword only gives meaningful results if both of the following considerations are met:

- The *select-item* list must contain at least one item that is a non-aggregate field reference. This field reference may be to any field in any table specified in the **FROM** clause, a field referenced using an implicit join (arrow syntax), the **%ID** alias, or an asterisk (*).
- The **HAVING** clause condition must apply at least one non-aggregate condition. Therefore, **HAVING Age>50**, **HAVING Age>AVG(Age)**, or **HAVING Age>50 AND MAX(Age)>75** are valid conditions, but **HAVING Age>50 OR MAX(Age)>75** is not a valid condition.

The following example uses a **HAVING** clause with a **GROUP BY** clause to return the state average age, and the state average age for people that are older than the average age for all rows in the table. It also uses a subquery to return the average age for all rows in the table:

```
SELECT Home_State, (SELECT AVG(Age) FROM Sample.Person) AS AvgAgeAllRecs,
                  AVG(Age) AS AvgAgeByState, AVG(Age %AFTERHAVING) AS AvgOlderByState
FROM Sample.Person
GROUP BY Home_State
HAVING Age > AVG(Age)
ORDER BY Home_State
```

Logical Predicates

The SQL predicates fall into the following categories:

- [Equality Comparison Predicates](#)
- [BETWEEN Predicate](#)
- [IN and %INLIST Predicates](#)

- [%STARTSWITH Predicate](#)
- [Contains Operator \(I\)](#)
- [FOR SOME Predicate](#)
- [NULL Predicate](#)
- [EXISTS Predicate](#)
- [LIKE, %MATCHES, and %PATTERN Predicates](#)
- [%INSET and %FIND Predicates](#)

Note: You cannot use the **%CONTAINS** or **%CONTAINSTERM** comparison predicate, or the **FOR SOME %ELEMENT** predicate in a **HAVING** clause. These comparison predicates can only be used in a **WHERE** clause.

Predicate Case-Sensitivity

A predicate uses the [collation type](#) defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

The **%INLIST**, Contains operator (I), **%MATCHES**, and **%PATTERN** predicates do not use the field’s default collation. They always uses EXACT collation, which is case-sensitive.

A predicate comparison of two literal strings is always case-sensitive.

Predicate Conditions and %NOINDEX

You can preface a predicate condition with the **%NOINDEX** keyword to prevent the query optimizer using an index on that condition. This is most useful when specifying a range condition that is satisfied by the vast majority of the rows. For example, `HAVING %NOINDEX Age >= 1`. For further details, refer to [Index Optimization Options](#) in the *Caché SQL Optimization Guide*.

Equality Comparison Predicates

The following are the available comparison predicates:

Table B–1: SQL Equality Comparison Predicates

Predicate	Operation
=	Equals
<>	Does not equal
!=	Does not equal
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

The following example uses a comparison predicate. It returns one record for each Age less than 21:

```
SELECT Name, Age FROM Sample.Person
GROUP BY Age
HAVING Age < 21
ORDER BY Age
```

Note that SQL defines comparison operations in terms of collation: the order in which values are sorted. Two values are equal if they collate in exactly the same way. A value is greater than another value if it collates after the second value. String data type [field collation](#) is based on the field's default collation. By default, it is not case-sensitive. Thus, a comparison of two string field values or a comparison of a string field value with a string literal is (by default) not case-sensitive. For example, if Home_State field values are uppercase two-letter strings:

Expression	Value
'MA' = Home_State	TRUE for values MA.
'ma' = Home_State	TRUE for values MA.
'VA' < Home_State	TRUE for values VT, WA, WI, WV, WY.
'ar' >= Home_State	TRUE for values AK, AL, AR.

Note, however, that a comparison of two literal strings *is* case-sensitive: WHERE 'ma' = 'MA' is always FALSE.

BETWEEN Predicate

This is equivalent to a paired greater than or equal to and less than or equal to. The following example uses a BETWEEN predicate. It returns one record for each Age between 18 and 35, inclusive of 18 and 35:

```
SELECT Name, Age FROM Sample.Person
GROUP BY Age
HAVING Age BETWEEN 18 AND 35
ORDER BY Age
```

For further details, refer to the [BETWEEN](#) reference page in this manual.

IN and %INLIST Predicates

The [IN](#) predicate is used for matching a value to an unstructured series of items.

The [%INLIST](#) predicate is a Caché extension for matching a value to the elements of a list structure.

With either predicate you can perform equality comparisons and subquery comparisons.

IN has two formats. The first serves as shorthand for the use of multiple equality comparisons linked together with the OR operator. For instance:

```
SELECT Name, Home_State FROM Sample.Person
GROUP BY Home_State
HAVING Home_State IN ('ME', 'NH', 'VT', 'MA', 'RI', 'CT')
```

evaluates true if Home_State equals any of the values inside the parenthetical list. The list elements can be constants or expressions. [Collation](#) applies to the IN comparison as it applies to an equality test. By default, IN comparisons use the collation type of the field definition; by default string fields are defined as SQLUPPER, which is not case-sensitive.

When dates or times are used for IN predicate equality comparisons, the appropriate data type conversions are automatically performed. If the **HAVING** clause field is type TimeStamp, values of type Date or Time are converted to Timestamp. If the **HAVING** clause field is type Date, values of type TimeStamp or String are converted to Date. If the **HAVING** clause field is type Time, values of type TimeStamp or String are converted to Time.

The following examples both perform the same equality comparisons and return the same data. The **GROUP BY** field specifies to return only one record for each successful equality comparison. The DOB field is of data type Date:

```
SELECT Name, DOB FROM Sample.Person
GROUP BY DOB
HAVING DOB IN ({d '1951-02-02'}, {d '1987-02-28'})

SELECT Name, DOB FROM Sample.Person
GROUP BY DOB
HAVING DOB IN ({ts '1951-02-02 02:37:00'}, {ts '1987-02-28 16:58:10'})
```

For further details refer to [Date and Time Constructs](#).

The **%INLIST** predicate can be used to perform an equality comparison on the elements of a list structure. **%INLIST** uses EXACT collation. Therefore, by default, **%INLIST** string comparisons are case-sensitive. For further details on list structures, see the SQL [\\$LIST](#) function.

The following example uses **%INLIST** to match a string value to the elements of the FavoriteColors list field:

```
SELECT Name,FavoriteColors FROM Sample.Person
HAVING 'Red' %INLIST FavoriteColors
```

It returns all records where FavoriteColors includes the element “Red”.

The following embedded SQL example matches Home_State column values to the elements of the *northne* (northern New England states) list:

```
SET northne=$LISTBUILD("VT","NH","ME")
&sql(DECLARE StateCursor CURSOR FOR
      SELECT Name,Home_State
      INTO :name,:state FROM Sample.Person
      HAVING Home_State %INLIST :northne)
&sql(OPEN StateCursor)
      QUIT:(SQLCODE'=0)
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH StateCursor)
      QUIT:SQLCODE
      WRITE !,"#",&ROWCOUNT," Name=",name," State=",state,!
}
WRITE !,"Final Fetch SQLCODE: ",SQLCODE
&sql(CLOSE StateCursor)
```

You can also use IN or **%INLIST** with a subquery to test whether a column value (or any other expression) equals any of the subquery row values. For example:

```
SELECT Name,Home_State FROM Sample.Person
HAVING Name IN
  (SELECT Name FROM Sample.Employee
   HAVING Salary < 50000)
```

Note that the subquery must have exactly one item in the SELECT list.

For further details, refer to the [IN](#) and [%INLIST](#) reference pages in this manual.

%STARTSWITH Predicate

The Caché **%STARTSWITH** comparison operator permits you to perform partial matching on the initial characters of a string or numeric. The following example uses **%STARTSWITH**. It selects by age, then returns a record for each Name that begins with “S”:

```
SELECT Name,Age FROM Sample.Person
WHERE Age > 30
HAVING Name %STARTSWITH 'S'
ORDER BY Name
```

Like other string field comparisons, **%STARTSWITH** comparisons are not case-sensitive. For further details, refer to the [%STARTSWITH](#) reference page in this manual.

Contains Operator (I)

The Contains operator is the open bracket symbol: [. It permits you to match a substring (string or numeric) to any part of a field value. The comparison is always case-sensitive. The following example uses the Contains operator in a HAVING clause to select those records in which the Home_State value contains a “K”, and then do an **%AFTERHAVING** count on those states:

```
SELECT Home_State,COUNT(Home_State) AS States,
      COUNT(Home_State %AFTERHAVING) AS KStates
FROM Sample.Person
HAVING Home_State [ 'K'
```

FOR SOME Predicate

The FOR SOME predicate of the HAVING clause determines whether or not to return a result set based on a condition test of one or more field values. This predicate has the following syntax:

```
FOR SOME (table[AS t-alias]) (fieldcondition)
```

FOR SOME specifies that *fieldcondition* must evaluate to true; at least one of the field values must match the specified condition. *table* can be a single table or a comma-separated list of tables, and can optionally take a table alias. *fieldcondition* specifies one or more conditions for one or more fields within the specified *table*. Both the *table* argument and the *fieldcondition* argument must be delimited by parentheses.

The following example shows the use of the FOR SOME predicate:

```
SELECT Name, Age
FROM Sample.Person
HAVING FOR SOME (Sample.Person)(Age>20)
ORDER BY Age
```

In the above example, if at least one field contains an Age value greater than 20, all of the records are returned. Otherwise, no records are returned.

For further details, refer to the [FOR SOME](#) reference page in this manual.

NULL Predicate

This detects undefined values. You can detect all null values, or all non-null values:

```
SELECT Name, FavoriteColors FROM Sample.Person
HAVING FavoriteColors IS NULL
```

```
SELECT Name, FavoriteColors FROM Sample.Person
HAVING FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

Using the **GROUP BY** clause, you can return one record for each non-null value for a specified field:

```
SELECT Name, FavoriteColors FROM Sample.Person
GROUP BY FavoriteColors
HAVING FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

For further details, refer to the [NULL](#) reference page in this manual.

EXISTS Predicate

This operates with subqueries to test whether a subquery evaluates to the empty set.

```
SELECT t1.disease FROM illness_tab t1 WHERE EXISTS
(SELECT t2.disease FROM disease_registry t2
WHERE t1.disease = t2.disease
HAVING COUNT(t2.disease) > 100)
```

For further details, refer to the [EXISTS](#) reference page in this manual.

LIKE, %MATCHES, and %PATTERN Predicates

These three predicates allow you to perform pattern matching.

- **LIKE** allows you to pattern match using literals and wildcards. Use LIKE when you wish to return data values that contain a known substring of literal characters, or contain several known substrings in a known sequence. LIKE uses the collation of its target for letter case comparisons.
- **%MATCHES** allows you to pattern match using literals, wildcards, and lists and ranges. Use %MATCHES when you wish to return data values that contain a known substring of literal characters, or contain one or more literal characters

that fall within a list or range of possible characters, or contain several such substrings in a known sequence.

`%MATCHES` uses EXACT collation for letter case comparisons.

- [%PATTERN](#) allows you to specify a pattern of character types. For example, `'1U4L1', '.A'` (1 uppercase letter, 4 lowercase letters, one literal comma, followed by any number of letter characters of either case). Use `%PATTERN` when you wish to return data values that contain a known sequence of character types. `%PATTERN` is especially useful when the data value is unimportant, but the character type format of those values is significant. `%PATTERN` can also specify known literal characters. It uses EXACT collation for literal comparisons, which are always case-sensitive.

To perform a comparison with the first characters of a string, use the [%STARTSWITH](#) predicate.

Examples

The following example returns a row for each state that has at least one person under the age of 21. For each row it returns the average, minimum, and maximum ages of all people in the state.

```
SELECT Home_State, MIN(Age) AS Youngest,
       AVG(Age) AS AvgAge, MAX(Age) AS Oldest
FROM Sample.Person
GROUP BY Home_State
HAVING Age < 21
ORDER BY Youngest
```

The following example returns a row for each state that has at least one person under the age of 21. For each row it returns the average, minimum, and maximum ages of all people in the state. Using the `%AFTERHAVING` keyword, it also returns the average age of those people in the state under the age of 21 (`AvgYouth`), and the age of the oldest person in the state under the age of 21 (`OldestYouth`).

```
SELECT Home_State,AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgYouth,
       MIN(Age) AS Youngest, MAX(Age) AS Oldest,
       MAX(Age %AFTERHAVING) AS OldestYouth
FROM Sample.Person
GROUP BY Home_State
HAVING Age < 21
ORDER BY AvgAge
```

For further examples of `%AFTERHAVING`, refer to the individual [aggregate functions](#).

See Also

- [SELECT](#) statement
- [WHERE](#) clause
- [GROUP BY](#) clause
- [Overview of Predicates](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*

INSERT

Adds a new row (or rows) to a table.

```
INSERT [%NOFPLAN] [restriction] [INTO] table
    SET column1 = scalar-expression1 {,column2 = scalar-expression2} ... |
    [ (column1{,column2} ...) ] VALUES (scalar-expression1 {,scalar-expression2}
... ) |
    VALUES :array() |
    [ (column1{,column2} ...) ] query |
    DEFAULT VALUES
```

Arguments

<code>%NOFPLAN</code>	<i>Optional</i> — The <code>%NOFPLAN</code> keyword specifies that Caché will ignore the frozen plan (if any) for this operation and generate a new query plan. The frozen plan is retained, but not used. For further details, refer to Frozen Plans in <i>Caché SQL Optimization Guide</i> .
<code>restriction</code>	<i>Optional</i> — One or more of the following keywords, separated by spaces: <code>%NOLOCK</code> , <code>%NOCHECK</code> , <code>%NOINDEX</code> , <code>%NOTRIGGER</code> .
<code>table</code>	The name of the table or view on which to perform the insert operation. This argument may be a subquery. The <code>INTO</code> keyword is optional. A table name (or view name) can be qualified (schema.table), or unqualified (table). An unqualified name is matched to its schema using either a schema search path (if provided) or the system-wide default schema name .
<code>column</code>	<i>Optional</i> — A column name or comma-separated list of column names that correspond in sequence to the supplied list of values. If omitted, the list of values is applied to all columns in column-number order.
<code>scalar-expression</code>	A scalar expression or comma-separated list of scalar expressions that supplies the data values for the corresponding <code>column</code> fields.
<code>:array()</code>	<i>Embedded SQL only</i> — A dynamic local array of values specified as a host variable . The lowest subscript level of the array must be unspecified. Thus <code>:myupdates()</code> , <code>:myupdates(5,)</code> , and <code>:myupdates(1,1,)</code> are all valid specifications.
<code>query</code>	A SELECT query the result set of which supplies the data values for the corresponding <code>column</code> fields for one or multiple new rows.

Description

The **INSERT** statement can be used in two ways:

- A single-row **INSERT** adds one new row to a table. It inserts data values for all specified columns (fields) and defaults unspecified column values to either `NULL` or the defined default value. It sets the `%ROWCOUNT` variable to the number of affected rows (always either 1 or 0).
- An **INSERT with a SELECT** adds multiple new rows to a table. It inserts data values for all specified columns (fields) for each row from the query result set and defaults unspecified column values to either `NULL` or the defined default value. This use of an **INSERT** statement combined with a **SELECT** query is commonly used to populate a table with existing data extracted from other tables, as described in the “[INSERT Query Results](#)” section below.

This reference page is structured as follows:

- [Restriction Keywords Argument](#)

- [Table Argument](#)
- [Column and Values Arguments](#), including single-row **INSERT** value assignment syntax options and the handling of special fields, data types, special values, and default values.
- [Insert Query Results](#), using **INSERT** with a **SELECT** to insert multiple rows.
- [SQLCODE Errors](#)
- [Operational Considerations](#), including privileges and security, atomicity and transaction locking, referential integrity.
- [Program Examples](#)

INSERT OR UPDATE

The **INSERT OR UPDATE** statement is a variant of the **INSERT** statement that performs both insert and update operations. First it attempts to perform an insert operation. If the insert request fails due to a **UNIQUE KEY** violation (for the field(s) of some unique key, there exists a row that already has the same value(s) as the row specified for the insert), then it automatically turns into an update request for that row, and **INSERT OR UPDATE** uses the specified field values to update the existing row.

Restriction Keywords Argument

To use a *restriction* argument, you must have the corresponding *admin-privilege* for the current namespace. Refer to [GRANT](#) for further details.

Specifying *restriction* argument(s) restricts processing as follows:

- **%NOCHECK** — foreign key referential integrity checking is not performed. Column data validation for data type, maximum length, data constraints, and other validation criteria is also not performed. The **WITH CHECK OPTION** validation for a view is not performed when performing an **INSERT** through a view.

Note: Because use of **%NOCHECK** can result in invalid data, this restriction argument should only be used when performing bulk inserts or updates from a reliable data source.
- **%NOLOCK** — the row is not locked upon **INSERT**. This should only be used when a single user/process is updating the database.
- **%NOINDEX** — the index maps are not set during **INSERT** processing.
- **%NOTRIGGER** — the base table triggers are not pulled during **INSERT** processing.

You can specify multiple *restriction* arguments in any order. Multiple arguments are separated by spaces.

Table Argument

You can specify the *table* argument to insert into a [table](#) directly, insert through a [view](#), or insert via a subquery. Inserting through a view is subject to requirements and restrictions, as described in [CREATE VIEW](#). The following is an example of an **INSERT** using a subquery in place of the *table* argument:

```
INSERT INTO (SELECT field1 AS ff1 FROM MyTable) (ff1) VALUES ('test')
```

The subquery target must be updateable, following the same criteria used to determine if a view's query is updateable. Attempting to **INSERT** using a view or a subquery that is not updateable generates an **SQLCODE -35** error.

You cannot specify a table-valued function or **JOIN** syntax in the *table* argument.

For required table privileges, refer to [Privileges](#). For error codes, refer to [SQLCODE Errors](#).

Value Assignment

This section describes how data values are assigned to columns (fields) during an **INSERT** operation:

- [Value Assignment Syntax](#) describing the various syntax options for specifying data values as literals to columns (fields).
- [Display to Logical Data Conversion](#)
- [%SerialObject Properties](#)
- [Non-Display Characters](#)
- [Special Variables](#)
- [Stream Data](#)
- [List Structured Data](#)
- [IDENTITY, ROWVERSION, and SERIAL Counters](#)
- [Computed Field Values](#)
- [DEFAULT VALUES Clause](#)

If you omit the *column* list argument, the **INSERT** assumes all columns are to be inserted, in column number order. If you specify a *column* list, the individual values must correspond positionally with the column names in the column list.

Value Assignment Syntax

When inserting a record, you can assign values to specified columns in a variety of ways. All non-specified columns must either accept NULL or have a defined default value.

- Using the SET keyword, specify one or more column = scalar-expression pairs as a comma-separated list. For example:

```
SET StatusDate='05/12/06',Status='Purged'
```

- Using the VALUES keyword, specify a list of columns equated to a corresponding scalar-expressions list. For example:

```
(StatusDate,Status) VALUES ('05/12/06','Purged')
```

When assigning scalar-expression values to a column list, there must be a scalar-expression for each specified column.

- Using the VALUES keyword without a column list, specify a list of scalar-expressions that implicitly correspond to the columns of the row in column order. For example:

```
VALUES ('Fred Wang',65342,'22 Main St. Anytown MA','123-45-6789')
```

Values must be specified in column number order. You must specify a value for *every* base table column that takes a user-supplied value; an **INSERT** using column order cannot take defined field default values. The RowID column cannot be user-specified, and is therefore not included in this syntax.

- Using the VALUES keyword without a column list, specify a dynamic local array of scalar-expressions that implicitly correspond to the columns of the row in column order. For example:

```
VALUES :myarray()
```

This value assignment can only be performed from [Embedded SQL](#) using a host variable. Unlike all other value assignments, this usage allows you to delay specifying which columns are to be inserted until runtime (by populating the array at runtime). All other types of insert require that you specify which columns are to be inserted at compile time.

Values must be specified in column number order. You must specify a value for *every* base table column that takes a user-supplied value; an **INSERT** using column order cannot take defined field default values. Supplied array values

must begin with array(2). Column 1 is the RowID field; you cannot specify a value for the RowID field. For further details, see “[Host Variable as a Subscripted Array](#)” in the “Using Embedded SQL” chapter of *Using Caché SQL*.

If you specify column names and corresponding data values, you can omit columns for which there is a defined default value. An **INSERT** can insert a default value for most field data types, including stream fields.

If you do not specify column names, data values must correspond positionally to the defined column list. You must specify a value for *every* user-specifiable base table column; defined default values cannot be used. (You can, of course, specify an empty string as a column value.)

To list all of the column names and column numbers defined for a specified table, refer to [Column Names and Numbers](#) in the “Defining Tables” chapter of *Using Caché SQL*.

For required column privileges, refer to [Privileges](#). For error codes, refer to [SQLCODE Errors](#).

DISPLAY to LOGICAL Data Conversion

Data is stored in LOGICAL mode format. For example, a date is stored as an integer count of days. Input data that is not in LOGICAL mode format must be converted to LOGICAL mode format. Compiled SQL supports automatic conversion of input values from DISPLAY or ODBC format to LOGICAL format. Automatic conversion of input data requires two factors: when compiled, the SQL must specify RUNTIME mode; when executed, the SQL must execute in a LOGICAL mode environment.

- In [Embedded SQL](#), if you specify `#SQLCompile Select=runtime`, Caché will compile the SQL statement with code that converts input values from a display format to LOGICAL mode storage format. Caché performs this mode conversion both for single values and for arrays of values. For further details, see [#SQLCompile Select](#) in the “ObjectScript Macros and the Macro Preprocessor” chapter of *Using Caché ObjectScript*.
- In an SQL [CREATE FUNCTION](#), [CREATE METHOD](#), or [CREATE PROCEDURE](#) statement, if you specify `SELECTMODE RUNTIME`, Caché will compile the SQL statement with code that converts input values from a display format to LOGICAL mode storage format.

The input data may be in any format: DISPLAY format (for example, 6/17/2011), ODBC format (for example, 2011-06-17), or LOGICAL format (for example, 62259). The data is stored in LOGICAL format if the SQL execution environment is in LOGICAL mode. This is the default mode for all Caché SQL execution environments.

You can explicitly set the select mode to LOGICAL in SQL execution environments as follows:

- In an ObjectScript program or from the Terminal interface: invoke the `$$SYSTEM.SQL.SetSelectMode(0)` method.
- In [Dynamic SQL](#), specify `%SelectMode 0`.
- From the [SQL Shell](#), specify `SET SELECTMODE LOGICAL`.
- From the Management Portal select the **[System] > [SQL]** interface, then use the **Display Mode** drop-down list to specify Logical Mode.

%SerialObject Properties

When inserting data into a [%SerialObject](#), you must insert into the table (persistent class) that references the embedded [%SerialObject](#); you cannot insert into a [%SerialObject](#) directly. From the referencing table, you can either:

- Use the referencing field to insert values for multiple [%SerialObject](#) properties as a [%List](#) structure. For example, if the persistent class has a property PAddress that references a serial object contain the properties Street, City, and Country (in that order), you insert `SET PAddress=$LISTBUILD('123 Main St.', 'Newtown', 'USA')`. The [%List](#) must contain values for the properties of the serial object (or placeholder commas) in the order that these properties are specified in the serial object.
- Use underscore syntax to insert values for individual [%SerialObject](#) properties in any order. For example, if the persistent class has a property PAddress that references a serial object contain the properties Street, City, and Country, you insert

SET PAddress_City='Newtown',PAddress_Street='123 Main St.',PAddress_Country='USA'.
Unspecified serial object properties default to NULL.

Non-Display Characters

You can insert non-display characters using the [CHAR](#) function and the [concatenation operator](#). For example, the following example inserts a string consisting of the letter “A”, a line feed, and the letter “B”:

```
INSERT INTO MyTable (Text) VALUES ('A' || CHAR(10) || 'B')
```

Note that to concatenate the results of a function you must use the `||` concatenation operator, not the `_` concatenation operator. A query can determine if a non-display character is present using the [LENGTH](#) or [\\$LENGTH](#) function.

Special Variables

You can insert into a column the value of the following special variables:

A [%TABLENAME](#), or [%CLASSNAME](#) pseudo-field variable keyword. [%TABLENAME](#) returns the current table name. [%CLASSNAME](#) returns the name of the class corresponding to the current table.

One or more of the following ObjectScript special variables (or their abbreviations): [\\$HOROLOGY](#), [\\$JOB](#), [\\$NAMESPACE](#), [\\$TLEVEL](#), [\\$USERNAME](#), [\\$ZHOROLOGY](#), [\\$ZJOB](#), [\\$ZNSPACE](#), [\\$ZPI](#), [\\$ZTIMESTAMP](#), [\\$ZTIMEZONE](#), [\\$ZVERSION](#).

Stream Data

You cannot use a single **INSERT** to insert multiple rows containing a stream value. Rows containing stream data must be inserted one row at a time.

You can insert the following types of data values into a stream field:

- An object reference (OREF) to a stream object. Caché opens this object and copies its contents into the new stream field. For example:

```
set oref=##class(%Stream.GlobalCharacter).%New()
do oref.Write("Technique 1")

//do the insert; use an actual OREF
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:oref))
```

- A string version of an OREF of a stream, for example:

```
set oref=##class(%Stream.GlobalCharacter).%New()
do oref.Write("Technique 2")

//next line converts OREF to a string OREF
set string=oref_" "

//do the insert
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:string))
```

- A numeric value, such as 1 or -1.
- A string literal whose first character is not numeric, for example:

```
set literal="Technique 3"

//do the insert; use a string
&sql(INSERT INTO MyStreamTable (MyStreamField) VALUES (:literal))
```

If the first character is numeric, SQL interprets the literal as the string form of an OREF instead. For example, the value `2@User.MyClass` would be considered the string version of an OREF, and not a string literal.

Attempting to insert an improperly defined stream value results in an `SQLCODE -412 error: General Stream Error`.

List Structured Data

Caché supports the list structure data type %List (data type class %Library.List). This is a compressed binary format, which does not map to a corresponding native data type for Caché SQL. It corresponds to data type VARBINARY with a default MAXLEN of 32749. For this reason, [Dynamic SQL](#) cannot use **INSERT** or **UPDATE** to set a property value of type %List. For further details, refer to the [Data Types](#) reference page in this manual.

Insert Counter Values

A table can optionally have one field defined as **IDENTITY**. By default, this field receives an integer from an automatically incremented table counter whenever a row is inserted into the table. By default, an insert cannot specify a value for this field. However, this default is configurable. An **IDENTITY** field value cannot be modified by an update operation. This counter is reset by a **TRUNCATE TABLE** operation.

A table can optionally have one or more fields defined as data type **SERIAL** (%Library.Counter). By default, this field receives an integer from an automatically incremented table counter whenever a row is inserted into the table. However, a user can specify an integer value for this field during an insert, overriding the table counter default. A %Counter field value cannot be modified by an update operation. This counter is reset by a **TRUNCATE TABLE** operation.

A table can optionally have one field defined as data type **ROWVERSION**. If this field is defined, an insert operation automatically inserts an integer from the namespace-wide RowVersion counter into this field. An update operation automatically updates this integer with the current namespace-wide RowVersion counter value. No user-specified, calculated, or default value can be inserted for a **ROWVERSION** field. This counter cannot be reset.

Inserting SERIAL Values

An **INSERT** operation can specify one of the following values for a field with the **SERIAL** data type, with the following results:

- No value, 0 (zero), or a nonnumeric value: Caché ignores the specified value, and instead increments this field's current serial counter value by 1, and inserts the resulting integer into the field.
- A positive integer value: Caché inserts the user-specified value into the field, and changes the serial counter value for this field to this integer value.

Thus a **SERIAL** field contains a series incremental integer values. These values are not necessarily continuous or unique. For example, the following is a valid series of values for a **SERIAL** field: 1, 2, 3, 17, 18, 25, 25, 26, 27. Sequential integers are either Caché-generated or user-supplied; nonsequential integers are user-supplied. If you wish **SERIAL** field values to be unique, you must apply a **UNIQUE** constraint on the field.

Insert Computed Values

A field defined with **COMPUTECODE** may insert a value as part of the **INSERT** operation, unless the field is **CALCULATED**. If you supply a value for a **COMPUTED** field or if this field has a default value, **INSERT** stores this explicit value. Otherwise, the field value is computed, as follows:

- **COMPUTECODE**: value is computed and stored upon **INSERT**, value is not changed upon **UPDATE**.
- **COMPUTECODE** with **COMPUTEONCHANGE**: value is computed and stored upon **INSERT**, is recomputed and stored upon **UPDATE**.
- **COMPUTECODE** with **DEFAULT** and **COMPUTEONCHANGE**: default value is stored upon **INSERT**, value is computed and stored upon **UPDATE**.
- **COMPUTECODE** with **CALCULATED** or **TRANSIENT**: you cannot **INSERT** a value for this field because no value is stored. The value is computed when queried. However, if you attempt to insert a value into a calculated field, Caché performs validation on the supplied value and issues an error if the value is invalid. If the value is valid, Caché performs no insert operation, issues no **SQLCODE** error, and increments **ROWCOUNT**.

If the compute code contains a programming error (for example, divide by zero), the INSERT operation fails with an SQLCODE -415 error.

DEFAULT VALUES Clause

You can insert a row into a table that has all of its field values set to default values. Fields that have a defined default value are set to that value. Fields without a defined default value are set to NULL. This is done using the following command:

```
INSERT INTO Mytable DEFAULT VALUES
```

Fields defined with the NOT NULL constraint and no defined DEFAULT fail this operation with an SQLCODE -108.

Fields defined with the UNIQUE constraint can be inserted using this statement. If a field is defined with a UNIQUE constraint and no DEFAULT value, repeated invocations insert multiple rows with this UNIQUE field set to NULL. If a field is defined with a UNIQUE constraint and a DEFAULT value, this statement can only be used once. A second invocation fails with an SQLCODE -119.

DEFAULT VALUES inserts a row with a system-generated integer values for counter fields. These include the RowID, and the optional IDENTITY field, %Counter field, and ROWVERSION field.

Insert Query Results: INSERT with SELECT

A single INSERT can be used to insert multiple rows into a table by combining it with a SELECT statement. The SELECT extracts column data from multiple rows of one table, and the INSERT creates corresponding new rows containing this column data in another table. However, you cannot use this technique to insert multiple rows if the data contains a stream value.

You can limit the number of rows inserted by specifying a TOP clause in the SELECT statement. You can also use an ORDER BY clause in the SELECT statement to determine which rows will be selected by the TOP clause.

An INSERT with SELECT operation sets the %ROWCOUNT variable to the number of rows inserted (either 0 or a positive integer).

The following example uses two embedded SQL programs to show this use of INSERT. The first example uses CREATE TABLE to create a new table SQLUser.MyStudents, and the second example populates this table with data extracted from Sample.Person. (Alternatively, you can create a new table from an existing table definition and insert data from the existing table in a single operation using the \$SYSTEM.SQL.QueryToTable() method.)

To demonstrate this, please run the first embedded SQL program, then run the second. (It is necessary to use two embedded SQL programs here because embedded SQL cannot compile an INSERT statement unless the referenced table already exists.)

The following program creates the MyStudents table with two stored data fields, and one calculated field:

```
ZNSPACE "Samples"
WRITE !,"Creating table"
&sql(CREATE TABLE SQLUser.MyStudents (
  StudentName VARCHAR(32),
  StudentDOB DATE,
  StudentAge INTEGER COMPUTECODE {SET {StudentAge}=
    $PIECE(( $PIECE($H, ", ", 1) - {StudentDOB} ) / 365, ". ", 1) }
    CALCULATED )
)
IF SQLCODE=0 {
  WRITE !,"Created table, SQLCODE=",SQLCODE }
ELSEIF SQLCODE=-201 {
  WRITE !,"Table already exists, SQLCODE=",SQLCODE }
```

The following program uses INSERT to populate the MyStudents table with query results. Because the StudentAge field is calculated you cannot supply a value to this field; its value is calculated each time the MyStudents table is queried:

```

ZNSPACE "Samples"
WRITE !,"Populating table with data"
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB)
    SELECT Name,DOB
    FROM Sample.Person WHERE Age <= '21')
IF SQLCODE=0 {
    WRITE !,"Number of records inserted=",%ROWCOUNT
    WRITE !,"Row ID of last record inserted=",%ROWID }
ELSE {
    WRITE !,"Insert failed, SQLCODE=",SQLCODE }

```

Note that executing this **INSERT** program multiple times will succeed, but produces generally undesirable results. Each execution populates `SQLUser.MyStudents` with another set of records (%ROWCOUNT) with identical Name and DOB field values, automatically assigning each record a unique row ID (%ROWID).

To view the data, go to the Management Portal, select the Globals option for the SAMPLES namespace. Scroll to “`SQLUser.MyStudentsD`” and click the Data option.

The following programs display the `MyStudents` table data, then delete this table:

```

SELECT * FROM SQLUser.MyStudents ORDER BY StudentAge

&sql(DROP TABLE SQLUser.MyStudents)
IF SQLCODE=0 {WRITE !,"Table deleted" }
ELSE {WRITE !,"SQLCODE=",SQLCODE," ",%msg }

```

By default, an Insert Query Results operation is an atomic operation. Either all of the specified rows are inserted in a table, or none of the rows are inserted. For example, if inserting one of the specified rows would violate foreign key referential integrity, the **INSERT** fails and no rows are inserted. This default is modifiable, as described below.

SQLCODE Errors

By default, an **INSERT** is an all-or-nothing event: either the row (or rows) is inserted completely or not at all. Caché returns a status variable `SQLCODE`, indicating the success or failure of the **INSERT**. To insert a row into a table, the insert must meet all table, field name, and field value requirements, as follows.

Tables:

- The table must already exist. Attempting an insert to a nonexistent table results in an `SQLCODE -30` error. Because **INSERT** checks for the table's existence at compile time, a single compiled SQL program (such as an Embedded SQL program) cannot create a table (using [CREATE TABLE](#)) and then insert values into it.
- The table cannot be defined as `READONLY`. Attempting to compile an **INSERT** that references a `ReadOnly` table results in an `SQLCODE -115` error. Note that this error is now issued at compile time, rather than only occurring at execution time. See the description of `READONLY` objects in the [Other Options for Persistent Classes](#) chapter of *Using Caché Objects*.
- If updating a table through a view, the view cannot be defined as `WITH READ ONLY`. Attempting to do so results in an `SQLCODE -35` error. See the [CREATE VIEW](#) command for further details.

Field Names:

- The field must exist. Attempting an insert to a nonexistent field results in an `SQLCODE -29` error. To list all of the field names defined for a specified table, refer to [Column Names and Numbers](#) in the “Defining Tables” chapter of *Using Caché SQL*.
- The insert must specify all required fields. Attempting to insert a row without specifying a value for a required field results in an `SQLCODE -108` error.
- The insert cannot include duplicate field names. Attempting to insert a row containing two fields with the same name results in an `SQLCODE -377` error.

- The insert cannot include fields that are defined as READONLY. Attempting to compile an **INSERT** that references a ReadOnly field results in an SQLCODE -138 error. Note that this error is now issued at compile time, rather than only occurring at execution time.

Field Values:

- Every field value must pass **data type** validation. Attempting to insert a field value inappropriate to the field data type results in an SQLCODE -104 error. Note that this applies only to a inserted data value; a field's **DEFAULT** value, if taken, does not have to pass data type validation or data size validation.
 - **Data Type Mismatch:** The field's data type, not the type of the inserted data, determines appropriateness. For example, attempting to insert a string data type value into a date field fails unless the string passes date validation for the current mode; however, attempting to insert a date data type value into a string field succeeds, inserting the date as a literal string.
 - **Data Size Mismatch:** A data value must be within the MAXLEN, MAXVAL, and MINVAL for the field. For example, attempting to insert a string longer than 24 characters into a field defined as VARCHAR(24), or attempting to insert a number larger than 127 into a field defined as TINYINT result an SQLCODE -104 error.
 - **Numeric Type Mismatch:** If an invalid DOUBLE number is supplied via ODBC or JDBC an SQLCODE -104 error occurs.
- Every field value must pass data value validation:
 - A field **defined as NOT NULL** must be provided with a data value. If there is no **DEFAULT** value, not specifying a data value results in an SQLCODE -108 error, indicating that you have not specified a required field.
 - A field value must obey uniqueness constraints. Attempting to insert a duplicate field value in a field (or group of fields) with a uniqueness constraint results in an SQLCODE -119 error. This error is returned if the field has a **UNIQUE data constraint**, or if the **unique fields constraint** has been applied to a group of fields. This error can occur when you specify a duplicate value to a unique field, or to a primary key field, or when you do not specify a value and a second use of the field's **DEFAULT** would supply a duplicate value. The SQLCODE -119 %msg string includes both the field and the value that violate the uniqueness constraint. For example <Table 'Sample.MyTable', Constraint 'MYTABLE_UNIQUE3', Field(s) FullName="Molly Bloom"; failed unique check> or <Table 'Sample.MyTable', Constraint 'MYTABLE_PKEY2', Field(s) FullName="Molly Bloom"; failed unique check>. For details on listing a table's unique value and primary key field constraints and the naming of constraints, refer to [Catalog Details: Constraints](#).
 - A field defined as a persistent class property with the **VALUELIST parameter** can only accept as a valid value one of the values listed in VALUELIST, or be provided with no value (NULL). VALUELIST valid values are case-sensitive. Specifying a data value that doesn't match the VALUELIST values results in an SQLCODE -104 field value failed validation error.
- Numbers are inserted in **canonical form**, but can be specified with leading and trailing zeros and multiple leading signs. However, in SQL, two consecutive minus signs are parsed as a single-line comment indicator. Therefore, attempting to specify a number with two consecutive leading minus signs results in an SQLCODE -12 error.
- By default, an insert cannot specify values for fields for which the value is system-generated, such as the RowID, IDKey, or IDENTITY field. By default, attempting to insert a non-NULL field value for any of these fields results in an SQLCODE -111 error. Attempting to insert a NULL for one of these fields causes Caché to override the NULL with a system-generated value; the insert completes successfully and no error code is issued.

If a field of data type **ROWVERSION** is defined, it is automatically assigned a system-generated counter value when a row is inserted. Attempting to insert a value into a ROWVERSION field results in an SQLCODE -138 error.

An IDENTITY field can be made to accept user-specified values. By setting the **SetIdentityInsert()** method of the %SYSTEM.SQL class you can override the IDENTITY field default constraint and allow inserts of unique integer values to IDENTITY fields. (You can return the current setting for this constraint by calling the **GetIdentityInsert()**

method.) Inserting an IDENTITY field value changes the IDENTITY counter so that subsequent system-generated values increment from this user-specified value. Attempting to insert a NULL for an IDENTITY field generates an SQLCODE -108 error.

IDKey data has the following restriction. Because multiple IDKey fields in an index are delimited using the “||” (double vertical bar) characters, you cannot include this character string in IDKey field data.

- An insert cannot include a field whose value violates [foreign key referential integrity](#), unless the %NOCHECK restriction argument is specified, or the foreign key was defined with the NOCHECK keyword. Otherwise, attempting an insert that violates foreign key referential integrity results in an SQLCODE -121 error, with a %msg such as the following: <Table 'Sample.MyTable', Foreign Key Constraint 'MYTABLEFKKey2', Field(s) FULLNAME failed referential integrity check>. For details on listing a table's foreign key constraints and the naming of foreign key constraints, refer to [Catalog Details: Constraints](#).
- A field value cannot be a subquery. Attempting to specify a subquery as a field value results in an SQLCODE -144 error.

The INSERT Operation

This section describes the operational considerations when performing an **INSERT**:

- [Privileges](#)
- [Referential Integrity](#)
- [Child Table Insert](#)
- [Atomicity](#)
- [Transaction Locking](#)
- [Row-Level Security](#)
- [Microsoft Access](#)

Privileges

To insert one or more rows of data into a table, you must have either table-level privileges or column-level privileges for that table.

Table-level Privileges

You must have both INSERT and SELECT privileges for the table. Failing to have these privileges results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has these privileges by invoking the %CHECKPRIV command. You can determine if a specified user has these privileges by invoking the \$SYSTEM.SQL.CheckPriv() method. For privilege assignment, refer to the [GRANT](#) command.

Table-level privileges are equivalent to (but not identical to) having column-level privileges on all columns of the table.

Column-level Privileges

If you do not have table-level INSERT privilege, you must have column-level INSERT privilege for at least one column of the table. To insert a specified value into a column, you must have column-level INSERT privilege for that column. Only those columns for which you have INSERT privilege receive the value specified in the **INSERT** command.

If you do not have column-level INSERT privilege for a specified column, Caché SQL inserts the column's default value (if defined), or NULL (if no default is defined). If you do not have INSERT privilege for a column that has no default and is defined as NOT NULL, Caché issues an SQLCODE -99 (Privilege Violation) error at Prepare time.

If the **INSERT** command specifies fields in a WHERE clause of a result set **SELECT**, you must have SELECT privilege for those fields if they are not data insert fields, and both SELECT and INSERT privileges for those fields if they are included in the result set.

When a property is defined as [ReadOnly](#), the corresponding table field is also defined as `ReadOnly`. A `ReadOnly` field may only be assigned a value using [InitialExpression](#) or [SqlComputed](#). Attempting to insert a value for a field for which you have column-level `ReadOnly` (`SELECT` or `REFERENCES`) privilege results in an `SQLCODE -138` error: `Cannot INSERT/UPDATE a value for a read only field.`

You can use [%CHECKPRIV](#) to determine if you have the appropriate column-level privileges. See the [GRANT](#) command for privilege assignment.

Referential Integrity

If you do not specify `%NOCHECK`, Caché uses the system configuration setting to determine whether to perform foreign key referential integrity checking. You can set the system default as follows:

- The `$$SYSTEM.SQL.SetFilerRefIntegrity()` method call.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Perform Referential Integrity Checks on Foreign Keys for INSERT, UPDATE, and DELETE**. The default is “Yes”. If you change this setting, any new process started after changing it will have the new setting.

This setting does not apply to foreign keys that have been defined with the `NOCHECK` keyword.

During an **INSERT** operation, for every foreign key reference a shared lock is acquired on the corresponding row in the referenced table. This row is locked while performing referential integrity checking and inserting the row. The lock is then released (it is not held until the end of the transaction). This ensures that the referenced row is not changed between the referential integrity check and the completion of the insert operation.

However, if you specify the `%NOLOCK restriction` argument, no locking is performed either on the specified table or on the corresponding foreign key row in the referenced table.

Child Table Insert

During an **INSERT** operation on a child table, a shared lock is acquired on the corresponding row in the parent table. This row is locked while inserting the child table row. The lock is then released (it is not held until the end of the transaction). This ensures that the referenced parent row is not changed during the insert operation.

Atomicity

By default, **INSERT**, **UPDATE**, **DELETE**, and **TRUNCATE TABLE** are atomic operations. An **INSERT** either completes successfully or the whole operation is rolled back. If any of the specified rows cannot be inserted, none of the specified rows are inserted and the database reverts to its state before issuing the **INSERT**.

You can modify this default for the current process within SQL by invoking [SET TRANSACTION %COMMITMODE](#). You can modify this default for the current process in ObjectScript by invoking the `SetAutoCommit()` method. The following options are available:

- **IMPLICIT** or 1 (autocommit on) — The default behavior, as described above. Each **INSERT** constitutes a separate transaction.
- **EXPLICIT** or 2 (autocommit off) — If no transaction is in progress, an **INSERT** automatically initiates a transaction, but you must explicitly **COMMIT** or **ROLLBACK** to end the transaction. In **EXPLICIT** mode the number of database operations per transaction is user-defined.
- **NONE** or 0 (no auto transaction) — No transaction is initiated when you invoke **INSERT**. A failed **INSERT** operation can leave the database in an inconsistent state, with some of the specified rows inserted and some not inserted. To provide transaction support in this mode you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

You can determine the atomicity setting for the current process using the `GetAutoCommit()` method, as shown in the following ObjectScript example:

```

DO $SYSTEM.SQL.SetAutoCommit($RANDOM(3))
SET x=$SYSTEM.SQL.GetAutoCommit()
IF x=1 {
  WRITE "Default atomicity behavior",!
  WRITE "automatic commit or rollback" }
ELSEIF x=0 {
  WRITE "No transaction initiated, no atomicity:",!
  WRITE "failed DELETE can leave database inconsistent",!
  WRITE "rollback is not supported" }
ELSE { WRITE "Explicit commit or rollback required" }

```

Transaction Locking

If you do not specify %NOLOCK, the system automatically performs standard record locking on **INSERT**, **UPDATE**, and **DELETE** operations. Each affected record (row) is locked for the duration of the current transaction.

The default lock threshold is 1000 locks per table. This means that if you insert more than 1000 records from a table during a transaction, the lock threshold is reached and Caché automatically escalates the locking level from record locks to a table lock. This permits large-scale inserts during a transaction without overflowing the lock table.

Caché applies one of the two following lock escalation strategies:

- “E”-type lock escalation: Caché uses this type of lock escalation if the following are true: (1) the class uses %CacheStorage (you can determine this from the [Catalog Details](#) in the Management Portal SQL schema display). (2) the class either does not specify an IDKey index, or specifies a single-property IDKey index. “E”-type lock escalation is described in the [LOCK](#) command in the *Caché ObjectScript Reference*.
- Traditional SQL lock escalation: The most likely reason why a class would not use “E”-type lock escalation is the presence of a multi-property IDKey index. In this case, each %Save increments the lock counter. This means if you do 1001 saves of a single object within a transaction, Caché will attempt to escalate the lock.

For both lock escalation strategies, you can determine the current systemwide lock threshold value using the `$$SYSTEM.SQL.GetLockThreshold()` method. The default is 1000. This systemwide lock threshold value is configurable:

- Using the `$$SYSTEM.SQL.SetLockThreshold()` method.
- Using the Management Portal. Go to **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Lock Threshold**. The default is 1000 locks. If you change this setting, any new process started after changing it will have the new setting.

You must have USE permission on the %Admin Manage Resource to change the lock threshold. Caché immediately applies any change made to the lock threshold value to all current processes.

One potential consequence of automatic lock escalation is a deadlock situation that might occur when an attempt to escalate to a table lock conflicts with another process holding a record lock in that table. There are several possible strategies to avoid this: (1) increase the lock escalation threshold so that lock escalation is unlikely to occur within a transaction. (2) substantially lower the lock escalation threshold so that lock escalation occurs almost immediately, thus decreasing the opportunity for other processes to lock a record in the same table. (3) apply a table lock for the duration of the transaction and do not perform record locks. This can be done at the start of the transaction by specifying **LOCK TABLE**, then **UNLOCK TABLE** (without the **IMMEDIATE** keyword, so that the table lock persists until the end of the transaction), then perform inserts with the %NOLOCK option.

Automatic lock escalation is intended to prevent overflow of the lock table. However, if you perform such a large number of inserts that a <LOCKTABLEFULL> error occurs, **INSERT** issues an SQLCODE -110 error.

For further details on transaction locking refer to [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

Row-Level Security

Caché row-level security permits **INSERT** to add a row even if the row security is defined so that you will not be permitted to subsequently access the row. To ensure that an **INSERT** does not prevent you from subsequent **SELECT** access to the

row, it is recommended that you perform the **INSERT** through a view that has a WITH CHECK OPTION. For further details, refer to [CREATE VIEW](#).

Microsoft Access

To use **INSERT** to add data to a Caché table using Microsoft Access, either mark the table RowID as private or define a unique index on one or more additional fields.

Embedded SQL and Dynamic SQL Examples

The following Embedded SQL example creates a new table SQLUser.MyKids. The examples that follow use **INSERT** to populate this table with data. After the **INSERT** examples, an example is provided to delete SQLUser.MyKids.

```
CreateTable
&sql(CREATE TABLE SQLUser.MyKids (
  KidName VARCHAR(16) UNIQUE NOT NULL,
  KidDOB INTEGER NOT NULL,
  KidPetName VARCHAR(16) DEFAULT 'no pet' )
IF SQLCODE=0 {
  WRITE !,"Table created" }
ELSEIF SQLCODE=-201 {WRITE !,"Table already exists"  QUIT}
ELSE {
  WRITE !,"CREATE TABLE failed. SQLCODE=",SQLCODE }
```

The following [Embedded SQL](#) example inserts a row with two field values (the third field, KidPetName, takes a default value). Note that the table schema name is supplied as a [schema search path](#) by the [#SQLCompile Path](#) macro directive:

```
EmbeddedSQLInsertByColName
#SQLCompile Path=Sample
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO MyKids (KidName,KidDOB) VALUES
('Molly',60000))
IF SQLCODE=0 {
  WRITE !,"Insert succeeded"
  WRITE !,"Row count=",%ROWCOUNT
  WRITE !,"Row ID=",%ROWID
  QUIT }
ELSEIF SQLCODE=-119 {
  WRITE !,"Duplicate record not written",!
  WRITE %msg,!
  QUIT }
ELSE {
  WRITE !,"Insert failed, SQLCODE=",SQLCODE }
```

The following [Embedded SQL](#) example inserts a row with three field values using the table's column order:

```
EmbeddedSQLInsertByColOrder
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO SQLUser.MyKids VALUES ('Josie','40100','Fido') )
IF SQLCODE=0 {
  WRITE !,"Insert succeeded"
  WRITE !,"Row count=",%ROWCOUNT
  WRITE !,"Row ID=",%ROWID
  QUIT }
ELSEIF SQLCODE=-119 {
  WRITE !,"Duplicate record not written",!
  WRITE %msg,!
  QUIT }
ELSE {
  WRITE !,"Insert failed, SQLCODE=",SQLCODE }
```

The following [Embedded SQL](#) example uses host variables to insert a row with two field values. The **INSERT** syntax used here specifies *column=value* pairs:

```

EmbeddedSQLInsertHostVars
#SQLCompile Path=Sample
NEW SQLCODE,%ROWCOUNT,%ROWID
SET x = "Sam"
SET y = "57555"
&sql(INSERT INTO MyKids SET KidName=:x,KidDOB=:y )
IF SQLCODE=0 {
  WRITE !,"Insert succeeded"
  WRITE !,"Row count=",%ROWCOUNT
  WRITE !,"Row ID=",%ROWID
  QUIT }
ELSEIF SQLCODE=-119 {
  WRITE !,"Duplicate record not written",!
  WRITE %msg,!
  QUIT }
ELSE {
  WRITE !,"Insert failed, SQLCODE=",SQLCODE }

```

The following [Embedded SQL](#) example uses a host variable array to insert a row with three field values. Array elements are numbered in column order. Note that user-supplied array values start with `myarray(2)`; the first array element corresponds to the RowID, which is automatically supplied and cannot be user-defined:

```

EmbeddedSQLInsertHostVarArray
#SQLCompile Path=Sample
NEW SQLCODE,%ROWCOUNT,%ROWID
SET myarray(2)="Deborah"
SET myarray(3)=60200
SET myarray(4)="Bowie"
&sql(INSERT INTO MyKids VALUES :myarray())
IF SQLCODE=0 {
  WRITE !,"Insert succeeded"
  WRITE !,"Row count=",%ROWCOUNT
  WRITE !,"Row ID=",%ROWID
  QUIT }
ELSEIF SQLCODE=-119 {
  WRITE !,"Duplicate record not written",!
  WRITE %msg,!
  QUIT }
ELSE {
  WRITE !,"Insert failed, SQLCODE=",SQLCODE }

```

The following [Dynamic SQL](#) example uses the `%SQL.Statement` class to insert a row with three field values. Note that the table schema name is supplied as a [schema search path](#) in the `%New()` method:

```

COSDynamicSQLInsert
SET x = "Noah"
SET y = "61000"
SET z = "Luna"
SET sqltext = "INSERT INTO MyKids (KidName,KidDOB,KidPetName) VALUES (?,?)"
SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
SET qStatus = tStatement.%Prepare(sqltext)
IF qStatus=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rtn = tStatement.%Execute(x,y,z)
IF rtn.%SQLCODE=0 {
  WRITE !,"Insert succeeded"
  WRITE !,"Row count=",rtn.%ROWCOUNT
  WRITE !,"Row ID=",rtn.%ROWID }
ELSEIF rtn.%SQLCODE=-119 {
  WRITE !,"Duplicate record not written",!,rtn.%Message
  QUIT }
ELSE {
  WRITE !,"Insert failed, SQLCODE=",rtn.%SQLCODE }

```

For further details, refer to the [Embedded SQL](#) and [Dynamic SQL](#) chapters in *Using InterSystems SQL*.

The following Embedded SQL example displays the inserted records, then deletes the `SQLUser.MyKids` table:

```

DisplayAndDeleteTable
SET myquery = "SELECT * FROM SQLUser.MyKids"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
&sql(DROP TABLE SQLUser.MyKids)
IF SQLCODE=0 {
  WRITE !,"Deleted table"
  QUIT }
ELSE {
  WRITE !,"Table delete failed, SQLCODE=",SQLCODE }

```

The following [Embedded SQL](#) example demonstrates the use of a host variable arrays. Note that with a host variable array, you can use a dynamic local array with an unspecified last subscript to pass an array of values to **INSERT** at runtime. For example:

```

NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO Sample.Employee VALUES :emp('profile',))
WRITE !,"SQL Error code: ",SQLCODE," Row Count: ",%ROWCOUNT

```

causes each field in the inserted "Employee" row to be set to:

```
emp("profile",col)
```

where "col" is the field's column number in the Sample.Employee table.

The following example shows how the results of a **SELECT** query can be used as the data input into an **INSERT** statement, supplying the data for multiple rows:

```

INSERT INTO StudentRoster (NAME,GPA,ID_NUM)
  SELECT FullName,GradeAvg,ID
  FROM temp WHERE SchoolYear = '2004'

```

See Also

- [INSERT OR UPDATE](#)
- [UPDATE](#)
- [DELETE](#)
- [SELECT](#)
- [CREATE TABLE](#)
- [JOIN](#)
- [SELECT](#)
- [VALUES](#)
- “[Modifying the Database](#)” chapter in *Using Caché SQL*
- “[Defining Tables](#)” chapter in *Using Caché SQL*
- “[Defining Views](#)” chapter in *Using Caché SQL*
- [Transaction Processing](#) in the “[Modifying the Database](#)” chapter of *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

INSERT OR UPDATE

Adds a new row or updates an existing row in a table.

```
INSERT OR UPDATE [%NOFPLAN] [restriction] [INTO] table
    SET column1 = scalar-expression1 {,column2 = scalar-expression2} ... |
    [ (column1{,column2} ...) ] VALUES (scalar-expression1 {,scalar-expression2}
    ...) |
    VALUES :array() |
    [ (column1{,column2} ...) ] query |
    DEFAULT VALUES
```

Arguments

<code>%NOFPLAN</code>	<i>Optional</i> — The <code>%NOFPLAN</code> keyword specifies that Caché will ignore the frozen plan (if any) for this operation and generate a new query plan. The frozen plan is retained, but not used. For further details, refer to Frozen Plans in <i>Caché SQL Optimization Guide</i> .
<code>restriction</code>	<i>Optional</i> — One or more of the following keywords, separated by spaces: <code>%NOLOCK</code> , <code>%NOCHECK</code> , <code>%NOINDEX</code> , <code>%NOTRIGGER</code> .
<code>table</code>	The name of the table or view on which to perform the insert operation. This argument may be a subquery. The <code>INTO</code> keyword is optional.
<code>column</code>	<i>Optional</i> — A column name or comma-separated list of column names that correspond in sequence to the supplied list of values. If omitted, the list of values is applied to all columns in column-number order.
<code>scalar-expression</code>	A scalar expression or comma-separated list of scalar expressions that supplies the data values for the corresponding <i>column</i> fields.
<code>:array()</code>	<i>Embedded SQL only</i> — A dynamic local array of values specified as a host variable . The lowest subscript level of the array must be unspecified. Thus <code>:myupdates()</code> , <code>:myupdates(5,)</code> , and <code>:myupdates(1,1,)</code> are all valid specifications.
<code>query</code>	A query's result set that supplies the data values for the corresponding <i>column</i> fields for one or more rows.

Description

The **INSERT OR UPDATE** statement is an extension of the [INSERT](#) statement (which it closely resembles):

- If the specified record does not exist, **INSERT OR UPDATE** performs an **INSERT**.
- If the specified record already exists, **INSERT OR UPDATE** performs an **UPDATE**. It updates the record with the specified field values. An update occurs even when the specified data is identical to the existing data.

INSERT OR UPDATE determines if a record exists by matching UNIQUE KEY field values to the existing data values. If a UNIQUE KEY violation occurs, **INSERT OR UPDATE** performs an update operation. Note that a UNIQUE KEY field value may not be a value explicitly specified in **INSERT OR UPDATE**; it may be the result of a column default value or a computed value.

INSERT OR UPDATE of a single record always sets the `%ROWCOUNT` variable to 1, and the `%ROWID` variable for the row that has been either inserted or updated.

An **INSERT OR UPDATE** statement combined with a **SELECT** statement can insert and/or update multiple table rows. For further details, refer to “[INSERT Query Results](#)” in the **INSERT** reference page.

INSERT OR UPDATE uses the same syntax, and generally has the same features and restrictions as the **INSERT** statement. Special considerations for **INSERT OR UPDATE** are described here. Unless otherwise stated here, refer to **INSERT** for details.

Privileges

INSERT OR UPDATE requires both **INSERT** and **UPDATE** privileges, as well as **SELECT** privilege. You must have these privileges either as table-level privileges or as column-level privileges.

IDKEY Fields

You can insert an **IDKEY** field value, but you cannot update an **IDKEY** field value. If the table has an **IDKEY** index and another unique key constraint, **INSERT OR UPDATE** matches these fields to determine whether to perform an insert or an update. If the other key constraint fails, this forces **INSERT OR UPDATE** to perform an update rather than an insert. However, if the specified **IDKEY** field values do not match the existing **IDKEY** field values, this update fails and generates an **SQLCODE -107** error, because the update is attempting to modify the **IDKEY** fields.

For example, the table **MyTest** is defined with four fields: **A**, **B**, **C**, **D**, with **IDKEY (A,B)** and **UNIQUE (C,D)** constraints. The table contains the following records:

```
Row 1: A=1, B=1, C=2, D=2
Row 2: A=1, B=2, C=3, D=4
```

You invoke `INSERT OR UPDATE ABC (A,B,C,D) VALUES (2,2,3,4)`. Because the **UNIQUE (C,D)** constraint failed, this statement cannot perform an insert. Instead, it attempts to update Row 2. The **IDKEY** for Row 2 is (1,2), so the **INSERT OR UPDATE** statement would attempt to change the field **A** value from 1 to 2. But you cannot change an **IDKEY** value, so the update fails with an **SQLCODE -107** error.

Counter Fields

When an **INSERT OR UPDATE** is executed, Caché initially assumes the operation will be an insert. Therefore, it increments by 1 the internal counters used to supply integers to **SERIAL** (%Library.Counter) fields. An insert uses these incremented counter values to assign integer values to these fields. If, however, Caché determines that the operation needs to be an update, **INSERT OR UPDATE** has already incremented the internal counters, but it does not assign these incremented integer values to counter fields. If the next operation is an insert, this results in a gap in the integer sequence for these fields. This is shown in the following example:

1. The internal counter value is 4. **INSERT OR UPDATE** increments the internal counter then inserts Row 5: internal counter=5, **SERIAL** field value=5.
2. **INSERT OR UPDATE** increments the internal counter then determines that it must perform an update on an existing row: internal counter=6, no change to field counters.
3. **INSERT OR UPDATE** increments internal counter then inserts a row: internal counter=7, **SERIAL** field value=7.

IDENTITY and RowID Fields

The effect of **INSERT OR UPDATE** on the assignment of **RowID** values depends on whether an **IDENTITY** field is present:

- If no **IDENTITY** field is defined for the table, an insert operation causes Caché to automatically assign the next sequential integer value to the **ID (RowID)** field. Update operations have no effect on subsequent inserts. Thus, **INSERT OR UPDATE** performs the same insert operation as **INSERT**.
- If an **IDENTITY** field is defined for the table, an **INSERT OR UPDATE** causes Caché to increment by 1 the internal counter used to supply integers to the **IDENTITY** field before determining if the operation will be an insert or an update. An insert operation assigns this incremented counter value to the **IDENTITY** field. If, however, Caché determines

that the **INSERT OR UPDATE** operation needs to be an update, it has already incremented the internal counter, but it does not assign these incremented integer value. If the next **INSERT OR UPDATE** operation is an insert, this results in a gap in the integer sequence for the **IDENTITY** field. The **RowID** field value is taken from the **IDENTITY** field value, resulting in a gap in the assignment of **ID (RowID)** integer values.

Examples

The following five examples: create a new table (`SQLUser.CaveDwellers`); use **INSERT OR UPDATE** to populate this table with data, use **INSERT OR UPDATE** to add new rows and update existing rows; use a `SELECT *` to display the data; and delete the table.

The following example uses **CREATE TABLE** to create a table with a unique field (`Num`):

```
CreateTable
  ZNSPACE "Samples"
  &sql(CREATE TABLE SQLUser.CaveDwellers (
    Num          INT UNIQUE,
    CaveCluster  CHAR(80) NOT NULL,
    Troglodyte   CHAR(50) NOT NULL,
    CONSTRAINT CaveDwellerPK PRIMARY KEY (Num))
  )
  IF SQLCODE=0 {WRITE !,"Table created" }
  ELSEIF SQLCODE=-201 {WRITE !,"Table already exists"}
  ELSE {WRITE !,"CREATE TABLE failed. SQLCODE=",SQLCODE }
```

The following example uses a class definition to define the same table, defining a unique key for `Num`:

```
Class SQLUser.CaveDwellers Extends %Persistent [
  DdlAllowed,Owner={UnknownUser},SqlRowIdPrivate,
  SqlTableName=CaveDwellers ]
{
  Property Num As %Integer;
  Property CaveCluster As %String(MAXLEN=80);
  Property Troglodyte As %String(MAXLEN=50);
  Index UniqueNumIdx On Num [ Type=index,Unique ];
}
```

```
SELECT * FROM SQLUser.CaveDwellers ORDER BY Num
```

Run the following two examples one or more times in any order. They will insert records 1 through 5. If record 4 already exists, **INSERT OR UPDATE** will update it. Use the `SELECT *` example to display the table data:

```
InsertOrUpdateIndividualRecords
  ZNSPACE "Samples"
  &sql(INSERT OR UPDATE INTO SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES
    (1,'Bedrock','Flintstone,Fred'))
  IF SQLCODE = 0 { SET rcount=%ROWCOUNT }
  &sql(INSERT OR UPDATE INTO SQLUser.CaveDwellers (Num,CaveCluster,Troglodyte) VALUES
    (4,'Bedrock','Flintstone,Wilma'))
  IF SQLCODE = 0 { SET rcount=rcount+%ROWCOUNT
    WRITE !,rcount," records inserted/updated" }
  ELSE { WRITE !,"Insert/Update failed, SQLCODE=",SQLCODE }
```

```
InsertOrUpdateWithQueryResults
  NEW SQLCODE,%ROWCOUNT,%ROWID
  &sql(INSERT OR UPDATE SQLUser.CaveDwellers
    (Num,CaveCluster,Troglodyte)
    SELECT %ID,Home_City,Name
    FROM Sample.Person
    WHERE %ID BETWEEN 2 AND 5)
  IF SQLCODE=0 {
    WRITE !,"Insert/Update succeeded"
    WRITE !,%ROWCOUNT," records inserted/updated"
    WRITE !,"Row ID=",%ROWID }
  ELSE {
    WRITE !,"Insert/Update failed, SQLCODE=",SQLCODE }
```

The following example deletes the table:

```
DeleteTable
  ZNSPACE "Samples"
  &sql(DROP TABLE SQLUser.CaveDwellers)
  IF SQLCODE=0 {WRITE !,"Table deleted" }
  ELSEIF SQLCODE=-30 {WRITE !,"Table does not exist"}
  ELSE {WRITE !,"DROP TABLE failed. SQLCODE=",SQLCODE }
```

See Also

- [CREATE TABLE](#)
- [INSERT](#)
- [UPDATE](#)
- “[Modifying the Database](#)” chapter in *Using Caché SQL*
- “[Defining Tables](#)” chapter in *Using Caché SQL*
- “[Defining Views](#)” chapter in *Using Caché SQL*
- [Transaction Processing](#) in the “[Modifying the Database](#)” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

INTO

A **SELECT** clause that specifies the storing of selected values in host variables.

```
INTO :hostvar1 [,:hostvar2]...
```

Arguments

<code>:hostvar1</code>	A host variable that has been declared in the host language. When specified in an INTO clause, the variable name is preceded by a colon (:). A host variable can be a local variable (unsubscripted or subscripted) or an object property. You can specify multiple variables as a comma-separated list, as a single subscripted array variable, or a combination of a comma-separated list and a single subscripted array variable.
------------------------	---

Description

The **INTO** clause and host variables are only used in [Embedded SQL](#). They are not used in [Dynamic SQL](#). In Dynamic SQL, similar functionality for output variables is provided by the %SQL.Statement class.

An **INTO** clause can be used in a [SELECT](#), [DECLARE](#), or [FETCH](#) statement. The **INTO** clause is identical for all three statements; examples on this page all refer to the **SELECT** statement. For usage with **DECLARE** and **FETCH**, refer to “[SQL Cursors](#)” in the “Using Embedded SQL” chapter of *Using Caché SQL*.

The **INTO** clause uses the values retrieved (or calculated) in the **SELECT** *select-item* list to set corresponding host variables, making these returned data values available to ObjectScript. In a **SELECT** the optional **INTO** clause appears after the *select-item* list and before the **FROM** clause.

Host Variables

A host variable can contain only a single value. Therefore, a **SELECT** in embedded SQL only retrieves one row of data. This defaults to the first row of the table. You can, of course, retrieve data from some other row of the table by limiting the eligible rows using a **WHERE** condition.

In embedded SQL you can return data from multiple rows by declaring a cursor and then issuing a **FETCH** for each successive row. The **INTO** clause host variables can be specified in the **DECLARE** query or specified in the **FETCH**.

INTO clause host variables can be specified in either of two ways (or a combination of both):

- A [host variable list](#), consisting of a comma-separated list of host variables, one for each *select-item*.
- A [host variable array](#), consisting of a single subscripted host variable.

For important restrictions on the use of host variable values in the containing program, refer to the [Host Variables](#) section of the “Embedded SQL” chapter of *Using Caché SQL*.

Note: If the host language declares data types for variables, all host variables must be declared in the host language before invoking the **SELECT** statement. The data types of the retrieved field values must match the host variable declarations. (ObjectScript does not declare data types for variables.)

Using a Host Variable List

The following rules apply when you specify a host variable list in the **INTO** clause:

- The number of host variables in the **INTO** clause must match the number of fields specified in the *select-item* list. If the number of selected fields and host variables differs, SQL returns a “cardinality mismatch” error.

- Selected fields and host variables are matched by relative position. Therefore, the corresponding items in these two lists must appear in the same sequence.
- The listed host variables may be any combination of unsubscripted or subscripted variables.
- A listed host variable can return an aggregate value (such as a count, sum, or average) or a function value.
- A listed host variable can return %CLASSNAME and %TABLENAME values.
- Listed host variables can return field values from a SELECT involving multiple tables, or return values from a SELECT with no FROM clause.

The following example selects four fields into a list of four host variables. The host variables in this example are subscripted:

```
&sql(SELECT %ID,Home_City,Name,SSN
      INTO :mydata(1),:mydata(2),:mydata(3),:mydata(4)
      FROM Sample.Person
      WHERE Home_State='MA' )
IF SQLCODE=0 {
  FOR i=1:1:15 {
    IF $DATA(mydata(i)) {
      WRITE "field ",i," = ",mydata(i),! }
    }
  }
ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

For further examples refer to [Host Variable List Examples](#), below.

Using a Host Variable Array

A host variable array uses a single subscripted variable to contain all of the selected field values. This array is populated according to the order of field definition in the table, not the order of fields in the *select-item* list.

The following rules apply when using a host variable array in the **INTO** clause:

- The fields specified in the *select-item* list are selected into subscripts of a single host variable. Therefore, you do not have to match the number of items in the *select-item* list with the host variable count.
- The host variable subscripts are populated by the corresponding field position in the table definition. For example, the 6th field, as defined in the table definition, corresponds to mydata(6). All subscripts that do not correspond to a specified *select-item* remain undefined. The order of the items in the *select-item* has no effect on how subscripts are populated.
- A host variable array can only return field values from a single table.
- A host variable array can only return field values. It cannot return an aggregate value (such as a count, sum, or average), a function value, or a %CLASSNAME or %TABLENAME value. (You can return these by specifying a host variable argument that combines host variable list items with the host variable array.)

The following example selects four fields into a host variable array:

```
&sql(SELECT %ID,Home_City,Name,SSN
      INTO :mydata()
      FROM Sample.Person
      WHERE Home_State='MA' )
IF SQLCODE=0 {
  FOR i=0:1:15 {
    IF $DATA(mydata(i)) {
      WRITE "field ",i," = ",mydata(i),! }
    }
  }
ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

For further examples refer to [Host Variable Array Examples](#), below.

For further details, refer to “[Host Variable as a Subscripted Array](#)” in the “Using Embedded SQL” chapter of *Using Caché SQL*.

Host Variable List Examples

The following Embedded SQL example selects three fields from the first record in the table (Embedded SQL always retrieves a single record), and uses **INTO** to set three corresponding unsubscripted host variables. These variables are then used by the ObjectScript **WRITE** commands. It is considered good program practice to immediately test the **SQLCODE** variable upon returning from Embedded SQL. If **SQLCODE** is not equal to 0, the values of output host variables are indeterminate.

```
WRITE !,"Going to get the first record"
&sql(SELECT Home_State, Name, Age
      INTO :state, :name, :age
      FROM Sample.Person)
IF SQLCODE=0 {
  WRITE !," Name=",name
  WRITE !," Age=",age
  WRITE !," Home State=",state }
ELSE {
  WRITE !,"SQL error ",SQLCODE }
```

The following Embedded SQL example passes a host variable (today) into the **SELECT** statement, where a calculation results in the **INTO** clause variable value (:tomorrow). This host variable is passed out to the containing program. This SQL query does not require a **FROM** clause.

```
SET today=$HOROLOG
&sql(SELECT :today+1
      INTO :tomorrow )
IF SQLCODE=0 {
  WRITE !,"Tomorrow is: ",$ZDATE(tomorrow) }
ELSE {
  WRITE !,"SQL error ",SQLCODE }
```

For restrictions on the use of input and output host variable values, refer to the [Host Variables](#) section of the “Embedded SQL” chapter of *Using Caché SQL*.

The following Embedded SQL example returns aggregate values. It uses the **COUNT** aggregate function to count the records in a table and **AVG** to average the **Salary** field values. The **INTO** clause returns these values to ObjectScript as two subscripted host variables:

```
WRITE !,"Counting the records"
&sql(SELECT COUNT(*),AVG(Salary)
      INTO :agg(1),:agg(2)
      FROM Sample.Employee)
IF SQLCODE=0 {
  WRITE !,"Total Employee records= ",agg(1)
  WRITE !,"Average Employee salary= ",agg(2) }
ELSE {
  WRITE !,"SQL error ",SQLCODE }
```

The following Embedded SQL example returns field values from a row resulting from the join of two tables. You must use a host variable list when returning fields from more than one table:

```
&sql(SELECT P.Name,E.Title,E.Name,P.%TABLENAME,E.%TABLENAME
      INTO :name(1),:title,:name(2),:ptname,:etname
      FROM Sample.Person AS P LEFT JOIN
           Sample.Employee AS E ON E.Name %STARTSWITH 'B'
      WHERE P.Name %STARTSWITH 'A')
IF SQLCODE=0 {
  WRITE ptname," = ",name(1),!
  WRITE etname," = ",title,!
  WRITE etname," = ",name(2) }
ELSE {
  WRITE !,"SQL error ",SQLCODE }
```

Host Variable Array Examples

The following two Embedded SQL examples use a host variable array to return the non-hidden data field values from a row. In these examples **%ID** is specified in the *select-item* list, because, by default, **SELECT *** does not return the **RowId** (though it does for **Sample.Person**); the **RowId** is always field 1. Note in **Sample.Person** fields 4 and 9 can take **NULL**, field 5 is not a data field (it references **Sample.Address**), and field 10 is hidden.

The first example returns a specified number of fields (*firstflds*); hidden and non-data fields are included in this count, though not displayed. Using *firstflds* would be appropriate when returning a row from a table with many fields. Note that this example can return Field 0, which is the parent reference. Sample.Person is not a child table, so *tflds(0)* is undefined:

```
&sql(SELECT *,%ID INTO :tflds()
      FROM Sample.Person )
IF SQLCODE=0 {
  SET firstflds=14
  FOR i=0:1:firstflds {
    IF $DATA(tflds(i)) {
      WRITE "field ",i," = ",tflds(i),! }
    } }
ELSE {WRITE "SQLCODE error=",SQLCODE,! }
```

The second example returns all the non-hidden data fields in Sample.Person. Note that this example does not attempt to return Field 0, the parent reference, because in Sample.Person *tflds(0)* is undefined, and would therefore generate an <UNDEFINED> error:

```
&sql(SELECT *,%ID INTO :tflds()
      FROM Sample.Person )
IF SQLCODE=0 {
  SET x=1
  WHILE x !="" {
    WRITE "field ",x," = ",tflds(x),!
    SET x=$ORDER(tflds(x)) }
}
ELSE { WRITE "SQLCODE error=",SQLCODE,! }
```

The following Embedded SQL example combines a comma-separated host variable list (for non-field values) and a host variable array (for field values):

```
&sql(SELECT %TABLENAME,Name,Age,AVG(Age)
      INTO :tname,:tflds(),:ageavg
      FROM Sample.Person
      WHERE Age > 50 )
IF SQLCODE=0 {
  WRITE "Table name is = ",tname,!
  FOR i=0:1:25 {
    IF $DATA(tflds(i)) {
      WRITE "field ",i," = ",tflds(i),! }
    }
  WRITE "Average age is = ",ageavg,! }
ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

See Also

- [SELECT](#), [DECLARE](#), [FETCH](#) statements
- [VALUES](#) clause
- “[Host Variables](#)” in the “Using Embedded SQL” chapter of *Using Caché SQL*
- ObjectScript: [SET](#) command

%INTRANSACTION

Shows transaction state.

```
%INTRANSACTION
%INTRANS
```

Arguments

None

Description

The **%INTRANSACTION** statement sets **SQLCODE** to indicate the transaction state:

- **SQLCODE=0** if currently in a transaction.
- **SQLCODE=100** if not in a transaction.

%INTRANSACTION returns **SQLCODE=0** while a transaction is in progress. This transaction can be an SQL transaction initiated by **START TRANSACTION** or **SAVEPOINT**. It can also be an ObjectScript transaction initiated by **TSTART**.

Transaction nesting has no effect on **%INTRANSACTION**. **SET TRANSACTION** has no effect on **%INTRANSACTION**.

You can also determine transaction state using **\$TLEVEL**. **%INTRANSACTION** only indicates whether a transaction is in progress. **\$TLEVEL** indicates both whether a transaction is in progress and the current number of transaction levels.

Examples

The following embedded SQL example shows how **%INTRANSACTION** sets **SQLCODE**:

```
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "Before %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "SetTran %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(START TRANSACTION)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "StartTran %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(SAVEPOINT a)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "Savepoint %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(ROLLBACK TO SAVEPOINT a)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "Rollback to Savepoint %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL,!
&sql(COMMIT)
NEW SQLCODE
&sql(%INTRANSACTION)
WRITE "After Commit %INTRANS SQLCODE=",SQLCODE," TL=", $TLEVEL
```

See Also

- [COMMIT ROLLBACK SAVEPOINT SET TRANSACTION START TRANSACTION \\$TLEVEL](#)
- [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

JOIN

A **SELECT** subclause that creates a table based on the data in two tables.

```

table1 [[AS] t-alias] CROSS JOIN table2 [[AS] t-alias] |
table1 [[AS] t-alias] , table2 [[AS] t-alias]

table1 [[AS] t-alias]
NATURAL [INNER] JOIN |
NATURAL LEFT [OUTER] JOIN |
NATURAL RIGHT [OUTER] JOIN |
table2 [[AS] t-alias]

table1 [[AS] t-alias]
[INNER] JOIN |
LEFT [OUTER] JOIN |
RIGHT [OUTER] JOIN |
FULL [OUTER] JOIN
table2 [[AS] t-alias]
ON condition-expression

table1 [[AS] t-alias]
[INNER] JOIN |
LEFT [OUTER] JOIN |
RIGHT [OUTER] JOIN |
table2 [[AS] t-alias]
USING (identifier-commalist)

```

(The above join syntax is used in the **SELECT** statement **FROM** clause. Other symbolic join syntax can be used in other **SELECT** statement clauses.)

Description

A join is an operation that combines two tables to produce a joined table, optionally subject to one or more restrictive conditions. Every row of the new table must satisfy the restrictive condition(s). Joins provide the means of linking data in one table with data in another table and are frequently used in defining reports and queries.

There are several syntactical forms for representing joins. The preferred form is specifying an explicit join expression in a **SELECT** statement as part of the **FROM** clause. A **FROM** clause join expression can contain multiple joins.

Note: Caché SQL also supports implicit joins using arrow syntax (\rightarrow) in the **SELECT** statement *select-item* list, **WHERE** clause, **ORDER BY** clause, and elsewhere. An implicit join is specified to perform a left outer join of a table with a field from another table; an explicit join is specified to join two tables. This implicit join syntax can be a useful substitute for explicit join syntax, or appear in the same query with explicit join syntax. There are, however some important restrictions on combining arrow syntax with explicit join syntax. These restrictions are described below. For information on using arrow syntax, refer to [Implicit Joins](#) in *Using Caché SQL*.

Caché uses complex optimization algorithms to maximize performance of join operations. It does not, necessarily, join tables in the order in which they are specified. Instead, the SQL optimizer determines the table join order based on Tune Table data for each table (among other factors). It is therefore important that [Tune Table](#) be run against a table before that table is used in complex SQL queries. For further details on query optimization, refer to “[Performance with Multiple Joins and Implicit Joins](#)” and the [Optimizing Query Performance](#) chapter of the *SQL Optimization Guide*.

In most cases, the SQL optimizer strategy provides optimal results. However, Caché also provides join optimization keywords such as **%FIRSTTABLE**, **%INORDER** and **%FULL** that you can use immediately after the **FROM** keyword to override the default optimization strategy for a specific query. For a description of these optimization keywords, refer to “[Query Optimization Options](#)” in the **FROM** clause documentation.

JOIN Definitions

Caché supports many different syntactical forms of **JOIN**. However, these many formulations refer to the following five types of joins.

ANSI Join Syntax	Syntactical Equivalents
CROSS JOIN	Same as symbolic representation: table1,table2 (a list of tables separated by commas) in the FROM clause.
INNER JOIN	Same as JOIN. Symbolic representation: "=" (in a WHERE clause).
LEFT OUTER JOIN	Same as LEFT JOIN. The symbolic representation: "=*" (in a WHERE clause) has been deprecated and should not be used in new code. Arrow syntax (->) also performs a left outer join.
RIGHT OUTER JOIN	Same as RIGHT JOIN. The symbolic representation: "*=" (in a WHERE clause) has been deprecated and should not be used in new code.
FULL OUTER JOIN	Same as FULL JOIN.

Unless otherwise indicated, all join syntax is specified in the FROM clause.

- A **CROSS JOIN** is a join that crosses every row of the first table with every row of the second table. This results in a Cartesian product, a large, logically comprehensive table with much data duplication. Usually this join is performed by providing a comma-separated list of tables in the FROM clause, then using the WHERE clause to specify restrictive conditions. The %INORDER or %STARTTABLE optimization keyword cannot be used with a cross join. Attempting to do so results in an SQLCODE -34 error. For further details on join optimization keywords, refer to the [FROM](#) clause.
- An **INNER JOIN** is a join that links rows of the first table with rows of a second table, excluding any row in the first table that finds no corresponding row in the second table.
- A **LEFT OUTER JOIN** and a **RIGHT OUTER JOIN** are in most respects functionally identical (with reversed syntax), and thus are frequently referred to collectively as *one-way outer joins*. A one-way outer join is a join that links rows of the first (source) table with rows of a second table, including all rows from the first table even if there is no match in the second table. This results in a table in which some fields of the first (source) table may be paired with NULL data.

When specifying a one-way outer join, the order in which you name the tables in the FROM clause is very important. For a **LEFT OUTER JOIN**, the first table you specify is the source table for the join. For a **RIGHT OUTER JOIN** the second table you specify is the source table for the join.

- A **FULL OUTER JOIN** is a join that combines the results of performing both a **LEFT OUTER JOIN** and a **RIGHT OUTER JOIN** on the two tables. It includes all rows found in either the first table or the second table, and fills in NULLs for missing matches on either side.

CROSS JOIN Considerations

The explicit use of the JOIN keyword has higher precedence than specifying a cross join using comma syntax. Caché thus interprets t1,t2 JOIN t3 as t1,(t2 JOIN t3). Earlier versions of Caché did not support this syntax precedence; join syntax was parsed in left-to-right order, so that t1,t2 JOIN t3 was interpreted as (t1,t2) JOIN t3. To maintain left-to-right parsing, this join must be re-specified as t1 CROSS JOIN t2 JOIN t3.

You cannot perform a cross join involving a local table and an [external table linked through an ODBC or JDBC gateway connection](#). For example, FROM Sample.Person,Mylink.Person. Attempting to do so results in SQLCODE -161: "References to an SQL connection must constitute a whole subquery". To perform this cross join you must specify the linked table as a subquery. For example, FROM Sample.Person,(SELECT * FROM Mylink.Person).

NATURAL Joins

A NATURAL JOIN is an INNER JOIN, LEFT OUTER JOIN, or RIGHT OUTER JOIN prefixed with the NATURAL keyword. Prefixing a join with the word NATURAL specifies that you are joining on all the columns of the two tables that have the same name. Because a NATURAL join automatically performs an equality condition on all columns having the same name, it is not possible to specify an ON clause or a USING clause. Attempting to do so results in an SQLCODE -25 error.

Only simple base table references (not views or subqueries) are supported for either operand of a NATURAL join.

A NATURAL join can only be specified as the first join within a join expression.

A NATURAL join does not merge columns with the same name.

A FULL JOIN cannot be prefixed with the NATURAL keyword. Attempting to do so results in an SQLCODE -94 error.

ON Clause

An INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, or FULL OUTER JOIN may have an ON clause. An ON clause contains one or more condition expressions used to limit the values returned by the join operation. A join with an ON clause can be specified anywhere within a join expression. A join with an ON clause can specify tables, views, or subqueries for either operand of the join.

The ON clause consists of one or more condition expression predicates. These include most of the [predicates](#) supported by Caché SQL. However, you cannot use a collection predicate to limit a join operation; the Caché SQL collection predicates are **FOR SOME %ELEMENT**, **%CONTAINS**, and **%CONTAINSTERM**.

You can associate multiple condition expressions using AND, OR, and NOT logical operators. AND takes precedence over OR. Parentheses can be used to nest and group condition expressions. Unless grouped by parentheses, predicates using the same logical operator are executed in strict left-to-right order.

An ON clause has the following restrictions:

- A join with an ON clause can only use ANSI join keyword syntax.
- A join with an ON clause cannot take the NATURAL keyword prefix. This results in an SQLCODE -25 error.
- A join with an ON clause cannot take a USING clause. This results in an SQLCODE -25 error.
- An ON clause cannot include =* or *= symbolic syntax. This results in an SQLCODE -68 error.
- An ON clause cannot include arrow syntax (->). This results in an SQLCODE -67 error. For a description of arrow syntax, refer to [Implicit Joins](#) in *Using Caché SQL*.
- An ON clause can only reference tables explicitly specified in the ANSI keyword JOIN operation. Other tables specified in the FROM clause may not be referenced in the ON clause. This results in an SQLCODE -23 error.
- An ON clause can only reference columns that are in the operands of the JOIN. Syntax precedence in multiple joins can cause the ON clause to fail. For example, the query `SELECT * FROM t1,t2 JOIN t3 ON t1.p1=t3.p3` fails because t1 and t3 are not operands of a join; t1 joins with the result set of t2 JOIN t3. Either of the following changes in syntax result in the successful execution of this query: `SELECT * FROM t1 CROSS JOIN t2 JOIN t3 ON t1.p1=t3.p3` or `SELECT * FROM t2,t1 JOIN t3 ON t1.p1=t3.p3`.
- OUTER JOIN with an ON clause restriction. If all the conditions affecting a table use comparisons that may pass null values, and that table is itself a target of an outer join, this can result in an SQLCODE -94 error: `Unsupported usage of OUTER JOIN`. The following is a LEFT OUTER JOIN example of this type of invalid join:

```
SELECT *
FROM Table1
  LEFT JOIN Table2 ON Table1.k = Table2.k
  LEFT JOIN Table3 ON COALESCE(Table1.k,Table2.k) = Table3.k
```

Similar examples using FULL OUTER JOIN or RIGHT OUTER JOIN also have this restriction.

ON Clause Indexing

For optimal performance, fields referenced in an ON clause should (in most cases) have an associated index. An ON clause can use an existing index that satisfies only some of the join conditions. An ON clause specifying conditions on multiple fields can use an index containing only a subset of those fields as subscripts to partially satisfy the join; Caché will test the join condition on the remaining fields directly from the table.

The [collation type](#) of a field referenced in an ON clause should match the collation type that it has in the corresponding index. A collation type mismatch can cause an index to not be used. However, if a join condition is on a %EXACT field value, but only an index on the collated field value is available, Caché can use that index to limit the rows to be checked for the exact value. For further details on collation type matching, refer to [Index Collation](#) in the “Defining and Building Indices” chapter of *Caché SQL Optimization Guide*.

In very specific situations you may wish to prevent the use of an index for an ON clause condition by prefacing it with the %NOINDEX keyword. For further details on indices and performance, refer to the [Index Analyzer](#) and [Index Optimization Options](#) in the *Caché SQL Optimization Guide*.

USING Clause

An INNER JOIN, LEFT OUTER JOIN, or RIGHT OUTER JOIN may have a USING clause. Only simple base table references (not views or subqueries) are supported for either operand of a join with a USING clause. A join with a USING clause can only be specified as the first join within a join expression. A join with a USING clause cannot take the NATURAL keyword prefix, or an ON clause.

A USING clause lists one or more column names, separated by commas and enclosed within parentheses. The parentheses are required. Duplicate column names are ignored. A USING clause does not merge columns with the same name.

A USING clause is a brief way to represent the equality conditions expressed in an ON clause. Thus: `t1 INNER JOIN t2 USING (a,b)` is equivalent to `t1 INNER JOIN t2 ON t1.a=t2.a AND t1.b=t2.b`

One-Way Outer Joins

Caché supports one-way outer joins: LEFT OUTER JOIN and RIGHT OUTER JOIN.

With standard "inner" joins, when rows of one table are linked with rows of a second table, a row in the first table that finds no corresponding row in the second table is excluded from the output table.

With one-way outer joins, all rows from the first table are included in the output table even if there is no match in the second table. With one-way outer joins, the first table pulls relevant information out of the second table but never sacrifices its own rows for lack of a match in the second table.

For example, if a query lists Table1 first and creates a left outer join, then it should be able to see all the rows in Table1 even if they don't have corresponding records in Table2.

When specifying a one-way outer join, the order in which you name the tables in the FROM clause is very important. For a left outer join, the first table you specify is the source table for the join. For a right outer join the second table you specify is the source table for the join. For this reason, the %INORDER or %STARTTABLE optimization keyword cannot be used with a right outer join. The following syntax is contradictory and results in an SQLCODE -34 error: `FROM %INORDER table1 RIGHT OUTER JOIN table2 ON...` For further details on join optimization keywords, refer to the [FROM clause](#).

Outer Join Syntax

Caché supports three formats for representing outer joins:

1. The ANSI standard syntax: LEFT OUTER JOIN and RIGHT OUTER JOIN. SQL Standard syntax puts the outer join in the FROM clause of the SELECT statement, rather than the WHERE clause, as shown in the following example:

```
FROM tbl1 LEFT OUTER JOIN tbl2 ON (tbl1.key = tbl2.key)
```

2. The ODBC Specification outer join extension syntax, using the escape-syntax `{oj join-expression }`, where *join-expression* is any ANSI standard join syntax.
3. Symbolic outer join extension syntax, using a condition such as `A=*B` in the WHERE clause. A left outer join is specified using the symbol `=*` in place of `=` in the WHERE clause. A right outer join is specified using the symbol `*=` in place of `=` in the WHERE clause. (Note that this is the reverse of the syntax used by Microsoft SQL Server and Sybase.)

Note: Use of symbolic outer join syntax (`=*` and `*=`) is strongly discouraged, and cannot be used with an ON clause. Use ANSI standard syntax: LEFT OUTER JOIN and RIGHT OUTER JOIN. Further restrictions on symbolic outer join syntax are listed below.

While the three outer join formats are interchangeable and can be mixed, we strongly recommend the use of ANSI standard syntax whenever possible, as it is the only one compatible with ODBC (and portable to the latest Microsoft products). Additionally, ANSI standard syntax can specify many operations not specifiable in principle with symbolic syntax. Finally, InterSystems has no intention of supporting new features, enhanced validation, and optimizer improvements for the older, symbolic outer join syntax.

Symbolic Syntax (`=*`, `*=`) Outer Join Restrictions

For a symbolic syntax outer join in the WHERE clause, the following operand values may be used:

- A column of a base table. No restrictions apply to specifying a base table column on either side of the outer join. However, you cannot specify an expression using a base table column (such as `substring(field1, 4, 3)`) on either side of a symbolic syntax outer join. Expressions are permitted with ANSI standard syntax.
- A column of a streamed view or subquery. A streamed view is a view that contains a DISTINCT, GROUP BY, or UNION keyword, or that has columns generated using aggregate functions. No restrictions apply to specifying a streamed view column on either side of the outer join.
- A column of a view or subquery that isn't streamed must expand to a base table column if it is used on the left side of a left outer join (`=*`) or the right side of a right outer join (`*=`).
- For a left outer join (`=*`), the right operand should not contain [arrow syntax](#). A construction such as `WHERE a =* b->c` is not supported. Arrow syntax is supported for the left operand.
- For a right outer join (`*=`), the left operand should not contain [arrow syntax](#). A construction such as `WHERE a->b *= c` is not supported. Arrow syntax is supported for the right operand.
- When using symbolic syntax, multiple condition expressions may only be joined with the AND logical operator. The OR and NOT logical operators may not be used.

The following is a valid query using symbolic syntax for a left outer join:

```
SELECT a.Name, a.Age, b.Name
FROM Sample.Person AS a, Sample.Employee AS b
WHERE a.Name =* b.Name AND a.Name %STARTSWITH 'G'
```

If a query contains both a FROM clause containing an ANSI standard outer join and a WHERE clause containing a symbolic syntax outer join, and these two joins are in conflict, Caché performs an inner join. Using both ANSI standard syntax and symbolic syntax in the same query is strongly discouraged.

Null Padding

A one-way outer join performs null padding. This means that if a row of the source table has a NULL value for the merged column, a null value is returned for the corresponding field from the non-source table.

The left outer join condition is expressed by the following syntax:

```
A LEFT OUTER JOIN B ON A.x=B.y
```

This specifies that every row in A is returned. For each A row returned, if there is a B row such that $A.x=B.y$, all of the corresponding B values are also returned.

If there is no B row such that $A.x=B.y$, null padding causes all B values for that A row to return as null.

For example, consider the Patient table that contains information about patients, including a field Patient.DocID specifying and ID code for the patient's primary doctor. Some patients in the database do not have a primary doctor, so for those patient records the Patient.DocID field is NULL. Now, we perform a join between the Patient table and the Doctor table to generate a table of patient names and corresponding doctor names.

The following example is an INNER JOIN.

```
SELECT Patient.PName,Doctor.DName
FROM Patient INNER JOIN Doctor
ON Patient.DocID=Doctor.DocID
```

An INNER JOIN does not perform null padding. Therefore, no patient name without a corresponding doctor name is returned.

A one-way outer join does perform null padding. Therefore, a patient name without a corresponding doctor name returns a NULL for Doctor.DName.

```
SELECT Patient.PName,Doctor.DName
FROM Patient LEFT OUTER JOIN Doctor
ON Patient.DocID=Doctor.DocID
```

Order of Operations

One-way outer join conditions, including the necessary null padding, are applied before other conditions. Therefore, a condition in the **WHERE** clause that cannot be satisfied by a null-padded value (for example, a range or equality condition on a field in B) effectively converts the one-way outer join of A and B into a regular join (an inner join).

For example, if you add the clause "WHERE Doctor.Age < 45" to the two "Patient" table queries above, it makes them equivalent. However, if you add the clause "WHERE Doctor.Age < 45 OR Doctor.Age IS NULL", it preserves the difference between the two queries.

Mixing Outer and Inner Joins

Caché supports all syntax of mixed inner joins and outer joins in any order.

Performance with Multiple Joins and Implicit Joins

By default, the query optimizer sequences multiple join operations in its best estimation of the optimal sequence. This is not necessarily the join sequence order that you specified in the query. You can specify the **%INORDER**, **%FIRSTTABLE**, or **%STARTTABLE** query optimization option in the **FROM** clause to explicitly specify the order in which the tables are joined.

The query optimizer may perform subquery flattening, converting certain subqueries to explicit joins. This substantially improves join performance when the number of subqueries is small. When the number of subqueries is more than one or two, subquery flattening may, in some cases, actually slightly degrade performance. You can specify the **%NOFLATTEN** query optimization option in the **FROM** clause to explicitly specify that subquery flattening should not be performed.

The query optimizer only performs subquery flattening when the total number of joins in a query, after subquery flattening, would not exceed 15 joins. Specifying more than 15 joins, when some of those joins are **implicit joins** or joined subqueries, can result in a significant degradation in query performance.

Examples

The following examples display the results of the **JOIN** operations performed on Table1 and Table2.

Table1		Table2	
Column1	Column2	Column1	Column3
aaa	bbb	ggg	hhh
ccc	ccc	xxx	zzz
xxx	yyy		
hhh	zzz		

CROSS JOIN Example

The statement:

```
SELECT * FROM Table1 CROSS JOIN Table2
```

yields the table:

Column1	Column2	Column1	Column3
aaa	bbb	ggg	hhh
aaa	bbb	xxx	zzz
ccc	ccc	ggg	hhh
ccc	ccc	xxx	zzz
xxx	yyy	ggg	hhh
xxx	yyy	xxx	zzz
hhh	zzz	ggg	hhh
hhh	zzz	xxx	zzz

NATURAL JOIN Example

The statement:

```
SELECT * FROM Table1 NATURAL JOIN Table2
```

yields the table

Column1	Column2	Column1	Column3
xxx	yyy	xxx	zzz

Note that the Caché implementation of NATURAL JOIN does not merge columns with the same name.

INNER JOIN with an ON Clause Example

The statement:

```
SELECT * FROM Table1 INNER JOIN Table2
ON Table1.Column1=Table2.Column3
```

yields the table:

Column1	Column2	Column1	Column3
hhh	zzz	ggg	hhh

INNER JOIN with a USING Clause Example

The statement:

```
SELECT * FROM Table1 INNER JOIN Table2
USING (Column1)
```

yields the table:

Column1	Column2	Column1	Column3
xxx	yyy	xxx	zzz

Note that the Caché implementation of a USING clause does not merge columns with the same name.

LEFT OUTER JOIN Example

The statement:

```
SELECT * FROM Table1 LEFT OUTER JOIN Table2
  ON Table1.Column1=Table2.Column3
```

yields the table:

Column1	Column2	Column1	Column3
aaa	bbb	null	null
ccc	ccc	null	null
xxx	yyy	null	null
hhh	zzz	ggg	hhh

RIGHT OUTER JOIN Example

The statement:

```
SELECT * FROM Table1 RIGHT OUTER JOIN Table2
  ON Table1.Column1=Table2.Column3
```

yields the table:

Column1	Column2	Column1	Column3
hhh	zzz	ggg	hhh
null	null	xxx	zzz

FULL OUTER JOIN Example

The statement:

```
SELECT * FROM Table1 FULL OUTER JOIN Table2
  ON Table1.Column1=Table2.Column3
```

yields the table:

Column1	Column2	Column1	Column3
aaa	bbb	null	null
ccc	ccc	null	null
xxx	yyy	null	null
hhh	zzz	ggg	hhh
null	null	xxx	zzz

See Also

- [SELECT](#) statement, [FROM](#) clause, [ORDER BY](#) clause, [WHERE](#) clause
- [ALTER TABLE](#), [CREATE TABLE](#), [DROP TABLE](#)
- [INSERT](#), [UPDATE](#)
- “[Defining Tables](#)” chapter in *Using Caché SQL*
- “[Querying the Database](#)” chapter in *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

LOCK

Locks a table.

```
LOCK [TABLE] tablename IN EXCLUSIVE MODE [WAIT seconds]
LOCK [TABLE] tablename IN SHARE MODE [WAIT seconds]
```

Arguments

<i>tablename</i>	The name of the table to be locked. <i>tablename</i> must be an existing table. A <i>tablename</i> can be qualified (schema.table), or unqualified (table). An unqualified table name takes the system-wide default schema name . A schema search path is ignored.
IN EXCLUSIVE MODE / IN SHARE MODE	The IN EXCLUSIVE MODE keyword phrase creates a regular Caché lock. The IN SHARE MODE keyword phrase creates a shared Caché lock.
<i>seconds</i>	<i>Optional</i> — An integer specifying the number of seconds to attempt to acquire the lock before timing out. If omitted, the system default timeout is applied.

Description

LOCK and **LOCK TABLE** are synonymous.

The **LOCK** command explicitly locks an SQL table. This table must be an existing table for which you have the necessary privileges. If *tablename* is a nonexistent table, **LOCK** fails with a compile error. If *tablename* is a temporary table, the command completes successfully, but performs no operation. If *tablename* is a view, the command fails with an SQLCODE -400 error.

The **UNLOCK** command reverses the **LOCK** operation. An explicit **LOCK** remains in effect until you issue an explicit **UNLOCK** for the same mode, or until the process terminates.

You can use **LOCK** to lock a table multiple times; you must explicitly **UNLOCK** the table as many times as it was explicitly locked. Each **UNLOCK** must specify the same mode as the corresponding **LOCK**.

Privileges

The **LOCK** command is a privileged operation. Prior to using **LOCK IN SHARE MODE** it is necessary for your process to have SELECT privilege for the specified table. Prior to using **LOCK IN EXCLUSIVE MODE** it is necessary for your process to have INSERT, UPDATE, or DELETE privilege for the specified table. For IN EXCLUSIVE MODE, the INSERT or UPDATE privilege must be on at least one field of the table. Failing to hold sufficient privileges results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has the necessary privileges by invoking the [%CHECKPRIV](#) command. You can determine if a specified user has the necessary privileges by invoking the [\\$SYSTEM.SQL.CheckPriv\(\)](#) method. For privilege assignment, refer to the [GRANT](#) command.

These privileges are required to acquire the lock; they do not define the nature of the lock. An IN EXCLUSIVE MODE lock prevents other processes from performing INSERT, UPDATE, or DELETE operations, regardless of whether the lock holder has the corresponding privilege.

LOCK Modes

LOCK supports two modes: SHARE and EXCLUSIVE. These lock modes are independent of each other. You can apply both a SHARE lock and an EXCLUSIVE lock to the same table. A lock in EXCLUSIVE mode can only be unlocked by an **UNLOCK** in EXCLUSIVE mode. A lock in SHARE mode can only be unlocked by an **UNLOCK** in SHARE mode.

- `LOCK mytable IN SHARE MODE` prevents other processes from issuing an EXCLUSIVE lock on mytable, or invoking a DDL operation, such as **DROP TABLE**.

- `LOCK mytable IN EXCLUSIVE MODE` prevents other processes from issuing an `EXCLUSIVE` lock or a `SHARE` lock on `mytable`, performing an insert, update, or delete operation, or invoking a DDL operation, such as **DROP TABLE**.

LOCK permits read access to the table. Neither **LOCK** mode prevents other processes from performing a **SELECT** on the table in `READ UNCOMMITTED` mode (the default **SELECT** mode).

Locking Conflicts

- If a table is already locked by another user `IN EXCLUSIVE MODE`, you cannot lock it in any mode.
- If a table is already locked by another user `IN SHARE MODE`, you can also lock the table `IN SHARE MODE`, but you cannot lock it `IN EXCLUSIVE MODE`.

These **LOCK** conflicts generate an `SQLCODE -110` error and generates a %msg such as the following: `Unable to acquire shared table-level lock for table 'Sample.Person'.`

Lock Timeout

LOCK attempts to acquire the specified SQL table lock until timeout occurs. When timeout occurs, **LOCK** generates an `SQLCODE -110` error.

- If you have specified `WAIT seconds`, SQL table lock timeout occurs when that number of seconds elapses.
- Otherwise, SQL table lock timeout occurs when the current process SQL timeout elapses. You can set the lock timeout for the current process using the `ProcessLockTimeout` methods **SetProcessLockTimeout()** and **GetProcessLockTimeout()**. You can also set the lock timeout for the current process using the SQL command **SET OPTION** with the `LOCK_TIMEOUT` option. (**SET OPTION** cannot be used from the SQL Shell.) The current process SQL lock timeout defaults to the system-wide SQL lock timeout.
- Otherwise, SQL table lock timeout occurs when the system-wide SQL timeout elapses. The system-wide default is 10 seconds. You can set the system-wide lock timeout in two ways:
 - Using the **SetLockTimeout()** method. This immediately changes the system-wide lock timeout default for new processes, and also resets the `ProcessLockTimeout` for the current process to this new system-wide value. Setting the system-wide lock timeout has no effect on the `ProcessLockTimeout` setting for other currently running processes.
 - Using the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Lock Timeout (in seconds)**. This changes the system-wide lock timeout default for new processes that start after you save the configuration change. It has no effect on currently running processes.

The **SetLockTimeout()** method sets a value and returns the previous value. You can use **GetLockTimeout()** to return the current system-wide lock timeout value:

```
GetSysTimeout
DO $SYSTEM.SQL.SetLockTimeout()
SET oldval=$SYSTEM.SQL.GetLockTimeout()
WRITE oldval," seconds initial system-wide lock setting",!
SetSysTimeout
DO $SYSTEM.SQL.SetLockTimeout(30,.oldval2)
WRITE "system-wide lock timeout changed from ",oldval2," to "
WRITE $SYSTEM.SQL.GetLockTimeout(),!
ResetSysTimeout
DO $SYSTEM.SQL.SetLockTimeout(oldval,.oldval3)
WRITE "system-wide lock timeout reset from ",oldval3," to "
WRITE $SYSTEM.SQL.GetLockTimeout()
```

The **SetProcessLockTimeout()** method sets a value and returns a status code. You can use **GetProcessLockTimeout()** to return the lock timeout value for the current process:

```

GetTimeoutDefaults
  SET sysinit=$SYSTEM.SQL.GetLockTimeout()
  WRITE sysinit," initial system-wide lock seconds",!
  SET procinit=$SYSTEM.SQL.GetProcessLockTimeout()
  WRITE procinit," initial process lock seconds",!
SetProcessTimeout
  DO $SYSTEM.SQL.SetProcessLockTimeout(50,.stat)
  IF stat {WRITE $SYSTEM.SQL.GetProcessLockTimeout()," set process lock seconds",! }
SetProcessTimeoutAgain
  DO $SYSTEM.SQL.SetProcessLockTimeout(60,.stat2)
  IF stat2 {WRITE $SYSTEM.SQL.GetProcessLockTimeout()," set process lock seconds",! }
SetProcessTimeoutMinimal
  DO $SYSTEM.SQL.SetProcessLockTimeout()
  WRITE $SYSTEM.SQL.GetProcessLockTimeout()," minimal process lock seconds",!
ResetToDefault
  DO $SYSTEM.SQL.SetProcessLockTimeout(procinit)
  WRITE $SYSTEM.SQL.GetProcessLockTimeout()," reset process lock seconds",!

```

Transaction Processing

A **LOCK** operation is not part of a transaction. Rolling back a transaction in which a **LOCK** is issued does not release the lock. An **UNLOCK** can be defined as occurring at the conclusion of the current transaction, or occurring immediately.

Other Locking Operations

Many DDL operations, including **ALTER TABLE** and **DELETE TABLE** acquire an exclusive table lock.

The **INSERT**, **UPDATE**, and **DELETE** commands also perform locking. By default they lock at the record level for the duration of the current transaction; if one of these commands locks a sufficiently large number of records (1000 is the default setting), the lock is automatically elevated to a table lock. The **LOCK** command allows you to explicitly set a table level lock, giving you greater control over the locking of data resources. An **INSERT**, **UPDATE**, or **DELETE** can override a **LOCK** by specifying the **%NOLOCK** keyword.

The Caché SQL **SET OPTION** with the **LOCK_TIMEOUT** option sets the timeout for the current process for an **INSERT**, **UPDATE**, **DELETE**, or **SELECT** operation.

Caché SQL supports the **SetCachedQueryLockTimeout()** method. See “[Cached Queries](#)” in the *Caché SQL Optimization Guide*.

Examples

The following embedded SQL examples create a table and then lock it:

```

ZNSPACE "Samples"
NEW SQLCODE,%msg
&sql(CREATE TABLE mytest (
  ID NUMBER(12,0) NOT NULL,
  CREATE_DATE DATE DEFAULT CURRENT_TIMESTAMP(2),
  WORK_START DATE DEFAULT SYSDATE) )
IF SQLCODE=0 { WRITE "Table created",! }
ELSEIF SQLCODE=-201 { WRITE "Table already exists",! }
ELSE { WRITE "CREATE TABLE error: ",SQLCODE
  QUIT }

ZNSPACE "Samples"
NEW SQLCODE,%msg
SET x=$ZHOROLOG
&sql(LOCK mytest IN EXCLUSIVE MODE WAIT 4)
IF SQLCODE=0 { WRITE !,"Table locked" }
ELSEIF SQLCODE=-110 { WRITE "waited ",$ZHOROLOG-x," seconds"
  WRITE !,"Table is locked by another process",!,%msg }
ELSE { WRITE !,"Unexpected LOCK error: ",SQLCODE,!,%msg }

```

SQL programs run from documentation, or from the Management Portal, spawn a process that terminates as soon as the program executes. Thus a lock is almost immediately released. Therefore, to observe a lock conflict, first issue a **LOCK mytest IN EXCLUSIVE MODE** command from a Terminal running the SQL Shell in the Samples namespace. Then run the above embedded SQL locking program. Issue an **UNLOCK mytest IN EXCLUSIVE MODE** from the Terminal SQL Shell. Then rerun the above embedded SQL locking program.

See Also

- [UNLOCK](#)
- [INSERT UPDATE DELETE](#)
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

OPEN

Opens a cursor.

```
OPEN cursor-name
```

Arguments

<i>cursor-name</i>	The name of the cursor, which has already been declared. The cursor name was specified in the DECLARE statement. Cursor names are case-sensitive.
--------------------	--

Description

An **OPEN** statement opens a [cursor](#) according to the parameters specified in the cursor's [DECLARE](#) statement. Once opened, a cursor can be fetched. An open cursor must be closed.

Attempting to open a cursor that is already open results in an `SQLCODE -101` error. Attempting to fetch or close a cursor that is not open results in an `SQLCODE -102` error. A successful **OPEN** sets `SQLCODE = 0`, even if the result set is empty.

OPEN does not support the `#SQLCompile Mode=Deferred` preprocessor directive. Attempting to use Deferred mode with a **DECLARE**, **OPEN**, **FETCH**, or **CLOSE** cursor statement generates a `#5663` compilation error.

As an SQL statement, this is only supported from embedded SQL. Equivalent operations are supported through ODBC using the ODBC API.

Example

The following embedded SQL example shows a cursor (named `EmpCursor`) being opened and closed:

```
SET name="LastName,FirstName",state="###"
&sql(DECLARE EmpCursor CURSOR FOR
    SELECT Name, Home_State
    INTO :name,:state FROM Sample.Person
    WHERE Home_State %STARTSWITH 'A')
WRITE !,"BEFORE: Name=",name," State=",state
&sql(OPEN EmpCursor)
IF SQLCODE '= 0 { WRITE "Open error: ",SQLCODE
    QUIT }
NEW %ROWCOUNT,%ROWID
FOR { &sql(FETCH EmpCursor)
    QUIT:SQLCODE
    WRITE !,"DURING: Name=",name," State=",state }
WRITE !,"FETCH status SQLCODE=",SQLCODE
WRITE !,"Number of rows fetched=",%ROWCOUNT
&sql(CLOSE EmpCursor)
WRITE !,"AFTER: Name=",name," State=",state
```

See Also

- [CLOSE, DECLARE, FETCH](#)
- [SQL Cursors](#) in the “Using Embedded SQL” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

ORDER BY

A **SELECT** clause that specifies the sorting of rows in a result set.

```
ORDER BY ordering-item [ASC | DESC]{,ordering-item [ASC | DESC] ...}
```

Arguments

<i>ordering-item</i>	A literal that determines the sort order. A column name, column alias, or column number. An ORDER BY clause can contain a single <i>ordering-item</i> or a comma-separated list of <i>ordering-items</i> specifying a sorting hierarchy.
ASC DESC	<i>Optional</i> — Sort in either ascending order (ASC) , or descending order (DESC) . The default is ascending order.

Description

The **ORDER BY** clause sorts the records in a query result set by the data values of a specified column or a comma-separated sequence of columns. This statement operates on a single result set, either from a [SELECT](#) statement or from a [UNION](#) of multiple [SELECT](#) statements.

ORDER BY sorts records by the Logical (internal storage) data value, regardless of the current [Select Mode](#) setting.

The **ORDER BY** clause is the last clause in a **SELECT** statement. It appears after the FROM, WHERE, GROUP BY, and HAVING clauses. Specifying **SELECT** clauses in the incorrect order generates an SQLCODE -25 error.

If a **SELECT** statement *does not* specify an **ORDER BY** clause, the returned record order is not predictable.

If a **SELECT** statement specifies an **ORDER BY** and a [TOP clause](#), the records that are returned as the “top” rows are in accord with the order specified in the **ORDER BY** clause. For example, `SELECT TOP 5 Name, Age FROM MyTable ORDER BY Age DESC` returns the 5 rows from MyTable with the highest age value, ordered from older to younger.

Specifying Sort Columns

You can specify a single column on which to sort, or multiple columns as a comma-separated list. Sorting is done by the first listed column, then within that column by the second listed column, and so on.

Columns can be specified by [column name](#), [column alias](#), or [column number](#). An **ORDER BY** clause can specify any combination of these. If the first character of the *ordering-item* is a number, Caché assumes you are specifying a column number. Otherwise a column name or column alias is assumed. Note that column names and column aliases are not case-sensitive.

With few exceptions, an *ordering-item* must be specified as a literal. If an *ordering-item* cannot be parsed as either a valid identifier (column name or column alias) or parsed as an unsigned integer (column number), that *ordering-item* is ignored and **ORDER BY** execution proceeds to the next *ordering-item* in the comma-separated list. Some examples of ignored *ordering-item* values are a Dynamic SQL ? input parameter or an Embedded SQL :var host variable, a subquery, an expression that resolves to a number, a signed number, or a number enclosed in parentheses.

Column Name

A column name can be specified as a literal. In some cases, an expression that operates upon a column name can be used as an *ordering-item*. You cannot use a variable or other expression that provides a column name as a string.

The following **ORDER BY** clause sorts by column names:

```
SELECT Name, Home_State, DOB
FROM Sample.Person
ORDER BY Home_State, Name
```

You can sort by column name whether or not the sort column is in the *select-item* list. (For obvious reasons, you cannot sort by column alias or column number unless the sort column is in the *select-item* list.) The following example returns the same records in the same order as the previous example:

```
SELECT Name,DOB
FROM Sample.Person
ORDER BY Home_State,Name
```

If the *ordering-item* is not an existing column name (or column alias) in the specified table, an SQLCODE -29 error is issued.

You can sort by the RowID value even if the RowID is private and not listed in the *select-item* list. You should specify the [%ID pseudo-column name](#) as the *ordering-item*, rather than the actual RowID field name. If the query contains a [TOP clause](#), sorting by RowID changes which rows are selected by the TOP clause. For example, if a table has 100 rows (with sequential RowIDs), `SELECT TOP 5 %ID FROM Table ORDER BY %ID` returns RowIDs 1, 2, 3, 4, 5; `SELECT TOP 5 %ID FROM Table ORDER BY %ID DESC` returns RowIDs 100, 99, 98, 97, 96.

An **ORDER BY** clause can specify a table name or [table alias](#) as part of the *ordering-item*:

```
SELECT P.Name AS People,E.Name As Employees
FROM Sample.Person AS P,Sample.Employee AS E
ORDER BY P.Name
```

An **ORDER BY** clause can use [arrow syntax](#) (`->`) operator to specify a field in a table that is not the base table:

```
SELECT Name,Company->Name AS CompName
FROM Sample.Employee ORDER BY Company->Name,Name
```

For further details, refer to [Implicit Joins](#) in *Using InterSystems SQL*.

Column Alias

A column alias must be specified as a literal. You cannot specify a column alias in an expression, or supply it using a variable.

The following **ORDER BY** clause sorts by [column alias](#):

```
SELECT Name,Home_State AS HS,DOB
FROM Sample.Person
ORDER BY HS,Name
```

A column alias can be the same as a column name (though this is not recommended). If column aliases are provided, **ORDER BY** first references column alias and then references any unaliased column names. If ambiguity between a column alias and a non-aliased column name exists, the **ORDER BY** clause generates an SQLCODE -24 error. However, if a column alias is the same as an *aliased* column name, this apparent ambiguity does not generate an error, but can produce unexpected results. This is shown in the following example:

```
SELECT Name AS Moniker,Home_City AS Name
FROM Sample.Person
ORDER BY Name
```

Because aliases are referenced first, this example orders the data by `Home_City`. This is probably not what was intended. If the `Name` column was not aliased, an SQLCODE -24 error would occur. If `Home_City` was given a different alias, **ORDER BY** would find no match on aliases, and would then check column names; it would order by the `Name` column.

You can use a column alias to sort by an expression in the *select-item* list, as shown in the following example:

```
SELECT Name,Age,$PIECE(AVG(Age)-Age,',' ,1) AS AgeDev
FROM Sample.Employee ORDER BY AgeDev,Name
```

Column Number

A column number must be specified as an unsigned numeric literal. You cannot specify a column number as a variable or the result of an expression. You cannot enclose a column number in parentheses. Integer truncation rules apply to resolve a non-integer value to an integer; for example, 1.99 resolves to 1.

The following **ORDER BY** clause sorts by column number (the numeric sequence of the retrieved columns, as specified in the **SELECT** *select-item* list):

```
SELECT Name,Home_State,DOB
FROM Sample.Person
ORDER BY 2,1
```

Column numbers refer to the position in the **SELECT** clause list. They do *not* refer to the positions of columns in the table itself. However, you can sort **SELECT** * results by column number; if the [RowID is public](#), it counts as column 1, if the RowID is hidden, it does not count as column 1.

Specifying a column number in **ORDER BY** that does not correspond to a **SELECT** list column results in an SQLCODE -5 error. **ORDER BY 0** results in an SQLCODE -5 error.

You can use a column number to sort by an expression in the *select-item* list, as shown in the following example:

```
SELECT Name,Age,$PIECE(AVG(Age)-Age,'.',1)
FROM Sample.Employee ORDER BY 3,Name
```

Specifying Collation

Sorting is done in [collation order](#). By default, ordering of string values is done based on the collation specified for the *ordering-item* field when it was created. Caché has a [default string collation for each namespace](#); the initial collation default for string data type fields is SQLUPPER, which is not case-sensitive. Therefore, commonly, **ORDER BY** collation is not case-sensitive.

Ordering of numeric data type fields is done based on numeric collation. For expressions, the default collation is EXACT.

You can override the default collation for a field by applying a collation function to a *ordering-item* field name. For example, ORDER BY %EXACT(Name). You cannot apply a collation function to a [column alias](#); attempting to do so generates an SQLCODE -29 error.

The default ascending collation sequence considers NULL to be the lowest value, followed by the empty string ("). **ORDER BY** does not distinguish between the empty string and strings that consist only of blank spaces.

If the collation specified for a column is alphanumeric, leading numbers are sorted in character collation sequence, not integer sequence. You can use the [%PLUS](#) collation function to order in integer sequence. However, the %PLUS collation function treats all non-numeric characters as 0.

Therefore, to properly sort mixed numeric strings in numeric sequence requires more than one *ordering-item*. For example, in Sample.Person a street address consists of an integer house number separated by a space from a street name. The street name consists of two parts separated by a space. Compare the following two examples. The first example sorts street addresses in character collation sequence:

```
SELECT Name,Home_Street FROM Sample.Person
ORDER BY Home_Street
```

The second example sorts the house number in integer sequence and the street name in character collation sequence:

```
SELECT Name,Home_Street FROM Sample.Person
ORDER BY $PIECE(%PLUS(Home_Street),' ',1),$PIECE(Home_Street,' ',2),$PIECE(Home_Street,' ',3)
```

Note that this example only works with a column name, not with a column alias or a column number.

ASC and DESC

Sorting can be specified for each column in ascending or descending collation sequence order, as specified by the optional ASC (ascending) or DESC (descending) keyword following the column identifier. If ASC or DESC is not specified, **ORDER BY** sorts that column in ascending order. You cannot specify the ASC or DESC keyword using a Dynamic SQL ? input parameter or an Embedded SQL :var host variable.

NULL is always the lowest value in ASC sequence and the highest value in DESC sequence.

Multiple comma-separated **ORDER BY** values specify a hierarchy of sort operations, as shown in the following example:

```
SELECT A,B,C,M,E,X,J
FROM LetterTable
ORDER BY 3,7 DESC,1 ASC
```

This example sorts the data values of the third-listed item (C) in the **SELECT** clause list in ascending order; within this sequence, it sorts the seventh-listed item (J) values in descending order; within this, it sorts the first-listed item (A) values in ascending order.

Duplicate columns in the list of **ORDER BY** values have no effect. This is because the second sort is within the order of the first sort. For example, ORDER BY Name ASC, Name DESC sorts the Name column in ascending order.

NLS Collation

If you have specified a non-default NLS collation, you must make sure that all collations are aligned and use the exact same national collation sequence. This includes not only globals used by the tables, but also globals used for indexes, in temporary files such as in CACHETEMP and process-private globals. For further details, refer to “[SQL Collation and NLS Collations](#)” in *Using InterSystems SQL*.

Restrictions

If your **SELECT** query specifies an **ORDER BY** clause, the resulting data is not updateable. Thus, if you specify a subsequent **DECLARE CURSOR FOR UPDATE** statement, the **FOR UPDATE** clause is ignored, and the cursor is declared read-only.

If the **ORDER BY** applies to a **UNION**, an ordering item must be a number or a simple column name. It cannot be an expression. If a column name is used, it refers to result columns as they are named in the first **SELECT** list of the **UNION**.

When used in a subquery, an **ORDER BY** clause must be paired with a **TOP** clause. This may be a **TOP ALL** clause. For example, the following FROM clause subquery is not valid: (SELECT DISTINCT age FROM table1 ORDER BY age); however, the following FROM clause subquery is valid: (SELECT DISTINCT TOP ALL age FROM table1 ORDER BY age).

Cached Queries

Each literal value used in an **ORDER BY** clause generates a different [cached query](#). [Literal substitution](#) is not performed on **ORDER BY** literals. This is because **ORDER BY** can use an integer to specify a column number. Changing this integer would result in a fundamentally different query.

ORDER BY and Long Global References

An **ORDER BY** *ordering-item* value should not exceed (approximately) between 400 and 500 characters, depending on the number of *ordering-items* and other factors. If an *ordering-item* value exceeds this maximum length, running a query with an **ORDER BY** clause may result in an SQLCODE -400 fatal error. This occurs because of a limitation in the maximum encoded length of a global reference, which is a fixed Caché system limit. To prevent this problem, use a truncation length in the collation setting for the field that is the basis of the **ORDER BY** clause. For example, the following query exceeds this limit:

```
TRY {
SET myquery = 3
SET myquery(1) = "SELECT LocationCity,NarrativeSummary FROM Aviation.Event "
SET myquery(2) = "WHERE LocationCity %Startswith 'Be' "
```

```

    SET myquery(3) = "ORDER BY NarrativeSummary"
    SET tStatement = ##class(%SQL.Statement).%New()
    SET qStatus = tStatement.%Prepare(.myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
    SET rset = tStatement.%Execute()
    IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
    ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)
          THROW badSQL }
    DO rset.%Display()
    WRITE !,"End of data"
    RETURN
}
CATCH exp { WRITE "In the CATCH block",!
            IF 1=exp.%IsA("%Exception.SQL") {
                WRITE "SQLCODE: ",exp.Code,!
                WRITE "Message: ",exp.Data,! }
            ELSE { WRITE "Not an SQL exception",! }
            RETURN
}
}

```

Adding a collation function with a *maxlen* truncation length allows this program to execute successfully:

```

TRY {
    SET myquery = 3
    SET myquery(1) = "SELECT LocationCity,NarrativeSummary FROM Aviation.Event "
    SET myquery(2) = "WHERE LocationCity %Startswith 'Be' "
    SET myquery(3) = "ORDER BY %SqlUpper(NarrativeSummary,400)"
    SET tStatement = ##class(%SQL.Statement).%New()
    SET qStatus = tStatement.%Prepare(.myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
    SET rset = tStatement.%Execute()
    IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
    ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)
          THROW badSQL }
    DO rset.%Display()
    WRITE !,"End of data"
    RETURN
}
CATCH exp { WRITE "In the CATCH block",!
            IF 1=exp.%IsA("%Exception.SQL") {
                WRITE "SQLCODE: ",exp.Code,!
                WRITE "Message: ",exp.Data,! }
            ELSE { WRITE "Not an SQL exception",! }
            RETURN
}
}

```

Caché truncates the collated value of the field at 400 characters. Remember that if the field contents are not unique within the first 400 characters, the data may be slightly misordered, but this is unlikely to occur. If this does occur, you can attempt to avoid displaying misordered data by using a larger value for truncation; however, if a value is too large, it will result in a <SUBSCRIPT> error.

Note also that the maximum length is for the entire encoded length of the global reference, including the length of the global name. It is not simply per subscript.

Examples

The following example sorts records in reverse RowID order:

```

SELECT %ID,Name
FROM Sample.Person
ORDER BY %ID DESC

```

The following two examples show different ways of specifying sort columns in an **ORDER BY** clause. The following two queries are equivalent; the first uses column names as sort items, the second uses column numbers (the sequence number of the items in the *select-item* list):

```

SELECT Name,Age,Home_State
FROM Sample.Person
ORDER BY Home_State,Age DESC

```

```

SELECT Name,Age,Home_State
FROM Sample.Person
ORDER BY 3,2 DESC

```

The following example sorts by a field containing Caché list data. Because a Caché list is an encoded character string that begins with formatting characters, this example uses `$LISTTOSTRING` to sort by the actual field data value, rather than the list element encoding:

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY $LISTTOSTRING(FavoriteColors)
```

Dynamic SQL can use an input parameter to supply a literal value to an **ORDER BY** clause; it cannot use an input parameter to supply a field name, field alias, field number, or collation keyword. The following Dynamic SQL example uses an input parameter to sort result set records by first name:

```
SET myquery = 4
SET myquery(1) = "SELECT TOP ? Name,Age,"
SET myquery(2) = "CURRENT_DATE AS Today"
SET myquery(3) = "FROM Sample.Person WHERE Age > ?"
SET myquery(4) = "ORDER BY $PIECE(Name,',' ,?)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(10,60,2)
DO rset.%Display()
WRITE !,"%Display SQLCODE=",rset.%SQLCODE
```

The following cursor-based Embedded SQL example performs the same operation:

```
SET topnum=10,agemin=60,firstname=2
&sql(DECLARE pCursor CURSOR FOR
      SELECT TOP :topnum Name,Age,CURRENT_DATE AS Today
      INTO :name,:years,:today FROM Sample.Person
      WHERE Age > :agemin
      ORDER BY $PIECE(Name,',' ,:firstname) )
&sql(OPEN pCursor)
QUIT:(SQLCODE'=0)
FOR { &sql(FETCH pCursor)
      QUIT:SQLCODE
      WRITE "Name=",name," Age=",years," today=",today,!
    }
&sql(CLOSE pCursor)
```

See Also

- [SELECT](#)
- [UNION](#)
- [TOP](#) clause
- “[Collation](#)” chapter in *Using Caché SQL*
- “[Querying the Database](#)” chapter in *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

REVOKE

Removes privileges from a user or role.

```
REVOKE admin-privilege FROM grantee

REVOKE role FROM grantee

REVOKE [GRANT OPTION FOR] object-privilege
      ON object-list FROM grantee [CASCADE | RESTRICT] [AS grantor]

REVOKE [GRANT OPTION FOR] SELECT ON CUBE[S] object-list
FROM grantee

REVOKE column-privilege (column-list) ON table FROM grantee [CASCADE | RESTRICT]
```

Arguments

<i>admin-privilege</i>	<p>An administrative-level privilege or a comma-separated list of administrative-level privileges previously granted to be revoked. The available <i>syspriv</i> options include sixteen object definition privileges and four data modification privileges.</p> <p>The object definition privileges are: %CREATE_FUNCTION, %DROP_FUNCTION, %CREATE_METHOD, %DROP_METHOD, %CREATE_PROCEDURE, %DROP_PROCEDURE, %CREATE_QUERY, %DROP_QUERY, %CREATE_TABLE, %ALTER_TABLE, %DROP_TABLE, %CREATE_VIEW, %ALTER_VIEW, %DROP_VIEW, %CREATE_TRIGGER, %DROP_TRIGGER. Alternatively, you can specify %DB_OBJECT_DEFINITION, which revokes all 16 object definition privileges.</p> <p>The data modification privileges are the %NOCHECK, %NOINDEX, %NOLOCK, %NOTRIGGER privileges for INSERT, UPDATE, and DELETE operations.</p>
<i>grantee</i>	<p>A list of one or more users having SQL System Privileges, SQL Object Privileges, or Roles. Valid values are a comma-separated list of users or roles, or "*". The asterisk (*) specifies all currently defined users who do not have the %All role.</p>
AS <i>grantor</i>	<p>This clause permits you to revoke a privilege granted by another user by specifying the name of the original grantor. Valid <i>grantor</i> values are a user name, a comma-separated list of user names, or "*". The asterisk (*) specifies all currently defined users who are grantors. To use the AS <i>grantor</i> clause, you must have the %All role or the %Admin_Secure resource.</p>
<i>role</i>	<p>A role or comma-separated list of roles whose privileges are being revoked from a user.</p>
<i>object-privilege</i>	<p>A basic-level privilege or comma-separated list of basic-level privileges previously granted to be revoked. The list may consist of one or more of the following: %ALTER, DELETE, SELECT, INSERT, UPDATE, EXECUTE, and REFERENCES. To revoke all privileges, use either "ALL [PRIVILEGES]" or "*" as the value for this argument. Note that you can only revoke SELECT privilege from cubes, because this is the only grantable cubes privilege.</p>

<i>object-list</i>	A comma-separated list of one or more tables , views , stored procedures, or cubes for which the <i>object-privilege(s)</i> are being revoked. You can use the SCHEMA keyword to specify revoking the <i>object-privilege</i> from all objects in the specified schema. You can use "*" to specify revoking the <i>object-privilege</i> from all objects in the current namespace.
<i>column-privilege</i>	A basic-level privilege being revoked from one or more <i>column-list</i> listed columns. Available options are SELECT, INSERT, UPDATE, and REFERENCES.
<i>column-list</i>	A list of one or more column names, separated by commas and enclosed in parentheses.
<i>table</i>	The name of the table or view that contains the <i>column-list</i> columns.

Description

The **REVOKE** statement revokes privileges that allow a user or role to perform specified tasks on specified tables, views, columns, or other entities. **REVOKE** can also revoke a role assignment from a user. **REVOKE** reverses the actions of the **GRANT** command; see that command for more details on privileges generally.

A privilege can only be revoked by the user who granted the privilege, or through a CASCADE operation (as described below).

You can revoke a role or privilege from a specified user, a list of users, or all users (using the * syntax).

Because **REVOKE** prepares and executes quickly, and is generally run only once, Caché does not create a cached query for **REVOKE** in ODBC, JDBC, or Dynamic SQL.

A **REVOKE** completes successfully, even if no actual revoke can be performed (for example, the specified privilege was never granted or has already been revoked). However, if an error occurs during the **REVOKE** operation, SQLCODE is set to a negative number.

Revoking Roles

Roles can be granted or revoked via either the SQL **GRANT** and **REVOKE** commands, or via ^SECURITY Caché System Security. You can use **REVOKE** to revoke a role from a user or to revoke a role from another role. You cannot use Caché System Security to grant or revoke roles to other roles. The \$ROLES special variable does not display roles granted to roles.

REVOKE can specify a single role, or a comma-separated list of roles to revoke. **REVOKE** can revoke one or more roles from a specified user (or role), a list of users (or roles), or all users (using the * syntax).

The **GRANT** command can grant a non-existent role to a user. You can use **REVOKE** to revoke a non-existent role from an existing user. However, the role name must be specified using the same letter case that was used to grant the role.

If you attempt to revoke an existing role from a non-existent user or role, Caché issues an SQLCODE -118 error. If you are not the SuperUser, and you attempt to revoke a role that you don't own and don't have ADMIN OPTION for, Caché issues an SQLCODE -112 error.

Revoking Object Privileges

Object privileges give a user or role some right to a particular object. You revoke an *object-privilege* ON an *object-list* FROM a *grantee*. An *object-list* can specify one or more tables, views, stored procedures, or cubes in the current namespace. By using comma-separated lists, a single **REVOKE** statement can revoke multiple object privileges on multiple objects from multiple users and/or roles.

You can use the asterisk (*) wildcard as the *object-list* value to revoke the *object-privilege* from all of the objects in the current namespace. For example, REVOKE SELECT ON * FROM Deborah revokes this user's SELECT privilege for

all tables and views. `REVOKE EXECUTE ON * FROM Deborah` revokes this user's `EXECUTE` privilege for all non-hidden Stored Procedures.

You can use `SCHEMA schema-name` as the *object-list* value to revoke the *object-privilege* for all of the tables, views, and stored procedures in the named schema, in the current namespace. For example, `REVOKE SELECT ON SCHEMA Sample FROM Deborah` revokes this user's `SELECT` privilege for all objects in the `Sample` schema. You can specify multiple schemas as a comma-separated list; for example, `REVOKE SELECT ON SCHEMA Sample,Cinema FROM Deborah` revokes `SELECT` privilege for all objects in both the `Sample` and the `Cinema` schemas.

You can revoke an object privilege from a user or from a role. If you revoke it from a role, a user that only had that privilege through the role no longer has the privilege. A user that no longer has a privilege can no longer execute an existing cached query that requires that object privilege.

When **REVOKE** revokes an object privilege, it completes successfully and sets `SQLCODE` to 0. If **REVOKE** does not perform an actual revoke (for example, the specified object privilege was never granted or has already been revoked), it completes successfully and sets `SQLCODE` to 100 (no more data). If an error occurs during the **REVOKE** operation, it sets `SQLCODE` to a negative number.

Cubes are SQL identifiers that are not qualified by a schema name. To specify a cubes *object-list*, you must specify the `CUBE` (or `CUBES`) keyword. Because cubes can only have `SELECT` privilege, you can only revoke `SELECT` privilege from a cube.

Object privileges can be revoked by any of the following:

- The **REVOKE** command.
- The `%SYSTEM.SQL RevokeObjPriv()` method.
- Via Caché System Security. Go to the Management Portal, select **System Administration, Security, Users** (or **System Administration, Security, Roles**) select **Edit** for the desired user or role, then select the **SQL Tables** or **SQL Views** tab. Select the desired **Namespace** from the drop-down list. Scroll down to the desired table, then click **revoke** to revoke privileges.

You can determine if the current user has a specified object privilege by invoking the `%CHECKPRIV` command. You can determine if a specified user has a specified table-level object privilege by invoking the `$SYSTEM.SQL.CheckPriv()` method.

Revoking Object Owner Privileges

If you revoke the privileges on an SQL object from the owner of the object, the owner will still implicitly have privileges on the object. In order to completely revoke all privileges on the object from the owner of the object, the object must be changed to specify a different owner or no owner.

Prior to Caché 2013.1, if you revoke the privileges on an SQL object from the owner of the object, the owner will cease to have privileges on the object. However, if the class is later recompiled, the privileges for the owner would be granted again.

Revoking Table-level and Column-level Privileges

REVOKE can be used to reverse the granting of table-level privileges or column-level privileges. A table-level privilege provides access to all of the columns in a table. A column-level privilege provides access to every specified column in the table. Granting a column-level privilege to all of the columns in a table is functionally equivalent to granting a table-level privilege. However, the two are not functionally identical. A column-level **REVOKE** can only revoke privileges granted at the column level. You cannot grant a table-level privilege to the table, then revoke this privilege at the column level for one or more columns. In this case, the **REVOKE** statement has no effect on granted privileges.

CASCADE or RESTRICT

Caché supports the optional `CASCADE` and `RESTRICT` keywords to specify **REVOKE** *object-privilege* behavior. If neither keyword is specified, the default is `RESTRICT`.

You can use **CASCADE** or **RESTRICT** to specify whether revoking an *object-privilege* or *column-privilege* from a user will also revoke that privilege from any other users that received it via the **WITH GRANT OPTION**. **CASCADE** revokes all such associated privileges. **RESTRICT** (the default) causes **REVOKE** to fail when an associated privilege is detected. Instead it sets the **SQLCODE -126** error “REVOKE with RESTRICT failed”.

The use of these keywords is shown by the following example:

```
--UserA
GRANT Select ON MyTable TO UserB WITH GRANT OPTION

--UserB
GRANT Select ON MyTable TO UserC

--UserA
REVOKE Select ON MyTable FROM UserB
-- This REVOKE fails with SQLCODE -126

--UserA
REVOKE Select ON MyTable FROM UserB CASCADE
-- This REVOKE succeeds
-- It revokes this privilege from UserB and UserC
```

Note that **CASCADE** and **RESTRICT** have no effect on a view created by UserB that references MyTable.

Effect on Cached Queries

When you revoke a privilege or role, Caché updates all [cached queries](#) on the system to reflect this change in privileges. However, when a namespace is inaccessible — for example, when an ECP connection to a database server is down — the **REVOKE** successfully completes but performs no operation on cached queries in that namespace. This is because **REVOKE** cannot update the cached queries in the unreachable namespace to revoke the privileges at the cached query level. No error is issued.

If the database server later comes up, the privileges for the cached queries in that namespace may be incorrect. It is advised that you purge cached queries in a namespace if a role or privilege might have been revoked while the namespace was not accessible.

Obsolete Privileges

At Caché Version 5.1 and all subsequent versions, **REVOKE** no longer supports the following general administrative privileges: **%GRANT_ANY_PRIVILEGE**, **%CREATE_USER**, **%ALTER_USER**, **%DROP_USER**, **%CREATE_ROLE**, **%GRANT_ANY_ROLE**, **%DROP_ANY_ROLE**. Control of these privileges is handled at the system level, rather than through SQL. These SQL privileges were available in prior versions of Caché, and may appear in existing code. An attempt to revoke one of these may execute, but it results not in the revoking of a privilege, but in attempting to revoke a role having this name from the specified user(s). Similarly, attempting to revoke **%THRESHOLD** now results in attempting to revoke a role having this name from the specified user(s).

Caché Security

The **REVOKE** command is a privileged operation. Prior to using **REVOKE** in embedded SQL, it is necessary to be logged in as a user with appropriate privileges. Failing to do so results in an **SQLCODE -99** error (Privilege Violation).

Use the **\$\$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login("_SYSTEM", "SYS")
&sql( )
```

You must have the **%Service_Login:Use** privilege to invoke the **\$\$SYSTEM.Security.Login** method. For further information, refer to **%SYSTEM.Security** in the *InterSystems Class Reference*.

Examples

The following embedded SQL example creates two users, creates a role, and assigns the role to the users. It then revokes the role from all users using the asterisk (*) syntax. If the user or the role already exists, the **CREATE** statement issues an

SQLCODE -118 error. If the user does not exist, the GRANT or REVOKE statement issues an SQLCODE -118 error. If the user exists but the role does not, the GRANT or REVOKE statement issues SQLCODE 100. If the user and role exist, the GRANT or REVOKE statement issues SQLCODE 0. This is true even when the granting or revoking of the role has already been done, or if you are attempting to revoke a role that was never granted.

```
DO $SYSTEM.Security.Login("_SYSTEM","SYS")
&sql(CREATE USER User1 IDENTIFY BY fredpw)
&sql(CREATE USER User2 IDENTIFY BY barneypw)
WRITE !,"CREATE USER error code: ",SQLCODE
&sql(CREATE ROLE workerbee)
WRITE !,"CREATE ROLE error code: ",SQLCODE
&sql(GRANT workerbee TO User1,User2)
WRITE !,"GRANT role error code: ",SQLCODE
&sql(REVOKE workerbee FROM *)
WRITE !,"REVOKE role error code: ",SQLCODE
```

In the following example, one user (Joe) grants a privilege and a different user (John) revokes that privilege, using the AS *grantor* clause:

```
/* User Joe */
GRANT SELECT ON Sample.Person TO Michael

/* User John */
REVOKE SELECT ON Sample.Person FROM Michael AS Joe
```

Note that John must have the %All role or the %Admin_Secure resource.

See Also

- SQL statements: [CREATE USER](#), [DROP USER](#), [CREATE ROLE](#), [DROP ROLE](#), [GRANT](#), [%CHECKPRIV](#)
- “Users, Roles, and Privileges” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*
- ObjectScript: [\\$ROLES](#) and [\\$USERNAME](#) special variables

ROLLBACK

Rolls back a transaction.

```
ROLLBACK [WORK]
ROLLBACK TO SAVEPOINT pointname
```

Arguments

<i>pointname</i>	The name of an existing savepoint, specified as an identifier . For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> .
------------------	---

Description

A **ROLLBACK** statement rolls back a [transaction](#), undoing work performed but not committed, decrementing the **\$TLEVEL** transaction level counter, and releasing locks. **ROLLBACK** is used to restore the database to a previous consistent state.

- A **ROLLBACK** rolls back all work completed during the current transaction, resets the **\$TLEVEL** transaction level counter to zero and releases all locks. This restores the database to its state before the beginning of the transaction. **ROLLBACK** and **ROLLBACK WORK** are equivalent statements; both versions are supported for compatibility.
- A **ROLLBACK TO SAVEPOINT** *pointname* rolls back all work done since the specified savepoint and decrements the **\$TLEVEL** transaction level counter by the number of savepoints undone. When all savepoints have been either rolled back or committed and the transaction level counter reset to zero, the transaction is completed. If the specified savepoint does not exist, or has already been rolled back, **ROLLBACK** issues an SQLCODE -375 error and rolls back the entire current transaction.

A **ROLLBACK TO SAVEPOINT** must specify a *pointname*. Failing to do so results in an SQLCODE -301 error.

For details on establishing savepoints, refer to [SAVEPOINT](#).

An SQLCODE -400 error is issued if a transaction operation fails to complete successfully.

Not Rolled Back

The following items are not affected by a **ROLLBACK** operation:

- A roll back does not decrement the IDKey counter for a default class. The IDKey is automatically generated by [\\$INCREMENT](#) (or [\\$SEQUENCE](#)), which maintains a count independent of the SQL transaction.
- A roll back does not reverse the creation, modification, or purging of a [Cached Query](#). These operations are not treated as part of a transaction.
- A DDL operation or a [Tune Table](#) operation that occur within a transaction may create and run a temporary routine. This temporary routine is treated the same as a Cached Query. That is, the creation, compilation, and deletion of a temporary routine are not treated as part of the transaction. The execution of the temporary routine is considered part of the transaction.

For non-SQL items rolled back or not rolled back, refer to the ObjectScript [TROLLBACK](#) command.

Rollback Logging

Messages indicating that a rollback occurred, and errors encountered during the rollback operation are logged in the `cconsole.log` file in the MGR directory. You can use the Management Portal **System Operation**, **System Logs**, **Console Log** option to view `cconsole.log`.

ObjectScript Transaction Commands

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with **TSTART**. If you need compatibility with the SQL standard, use **START TRANSACTION**.

ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

Examples

The following Embedded SQL example demonstrates how a **ROLLBACK** restores the transaction level counter (**\$TLEVEL**) to 0, the level immediately prior to the **START TRANSACTION**:

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT b)
WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT c)
WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(ROLLBACK)
WRITE !,"Rollback transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
```

The following Embedded SQL example demonstrates how a **ROLLBACK TO SAVEPOINT name** restores the transaction level (**\$TLEVEL**) to the level immediately prior to the specified **SAVEPOINT**:

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level at a=", $TLEVEL
&sql(SAVEPOINT b)
WRITE !,"Set Savepoint b, SQLCODE=",SQLCODE
WRITE !,"Transaction level at b=", $TLEVEL
&sql(ROLLBACK TO SAVEPOINT b)
WRITE !,"Rollback to b, SQLCODE=",SQLCODE
WRITE !,"Rollback transaction level=", $TLEVEL
&sql(SAVEPOINT c)
WRITE !,"Set Savepoint c, SQLCODE=",SQLCODE
WRITE !,"Transaction level at c=", $TLEVEL
&sql(SAVEPOINT d)
WRITE !,"Set Savepoint d, SQLCODE=",SQLCODE
WRITE !,"Transaction level at d=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
```

See Also

- SQL commands: [COMMIT](#), [SAVEPOINT](#), [SET TRANSACTION](#), [START TRANSACTION](#), [\\$TLEVEL](#)
- [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*
- ObjectScript: [TROLLBACK](#)

- ObjectScript: the [Transaction Processing](#) chapter of *Using Caché ObjectScript*

SAVEPOINT

Marks a point within a transaction.

```
SAVEPOINT pointname
```

Arguments

<i>pointname</i>	The name of the savepoint, specified as an identifier . For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> .
------------------	---

Description

A **SAVEPOINT** statement marks a point within a [transaction](#). Establishing a savepoint enables you to perform transaction roll back to the savepoint, undoing all work done and releasing all locks acquired during that period. In a long-running transaction, or a transaction with internal control structure, it is often desirable to be able to roll back part of the transaction without undoing all work submitted during the transaction.

The establishment of a savepoint increments the **\$TLEVEL** transaction level counter. Rolling back to a savepoint decrements the **\$TLEVEL** transaction level counter to its value immediately prior to the savepoint. You can establish up to 255 savepoints within a transaction. Exceeding this number of savepoints results in an SQLCODE -400 fatal error, a <TRANSACTION LEVEL> exception caught during SQL execution. The Terminal prompt displays the current transaction level as a TL*n* : prefix to the prompt, where *n* is an integer between 1 and 255 representing the current **\$TLEVEL** count.

Each savepoint is associated with an savepoint name, a unique [identifier](#). Savepoint names are not case-sensitive. A savepoint name can be a delimited identifier.

- If you specify a **SAVEPOINT** with no *pointname*, or with a *pointname* that is not a valid identifier or is an [SQL Reserved Word](#), a runtime SQLCODE -301 error is issued.
- If you specify a **SAVEPOINT** with a *pointname* that begins with “SYS”, a runtime SQLCODE -302 error is issued. These savepoint names are reserved.

Savepoint names are not case-sensitive; therefore `resetpt`, `ResetPt` and `"RESETPT"` are the same *pointname*. This duplication is detected during **ROLLBACK TO SAVEPOINT**, not during **SAVEPOINT**. When you specify a **SAVEPOINT** statement with a duplicate *pointname*, Caché increments the transaction level counter, just as if the *pointname* was unique. However, the most recent *pointname* overwrites all prior duplicate values in the table of savepoint names. Therefore, when you specify a **ROLLBACK TO SAVEPOINT** *pointname*, Caché rolls back to the most recently established **SAVEPOINT** with that *pointname*, and decrements the transaction level counter appropriately. However, if you again specify a **ROLLBACK TO SAVEPOINT** *pointname* with the same name, an SQLCODE -375 error is generated, with the %msg: `Cannot ROLLBACK to unestablished savepoint 'name'`, the full transaction is rolled back and the **\$TLEVEL** count reverts to 0.

Using Savepoints

The **SAVEPOINT** statement is supported for Embedded SQL, Dynamic SQL, ODBC, and JDBC. In JDBC, `connection.setSavepoint(pointname)` sets a savepoint, and `connection.rollback(pointname)` rolls back to the named savepoint.

If savepoints have been established:

- A **ROLLBACK TO SAVEPOINT** *pointname* rolls back work done since the specified savepoint, deletes that savepoint and all intermediate savepoints, and decrements the **\$TLEVEL** transaction level counter by the number of savepoints deleted. If *pointname* does not exist, or has already been rolled back, this command rolls back the entire transaction, resets **\$TLEVEL** to 0, and releases all locks.

- A **ROLLBACK** rolls back all work done during the current transaction, rolling back the work done since **START TRANSACTION**. It resets the **\$TLEVEL** transaction level counter to zero and releases all locks. Note that a generic **ROLLBACK** ignores savepoints.
- A **COMMIT** commits all work done during the current transaction. It resets the **\$TLEVEL** transaction level counter to zero and releases all locks. Note that a **COMMIT** ignores savepoints.

Issuing a second **START TRANSACTION** within a transaction has no effect on savepoints or the **\$TLEVEL** transaction level counter.

An **SQLCODE -400** error is issued if a transaction operation fails to complete successfully.

Examples

The following embedded SQL example creates a transaction with two savepoints:

```
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(START TRANSACTION)
&sql(DELETE FROM Sample.Person WHERE Name=NULL)
IF SQLCODE=100 { WRITE !,"No null name records to delete" }
ELSEIF SQLCODE'=0 {&sql(ROLLBACK)}
ELSE {WRITE !,%ROWCOUNT," null name records deleted"}
    &sql(SAVEPOINT svpt_age1)
    &sql(DELETE FROM Sample.Person WHERE Age=NULL)
    IF SQLCODE=100 { WRITE !,"No null age records to delete" }
    ELSEIF SQLCODE'=0 {&sql(ROLLBACK TO SAVEPOINT svpt_age1)}
    ELSE {WRITE !,%ROWCOUNT," null age records deleted"}
        &sql(SAVEPOINT svpt_age2)
        &sql(DELETE FROM Sample.Person WHERE Age>65)
        IF SQLCODE=0 { &sql(COMMIT)}
        ELSEIF SQLCODE=100 { &sql(COMMIT)}
        ELSE {
            &sql(ROLLBACK TO SAVEPOINT svpt_age2)
            WRITE !,"retirement age deletes failed"
        }
    &sql(COMMIT)
&sql(COMMIT)
```

ObjectScript and SQL Transactions

ObjectScript transaction processing, using **TSTART** and **TCOMMIT**, differs from, and is incompatible with, SQL transaction processing using the SQL statements **START TRANSACTION**, **SAVEPOINT**, and **COMMIT**. Both ObjectScript and Caché SQL provides limited support for nested transactions. ObjectScript transaction processing does not interact with SQL lock control variables; of particular concern is the SQL lock escalation variable. An application should not attempt to mix the two types of transaction processing.

If a transaction involves SQL update statements, then the transaction should be started by the SQL **START TRANSACTION** statement and committed with the SQL **COMMIT** statement. Methods that use **TSTART/TCOMMIT** nesting can be included in the transaction, as long as they don't initiate the transaction. Methods and stored procedures should not normally use SQL transaction control statements, unless, by design, they are the main controller of the transaction.

See Also

- SQL commands: [COMMIT ROLLBACK SET TRANSACTION START TRANSACTION \\$TLEVEL](#)
- [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*
- ObjectScript command: [TCOMMIT](#)

SELECT

Retrieves rows from one or more tables within a database.

```
[ ( ) SELECT [%NOFPLAN] [%NOLOCK]
  [DISTINCT [BY (item {,item})] | ALL]
  [TOP {int | ALL}]
  select-item {,select-item}
  [INTO host-variable-list]
  [FROM [optimize-option] table-ref [[AS] t-alias]
    {,table-ref [[AS] t-alias]} ]
  [WHERE condition-expression]
  [GROUP BY scalar-expression]
  [HAVING condition-expression]
  [ORDER BY item-order-list [ASC | DESC] ]
( ) ]

select-item ::=
  [t-alias.* |
  [t-alias.]scalar-expression [[AS] c-alias]
  {,[t-alias.]scalar-expression [[AS] c-alias]}
```

Arguments

%NOFPLAN	<i>Optional</i> — The %NOFPLAN keyword specifies that Caché will ignore the frozen plan (if any) for this query and generate a new query plan. The frozen plan is retained, but not used. For further details, refer to Frozen Plans in <i>Caché SQL Optimization Guide</i> .
%NOLOCK	<i>Optional</i> — The %NOLOCK keyword specifies that Caché will perform no locking on any of the specified tables. If you specify this keyword, the query retrieves data in READ UNCOMMITTED mode , regardless of current transaction's isolation mode. For further details, refer to Transaction Processing in the “Modifying the Database” chapter of <i>Using Caché SQL</i> .
DISTINCT DISTINCT BY (<i>item</i>) ALL	<i>Optional</i> — The DISTINCT clause specifies that each row returned must contain a unique value for the specified field or combination of fields. A DISTINCT keyword specifies that the <i>select-item</i> value(s) must be unique. A DISTINCT BY keyword clause specifies that <i>item</i> value(s) must be unique. An <i>item</i> (or a comma-separated list of <i>items</i>) is enclosed in parentheses. Commonly, an <i>item</i> is the name of a column. It may or may not also be listed as a <i>select-item</i> . <i>Optional</i> — The ALL keyword specifies that all rows that meet the SELECT criteria be returned. This is the default for Caché SQL. The ALL keyword performs no operation; it is provided for SQL compatibility. See DISTINCT clause for more details.

<p>TOP <i>int</i></p> <p>TOP ALL</p>	<p><i>Optional</i> — The TOP clause limits the number of rows returned to the number specified in <i>int</i>. If no ORDER BY clause is specified in the query, which records are returned as the “top” rows is unpredictable. If an ORDER BY clause is specified, the top rows accord to the specified order. The DISTINCT keyword (if specified) is applied before TOP, specifying that <i>int</i> number of unique values are to be returned. The <i>int</i> argument can be either a positive integer or a Dynamic SQL ? input parameter that resolves to a positive integer. If no TOP keyword is specified, the default is to display all the rows that meet the SELECT criteria.</p> <p>TOP ALL is only meaningful in a subquery or in a CREATE VIEW statement. It is used to support the use of an ORDER BY clause in these situations, fulfilling the requirement that an ORDER BY clause must be paired with a TOP clause in a subquery or a query used in a CREATE VIEW. TOP ALL does not restrict the number of rows returned.</p> <p>See TOP clause for more details.</p>
<p><i>select-item</i></p>	<p>One or more columns (or other values) to be retrieved. Multiple <i>select-items</i> are specified as a comma-separated list. You can also retrieve all columns by using the * symbol.</p>
<p>INTO <i>host-variable-list</i></p>	<p><i>Optional</i> — (Embedded SQL only): One or more host variables into which <i>select-item</i> values are placed. Multiple host variables are specified as a comma-separated list or as a single host variable array. See INTO clause for more details.</p>

FROM <i>table-ref</i>	<p><i>Optional</i> — A reference to one or more tables from which data is being retrieved. A valid <i>table-ref</i> is required for every FROM clause, even if the SELECT makes no reference to that table. A SELECT that makes no references to table data can omit the FROM clause.</p> <p>A <i>table-ref</i> can be specified as one or more tables, views, table-valued functions, or subqueries, specified as a comma-separated list or with JOIN syntax. Some restrictions apply on using views with JOIN syntax. A subquery must be enclosed in parentheses.</p> <p>A <i>table-ref</i> is either qualified (schema.tablename) or unqualified (tablename). An unqualified <i>table-ref</i> is supplied either the system-wide default schema name, or a schema name from the schema search path.</p> <p>Multiple tables can be specified as a comma-separated list or associated with ANSI join keywords. Any combination of tables or views can be specified. If you specify a comma between two <i>table-refs</i> here, Caché performs a CROSS JOIN on the tables and retrieves data from the results table of the JOIN operation. If you specify ANSI join keywords between two <i>table-refs</i> here, Caché performs the specified join operation. For further details, refer to the JOIN page of this manual.</p> <p>You can optionally assign an alias (<i>t-alias</i>) to each <i>table-ref</i>. The AS keyword is optional.</p> <p>You can optionally specify one or more <i>optimize-option</i> keywords to optimize query execution. The available options are: %ALLINDEX, %FIRSTTABLE, %FULL, %INORDER, %IGNOR-EINDEX, %NOFLATTEN, %NOMERGE, %NOREDUCE, %NOSVSO, %NOTOPOPT, %NOUNIONOROPT, %PARALLEL, and %STARTTABLE. See FROM clause for more details.</p>
WHERE <i>condition-expression</i>	<p><i>Optional</i> — A qualifier specifying one or more predicate conditions for what data is to be retrieved. See WHERE clause for more details.</p>
GROUP BY <i>scalar-expression</i>	<p><i>Optional</i> — A comma-separated list of one or more scalar expressions specifying how the retrieved data is to be organized; these may include column names. See GROUP BY clause for more details.</p>
HAVING <i>condition-expression</i>	<p><i>Optional</i> — A qualifier specifying one or more predicate conditions for what data is to be retrieved. See HAVING clause for more details.</p>

ORDER BY <i>item-order-list</i>	<i>Optional</i> — A select-item or a comma-separated list of items that specify the order in which rows are displayed. Each item can have an optional ASC (ascending order) or DESC (descending order). The default is ascending order. An ORDER BY clause is used on the results of a query. An ORDER BY clause in a subquery (for example, in a UNION statement) must be paired with a TOP clause. If no ORDER BY clause is specified, the order of the records returned is unpredictable. See ORDER BY clause for more details.
<i>scalar-expression</i>	A field identifier, an expression containing a field identifier, or a general expression, such as a function call or an arithmetic operation.
AS <i>t-alias</i>	<i>Optional</i> — An alias for a table or view name (<i>table-ref</i>). An alias must be a valid identifier ; it can be a delimited identifier. For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> . The AS keyword is optional.
AS <i>c-alias</i>	<i>Optional</i> — An alias for a column name (<i>select-item</i>). An alias must be a valid identifier . For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> . The AS keyword is optional.

Description

The **SELECT** statement performs a query that retrieves data from a Caché database. In its simplest form, it retrieves one or more items from a single table. The items are specified by the *select-item* list and the table is specified by the FROM *table-ref* clause. In more complex queries, a **SELECT** can retrieve data from multiple tables and can retrieve data using views.

A **SELECT** can also be used to return a value from an SQL function, a host variable, or a literal. A **SELECT** query can combine returning these non-database values with retrieving values from tables or views. When a **SELECT** is *only* used to return such non-database values, the FROM clause is optional. See [FROM clause](#) for more details.

The values returned from a **SELECT** query are known as a result set. In Dynamic SQL, **SELECT** retrieves values into the %SQL.Statement class. Refer to the [Dynamic SQL](#) chapter of *Using Caché SQL*, and the %SQL.Statement class in the *InterSystems Class Reference*.

Caché sets a status variable [SQLCODE](#), which indicates the success or failure of the **SELECT**. In addition, the **SELECT** operation sets the [%ROWCOUNT](#) local variable to the number of selected rows. Successful completion of a **SELECT** generally sets SQLCODE=0 and %ROWCOUNT to the number of rows selected. In the case of an embedded SQL containing a simple **SELECT**, data from (at most) one row is selected, so SQLCODE=0 and %ROWCOUNT is set to either 0 or 1. However, in the case of an embedded SQL **SELECT** that declares a cursor and fetches data from multiple rows, the operation completes when the cursor has been advanced to the end of the data (SQLCODE=100); at that point, %ROWCOUNT is set to the total number of rows selected. Refer to the [FETCH](#) command for further details.

Uses of SELECT

You can use a **SELECT** statement in the following contexts:

- As an independent query.
- As a subquery, a **SELECT** statement that supplies values to a clause of an enclosing **SELECT** statement. A subquery in a **SELECT** statement can be specified in the [select-item](#) list, in a [FROM](#) clause, or in a **WHERE** clause with an [EXISTS](#) or [IN](#) predicate. A subquery can also be specified in an [UPDATE](#) or [DELETE](#) statement. A subquery must be enclosed in parentheses.

- As a leg of a **UNION**. The **UNION** statement allows you to combine two or more **SELECT** statements into a single query.
- As part of a **CREATE VIEW** defining the data available to the view.
- As part of a **DECLARE CURSOR** used with Embedded SQL.
- As part of an **INSERT with a SELECT**. An **INSERT** statement can use a **SELECT** to insert data values for multiple rows into a table, selecting the data from another table.

You can enclose the entire **SELECT** statement with one or more sets of parentheses, as follows:

- Parentheses are optional for an independent **SELECT** query, a **UNION** leg **SELECT** query, a **CREATE VIEW SELECT** query, or a **DECLARE CURSOR SELECT** query. Enclosing a **SELECT** query in parentheses causes it to follow the syntax rules for a subquery; specifically, an **ORDER BY** clause must be paired with a **TOP** clause.
- Parentheses are mandatory for a subquery. One set of parentheses is mandatory; you can specify additional optional sets of parentheses.
- Parentheses are not permitted for an **INSERT** statement **SELECT** query.

Specifying optional parentheses generates a separate **cached query** for each set of parentheses added.

Privileges

To perform a **SELECT** query on one or more tables, you must either have column-level **SELECT** privileges for all of the specified *select-item* column(s), or table-level **SELECT** privileges for the specified *table-ref* table(s) or view(s). A *select-item* column specified using a table alias (such as `t .Name` or `"MyAlias" .Name`) only requires column-level **SELECT** privilege, not table-level **SELECT** privilege.

When using **SELECT ***, note that column-level privileges cover all table columns named in the **GRANT** statement; table-level privileges cover all table columns, including those added after the privilege was assigned.

Failing to have the necessary privileges results in an **SQLCODE -99** error (Privilege Violation). You can determine if the current user has **SELECT** privilege by invoking the **%CHECKPRIV** command. You can determine if a specified user has table-level **SELECT** privilege by invoking the **\$\$SYSTEM.SQL.CheckPriv()** method. For privilege assignment, refer to the **GRANT** command.

Note: Having table-level **SELECT** privilege for a table is not a sufficient test that the table actually exists. If the specified user has the **%All** role, **CheckPriv()** returns 1 even if the specified table or view does not exist.

A **SELECT** query that does not have a **FROM** clause does not require any **SELECT** privileges. A **SELECT** query that contains a **FROM** clause requires **SELECT** privilege, even if no column data is accessed by the query.

Required Clauses

The following are required clauses for all **SELECT** statements:

- A **select-item** list, a comma-separated list of one or more items (the *select-item* arguments) to be retrieved from the table or otherwise generated. Most commonly, these items are the names of columns in a table. The *select-item* consists of either a scalar expression specifying one or more individual items, or an asterisk (*) referring to all the columns of a base table.
- A **FROM clause** specifies one or more tables, views, or subqueries from which rows are to be retrieved. These tables may be associated by **JOIN** expressions. In Caché SQL a **FROM** clause with a valid *table-ref* is required for a **SELECT** that makes any reference to table data. A **FROM** clause is optional for a **SELECT** that does not access table data. The optional **FROM** clause is further described in the **FROM** clause reference page.

Optional Clauses

The following optional clauses operate on the virtual table that a FROM clause returns. All are optional, but, if used, must appear in the order specified:

- A **DISTINCT clause**, which specifies that only distinct (non-duplicate) values should be returned.
- A **TOP clause**, which specifies how many rows to return.
- A **WHERE clause**, which specifies boolean predicate conditions that rows must match. The WHERE clause predicate condition both determines which rows are returned and limits the values supplied to aggregate functions to the values from those rows. These conditions are specified by one or more **predicates** linked by logical operators; the WHERE clause returns all records that satisfy these predicate conditions. A WHERE clause predicate cannot include aggregate functions.
- A **GROUP BY clause**, which specifies a comma-delimited list of columns. These organize a query's result set into subsets with matching values for one or more columns and determine the ordering of the rows returned. GROUP BY allows scalar expressions as well as columns.
- A **HAVING clause**, which specifies boolean predicate conditions that rows must match. These conditions are specified by one or more **predicates** linked by logical operators. The HAVING clause predicate condition determines which rows are returned, but (by default) it does not limit the values supplied to aggregate functions to the values from those rows. This default can be overridden using the %AFTERHAVING keyword. A HAVING clause predicate can specify aggregate functions. These predicates typically operate on each group specified by a GROUP BY clause.
- An **ORDER BY clause**, which specifies the order in which rows should be displayed. An ORDER BY clause in a subquery or a **CREATE VIEW** query must be paired with a TOP clause.

Specifying **SELECT** clauses in the incorrect order generates an SQLCODE -25 error.

The DISTINCT Clause

The DISTINCT keyword clause causes redundant field values to be eliminated. It has two forms:

- **SELECT DISTINCT**: Returns one row for each unique combination of *select-item* values. You can specify one or more than one *select-items*. For example, the following query returns a row with Home_State and Age values for each unique combination of Home_State and Age values:

```
SELECT DISTINCT Home_State, Age FROM Sample.Person
```

- **SELECT DISTINCT BY (item)**: Returns one row for each unique combination of *item* values. You can specify a single *item* or a comma-separated list of *items*. The *select-item* list may, but does not have to, include the specified *item(s)*. For example, the following query returns a row with Name and Age values for each unique combination of Home_State and Age values:

```
SELECT DISTINCT BY (Home_State, Age) Name, Age FROM Sample.Person
```

An *item* can be any valid *select-item* value, except an asterisk. It cannot be a **column name alias**.

Either type of DISTINCT clause can specify more than one item to test for uniqueness. Listing more than one item retrieves all rows that are distinct for the combination of both items. DISTINCT does consider NULL a unique value. For further details, see the **DISTINCT clause** reference page.

The TOP Clause

The TOP keyword clause specifies that the **SELECT** statement return only a specified number of rows. It returns the specified number of rows that appear at the “top” of the returned virtual table. By default, which rows are the “top” rows of the table is unpredictable. However, Caché applies the **DISTINCT** and **ORDER BY** clauses (if specified) before selecting the TOP rows. For further details, see the **TOP clause** reference page.

The select-item

This is a mandatory element for all **SELECT** statements. Commonly, a *select-item* refers to a field in the table(s) specified in the FROM clause. A *select-item* consists of one or more of the following items, with multiple items separated by commas:

- A column name (field name), with or without a table name alias:

```
SELECT Name, Age FROM Sample.Person
```

Field names are not case-sensitive. However, the label associated with the field in the result set uses the letter case of the `SqlFieldName` as specified in the table definition, not the letter case specified in the *select-item*. See “[Field Column Alias](#)” for further details on letter case resolution.

A field name containing one or more underscores references an [embedded serial object](#) property. For example, for the field name `Home_City`, the table contains a referencing field `Home` that references an embedded serial object that defines the property `City`. For the field name `Home_Phone_AreaCode`, the table contains a referencing field `Home` that references an embedded serial object property `Phone` that references a nested embedded serial object that defines the property `AreaCode`. If you select a referencing field such as `Home` or `Home_Phone`, you receive the values of all of properties in the serial object in [%List data type](#) format.

To list all of the column names defined for a specified table, refer to [Column Names and Numbers](#) in the “[Defining Tables](#)” chapter of *Using Caché SQL*.

To display the [RowID](#) (record ID), you can use the [%ID pseudo-field variable](#) alias, which displays the RowID regardless of what name it is assigned. By default, the name of the RowID is `ID`, but Caché may rename it if there is a user-defined field named `ID`. By default, RowID is a hidden field.

A **SELECT** on a stream field returns the `oref` (object reference) of the opened stream object:

```
SELECT Name, Picture FROM Sample.Employee WHERE Picture IS NOT NULL
```

When the FROM clause specifies more than one table or view, you must include the table name (or a table name alias) as part of the *select-item*, using periods, as shown in the following two examples:

Full table name:

```
SELECT Sample.Person.Name, Sample.Employee.Company
FROM Sample.Person, Sample.Employee
```

Table name alias:

```
SELECT p.Name, e.Company
FROM Sample.Person AS p, Sample.Employee AS e
```

However, you cannot use a full table name as part of a *select-item* if an alias has been assigned to that table name.

Attempting to so results in an `SQLCODE -23` error.

You can use a [collation function](#) to specify the sorting and display of a *select-item* field. You can supply the collation function without parentheses (`SELECT %SQLUPPER Name`) or with parentheses (`SELECT %SQLUPPER (Name)`). If the collation function specifies truncation, the parentheses are required (`SELECT %SQLUPPER (Name, 10)`).

When the *select-item* references an [embedded serial object](#) property (embedded serial class data), use underline syntax. Underline syntax consists of the name of the object property, an underscore, and the property within the embedded object: for example, `Home_City` and `Home_State`. (In other contexts, index tables for example, these are represented using dot syntax: `Home.City`.)

```
SELECT Home_City, Home_State FROM Sample.Person
```

You can use **SELECT** to directly query a referencing field (such as `Home`), rather than using underline syntax. Because the data returned is in list format, you may want to use a `$LISTTOSTRING` or `$LISTGET` function to display the data. For example:

```
SELECT $LISTTOSTRING(Home, '^') AS HomeAddress FROM Sample.Person
```

- A subquery. A subquery returns a single column from a specified table. This column can be the values of a single table field (SELECT Name), or the values of multiple table fields returned as a single column, either by using concatenation (SELECT Home_City||Home_State) or by specifying a container field (SELECT Home). A subquery can use arrow syntax. A subquery cannot use asterisk syntax, even when the table cited in the subquery has only one data field.

One common use of a subquery is to specify an aggregate function that is not subject to the GROUP BY clause. In the following example, the GROUP BY clause groups ages by decades (for example, 25 through 34). The AVG(Age) *select-item* gives the average age of each group, as defined by the GROUP BY clause. In order to get the average age of all of the records in all groups, it uses a subquery:

```
SELECT Age AS Decade,
       COUNT(Age) AS PeopleInDecade,
       AVG(Age) AS AvgAgeForDecade,
       (SELECT AVG(Age) FROM Sample.Person) AS AvgAgeAllDecades
FROM Sample.Person
GROUP BY ROUND(Age, -1)
ORDER BY Age
```

- Arrow syntax, used to access a field from a table other than the FROM clause table. This is known as an implicit join. In the following example, the Sample.Employee table contains a Company field containing the RowID for the corresponding company name in the Sample.Company table. The arrow syntax retrieves the company name from that table:

```
SELECT Name, Company->Name AS CompanyName
FROM Sample.Employee
```

In this case, you must have SELECT privileges for the referenced table: either table-level SELECT privilege, or column-level SELECT privilege for both the referenced field and the RowID column of the referenced table. For further details on arrow syntax, refer to [Implicit Joins \(Arrow Syntax\)](#) in *Using Caché SQL*.

- Asterisk syntax (*), which selects all the columns in a table in [column number order](#):

```
SELECT TOP 5 * FROM Sample.Person
```

Asterisk syntax selects [embedded serial object](#) properties (fields), including properties from a serial object nested within a serial object. A field referencing a serial object is not selected. For example, the Home_City property from an embedded serial object is selected, but the Home referencing field used to access the Sample.Address embedded serial class (which contains the City property) is not selected.

Asterisk syntax does not select hidden fields. By default, the [RowID](#) is hidden (not displayed by SELECT *). However, if the table was defined with %PUBLICROWID, SELECT * returns the RowID field and all non-hidden fields. By default, the name of this field is ID, but Caché may rename it if there is a user-defined field named ID.

Note: Most of the example tables supplied in the Samples namespace are defined with %PUBLICROWID.

If the *select-item* is an asterisk and more than one table is specified, it selects all the columns in all of the joined tables:

```
SELECT TOP 5 * FROM Sample.Company, Sample.Employee
```

Asterisk syntax can be qualified or unqualified. If the *select-item* is qualified by prefixing a table name (or table name alias) and period (.) before the asterisk, the *select-item* selects all the columns in the specified table. Qualified asterisk syntax can be combined with other select items for other tables.

In the following example, *select-item* consists of an unqualified asterisk syntax that selects all columns from the table. Note that you can also specify duplicate column names (in this case Name) and non-column *select-item* elements (in this case {fn NOW}):

```
SELECT TOP 5 {fn NOW} AS QueryDate,
           Name AS Client,
           *
FROM Sample.Person
```

In the following example, *select-item* consists of qualified asterisk syntax that selects all columns from one table, and a list of column names from another table.

```
SELECT TOP 5 E.Name AS EmpName,
            C.*,
            E.Home_State AS EmpState
FROM Sample.Employee AS E, Sample.Company AS C
```

Note: `SELECT *` is a fully supported part of Caché SQL that can be extremely convenient during application development and debugging. However, in production applications the preferred programming practice is to explicitly list the selected fields, rather than using the asterisk syntax form. Explicitly listing fields makes your application clearer and easier to understand, easier to maintain, and easier to search for fields by name.

- A *select-item* containing one or more SQL [aggregate functions](#). An aggregate function always returns a single value. The argument of an aggregate function may be any of the following:
 - A single column name—computes the aggregate for all non-null values of the rows selected by the query:


```
SELECT AVG(Age) FROM Sample.Person
```
 - A scalar expression is also permitted to compute an aggregate:


```
SELECT SUM(Age) / COUNT(*) FROM Sample.Person
```
 - Asterisk syntax (*) — used with the **COUNT** function to compute the number of rows in the table:


```
SELECT COUNT(*) FROM Sample.Person
```
 - A select distinct function — computes the aggregate by eliminating redundant values:


```
SELECT COUNT(DISTINCT Home_State) FROM Sample.Person
```
 - While ANSI SQL does not allow the combination of column names and aggregate functions in a single **SELECT** statement, Caché SQL extends the standard by allowing this:


```
SELECT Name, COUNT(DISTINCT Home_State) FROM Sample.Person
```
 - An aggregate function using `%FOREACH`. This causes the aggregate to be computed for each distinct value of a column or columns:


```
SELECT DISTINCT Home_State, AVG(Age %FOREACH(Home_State))
FROM Sample.Person
```
 - An aggregate function using `%AFTERHAVING`. This causes the aggregate to be computed on a sub-population specified with the `HAVING` clause:


```
SELECT Name, AVG(Age %AFTERHAVING)
FROM Sample.Person
HAVING (Age > AVG(Age))
```

would return those records where Age is greater than average age, giving the average age for those persons whose age is above the average for all persons in the database.
- A user-defined class method stored as a [procedure](#). May be an unqualified method name or a qualified method name. The following are all valid class method names: `RandLetter()` an [unqualified name](#) for which a schema is supplied; `Sample.RandLetter()` a qualified class method name; and `Sample.Rand_Letter()` invoking class method "Rand_Letter"(). In the following example, `RandCaseLetter()` is a class method that returns a random letter, in either uppercase ('U') or lowercase ('L'):


```
SELECT RandCaseLetter('U')
```

The return value from the method is automatically converted from Logical format to Display/ODBC format. An input value to the method is, by default, not converted from Display/ODBC format to Logical format. However, input display-to-logical conversion can be configured system-wide using the `$$SYSTEM.SQL.SetSQLFunctionArgConversion()` method. You can use `$$SYSTEM.SQL.GetSQLFunctionArgConversion()` to determine the current configuration of this option.

If the specified method does not exist in the current namespace, the system generates a SQLCODE -359 error. If the specified method is ambiguous (could refer to more than one method), the system generates a SQLCODE -358 error. For further details on creating a class method, refer to [CREATE METHOD](#).

- A user-supplied ObjectScript function call (extrinsic function) operating on a database column:

```
SELECT $$REFORMAT^ABC(name)FROM MyTable
```

```
MySQL
&sql(SELECT Name,$$MyFunc() INTO :n,:f FROM Sample.Person)
WRITE "name is: ",n,!
WRITE "function value is: ",f,!
QUIT
MyFunc()
SET x="my text"
QUIT x
```

You can only invoke user-supplied (extrinsic) functions within an SQL statement if you configured this option system-wide. The default is “No”; by default, attempting to invoke user-supplied functions issues an SQLCODE -372 error. To configure use of extrinsic functions either use the `SetAllowExtrinsicFunctions()` method of the `%SYSTEM.SQL` class, or the Management Portal SQL configuration interface. From the Management Portal, select **System Administration, Configuration**, then **SQL and Object Settings**, then **General SQL Settings**. View and edit the current setting of **Allow Extrinsic Functions in SQL Statements**. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`.

You cannot use a user-supplied function to call a % routine (a routine with a name that begins with the % character). Attempting to do so issues an SQLCODE -373 error.

- A *select-item* that applies additional processing to a field value:

Arithmetic operations:

```
SELECT Name, Age, Age-AVG(Age) FROM Sample.Person
```

If a *select-item* arithmetic operation includes division, and there are any values for that field in the database that could produce a divisor with a value of zero or a NULL value, you cannot rely on order of testing to avoid division by zero. Instead, use a case statement to suppress the risk.

SQL functions:

```
SELECT Name,$LENGTH(Name) FROM Sample.Person
```

SQL case conversion functions:

```
SELECT Name,UCASE(Name) FROM Sample.Person
```

An **XMLELEMENT**, **XMLFOREST**, or **XMLCONCAT** function, which place XML (or HTML) tags around the data values retrieved from specified column names. Refer to [XMLELEMENT](#) for further details.

- A *select-item* that returns the same value for all records.

When all of the *select-items* references *no* table data, the FROM clause is optional. If you include the FROM clause, the specified table must exist. For further details on optional FROM clause, refer to the [FROM](#) clause reference page.

– Arithmetic operations:

```
SELECT 7 * 7, 7 * 8 FROM Sample.Person
```

```
SELECT Name, Age, 9 - 6 FROM Sample.Person
```

- A string literal or a function operating on a string literal:

```
SELECT UCASE('fred') FROM Sample.Person
```

String literals can be used to produce a more readable output, as shown in the following example:

```
SELECT TOP 10 Name, 'was born on', %EXTERNAL(DOB)
FROM Sample.Person
```

The way a numeric literal is specified determines its data type. Therefore, the string '123' is reported as data type VARCHAR, and the numeric 123 is reported as data type INTEGER or NUMERIC.

- A [%TABLENAME](#), or [%CLASSNAME](#) pseudo-field variable keyword. %TABLENAME returns the current table name. %CLASSNAME returns the name of the class corresponding to the current table. If the query references multiple tables, you can prefix the keyword with a table alias. For example, t1.%TABLENAME.
- One of the following ObjectScript special variables (or their abbreviations): [\\$HOROLOGY](#), [\\$JOB](#), [\\$NAMESPACE](#), [\\$TLEVEL](#), [\\$USERNAME](#), [\\$ZHOROLOGY](#), [\\$ZJOB](#), [\\$ZNSPACE](#), [\\$ZPI](#), [\\$ZTIMESTAMP](#), [\\$ZTIMEZONE](#), [\\$ZVERSION](#).

The Column Alias

When specifying a *select-item*, you can use the AS keyword to specify an alias for the name of a column:

```
SELECT Name AS PersonName, DOB AS BirthDate, ...
```

The column alias is displayed as the column header in the result set. Specifying a column alias is optional; a default is always provided. A column alias is displayed with the specified letter case; it is not, however, case-sensitive when referenced in an ORDER BY clause. The *c-alias* name must be a valid [identifier](#). A *c-alias* name can be a [delimited identifier](#). For further details see the “Identifiers” chapter of *Using Caché SQL*.

The AS keyword is not required, but makes the query text easier to read. Thus the following is also valid syntax:

```
SELECT Name PersonName, DOB BirthDate, ...
```

SQL does not perform uniqueness checking for column aliases. It is possible (though not desirable) for a field column and a column alias to have the same name, or for two column aliases to be identical. Such non-unique column aliases may cause an SQLCODE -24 “Ambiguous sort column” error when referenced by an ORDER BY clause. Column aliases, like all SQL identifiers, are not case-sensitive.

Use of column aliases in other **SELECT** clauses is governed by [query semantic processing order](#). You can reference a column by its column alias in an **ORDER BY** clause. You cannot reference a column alias in another *select-item* in the select list, in a **DISTINCT BY** clause, a **WHERE** clause, a **GROUP BY** clause, or a **HAVING** clause. You cannot reference a column alias in a **JOIN** operation’s **ON** clause or **USING** clause. You can, however, use a subquery to make a column alias available for use by other these other **SELECT** clauses, as shown in the “[Querying the Database](#)” chapter of *Using Caché SQL*.

Field Column Aliases

A *select-item* field name is not case-sensitive. However, unless you supply a column alias, the name of a field column in the result set follows the letter case of the [SqlFieldName](#) associated with the column property. The letter case of the [Sql-FieldName](#) corresponds to the field name as specified in the table definition, not as specified in the *select-item* list. Therefore, `SELECT name FROM Sample.Person` returns the field column label as Name. Using a field column alias allows you to specify the letter case to display, as shown in the following example:

```
SELECT name,name AS NAME
FROM Sample.Person
```

Letter case resolution takes time. To maximize **SELECT** performance, you can specify the exact letter case of the field name, as specified in the table definition. However, determining the exact letter case of a field in the table definition is often inconvenient and prone to error. Instead, you can use a field column alias to avoid letter case issues. Note that all references to the field column alias must match in letter case.

The following Dynamic SQL example requires letter case resolution (the `SqlFieldNames` are “Latitude” and “Longitude”):

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT latitude,longitude FROM Sample.USZipCode"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {WRITE rset.latitude," ",rset.longitude,! }
```

The following Dynamic SQL example does not require letter case resolution, and therefore executes faster:

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT latitude AS northsouth,longitude AS eastwest FROM Sample.USZipCode"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {WRITE rset.northsouth," ",rset.eastwest,! }
```

The *t-alias* table alias prefix is not included in the column name. Therefore, in the following example, both columns are labeled as Name:

```
SELECT p.Name,e.Name
FROM Sample.Person AS p LEFT JOIN Sample.Employee AS e ON p.Name=e.Name
```

To distinguish the columns in a query that specifies multiple tables, you should specify column aliases:

```
SELECT p.Name AS PersonName,e.Name AS EmployeeName
FROM Sample.Person AS p LEFT JOIN Sample.Employee AS e ON p.Name=e.Name
```

You may also wish to provide a column alias to make the data easier to understand. In the following example, the table column “Home_State” is renamed “US_State_Abbrev”:

```
SELECT Name,Home_State AS US_State_Abbrev
FROM Sample.Person
```

Note that `%ID` references a specific column, and therefore returns that field name (ID, by default), or a specified column alias, as shown in the following example:

```
SELECT %ID,%ID AS Ident,Name
FROM Sample.Person
```

Non-Field Column Aliases

Non-field columns are automatically assigned a column name. If you provide no alias for such fields, Caché SQL supplies a unique column name, such as “Expression_1”, or “Aggregate_3”. The integer suffix refers to the *select-item* position as specified in the **SELECT** statement. They are *not* a count of fields of that type.

The following are automatically assigned column names (*n* is an integer):

- **Aggregate_n**: an **aggregate function**, such as `AVG(Age)` or `COUNT(*)`. A column is named `Aggregate_n` if the outermost operation is an aggregate function, even when this aggregate contains an expression. For example, `COUNT(Name)+COUNT(Spouse)` is `Expression_n`, but `MAX(COUNT(Name)+COUNT(Spouse))` is `Aggregate_n`.
- **HostVar_n**: a host variable. This may be a literal, such as ‘text’, 123, or the empty string (‘’), an input variable (`:myvar`), or a **? input parameter** replaced by a literal. Note that any expression evaluation on a literal, such as appending a sign to a number, string concatenation, or an arithmetic operation, makes it an `Expression_n`. A literal value supplied to a `?` parameter is returned unchanged without expression evaluation. For example, supplying `5+7` returns the string ‘5+7’ as `HostVar_n`.

- `Literal_n`: a [pseudo-field variable](#) such as `%TABLENAME`, or the NULL specifier. Note that `%ID` is not `Literal_n`; it is given the column name of the actual `RowID` field.
- `Subquery_n`: the result of a subquery that specifies a single *select-item*. The *select-item* may be a field, aggregate function, expression, or literal. You specify the column alias after the subquery, not within the subquery.
- `Expression_n`: any operation in the *select-item* list on a literal, a field, or on an `Aggregate_n`, `HostVar_n`, `Literal_n`, or `Subquery_n` *select-item* changes its column name to `Expression_n`. This includes unary operations on numbers (`-Age`), arithmetic operations (`Age+5`), concatenation (`'USA: ' || Home_State`), data type **CAST** operations, SQL collation functions (`%UPPER(Name)`), SQL scalar functions (`$LENGTH(Name)`), user-defined class methods, **CASE** expressions, and special variables (such as `CURRENT_DATE` or `$ZPI`).

In the following example, the aggregate field column created by the `AVG` function is given the column alias “`AvgAge`”; its default name is “`Aggregate_3`” (an aggregate field in position 3 in the **SELECT** list).

```
SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person
```

The following example is identical to the previous, except that the `AS` keyword is here omitted. The use of this keyword is recommended, but not required.

```
SELECT Name, Age, AVG(Age) AvgAge FROM Sample.Person
```

The following example show how to specify a column alias for a *select-item* subquery:

```
SELECT Name AS PersonName,
       (SELECT Name FROM Sample.Employee) AS EmpName,
       Age AS YearsOld
FROM Sample.Person
```

FROM Clause

The `FROM` *table-ref* clause specifies one or more [tables](#), [views](#), table-valued functions, or [subqueries](#). You can specify any combination of these *table-ref* types as a comma-separated list or with `JOIN` syntax. If you specify a single *table-ref*, the specified data is retrieved from that table or view. If you specify multiple *table-refs*, SQL performs a join operation on the tables, merging their data into a results table from which the specified data is retrieved.

If you specify more than one *table-ref*, you can separate these table names with commas or with explicit join syntax keywords. For further details on specifying a comma-separated list of table names, see the [FROM](#) clause reference page. For further details on specifying multiple table names with explicit `JOIN` syntax (such as `RIGHT JOIN` or `*=`) see the [JOIN](#) reference page.

You can use the `$$SYSTEM.SQL.TableExists()` or `$$SYSTEM.SQL.ViewExists()` method to determine whether a table or view exists in the current namespace. You can use the `$$SYSTEM.SQL.CheckPriv()` method to determine if you have [SELECT](#) privileges for that table or view.

The Table Alias

When specifying a *table-ref*, you can use the `AS` keyword to specify an alias for that table name or view name:

```
FROM Sample.Person AS P
```

The `AS` keyword is not required, but makes the query text easier to read. The following is valid equivalent syntax:

```
FROM Sample.Person P
```

The *t-alias* name must be a valid [identifier](#). A *t-alias* name can be a [delimited identifier](#). A *t-alias* must be unique among table aliases within the query. A *t-alias*, like all identifiers, is not case-sensitive. Therefore, you cannot specify two *t-alias* names that differ only in letter case. This results in an `SQLCODE -20` “Name conflict” error. For further details see the “Identifiers” chapter of *Using Caché SQL*.

The table alias is used as a prefix (with a period) to a field name to indicate the table to which the field belongs. For example:

```
SELECT P.Name, E.Name
FROM Sample.Person AS P, Sample.Employee AS E
```

You must use a table reference prefix when a query specifies multiple tables that have the same field name. A table reference prefix can be a *t-alias* (as shown above) or it can be the fully qualified table name, as shown in the following equivalent example:

```
SELECT Sample.Person.Name, Sample.Employee.Name
FROM Sample.Person, Sample.Employee
```

However, you cannot use a full table name as part of a *select-item* if a *t-alias* has been assigned to that table name. Attempting to so results in an SQLCODE -23 error.

Specifying a table alias is optional when a query references only one table (or view). Specifying a table alias is optional (but recommended) when a query references multiple tables (and/or views) and the field names referenced are unique to each table. Specifying a table alias is required when a query references multiple tables (and/or views) and the field names referenced are the same in different tables. Failing to specify a *t-alias* (or fully qualified table name) prefix results in an SQLCODE -27 “Field %1 is ambiguous among the applicable tables” error.

A *t-alias* can be used, but is not required, when specifying a subquery such as the following:

```
SELECT Name, (SELECT Name FROM Sample.Vendor)
FROM Sample.Person
```

A *t-alias* only uniquely identifies a field for query execution; to uniquely identify a field for query result set display you must also use a column alias (*c-alias*). The following example uses both table aliases (Per and Emp) and column aliases (PName and EName):

```
SELECT Per.Name AS PName, Emp.Name AS EName
FROM Sample.Person AS Per, Sample.Employee AS Emp
WHERE Per.Name %STARTSWITH 'G'
```

You can use the same name for a field, a column alias, and/or a table alias without a naming conflict.

A *t-alias* prefix is used wherever it is necessary to distinguish which table is being referred to. Some examples of this are shown in the following:

```
SELECT P.%ID As PersonID,
       AVG(P.Age) AS AvgAge,
       Z.%TABLENAME||'=' AS Tablename,
       Z.*
FROM Sample.Person AS P, Sample.USZipCode AS Z
WHERE P.Home_City = Z.City
GROUP BY P.Home_City
ORDER BY Z.City
```

WHERE Clause

The WHERE clause qualifies or disqualifies specific rows from the query selection. The rows that qualify are those for which the *condition-expression* is true. The *condition-expression* is a list of logical tests (predicates) which can be linked by the AND and OR logical operators. These predicates may be inverted using the NOT unary logical operator.

The SQL predicates fall into the following categories:

- [Comparison Predicates](#)
- [BETWEEN Predicate](#)
- [LIKE Predicate](#)
- [NULL Predicate](#)
- [IN and %INLIST Predicates](#)

- [EXISTS Predicate](#)
- [FOR SOME Predicate](#)
- [FOR SOME %ELEMENT Predicate](#)

For further details on these logical predicates, see the [WHERE](#) clause reference page. The *condition-expression* cannot contain aggregate functions. If you wish to specify a selection condition using a value returned by an aggregate function, use a [HAVING](#) clause.

A WHERE clause can specify an explicit join between two tables using the = (inner join), =* (left outer join), and *= (right outer join) symbolic join operators. For further details, refer to the [JOIN](#) page of this manual.

A WHERE clause can specify an implicit join between the base table and a field from another table using the arrow syntax (→) operator. For further details, refer to [Implicit Joins](#) in *Using Caché SQL*.

GROUP BY Clause

The GROUP BY clause takes the resulting rows of a query and breaks them up into individual groups according to one or more database columns. When you use **SELECT** in conjunction with GROUP BY, one row is retrieved for each distinct value of the GROUP BY fields. The GROUP BY clause is conceptually similar to the Caché extension %FOREACH, but GROUP BY operates on an entire query, while %FOREACH allows selection of aggregates on sub-populations without restricting the entire query population. For instance:

```
SELECT Home_State, COUNT(Home_State) AS Population
FROM Sample.Person
GROUP BY Home_State
```

This query returns one row for each distinct Home_State.

For further details, see the [GROUP BY](#) clause reference page.

HAVING Clause

The HAVING clause is like a WHERE clause that operates on groups. It is typically used in combination with the GROUP BY clause, or with the %AFTERHAVING keyword. The HAVING clause qualifies or disqualifies specific rows from the query selection. The rows that qualify are those for which the *condition-expression* is true. The *condition-expression* is a list of logical tests (predicates) which can be linked by the AND and OR logical operators. The *condition-expression* can contain aggregate functions. For further details, see the [HAVING](#) clause reference page.

ORDER BY Clause

An ORDER BY clause consists of the ORDER BY keywords followed by a select-item or a comma-separated list of items that specify the order in which rows are displayed. Each item can have an optional ASC (ascending order) or DESC (descending order). The default is ascending order. An ORDER BY clause is applied to the results of a query, and is frequently paired with a TOP clause. For further details, see the [ORDER BY](#) clause reference page.

The following example returns the selected fields for all rows in the database, and orders these rows in ascending order by age:

```
SELECT Home_State, Name, Age
FROM Sample.Person
ORDER BY Age
```

SELECT and Transaction Processing

A transaction performing a query is defined as either READ COMMITTED or READ UNCOMMITTED. The default is READ UNCOMMITTED. A query that is not in a transaction is defined as READ UNCOMMITTED.

- If READ UNCOMMITTED, a **SELECT** returns the current state of the data, including changes made to the data by transactions in progress which have not been committed. These changes may be subsequently rolled back.

- If READ COMMITTED, the behavior depends on the contents of the **SELECT** statement. Normally, a **SELECT** statement in read committed mode would only return insert and update changes to data that has been committed. Data rows that have been deleted by a transaction in progress are not returned, even though these deletes have not been committed and may be rolled back.

However, if the **SELECT** statement contains a %NOLOCK keyword, a **DISTINCT clause**, or a **GROUP BY** clause, the **SELECT** returns the current state of the data, including changes made to data during the current transaction which have not been committed. An aggregate function in a **SELECT** also returns the current state of the data for the specified column(s), including uncommitted changes.

For further details, refer to [SET TRANSACTION](#) and [START TRANSACTION](#).

Query Metadata

You can use Dynamic SQL to return metadata about the query, such as the number of columns specified in the query, the name (or alias) of a column specified in the query, and the data type of a column specified in the query. For further details, refer to the [Dynamic SQL](#) chapter of *Using Caché SQL*, and the %SQL.Statement class in the *InterSystems Class Reference*.

Examples

The following four examples perform similar queries, using different combinations of **SELECT** clauses. Note that these clauses must be specified in the correct order. In all four examples, three fields are selected from the Sample.Person table: Name, Home_State, and Age, and two fields (AvgAge and AvgMiddleAge) are computed.

HAVING/ORDER BY

In the following example, the AvgAge computed field is computed on all records in Sample.Person. The HAVING clause governs the AvgMiddleAge computed field, calculating the average age of those over 40 from all records in Sample.Person. Thus, every row has the same value for AvgAge and AvgMiddleAge. The ORDER BY clause sequences the display of the rows alphabetically by the Home_State field value.

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
HAVING Age > 40
ORDER BY Home_State
```

WHERE/HAVING/ORDER BY

In the following example, the WHERE clause limits the selection to the seven specified northeastern states. The AvgAge computed field is computed on the records from those Home_States. The HAVING clause governs the AvgMiddleAge computed field, calculating the average age of those over 40 from the records from the specified Home_States. Thus, every row has the same value for AvgAge and AvgMiddleAge. The ORDER BY clause sequences the display of the rows alphabetically by the Home_State field value.

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
WHERE Home_State IN ('ME','NH','VT','MA','RI','CT','NY')
HAVING Age > 40
ORDER BY Home_State
```

GROUP BY/HAVING/ORDER BY

The GROUP BY clause causes the AvgAge computed field to be separately computed for each Home_State group. The GROUP BY clause also limits the output display to the first record encountered from each Home_State. The HAVING clause governs the AvgMiddleAge computed field, calculating the average age of those over 40 in each Home_State group. The ORDER BY clause sequences the display of the rows alphabetically by the Home_State field value.

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
GROUP BY Home_State
HAVING Age > 40
ORDER BY Home_State
```

WHERE/GROUP BY/HAVING/ORDER BY

The WHERE clause limits the selection to the seven specified northeastern states. The GROUP BY clause causes the AvgAge computed field to be separately computed for each of these seven Home_State groups. The GROUP BY clause also limits the output display to the first record encountered from each specified Home_State. The HAVING clause governs the AvgMiddleAge computed field, calculating the average age of those over 40 in each of the seven Home_State groups. The ORDER BY clause sequences the display of the rows alphabetically by the Home_State field value.

```
SELECT Name,Home_State,Age,AVG(Age) AS AvgAge,
AVG(Age %AFTERHAVING) AS AvgMiddleAge
FROM Sample.Person
WHERE Home_State IN ('ME','NH','VT','MA','RI','CT','NY')
GROUP BY Home_State
HAVING Age > 40
ORDER BY Home_State
```

Embedded SQL and Dynamic SQL Examples

Embedded SQL and Dynamic SQL can be used to issue a **SELECT** query from within a program in another language. Embedded SQL can be included in ObjectScript code. Dynamic SQL can be included in either ObjectScript code or Caché Basic code.

The following embedded SQL program retrieves data values from one record and places them in the output [host variables](#) specified in the [INTO clause](#).

```
NEW SQLCODE,%ROWCOUNT
&sql(SELECT Home_State,Name,Age
      INTO :a, :b, :c
      FROM Sample.Person)
IF SQLCODE=0 {
  WRITE !,"  Name=",b
  WRITE !,"  Age=",c
  WRITE !,"  Home State=",a
  WRITE !,"Row count is: ",%ROWCOUNT }
ELSE {
  WRITE !,"SELECT failed, SQLCODE=",SQLCODE }
```

This program retrieves (at most) one row, so the %ROWCOUNT variable is set to either 0 or 1. To retrieve multiple rows, you must declare a cursor and use the [FETCH](#) command. For further details, refer to the [Embedded SQL](#) chapter in *Using Caché SQL*.

The following Dynamic SQL example first tests whether the desired table exists and checks the current user's SELECT privilege for that table. It then executes the query and returns a result set. It uses the **WHILE** loop to repeatedly invoke the **%Next** method for the first 10 records of the result set. It displays three field values using **%GetData** methods that specify the field position as specified in the **SELECT** statement:

```
ZNSPACE "Samples"
SET tname="Sample.Person"
IF $SYSTEM.SQL.TableExists(tname)
  & $SYSTEM.SQL.CheckPriv($USERNAME,"1","_tname","s")
  {GOTO SpecifyQuery}
ELSE {WRITE "Table unavailable" QUIT}
SpecifyQuery
SET myquery = 3
SET myquery(1) = "SELECT Home_State,Name,SSN,Age"
SET myquery(2) = "FROM "_tname
SET myquery(3) = "ORDER BY Name"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 {
  SET x=0
  WHILE x < 10 {
```

```
SET x=x+1
SET status=rset.%Next()
WRITE rset.%GetData(2)," " /* Name field */
WRITE rset.%GetData(1)," " /* Home_State field */
WRITE rset.%GetData(4),! /* Age field */
}
WRITE !,"End of Data"
WRITE !,"SQLCODE=",rset.%SQLCODE," Row Count=",rset.%ROWCOUNT
}
ELSE {
WRITE !,"SELECT failed, SQLCODE=",rset.%SQLCODE }
```

For further details, refer to the [Dynamic SQL](#) chapter in *Using Caché SQL*.

See Also

- SELECT clauses: [DISTINCT](#), [FROM](#), [GROUP BY](#), [HAVING](#), [INTO](#), [ORDER BY](#), [TOP](#), [WHERE](#)
- [JOIN](#), [UNION](#)
- [CREATE VIEW](#)
- [CREATE TABLE](#), [ALTER TABLE](#), [DROP TABLE](#)
- [CREATE QUERY](#), [DROP QUERY](#)
- [INSERT](#), [INSERT OR UPDATE](#), [UPDATE](#), [DELETE](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

SET OPTION

Sets an execution option.

```
SET OPTION option_keyword = value
```

Description

The **SET OPTION** statement is used to set execution options, such as the compile mode, SQL configuration settings, and the locale settings governing date, time, and numeric conventions. Only one keyword option can be set by each **SET OPTION** statement.

SET OPTION can be used in [Dynamic SQL](#) or [Embedded SQL](#). **SET OPTION** cannot be issued from the [SQL Shell](#).

Other **SET OPTION** arguments (not documented here) are parsed by Caché for SQL compatibility, but perform no operation.

Because **SET OPTION** prepares and executes quickly, and is generally run only once, Caché does not create a cached query for **SET OPTION** in ODBC, JDBC, or Dynamic SQL.

The following options are supported by Caché:

COMPILEMODE

The COMPILEMODE option sets the compile mode to DEFERRED, IMMEDIATE, INSTALL, or NOCHECK for the current namespace. The default is IMMEDIATE. Changing from DEFERRED to IMMEDIATE compile mode causes any classes in the Deferred Compile Queue to be compiled immediately. If all class compilations are successful, Caché sets SQLCODE to 0. If there are any errors, SQLCODE is set to -400. Class compilation errors are logged in the ^mtemp2 ("Deferred Compile Mode", "Error"). If SQLCODE is set to -400, you should view this global structure for more precise error messages. The INSTALL compile mode is similar to the DEFERRED compile mode, but it should only be used for DDL installations where there is no data in the tables.

The NOCHECK compile mode is similar to IMMEDIATE, except that it skips checking of the following constraints when compiling: If a table is dropped, Caché does not check foreign key constraints in other tables that reference the dropped table. If a foreign key constraint is added, Caché does not check existing data to ensure that it is valid for this foreign key. If a NOT NULL constraint is added, Caché does not check existing data for NULLs or assign the field's default value. If a UNIQUE or Primary Key constraint is deleted, Caché does not check if a foreign key in this table or another table references the dropped key.

LOCK_TIMEOUT

The LOCK_TIMEOUT numeric option lets you set the default lock timeout for the current process. The LOCK_TIMEOUT *value* is the number of seconds to wait when trying to establish a lock during SQL execution. This lock timeout is used when a locking conflict prevents the current process from immediately locking a record, table, or other entity for a **LOCK**, **INSERT**, **UPDATE**, **DELETE**, or **SELECT** operation. Caché SQL continue to try to establish the lock until the timeout expires, at which point an SQLCODE -110 or -114 error is generated.

Available values are positive integers and zero. The timeout setting is per process. You can determine the lock timeout setting for the current process using the **GetProcessLockTimeout()** method.

If you do not set the lock timeout for the current process, it defaults to the current system-wide lock timeout setting. If your ODBC connection disconnects and reconnects, the reconnected process uses the current system-wide lock timeout setting. The default system-wide lock timeout is 10 seconds.

For further details on locking conflicts and per-process and system-wide SQL lock timeout settings, refer to the [LOCK](#) command.

PKEY_IS_IDKEY

The PKEY_IS_IDKEY boolean option specifies whether primary keys are also ID keys system-wide. Available values are TRUE and FALSE. If TRUE, the primary key is created as an ID key. (That is, the primary key of the table also becomes the IDKey index in the class definition.) This may improve performance, but has the limitation that a primary key thus created cannot be subsequently modified. Once set, you cannot change the value assigned to a primary key, nor can you assign a different key as the primary key. Use of this option also changes the primary key collation default; primary key string values default to EXACT collation. If FALSE, the primary key and ID key are defined as independent, primary key values are changeable, and primary key string values default to the current [collation type default](#), which is SQLUPPER by default.

To set the PKEY_IS_IDKEY option, you must have the %Admin_Manage:USE privilege. Otherwise, you receive an SQLCODE -99 error (Privilege Violation). Once set, this option takes effect system-wide for all processes. The system-wide default for this option can be set using:

- The `$$SYSTEM.SQL.SetDDLPrimaryKeyNotIDKey()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Are Primary Keys Created through DDL not ID Keys**. If set to “Yes” (1), when a Primary Key constraint is specified through DDL it does not automatically become the IDKey index in the class definition. If “No” (0), it does become the IDKey index. Setting this value to “No” can give better performance, but means that the Primary Key fields cannot be updated. “Yes” is the default.

The PKEY_IS_IDKEY setting remains in effect until reset through another **SET OPTION PKEY_IS_IDKEY** or until the Caché Configuration is reactivated, which resets this parameter to the Caché System Configuration setting.

SUPPORT_DELIMITED_IDENTIFIERS

The SUPPORT_DELIMITED_IDENTIFIERS boolean option specifies whether delimited identifiers are supported system-wide. Available values are TRUE and FALSE. If TRUE, a string delimited by double quotation marks is considered an identifier within an SQL statement. If FALSE, a string delimited by double quotation marks is considered a string literal within an SQL statement.

To set the SUPPORT_DELIMITED_IDENTIFIERS option, you must have the %Admin_Manage:USE privilege. Otherwise, you receive an SQLCODE -99 error (Privilege Violation). Once set, this option takes effect system-wide for all processes. The SUPPORT_DELIMITED_IDENTIFIERS setting remains in effect until reset through another **SET OPTION SUPPORT_DELIMITED_IDENTIFIERS** or until the Caché Configuration is reactivated, which will reset this parameter to the System Configuration setting in the Management Portal.

The system-wide default for this option can be set as follows:

- The `$$SYSTEM.SQL.SetDelimitedIdentifiers()` method call. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View the current setting of **Support Delimited Identifiers**.

The default is “Yes” (1). If set to “Yes”, delimited identifiers are supported system-wide. For further details on [delimited identifiers](#), see the “Identifiers” chapter of *Using Caché SQL*.

Locale Options

Locale options are keyword options used to set your Caché Locale settings for date, time, and numeric conventions for the current process. The available keyword options are AM, DATE_FORMAT, DATE_MAXIMUM, DATE_MINIMUM, DATE_SEPARATOR, DECIMAL_SEPARATOR, MIDNIGHT, MINUS_SIGN, MONTH_ABBR, MONTH_NAME, NOON, NUMERIC_GROUP_SEPARATOR, NUMERIC_GROUP_SIZE, PM, PLUS_SIGN, TIME_FORMAT, TIME_PRECISION, TIME_SEPARATOR, WEEKDAY_ABBR, WEEKDAY_NAME, and YEAR_OPTION. All of these

options can be set to a literal, and all take a default (American English conventions). The `TIME_PRECISION` option is configurable (see below). If you set any of these options to an invalid value, Caché issues an `SQLCODE -129` error (Illegal value for **SET OPTION** locale property). See the ObjectScript [\\$ZDATETIME](#) function for an explanation of date and time formats and options.

Date/Time Option Keyword	Description
AM	String. Default is 'AM'
DATE_FORMAT	Integer. Default is 1. Available values are 0 through 15. For an explanation of these date formats, see the ObjectScript \$ZDATE function.
DATE_MAXIMUM	Integer. Default is 2980013 (12/31/9999). Can be set to an earlier date, but not to a later date.
DATE_MINIMUM	Positive Integer. Default is 0 (12/31/1840). Can be set to a later date, but not to an earlier date.
DATE_SEPARATOR	Character. Default is '/'
DECIMAL_SEPARATOR	Character. Default is '.'
MIDNIGHT	String. Default is 'MIDNIGHT'
MINUS_SIGN	Character. Default is '-'
MONTH_ABBR	String. Default is ' Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec'. (Note that this string begins with a space character, which is the default separator character.)
MONTH_NAME	String. Default is ' January February March April May June ... November December'. (Note that this string begins with a space character, which is the default separator character.)
NOON	String. Default is 'NOON'
NUMERIC_GROUP_SEPARATOR	Character. Default is ','
NUMERIC_GROUP_SIZE	Integer. Default is 3.
PM	String. Default is 'PM'
PLUS_SIGN	Character. Default is '+'
TIME_FORMAT	Integer. Default is 1. Available values are 1 through 4. For an explanation of these time formats, see the ObjectScript \$ZTIME function.
TIME_PRECISION	Integer from 0 through 9 (inclusive). Default is 0. The number of digits of fractional seconds. Configurable, as described below.
TIME_SEPARATOR	Character. Default is ':'
WEEKDAY_ABBR	String. Default is ' Sun Mon Tue Wed Thu Fri Sat'. (Note that this string begins with a space character, which is the default separator character.)
WEEKDAY_NAME	String. Default is ' Sunday Monday Tuesday Wednesday Thursday Friday Saturday'. (Note that this string begins with a space character, which is the default separator character.)

Date/Time Option Keyword	Description
YEAR_OPTION	Integer. Default is 0. Available values are 0 through 6. For an explanation of these ways of representing 2-digit and 4-digit years, see the ObjectScript \$ZDATE function.

To configure TIME_PRECISION system-wide, go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**. This specifies the number of digits of precision for fractional seconds. The default is 0. The range of allowed values is 0 through 9 digits of precision. The actual number of meaningful digits of fractional seconds is platform-dependent.

See Also

- SQL date and time functions: [CURRENT_TIMESTAMP](#) [DATEPART](#) [DATENAME](#) [GETDATE](#) [NOW](#)
- SQL date functions: [DAYNAME](#) [DAYOFWEEK](#) [DAYOFMONTH](#) [DAYOFYEAR](#) [WEEK](#) [MONTH](#) [MONTHNAME](#) [QUARTER](#) [YEAR](#) [CURDATE](#) [CURRENT_DATE](#) [TO_DATE](#)
- SQL time functions: [HOUR](#) [MINUTE](#) [SECOND](#) [CURTIME](#) [CURRENT_TIME](#)
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*
- ObjectScript functions: [\\$ZDATE](#) [\\$ZDATETIME](#) [\\$ZTIME](#)

SET TRANSACTION

Sets parameters for transactions.

```
SET TRANSACTION [%COMMITMODE commitmode]
```

```
SET TRANSACTION [transactionmodes]
```

Arguments

<code>%COMMITMODE <i>commitmode</i></code>	<i>Optional</i> — Specifies the manner in which transactions are committed to the database. Available values are EXPLICIT, IMPLICIT, and NONE. The default is IMPLICIT.
<code><i>transactionmodes</i></code>	<p><i>Optional</i> — Specifies the isolation mode and access mode for the transaction. You can specify a value for either an isolation mode, an access mode, or for both modes as a comma-separated list.</p> <p>Valid values for isolation mode are ISOLATION LEVEL READ COMMITTED, ISOLATION LEVEL READ UNCOMMITTED, and ISOLATION LEVEL READ VERIFIED. The default is ISOLATION LEVEL READ UNCOMMITTED.</p> <p>Valid values for access mode are READ ONLY and READ WRITE. Note that only ISOLATION LEVEL READ COMMITTED is compatible with access mode READ WRITE.</p>

Description

A **SET TRANSACTION** statement sets parameters that govern SQL transactions for the current process. These parameters take effect at the beginning of the next transaction and continue in effect for the duration of the current process or until explicitly reset. They do not automatically reset to defaults at the end of a transaction.

A single **SET TRANSACTION** statement can be used to set either the *commitmode* parameter or the *transactionmodes* parameters, but not both.

The same parameters can be set using the **START TRANSACTION** command, which can both set parameters and begin a new transaction. The parameters can also be set using method calls.

SET TRANSACTION does not begin a transaction, and therefore does not increment the **\$TLEVEL** transaction level counter.

%COMMITMODE

The **%COMMITMODE** keyword allows you to specify whether or not automatic transaction commitment is performed. The available options are:

- **IMPLICIT**: automatic transaction commitment is on (the default). SQL automatically initiates a transaction when a program issues a database modification operation (**INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE TABLE**). The transaction continues until either the operation completes successfully and SQL automatically commits the changes, or the operation is unable to complete successfully on all rows and SQL automatically rolls back the entire operation. Each database operation (**INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE TABLE**) constitutes a separate transaction.

Successful completion of the database operation automatically clears the rollback journal, releases locks, and decrements \$TLEVEL. No **COMMIT** statement is needed. This is the default setting.

- **EXPLICIT**: automatic transaction commitment is off. SQL automatically initiates a transaction when a program issues the first database modification operation (**INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE TABLE**). This transaction continues until it is explicitly concluded. Upon successful completion you issue a **COMMIT** statement. If a database modification operation fails you issue a **ROLLBACK** statement to revert the database to the point prior to the beginning of the transaction. In **EXPLICIT** mode the number of database operations per transaction is user-defined.
- **NONE**: no automatic transaction processing. A transaction is not initiated unless explicitly invoked by a **START TRANSACTION** statement. The transaction must be explicitly concluded by issuing either a **COMMIT** or **ROLLBACK** statement. Thus whether a database operation is included in a transaction, and the number of database operations in a transaction are both user-defined.

You can determine the %COMMITMODE setting for the current process using the **GetAutoCommit()** method, as shown in the following ObjectScript example:

```
DO $SYSTEM.SQL.SetAutoCommit($RANDOM(3))
SET x=$SYSTEM.SQL.GetAutoCommit()
IF x=1 {
    WRITE "%COMMITMODE IMPLICIT (default behavior):",!,
        "each database operation is a separate transaction",!,
        "with automatic commit or rollback" }
ELSEIF x=0 {
    WRITE "%COMMITMODE NONE:",!,
        "No automatic transaction support",!,
        "You must use START TRANSACTION to start a transaction",!,
        "and COMMIT or ROLLBACK to conclude one" }
ELSE {
    WRITE "%COMMITMODE EXPLICIT:",!,
        "the first database operation automatically",!,
        "starts a transaction; to end the transaction",!,
        "explicit COMMIT or ROLLBACK required" }
```

The %COMMITMODE can be set in ObjectScript using the **SetAutoCommit()** method call. The available method values are 0 (NONE), 1 (IMPLICIT), and 2 (EXPLICIT).

ISOLATION LEVEL

You specify an ISOLATION LEVEL for a process that is issuing a query. The ISOLATION LEVEL options permit you to specify whether or not changes that are in progress should be available for read access by the query. If another concurrent process is performing inserts or updates to a table and those changes to the table are in a transaction, those changes are in progress, and could, potentially, be rolled back. By setting the ISOLATION LEVEL for your process that is querying that table, you can specify whether you wish to include or exclude these changes in progress from the query results.

- **READ UNCOMMITTED** states that all changes are immediately available for query access. This includes changes that may subsequently be rolled back. **READ UNCOMMITTED** insures that your query will return results without waiting for a concurrent insert or update process, and will not fail due to a lock timeout error. However, the results of a **READ UNCOMMITTED** may include values that are not committed; these values may be internally inconsistent because the insert or update operation has only partially completed, and these values may be subsequently rolled back. **READ UNCOMMITTED** is the default if your query process is not in an explicit transaction, or if the transaction does not specify an ISOLATION LEVEL. **READ UNCOMMITTED** is incompatible with **READ WRITE** access; attempting to specify both in the same statement results in an SQLCODE -92 error.
- **READ VERIFIED** states that uncommitted data from other transactions is immediately available, and no locking is performed. This includes changes that may subsequently be rolled back. However, unlike **READ UNCOMMITTED**, a **READ VERIFIED** transaction will re-check any conditions that could be invalidated by uncommitted or newly committed data which would result in output that does not satisfy the query conditions. Because of this condition re-check, **READ VERIFIED** is more accurate but less efficient than **READ UNCOMMITTED** and should only be used when concurrent updates to the data being checked by the conditions is likely to occur. **READ VERIFIED** is incompatible with **READ WRITE** access; attempting to specify both in the same statement results in an SQLCODE -92 error.

- `READ COMMITTED` states that only those changes that have been committed are available for query access. This ensures that a query is performed on the database in a consistent state, not while a group of changes are being made, a group of changes which may be subsequently rolled back. If requested data has been changed, but the changes have not been committed (or rolled back), the query waits for transaction completion. If a lock timeout occurs while waiting for this data to be available, an `SQLCODE -114` error is issued.

READ UNCOMMITTED or READ VERIFIED?

The difference between `READ UNCOMMITTED` and `READ VERIFIED` is demonstrated by the following example:

```
SELECT Name,SSN FROM Sample.Person WHERE Name >= 'M'
```

The query optimizer may choose first to collect all RowID's containing Names meeting the `>= 'M'` condition from a Name index. Once collected, the Person table is accessed one RowID at a time to retrieve the Name and SSN fields for output. A concurrently running updating transaction could change the Name field of a Person with RowID 72 from 'Smith' to 'Abel' in-between the query's collection of RowID's from the index and its row-by-row access to the table. In this case, the collection of RowID's from the index would contain the RowID for a row that no longer conforms to the `Name >= 'M'` condition.

`READ UNCOMMITTED` query processing assumes that the `Name >= 'M'` condition has been satisfied by the index, and will output whatever Name is present in the table for each RowID it collected from the index. In this example it would therefore output a row with a Name of 'Abel', which does not satisfy the condition.

`READ VERIFIED` query processing notes that it is retrieving a field from a table for output (Name) that participates in a condition which should have been previously satisfied by the index, and re-checks the condition in case the field value has changed since the index was examined. Upon re-check, it notes that the row no longer satisfies the condition and omits it from the output. Only values that are needed for output have their conditions re-checked: `SELECT SSN FROM Person WHERE Name >= 'M'` would output the row with RowID 72 in this example.

Exceptions to READ COMMITTED

When `ISOLATION LEVEL read committed` is in effect, either through setting `ISOLATION LEVEL READ COMMITTED` or `$SYSTEM.SQL.SetIsolationMode(1)`, SQL can retrieve only those changes to the data that have been committed. However, there are significant exceptions to this rule:

- A [deleted row](#) is never returned by a query, even when the transaction that deleted the row is in progress and the delete may be subsequently rolled back. `ISOLATION LEVEL READ COMMITTED` ensures that inserts and updates are in a consistent state, but not deletes.
- If you query contains an [aggregate function](#), the aggregate result returns the current state of the data, regardless of the specified `ISOLATION LEVEL`. Therefore, inserts and updates are in progress (and may subsequently be rolled back) are included in aggregate results. Deletes that are in progress (and may subsequently be rolled back) are *not* included in aggregate results. This is because an aggregate operation requires access to data from many rows of a table.
- A `SELECT` query that contains a [DISTINCT clause](#) or a [GROUP BY clause](#) is unaffected by the `ISOLATION LEVEL` setting. A query containing one of these clauses returns the current state of the data, including changes in progress that may be subsequently rolled back. This is because these query operations require access to data from many rows of a table.
- A query with the [%NOLOCK keyword](#).

Note: On Caché implementations with ECP (Enterprise Cache Protocol) use of `READ COMMITTED` may result in significantly slower performance when compared to `READ UNCOMMITTED`. Developers should weigh the superior performance of `READ UNCOMMITTED` against the greater data accuracy of `READ COMMITTED` when defining transactions that involve ECP.

For further details, refer to [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

ISOLATION LEVEL in Effect

You can set the ISOLATION LEVEL for a process using **SET TRANSACTION** (without starting a transaction), **START TRANSACTION** (setting isolation mode and starting a transaction), or a **SetIsolationMode()** method call.

The specified ISOLATION LEVEL remains in effect until explicitly reset by a **SET TRANSACTION**, **START TRANSACTION**, or a **SetIsolationMode()** method call. Because **COMMIT** or **ROLLBACK** is only meaningful for changes to the data, not data queries, a **COMMIT** or **ROLLBACK** operation has no effect on the ISOLATION LEVEL setting.

The ISOLATION LEVEL in effect at the start of a query remains in effect for the duration of the query.

you can determine the ISOLATION LEVEL for the current process using the **GetIsolationMode()** method call. You can also set the isolation mode for the current process using the **SetIsolationMode()** method call. These methods specify READ UNCOMMITTED (the default) as 0, READ COMMITTED as 1, and READ VERIFIED as 3. Specifying any other numeric value leaves the isolation mode unchanged. No error or change occurs if you set the isolation mode to the current isolation mode. Use of these methods is shown in the following example:

```
WRITE $$SYSTEM.SQL.GetIsolationMode()," default",!
&sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $$SYSTEM.SQL.GetIsolationMode()," after START TRANSACTION",!
DO $$SYSTEM.SQL.SetIsolationMode(0,.stat)
IF stat=1 {
  WRITE $$SYSTEM.SQL.GetIsolationMode()," after SetIsolationMode(0) call",! }
ELSE { WRITE "SetIsolationMode() error" }
&sql(COMMIT)
```

The isolation mode and the access mode must always be compatible. Changing the access mode changes the isolation mode, as shown in the following example:

```
WRITE $$SYSTEM.SQL.GetIsolationMode()," default",!
&sql(SET TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $$SYSTEM.SQL.GetIsolationMode()," after SET TRANSACTION",!
&sql(START TRANSACTION READ ONLY)
WRITE $$SYSTEM.SQL.GetIsolationMode()," after changing access mode",!
&sql(COMMIT)
```

Examples

The following Embedded SQL example uses two **SET TRANSACTION** statements to set transaction parameters. Note that **SET TRANSACTION** does not increment the transaction level (*\$TLEVEL*). The **START TRANSACTION** command initiates a transaction and increments *\$TLEVEL*:

```
&sql(SET TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Set transaction commit mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED)
WRITE !,"Set transaction isolation mode, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION)
WRITE !,"Start transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=",SQLCODE
WRITE !,"Transaction level=", $TLEVEL
```

See Also

- [COMMIT ROLLBACK SAVEPOINT START TRANSACTION \\$TLEVEL](#)
- [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*

START TRANSACTION

Begins a transaction.

```
START TRANSACTION [%COMMITMODE commitmode ]
START TRANSACTION [transactionmodes ]
```

Arguments

<i>commitmode</i>	<i>Optional</i> — Specifies how future transactions will be committed to the database during the current process. Valid values are EXPLICIT, IMPLICIT, and NONE. The default is to maintain the existing commit mode; the initial commit mode default for a process is IMPLICIT.
<i>transactionmodes</i>	<p><i>Optional</i> — Specifies the isolation mode and access mode for the transaction. You can specify a value for either an isolation mode, an access mode, or for both modes as a comma-separated list.</p> <p>Valid values for isolation mode are ISOLATION LEVEL READ COMMITTED, ISOLATION LEVEL READ UNCOMMITTED, and ISOLATION LEVEL READ VERIFIED. The default is ISOLATION LEVEL READ UNCOMMITTED.</p> <p>Valid values for access mode are READ ONLY and READ WRITE. Note that only ISOLATION LEVEL READ COMMITTED is compatible with access mode READ WRITE.</p>

Description

A **START TRANSACTION** statement initiates a [transaction](#). **START TRANSACTION** immediately initiates a transaction, regardless of the current commit mode setting. A transaction begun with **START TRANSACTION** must be concluded by issuing an explicit **COMMIT** or **ROLLBACK**, regardless of the current commit mode setting.

START TRANSACTION is optional.

- If your process is only querying the data (**SELECT** statements), you can use [SET TRANSACTION](#) to establish the ISOLATION LEVEL. A **START TRANSACTION** is not needed.
- If your process is modifying the data, whether you need to explicitly begin an SQL transaction by issuing a **START TRANSACTION** depends on the current commit mode setting for the process (also referred to as the AutoCommit setting). If the commit mode for the current process is IMPLICIT or EXPLICIT, issuing a **START TRANSACTION** is optional. If you omit **START TRANSACTION**, the system automatically initiates a transaction when you invoke a modify data operation (**DELETE**, **UPDATE**, **INSERT**, or **TRUNCATE TABLE**). If you specify **START TRANSACTION** a transaction is immediately initiated, and must be concluded by an explicit **COMMIT** or **ROLLBACK**.

When **START TRANSACTION** initiates a transaction it increments the [\\$TLEVEL](#) transaction level counter from 0 to 1, indicating a transaction is in progress. You can also determine if a transaction is in progress by checking the SQLCODE set by the [%INTRANSACTION](#) statement. Issuing a **START TRANSACTION** when a transaction is in progress has no effect on [\\$TLEVEL](#) or [%INTRANSACTION](#).

Caché SQL does not support nested transactions. Issuing a **START TRANSACTION** when a transaction is already in progress does not initiate another transaction and does not return an error code. Caché SQL does support savepoints, allowing a partial rollback of a transaction.

If a transaction is not in progress when you issue a **SAVEPOINT** statement, **SAVEPOINT** initiates a transaction. However, this means of initiating a transaction is not recommended.

An SQLCODE -400 is issued if a transaction operation fails to complete successfully.

%BEGTRANS (*deprecated*)

The **%BEGTRANS** statement is functionally identical to **START TRANSACTION** with no arguments. **%BEGTRANS** cannot take arguments, and is considered deprecated. Use **START TRANSACTION** for all new SQL program code.

Setting Parameters

Optionally, **START TRANSACTION** can be used to set parameters. The parameter settings you specify take effect immediately. However, any transaction initiated with a **START TRANSACTION** must be concluded with an explicit **COMMIT** or **ROLLBACK**, regardless of how you set the *commitmode* parameter. Parameter settings continue in effect for the duration of the current process or until explicitly reset. They do not automatically reset to defaults at the end of a transaction.

A single **START TRANSACTION** statement can be used to set either the *commitmode* parameter or the *transactionmodes* parameters, but not both. To set both, you may issue a **SET TRANSACTION** and a **START TRANSACTION**, or two **START TRANSACTION** statements. Only the first **START TRANSACTION** initiates a transaction.

After issuing a **START TRANSACTION**, you can change these parameter settings during the transaction by issuing another **START TRANSACTION**, a **SET TRANSACTION**, or a method call. Changing the *commitmode* parameter does not remove the requirement to conclude the current transaction with an explicit **COMMIT** or **ROLLBACK**.

You can use the **SET TRANSACTION** statement to set the *commitmode* or *transactionmodes* parameters without starting a transaction. These parameters can also be set using method calls, either outside of a transaction or within a transaction.

%COMMITMODE

The **%COMMITMODE** keyword allows you to specify automatic transaction initiation and commitment behavior for the current process. A **START TRANSACTION %COMMITMODE** changes the commit mode setting for all future transactions on the current process. It does not affect the transaction initiated by the **START TRANSACTION** statement. Regardless of the current or set commit mode, a **START TRANSACTION** immediately initiates a transaction, and this transaction must be concluded by issuing an explicit **COMMIT** or **ROLLBACK**.

The available **%COMMITMODE** options are:

- **IMPLICIT**: automatic transaction commitment is on (the initial process default). SQL automatically initiates a transaction when a program issues a database modification operation (**INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE TABLE**). The transaction continues until either the operation completes successfully and SQL automatically commits the changes, or the operation is unable to complete successfully on all rows and SQL automatically rolls back the entire operation. Each database operation (**INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE TABLE**) constitutes a separate transaction. Successful completion of the database operation automatically clears the rollback journal, releases locks, and decrements \$TLEVEL. No **COMMIT** statement is needed.
- **EXPLICIT**: automatic transaction commitment is off. SQL automatically initiates a transaction when a program issues the first database modification operation (**INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE TABLE**). This transaction continues until it is explicitly concluded. Upon successful completion you issue a **COMMIT** statement. If a database modification operation fails you issue a **ROLLBACK** statement to revert the database to the point prior to the beginning of the transaction. In **EXPLICIT** mode multiple database modification operations can constitute a single transaction.
- **NONE**: no automatic transaction processing. Transactions are not initiated unless explicitly invoked by a **START TRANSACTION**. All transactions must be explicitly concluded by issuing either a **COMMIT** or **ROLLBACK** statement. Thus whether a database operation is included in a transaction, and the number of database operations in a transaction are both user-defined.

You can set the %COMMITMODE in ObjectScript using the **SetAutoCommit()** method call. The available method values are 0 (NONE), 1 (IMPLICIT), and 2 (EXPLICIT).

ISOLATION LEVEL

You specify an ISOLATION LEVEL for a process that is issuing a query. The ISOLATION LEVEL options permit you to specify whether or not changes that are in progress should be available for read access by the query. If another concurrent process is performing inserts or updates to a table and those changes to the table are in a transaction, those changes are in progress, and could, potentially, be rolled back. By setting the ISOLATION LEVEL for your process that is querying that table, you can specify whether you wish to include or exclude these changes in progress from the query results.

- **READ UNCOMMITTED** states that all changes are immediately available for query access. This includes changes that may subsequently be rolled back. **READ UNCOMMITTED** insures that your query will return results without waiting for a concurrent insert or update process, and will not fail due to a lock timeout error. However, the results of a **READ UNCOMMITTED** may include values that are not committed; these values may be internally inconsistent because the insert or update operation has only partially completed, and these values may be subsequently rolled back. **READ UNCOMMITTED** is the default if your query process is not in an explicit transaction, or if the transaction does not specify an ISOLATION LEVEL. **READ UNCOMMITTED** is incompatible with **READ WRITE** access; attempting to specify both in the same statement results in an SQLCODE -92 error.
- **READ VERIFIED** states that uncommitted data from other transactions is immediately available, and no locking is performed. This includes changes that may subsequently be rolled back. However, unlike **READ UNCOMMITTED**, a **READ VERIFIED** transaction will re-check any conditions that could be invalidated by uncommitted or newly committed data which would result in output that does not satisfy the query conditions. Because of this condition re-check, **READ VERIFIED** is more accurate but less efficient than **READ UNCOMMITTED** and should only be used when concurrent updates to the data being checked by the conditions is likely to occur. **READ VERIFIED** is incompatible with **READ WRITE** access; attempting to specify both in the same statement results in an SQLCODE -92 error.
- **READ COMMITTED** states that only those changes that have been committed are available for query access. This ensures that a query is performed on the database in a consistent state, not while a group of changes are being made, a group of changes which may be subsequently rolled back. If requested data has been changed, but the changes have not been committed (or rolled back), the query waits for transaction completion. If a lock timeout occurs while waiting for this data to be available, an SQLCODE -114 error is issued.

READ UNCOMMITTED or READ VERIFIED?

The difference between **READ UNCOMMITTED** and **READ VERIFIED** is demonstrated by the following example:

```
SELECT Name,SSN FROM Sample.Person WHERE Name >= 'M'
```

The query optimizer may choose first to collect all RowID's containing Names meeting the \geq 'M' condition from a Name index. Once collected, the Person table is accessed one RowID at a time to retrieve the Name and SSN fields for output. A concurrently running updating transaction could change the Name field of a Person with RowID 72 from 'Smith' to 'Abel' in-between the query's collection of RowID's from the index and its row-by-row access to the table. In this case, the collection of RowID's from the index would contain the RowID for a row that no longer conforms to the Name \geq 'M' condition.

READ UNCOMMITTED query processing assumes that the Name \geq 'M' condition has been satisfied by the index, and will output whatever Name is present in the table for each RowID it collected from the index. In this example it would therefore output a row with a Name of 'Abel', which does not satisfy the condition.

READ VERIFIED query processing notes that it is retrieving a field from a table for output (Name) that participates in a condition which should have been previously satisfied by the index, and re-checks the condition in case the field value has changed since the index was examined. Upon re-check, it notes that the row no longer satisfies the condition and omits it from the output. Only values that are needed for output have their conditions re-checked: `SELECT SSN FROM Person WHERE Name >= 'M'` would output the row with RowID 72 in this example.

Exceptions to READ COMMITTED

When ISOLATION LEVEL read committed is in effect, either through setting ISOLATION LEVEL READ COMMITTED or `$$SYSTEM.SQL.SetIsolationMode(1)`, SQL can retrieve only those changes to the data that have been committed. However, there are significant exceptions to this rule:

- A [deleted row](#) is never returned by a query, even when the transaction that deleted the row is in progress and the delete may be subsequently rolled back. ISOLATION LEVEL READ COMMITTED ensures that inserts and updates are in a consistent state, but not deletes.
- If you query contains an [aggregate function](#), the aggregate result returns the current state of the data, regardless of the specified ISOLATION LEVEL. Therefore, inserts and updates are in progress (and may subsequently be rolled back) are included in aggregate results. Deletes that are in progress (and may subsequently be rolled back) are *not* included in aggregate results. This is because an aggregate operation requires access to data from many rows of a table.
- A **SELECT** query that contains a [DISTINCT clause](#) or a [GROUP BY clause](#) is unaffected by the ISOLATION LEVEL setting. A query containing one of these clauses returns the current state of the data, including changes in progress that may be subsequently rolled back. This is because these query operations require access to data from many rows of a table.
- A query with the [%NOLOCK keyword](#).

Note: On Caché implementations with ECP (Enterprise Cache Protocol) use of READ COMMITTED may result in significantly slower performance when compared to READ UNCOMMITTED. Developers should weigh the superior performance of READ UNCOMMITTED against the greater data accuracy of READ COMMITTED when defining transactions that involve ECP.

For further details, refer to [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

ISOLATION LEVEL in Effect

You can set the ISOLATION LEVEL for a process using **SET TRANSACTION** (without starting a transaction), **START TRANSACTION** (setting isolation mode and starting a transaction), or a `SetIsolationMode()` method call.

The specified ISOLATION LEVEL remains in effect until explicitly reset by a **SET TRANSACTION**, **START TRANSACTION**, or a `SetIsolationMode()` method call. Because **COMMIT** or **ROLLBACK** is only meaningful for changes to the data, not data queries, a **COMMIT** or **ROLLBACK** operation has no effect on the ISOLATION LEVEL setting.

The ISOLATION LEVEL in effect at the start of a query remains in effect for the duration of the query.

you can determine the ISOLATION LEVEL for the current process using the `GetIsolationMode()` method call. You can also set the isolation mode for the current process using the `SetIsolationMode()` method call. These methods specify READ UNCOMMITTED (the default) as 0, READ COMMITTED as 1, and READ VERIFIED as 3. Specifying any other numeric value leaves the isolation mode unchanged. No error or change occurs if you set the isolation mode to the current isolation mode. Use of these methods is shown in the following example:

```
WRITE $$SYSTEM.SQL.GetIsolationMode()," default",!
&sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $$SYSTEM.SQL.GetIsolationMode()," after START TRANSACTION",!
DO $$SYSTEM.SQL.SetIsolationMode(0,.stat)
IF stat=1 {
  WRITE $$SYSTEM.SQL.GetIsolationMode()," after SetIsolationMode(0) call",! }
ELSE { WRITE "SetIsolationMode() error" }
&sql(COMMIT)
```

The isolation mode and the access mode must always be compatible. Changing the access mode changes the isolation mode, as shown in the following example:

```

WRITE $$SYSTEM.SQL.GetIsolationMode()," default",!
&sql(SET TRANSACTION ISOLATION LEVEL READ COMMITTED,READ WRITE)
WRITE $$SYSTEM.SQL.GetIsolationMode()," after SET TRANSACTION",!
&sql(START TRANSACTION READ ONLY)
WRITE $$SYSTEM.SQL.GetIsolationMode()," after changing access mode",!
&sql(COMMIT)

```

ObjectScript and SQL Transactions

ObjectScript and SQL transaction commands are fully compatible and interchangeable, with the following exception:

ObjectScript **TSTART** and SQL **START TRANSACTION** both start a transaction if no transaction is current. However, **START TRANSACTION** does not support nested transactions. Therefore, if you need (or may need) nested transactions, it is preferable to start the transaction with **TSTART**. If you need compatibility with the SQL standard, use **START TRANSACTION**.

ObjectScript transaction processing provides limited support for nested transactions. SQL transaction processing supplies support for savepoints within transactions.

If a transaction involves SQL data modification statements, the transaction should be started with the SQL **START TRANSACTION** statement and committed with the SQL **COMMIT** statement. (These statements may be explicit or implicit, depending on the %COMMITMODE setting.) Methods that use **TSTART/TCOMMIT** nesting can be included in the transaction, as long as they don't initiate the transaction. Methods and stored procedures should not normally use SQL transaction control statements, unless, by design, they are the main controller of the transaction. Stored procedures should not normally use SQL transaction control statements, because these stored procedures are normally called from ODBC/JDBC, which has its own model of transaction control.

Examples

The following Embedded SQL example uses two **START TRANSACTION** statements to start a transaction and set its parameters. Note that the first **START TRANSACTION** initiates a transaction, setting the commit mode and incrementing the **\$TLEVEL** transaction level counter. The second **START TRANSACTION** sets the isolation mode for query read operations in the current transaction, but does not increment **\$TLEVEL**, because the transaction has already been started. The **SAVEPOINT** statement increments **\$TLEVEL**:

```

WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION %COMMITMODE EXPLICIT)
WRITE !,"Start transaction commit mode, SQLCODE=", SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(START TRANSACTION ISOLATION LEVEL READ COMMITTED)
WRITE !,"Start transaction isolation mode, SQLCODE=", SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(SAVEPOINT a)
WRITE !,"Set Savepoint a, SQLCODE=", SQLCODE
WRITE !,"Transaction level=", $TLEVEL
&sql(COMMIT)
WRITE !,"Commit transaction, SQLCODE=", SQLCODE
WRITE !,"Transaction level=", $TLEVEL

```

See Also

- [COMMIT %INTRANSACTION ROLLBACK SAVEPOINT SET TRANSACTION \\$TLEVEL](#)
- [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

TOP

A **SELECT** clause that specifies how many rows to return.

```
SELECT [DISTINCT clause]
      [TOP {[(int)] | ALL}]
      select-item {,select-item2}
```

Arguments

<i>int</i>	<p>Limits the number of rows returned to the specified integer number. The <i>int</i> argument can be either a positive integer, a Dynamic SQL input parameter (?) or an Embedded SQL host variable (:var) that resolve to a positive integer.</p> <p>In Dynamic SQL, the <i>int</i> value can optionally be enclosed with single parentheses or double parentheses (double parentheses are the preferred syntax); these parentheses suppress literal substitution of the <i>int</i> value in the corresponding cached query.</p>
ALL	<p>TOP ALL is only meaningful in a subquery or in a CREATE VIEW statement. It is used to support the use of an ORDER BY clause in these situations, fulfilling the requirement that an ORDER BY clause must be paired with a TOP clause in a subquery or a query used in a CREATE VIEW. TOP ALL does not restrict the number of rows returned.</p>

Description

The optional TOP clause appears after the **SELECT** keyword and the optional DISTINCT clause, and before the first *select-item*.

The TOP keyword is used in [Dynamic SQL](#) and in [cursor-based Embedded SQL](#). In non-cursor Embedded SQL the only meaningful use of the TOP keyword is TOP 0. Any other TOP *int* (where *int* is any non-zero integer) is valid but not meaningful because a **SELECT** in non-cursor Embedded SQL always returns at most one row of data.

The TOP clause of a **SELECT** statement limits the number of rows returned to the number specified in *int*. If no TOP clause is specified, the default is to display all the rows that meet the **SELECT** criteria. If a TOP clause is specified, the number or rows displayed is either *int* or all of the rows that fulfill the query predicate requirements, whichever is smaller. If you specify ALL, **SELECT** returns all the rows in the table that fulfill the query predicate requirements.

If no **ORDER BY** clause is specified in the query, which records are returned as the “top” rows is unpredictable. If an **ORDER BY** clause is specified, the top rows accord to the order specified in that clause.

The **DISTINCT** clause (if specified) is applied before TOP, specifying that (at most) *int* number of unique values are to be returned.

TOP short circuits when all rows have been delivered. Thus, if you select until you get SQLCODE 100, the **FETCH** that sets SQLCODE 100 is instant.

When accessing data through a view, or through a **FROM** clause subquery, you can limit the number of rows returned by using the %vid view ID, rather than (or in addition to) the TOP clause. For further details on using %vid, refer to the [Defining and Using Views](#) chapter of *Using Caché SQL*.

The TOP int Value

The *int* numeric value can be an integer, or a numeric string, a Dynamic SQL [input parameter \(?\)](#), or an [input host variable \(:var\)](#) that resolve to an integer value.

The *int* value specifies the number of rows to return. Permitted values are 0 and positive numbers. A fractional number or a numeric string is parsed as its integer value. Zero (0) is a valid *int* value. TOP 0 executes the query but returns no data.

TOP ALL must be specified as a keyword in the query. You cannot specify ALL as a ? input parameter or :var host variable value. The query parser interprets the string “ALL” supplied in this way as a numeric string with a value of 0.

Note that the TOP [argument metadata](#) is returned as [xDBC data type 12 \(VARCHAR\)](#) rather than 4 (INTEGER) because it is possible to specify TOP *int* as a numeric string or an integer.

TOP and Cached Queries

An *int* value can be specified with or without enclosing parentheses. These parentheses affect how a Dynamic SQL query is [cached](#) (non-cursor Embedded SQL queries are not cached). An *int* value without parentheses is converted to a ? parameter variable in the cached query. This means that repeatedly invoking the same query with different TOP *int* values invokes the same cached query, rather than preparing and optimizing the query each time.

Enclosing parentheses [suppress literal substitution](#). For example, TOP ((7)). When *int* is enclosed in parentheses, the cached query preserves the specific *int* value. Re-invoking the query with the same TOP *int* value uses the cached query; invoking the query with a different TOP *int* value causes SQL to prepare, optimize, and cache this new version of the query.

TOP ALL is not cached as a ? parameter variable. ALL is parsed as a keyword, not a literal. Therefore, the same query with TOP 7 and with TOP ALL will generate two different cached queries.

TOP and ORDER BY

TOP is generally used in a **SELECT** with an **ORDER BY** clause. Note that the default ascending ORDER BY collation sequence considers NULL to be the lowest (“top”) value, followed by the empty string (“”).

TOP is required in a subquery **SELECT** or a **CREATE VIEW SELECT** when specifying an ORDER BY clause. In these cases you can specify either TOP *int* (to limit the number of rows to return) or TOP ALL.

TOP ALL is only used in a subquery or in a **CREATE VIEW** statement. It is used to support the use of an ORDER BY clause in these situations, fulfilling the requirement that an ORDER BY clause must be paired with a TOP clause in a subquery or a **CREATE VIEW** query. TOP ALL does not restrict the number of rows returned. TOP ALL ... ORDER BY *does not* change default **SELECT** optimization. The ALL keyword cannot be enclosed in parentheses.

TOP Optimization

By default, a **SELECT** optimizes for fastest time to return all data. Adding both a TOP *int* clause and an ORDER BY clause optimizes for fastest time to return first row. (Note that both clauses are required to change the optimization.) You can use the %SYS.PTools.SQLStats class TimeToFirstRow property to return the time required to return the first row.

The following are special case optimizations:

- You may wish to use the TOP and ORDER BY optimization strategy without limiting the number of rows returned; for example, if you are returning data that is displayed in page units. In such a case, you may want to issue a TOP clause with an *int* value larger than the total number of rows.
- You may wish to limit the number of rows returned and specify their order without changing the default **SELECT** optimization. In this case, specify a TOP clause, an ORDER BY clause, and the %NOTOPOPT keyword to preserve fastest time to return all data optimization. See the [FROM](#) clause for more details.

TOP with Aggregates and Functions

An aggregate function or a scalar function can only return a single value. If the query *select-item* list contains *only* aggregates and functions, the application of the TOP clause is as follows:

- If the *select-item* list contains an aggregate function, for example COUNT(*) or AVG(Age), and does not contain any field references, no more than one row is returned, regardless of the TOP *int* value or the presence of an ORDER BY clause. These clauses are validated, but ignored. This is shown in the following examples:

```
SELECT TOP 5 AVG(Age),CURRENT_TIMESTAMP(3) FROM Sample.Person
/* returns 1 row */
```

```
SELECT TOP 1 AVG(Age),CURRENT_TIMESTAMP(3) FROM Sample.Person ORDER BY Age
/* returns 1 row */
```

- If the *select-item* list contains one or more scalar functions, expressions, literals (such as %TABLENAME), subqueries, or host variables, and does not contain any field references or aggregates, the TOP clause is applied. This is shown in the following example:

```
SELECT TOP 5 ROUND(678.987,2),CURRENT_TIMESTAMP(3) FROM Sample.Person
/* returns 5 identical rows */
```

The actual number of rows returned depends on the number of rows in the table, even when table fields are not referenced. For example:

```
SELECT TOP 300 CURRENT_TIMESTAMP(3) FROM Sample.Person
/* returns either the number of rows in Sample.Person
or 300 rows, whichever is smaller */
```

When the query is restricted by a predicate condition, the number of rows returned is restricted by that condition, even when table fields are not referenced in the *select-item* list. For example:

```
SELECT TOP 300 CURRENT_TIMESTAMP(3) FROM Sample.Person WHERE Home_State = 'MA'
/* returns either the number of rows in Sample.Person
where Home_State = 'MA'
or 300 rows, whichever is smaller */
```

- If the **SELECT** statement does not contain a FROM clause, at most one row is returned, regardless of the TOP value. For example:

```
SELECT TOP 5 ROUND(678.987,2),CURRENT_TIMESTAMP(3)
/* returns 1 row */
```

- The DISTINCT clause further limits the TOP clause. If there are fewer distinct values than the TOP value, only the rows with distinct values are returned. When only scalar functions are referenced, only one row is returned. For example:

```
SELECT DISTINCT TOP 15 CURRENT_TIMESTAMP(3) FROM Sample.Person
/* returns 1 row */
```

- TOP 0 always returns no rows, regardless of the contents of the *select-item* list, or whether the **SELECT** statement contains a FROM clause or a DISTINCT clause.

In non-cursor Embedded SQL, a query with TOP 0 returns no rows and sets SQLCODE=100; a non-cursor Embedded SQL query with TOP 1 (or any other TOP *int* value) returns one row and sets SQLCODE=0. In cursor-based Embedded SQL, completion of the fetch loop always sets SQLCODE=100, regardless of the TOP *int* value.

Examples

The following query returns the first 20 rows retrieved from Sample.Person in the order that they are stored in the database. This record order is generally not predictable.

```
SELECT TOP 20 Home_State,Name FROM Sample.Person
```

The following query returns the first 20 distinct Home_State values retrieved from Sample.Person in ascending collation sequence order.

```
SELECT DISTINCT TOP 20 Home_State FROM Sample.Person ORDER BY Home_State
```

The following query returns the first 40 distinct FavoriteColor values. The “top” rows reflect the ORDER BY clause sequencing of all of the rows in Sample.Person in descending (DESC) collation sequence. Descending collation sequence is used rather than the default ascending collation sequence because the FavoriteColors field is known to have NULLs, which would appear at the top of the ascending collation sequence.

```
SELECT DISTINCT TOP 40 FavoriteColors FROM Sample.Person
ORDER BY FavoriteColors DESC
```

Also note in the preceding example that because FavoriteColors is a list field, the collation sequence includes the element length byte. Thus six-letter elements (YELLOW, PURPLE, ORANGE) collate together, listed before five-letter elements (WHITE, GREEN, etc.).

[Dynamic SQL](#) can specify the *int* value as an input parameter (indicated by “?”). In the following example, the TOP ? input parameter is set to 10 by the **%Execute** method:

```
SET myquery = "SELECT TOP ? Name, Age FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(10)
DO rset.%Display()
```

The following [cursor-based Embedded SQL](#) example performs the same operation:

```
SET topnum=10
&sql(DECLARE pCursor CURSOR FOR
SELECT TOP :topnum Name, Age INTO :name, :years FROM Sample.Person
)
&sql(OPEN pCursor)
QUIT: (SQLCODE'=0)
FOR { &sql(FETCH pCursor)
QUIT:SQLCODE
WRITE "Name=", name, " Age=", years, !
}
&sql(CLOSE pCursor)
```

See Also

- [SELECT](#) statement
- [DISTINCT](#) clause
- [ORDER BY](#) clause
- “[Querying the Database](#)” chapter in *Using Caché SQL*

TRUNCATE TABLE

Removes all data from a table and resets counters.

```
TRUNCATE TABLE tablename
```

Arguments

<i>tablename</i>	The table from which you are deleting all rows. You can also specify an updateable view through which you can delete all of the rows of a table. A table name (or view name) can be qualified (schema.table), or unqualified (table). An unqualified name is matched to its schema using either a schema search path (if provided) or the system-wide default schema name .
------------------	---

Description

The **TRUNCATE TABLE** command removes all rows from a table, and resets all table counters. You can truncate a table directly, or through a view. Truncating a table through a view is subject to delete requirements and restrictions, as described in [CREATE VIEW](#).

The **TRUNCATE TABLE** operation resets the internal counters used for generating [RowID field](#), [IDENTITY field](#), and [SERIAL \(%Library.Counter\) field](#) sequential integer values. Caché assigns a value of 1 for these fields in the first row inserted into a table following a **TRUNCATE TABLE**. Performing a [DELETE](#) on all rows of a table does not reset these internal counters.

TRUNCATE TABLE does not reset the [ROWVERSION](#) counter.

TRUNCATE TABLE suppresses the pulling of base table triggers that are otherwise pulled during **DELETE** processing. Because **TRUNCATE TABLE** performs a delete with %NOTRIGGER behavior, the user must have been granted the %NOTRIGGER privilege (using the [GRANT](#) statement) in order to run **TRUNCATE TABLE**. This aspect of **TRUNCATE TABLE** is functionally identical to:

```
DELETE %NOTRIGGER FROM tablename
```

TRUNCATE TABLE sets the [%ROWCOUNT](#) local variable to the number of deleted rows.

TRUNCATE TABLE sets the [%ROWID](#) local variable to the RowID of the last row deleted. However, **TRUNCATE TABLE** does not initialize or set %ROWID if no rows are deleted. Therefore, the use of %ROWID with **TRUNCATE TABLE** should be avoided.

Note: The [DELETE](#) command can also be used to delete all rows from a table. **DELETE** provides more functionality than **TRUNCATE TABLE**, but does not reset internal counters. **TRUNCATE TABLE** provides compatibility for code migration from other database software.

To truncate a table:

- The table must exist in the current (or specified) namespace. If the specified table cannot be located, Caché issues an SQLCODE -30 error.
- You must have DELETE privilege for the table. Failing to have this privilege results in an SQLCODE -99 (Privilege Violation) error. You can determine if the current user has DELETE privilege by invoking the [%CHECKPRIV](#) command. You can determine if a specified user has DELETE privilege by invoking the [\\$SYSTEM.SQL.CheckPriv\(\)](#) method. For privilege assignment, refer to the [GRANT](#) command.
- The table cannot be defined as READONLY. Attempting to compile a **TRUNCATE TABLE** that references a read-only table results in an SQLCODE -115 error. Note that this error is now issued at compile time, rather than only

occurring at execution time. See the description of READONLY objects in the [Other Options for Persistent Classes](#) chapter of *Using Caché Objects*.

- If deleting through a view, the view must be updateable; it cannot be defined as WITH READ ONLY. Attempting to do so results in an SQLCODE -35 error. See the [CREATE VIEW](#) command for further details.
- All of the rows must be available for deletion. By default, if one or more rows cannot be deleted the **TRUNCATE TABLE** operation fails and no rows are deleted. If a row cannot be locked, **TRUNCATE TABLE** fails to delete any rows and issues an error. If deleting a row would violate foreign key referential integrity, **TRUNCATE TABLE** fails to delete any rows and instead issues an SQLCODE -124 error. This default behavior is modifiable, as described below.

Atomicity

By default, **TRUNCATE TABLE**, **DELETE**, **UPDATE**, and **INSERT** are atomic operations. A **TRUNCATE TABLE** either completes successfully or the whole operation is rolled back. If any row cannot be deleted, none of the rows are deleted and the database reverts to its state before issuing the **TRUNCATE TABLE**.

You can modify this default for the current process within SQL by invoking [SET TRANSACTION %COMMITMODE](#). You can modify this default for the current process in ObjectScript by invoking the `SetAutoCommit()` method. The following options are available:

- **IMPLICIT** or 1 (autocommit on) — The default behavior, as described above. Each **TRUNCATE TABLE** constitutes a separate transaction.
- **EXPLICIT** or 2 (autocommit off) — If no transaction is in progress, a **TRUNCATE TABLE** automatically initiates a transaction, but you must explicitly **COMMIT** or **ROLLBACK** to end the transaction. In **EXPLICIT** mode the number of database operations per transaction is user-defined.
- **NONE** or 0 (no auto transaction) — No transaction is initiated when you invoke **TRUNCATE TABLE**. A failed **TRUNCATE TABLE** operation can leave the database in an inconsistent state, with some rows deleted and some not deleted. To provide transaction support in this mode you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

You can determine the atomicity setting for the current process using the `GetAutoCommit()` method, as shown in the following ObjectScript example:

```
DO $SYSTEM.SQL.SetAutoCommit($RANDOM(3))
SET x=$SYSTEM.SQL.GetAutoCommit()
IF x=1 {
  WRITE "Default atomicity behavior",!
  WRITE "automatic commit or rollback" }
ELSEIF x=0 {
  WRITE "No transaction initiated, no atomicity:",!
  WRITE "failed DELETE can leave database inconsistent",!
  WRITE "rollback is not supported" }
ELSE { WRITE "Explicit commit or rollback required" }
```

Referential Integrity

Caché uses the system configuration setting to determine whether to perform foreign key referential integrity checking. You can set the system default as follows:

- The `$SYSTEM.SQL.SetFilerRefIntegrity()` method call.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Perform Referential Integrity Checks on Foreign Keys for INSERT, UPDATE, and DELETE**. The default is “Yes”. If you change this setting, any new process started after changing it will have the new setting.

During a **TRUNCATE TABLE** operation, for every foreign key reference a shared lock is acquired on the corresponding row in the referenced table. This row is locked until the end of the transaction. This ensures that the referenced row is not changed before a potential rollback of the **TRUNCATE TABLE**.

Transaction Locking

Caché performs standard locking on a **TRUNCATE TABLE** operation. Unique field values are locked for the duration of the current transaction.

The default lock threshold is 1000 locks per table. This means that if you delete more than 1000 unique field values from a table during a transaction, the lock threshold is reached and Caché automatically elevates the locking level from unique field value locks to a table lock. This permits large-scale deletes during a transaction without overflowing the lock table.

You can determine the current system-wide lock threshold value using the **GetLockThreshold()** method. This system-wide lock threshold value is configurable:

- Using the **SetLockThreshold()** method.
- Using the Management Portal. Go to **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Lock Threshold**.

You must have USE permission on the %Admin Manage Resource to change the lock threshold. Caché immediately applies any change made to the lock threshold value to all current processes.

For further details on transaction locking refer to [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

Imported SQL Code

The **DDLImport("CACHE")** and **Cache()** methods do not support the **TRUNCATE TABLE** command. A **TRUNCATE TABLE** command found in an SQL code file imported by these methods is ignored. These import methods do support the **DELETE** command.

Examples

The following two Dynamic SQL examples compare **DELETE** and **TRUNCATE TABLE**. Each example creates a table, inserts rows into the table, deletes all the rows in the table, then inserts a single row into the now empty table.

The first example uses **DELETE** to delete all the records in the table. Note that **DELETE** does not reset the RowID counter:

```
SET tcreate = "CREATE TABLE SQLUser.MyStudents (StudentName VARCHAR(32),StudentDOB DATE)"
SET tinsert = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "_
              "SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
SET tinsert1 = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) VALUES ('Bob Jones',60123)"
SET tdelete = "DELETE SQLUser.MyStudents"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(tcreate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName,!

NEW %ROWCOUNT,%ROWID
SET qStatus = tStatement.%Prepare(tinsert)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName," rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(tdelete)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName," rowcount ",rset.%ROWCOUNT,!

SET qStatus = tStatement.%Prepare(tinsert1)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName," rowcount ",rset.%ROWCOUNT," RowID ",rset.%ROWID,!
&sql(DROP TABLE SQLUser.MyStudents)
```

The second example uses **TRUNCATE TABLE** to delete all the records in the table. Note that *%StatementTypeName* returns “DELETE” for **TRUNCATE TABLE**. Note that **TRUNCATE TABLE** does reset the RowID counter:

```
SET tcreate = "CREATE TABLE SQLUser.MyStudents (StudentName VARCHAR(32),StudentDOB DATE)"
SET tinsert = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "_
              "SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
SET tinsert1 = "INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) VALUES ('Bob Jones',60123)"
SET ttrunc = "TRUNCATE TABLE SQLUser.MyStudents"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(tcreate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, !

NEW %ROWCOUNT,%ROWID
SET qStatus = tStatement.%Prepare(tinsert)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT, !

SET qStatus = tStatement.%Prepare(ttrunc)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " (TRUNCATE TABLE) rowcount ",rset.%ROWCOUNT, !

SET qStatus = tStatement.%Prepare(tinsert1)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WRITE rset.%StatementTypeName, " rowcount ",rset.%ROWCOUNT, " RowID ",rset.%ROWID, !
&sql(DROP TABLE SQLUser.MyStudents)
```

See Also

- [DELETE, INSERT, UPDATE](#)
- [CREATE VIEW](#)
- “[Defining Tables](#)” chapter in *Using Caché SQL*
- “[Defining Views](#)” chapter of *Using Caché SQL*
- [Transaction Processing](#) in the “[Modifying the Database](#)” chapter of *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

UNION

Combines two or more **SELECT** statements.

```
select-statement {UNION [ALL] [%PARALLEL] select-statement}
select-statement {UNION [ALL] [%PARALLEL] (query)}
(query) {UNION [ALL] [%PARALLEL] select-statement}
(query) {UNION [ALL] [%PARALLEL] (query)}
```

Arguments

ALL	<i>Optional</i> — A keyword literal. If specified, duplicate data values are returned. If omitted, duplicate data values are suppressed.
%PARALLEL	<i>Optional</i> — The %PARALLEL keyword. If specified, each side of the union is run in parallel as a separate process.
<i>select-statement</i>	A SELECT statement, which retrieves data from a database.
<i>query</i>	A query that combines one or more SELECT statements.

Description

A **UNION** combines two or more queries into a single query that retrieves data into a result. The queries that are combined by a **UNION** can be simple queries, consisting of a single **SELECT** statement, or compound queries.

For a union to be possible between **SELECT** statements, the number of columns specified in each must match. Specifying **SELECT**s with different numbers of columns results in an SQLCODE -9 error. You can specify a NULL column in one **SELECT** to pair with a data column in another **SELECT** in order to match the number of columns. This use of NULL is shown in the “Examples” section below.

CAUTION: To use the **SELECT *** syntax in a **UNION**, the tables must contain the same number of columns. Therefore, future changes to the table definition by adding or deleting a column may cause unforeseen errors in unions of this sort.

Result column names and data types are taken from the column names and data types of the first leg of the **UNION** query. In situations where the corresponding columns in the two legs do not have the same names, it is may be useful to use the **AS** clause to identify the result columns.

An ordinary **UNION** eliminates duplicate rows (all values identical) from the result. A **UNION ALL** preserves duplicate rows in the result.

String fields in the **UNION** result have the **collation type** of the corresponding **SELECT** fields, but are assigned EXACT collation if the field collations do not match.

TOP and ORDER BY Clauses

A **UNION** statement can conclude with an **ORDER BY** clause which orders the result. This **ORDER BY** applies to the whole statement; it must be part of the outermost query, not a subquery. It does not have to be paired with a **TOP** clause. The following example shows this use of **ORDER BY**: the two **SELECT** statements select data, the data is combined by the **UNION**, then the **ORDER BY** sequences the results:

```
SELECT Name,Home_Zip FROM Sample.Person
WHERE Home_Zip %STARTSWITH 9
UNION
SELECT Name,Office_Zip FROM Sample.Employee
WHERE Office_Zip %STARTSWITH 8
ORDER BY Home_Zip
```

Using a column number in **ORDER BY** that does not correspond to a **SELECT** list column results in an SQLCODE -5 error. Using a column name in **ORDER BY** that does not correspond to a **SELECT** list column results in an SQLCODE -6 error.

Either **SELECT** statements (or both) in a union can also contain an **ORDER BY** clause, but it must be paired with a **TOP** clause. This **ORDER BY** is applied to determine which rows are selected by the **TOP** clause. The following example shows this use of **ORDER BY**: the two **SELECT** statements each use an **ORDER BY** to sequence their rows, which determines which rows are selected as the top rows. The selected data is combined by the **UNION**, then the final **ORDER BY** sequences the results:

```
SELECT TOP 5 Name,Home_Zip FROM Sample.Person
WHERE Home_Zip %STARTSWITH 9
ORDER BY Name
UNION
SELECT TOP 5 Name,Office_Zip FROM Sample.Employee
WHERE Office_Zip %STARTSWITH 8
ORDER BY Office_Zip
ORDER BY Home_Zip
```

TOP may apply to the first **SELECT** in the union, or to the result of the union, depending on the placement of the **ORDER BY** clause:

- **TOP..ORDER BY** applies to **UNION** result: If the **UNION** is within a FROM clause subquery, and TOP and ORDER BY are applied to the results of the **UNION**. For example:

```
SELECT TOP 10 Name,Home_Zip
FROM (SELECT Name,Home_Zip FROM Sample.Person
WHERE Name %STARTSWITH 'A'
UNION
SELECT Name,Home_Zip FROM Sample.Person
WHERE Home_Zip %STARTSWITH 8)
ORDER BY Home_Zip
```

- **TOP** applies to first **SELECT**; **ORDER BY** applies to **UNION** result. For example:

```
SELECT TOP 10 Name,Home_Zip
FROM Sample.Person
WHERE Name %STARTSWITH 'A'
UNION
SELECT Name,Home_Zip FROM Sample.Person
WHERE Home_Zip %STARTSWITH 8
ORDER BY Home_Zip
```

Enclosing Parentheses

UNION supports optional enclosing parentheses for either or both of its **SELECT** statements, or for the entire **UNION** statement. You may specify one or more pairs of enclosing parentheses. The following are all valid uses of enclosing parentheses:

```
(SELECT ...) UNION SELECT ...
(SELECT ...) UNION (SELECT ...)
((SELECT ...)) UNION ((SELECT ...))
(SELECT ... UNION SELECT ...)
((SELECT ...) UNION (SELECT ...))
```

Each use of parentheses generates a separate [cached Query](#).

UNION/OR Optimization

By default, SQL automatic optimization transforms UNION subqueries to OR conditions, where deemed appropriate. This UNION/OR transformation allows EXISTS and other low-level predicates to migrate to top-level conditions where they

are available to Caché query optimizer indexing. This default transformation is desirable in most situations. However, in some situations this UNION/OR transformation imposes a significant overhead burden. The %NOUNIONOROPT query optimization option disables this automatic UNION/OR transformation for all conditions in the WHERE clause associated with the FROM clause. Thus, in a complex query, you can disable automatic UNION/OR optimization for one subquery while allowing it in other subqueries. For further information on %NOUNIONOROPT, refer to the [FROM clause](#).

If a condition involving a subquery is applied to a UNION, it is applied within each union operand, rather than at the end. This allows subquery optimizations to be applied in each UNION operand. For descriptions of subquery optimization options, refer to the [FROM clause](#). In the following example, the WHERE clause condition is applied to each of the subqueries in the union, rather than to the result of the union:

```
SELECT Name, Age FROM
  (SELECT Name, Age FROM Sample.Person
   UNION SELECT Name, Age FROM Sample.Employee)
WHERE Age IN (SELECT TOP 5 Age FROM Sample.Employee WHERE Age>55 ORDER BY Age)
```

UNION ALL Aggregate Optimization

SQL automatic optimization of a UNION ALL pushes a top-level aggregate into the legs of the union. This can result in significantly improved performance with or without the %PARALLEL keyword, For example:

```
SELECT COUNT(*) FROM (SELECT item1 FROM table1 UNION ALL SELECT item2 FROM table2)
```

is optimized as:

```
SELECT SUM(y) FROM (SELECT COUNT(*) AS y FROM table1 UNION ALL SELECT COUNT(*) AS y FROM table2)
```

This optimization applies to all top-level aggregate functions (not just COUNT), including queries with multiple top-level aggregate functions. For this optimization to be applied, the outer query must be a "onerow" query, with no WHERE or GROUP BY clause, it cannot reference %VID, and the UNION ALL must be the only stream in its FROM clause. The aggregates cannot be nested, and any aggregate function used cannot use %FOREACH() grouping or DISTINCT.

Parallel Processing

The %PARALLEL keyword supports parallelism and distributed processing on a multiprocessor system. It causes Caché to perform parallel processing on the UNION queries, assigning each query to a separate process on the same machine. In some cases that process will send the query to a different machine to be processed. These processes communicate via pipes, with Caché creating one or more temporary files to hold subquery results. The main process combines the resulting rows and returns the final results. For further details, refer to the [Show Plan](#) for a UNION query, comparing the Show Plan with and without the %PARALLEL keyword.

In general, the more effort expended to produce each row, the more beneficial %PARALLEL becomes.

Specifying the %PARALLEL keyword disables [automatic UNION-to-OR optimizations](#).

The following examples show the use of the %PARALLEL keyword:

```
SELECT Name FROM Sample.Employee WHERE Name %STARTSWITH 'A'
UNION %PARALLEL
SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

```
SELECT Name FROM Sample.Employee WHERE Name %STARTSWITH 'A'
UNION ALL %PARALLEL
SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

%PARALLEL is intended for **SELECT** queries and their subqueries. An **INSERT** command subquery cannot use %PARALLEL.

Adding the %PARALLEL keyword may not be appropriate for all UNION queries, and may result in an error. The following SQL constructs generally do not support UNION %PARALLEL execution: an outer join, a correlated field, an IN predicate condition containing a subquery, or a collection predicate. UNION %PARALLEL is supported for a FOR SOME predicate,

but not for a FOR SOME %ELEMENT collection predicate. To determine if a UNION query can successfully use %PARALLEL, test each leg of the UNION separately. Separately test each leg query by adding a FROM %PARALLEL keyword. If one of the FROM %PARALLEL queries generates a query plan that does not show parallelization, then the UNION query will not support %PARALLEL.

UNION ALL and Aggregate Functions

SQL automatic optimization pushes UNION ALL aggregate functions into the union leg subqueries. SQL calculates the aggregate value for each subquery, and then combines the results to return the original aggregate value. For example:

```
SELECT COUNT(Name) FROM (SELECT Name FROM Sample.Person
                        UNION ALL SELECT Name FROM Sample.Employee)
```

Is optimized as:

```
SELECT SUM(y) FROM (SELECT COUNT(Name) AS y FROM Sample.Person
                  UNION ALL SELECT COUNT(Name) AS y FROM Sample.Employee)
```

This can result in substantial performance improvement. This optimization is applied with or without the %PARALLEL keyword. This optimization is applied to multiple aggregate functions.

This optimization transform only occurs under the following circumstances:

- The outer query FROM clause must contain only a UNION ALL statement.
- The outer query cannot contain a WHERE clause or a GROUP BY clause.
- The outer query cannot contain a %VID (view ID) field.
- Aggregate functions cannot contain a DISTINCT or %FOREACH keyword.
- Aggregate functions cannot be nested.

Examples

The following example creates a result that contains a row for every Name found in each of the two tables; if a Name is found in both tables, two rows are created. When the Name is an employee, it lists the office location, concatenated with the word “office” as State, and the employee’s Title. When Name is a person, it lists the home location, concatenated with the word “home” as State, and <null> for Title. The ORDER BY clause operates on the result; the combined rows are ordered by Name:

```
SELECT Name,Office_State||' office' AS State,Title
FROM Sample.Employee
UNION
SELECT Name,Home_State||' home',NULL
FROM Sample.Person
ORDER BY Name
```

The following two examples show the effects of the ALL keyword. In the first example, UNION returns only unique values. In the second example, UNION ALL returns all values, including duplicates:

```
SELECT Name
FROM Sample.Employee
WHERE Name %STARTSWITH 'A'
UNION
SELECT Name
FROM Sample.Person
WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

```
SELECT Name
FROM Sample.Employee
WHERE Name %STARTSWITH 'A'
UNION ALL
SELECT Name
FROM Sample.Person
WHERE Name %STARTSWITH 'A'
ORDER BY Name
```

See Also

- [SELECT](#)
- [ORDER BY](#) clause, [TOP](#) clause
- [CREATE QUERY](#), [CREATE PROCEDURE](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

UNLOCK

Unlocks a table.

```
UNLOCK [TABLE] tablename IN EXCLUSIVE MODE [IMMEDIATE]
UNLOCK [TABLE] tablename IN SHARE MODE [IMMEDIATE]
```

Arguments

<i>tablename</i>	The name of the table to be unlocked. <i>tablename</i> must be an existing table. A <i>tablename</i> can be qualified (schema.table), or unqualified (table). An unqualified table name takes the system-wide default schema name . A schema search path is ignored.
IN EXCLUSIVE MODE / IN SHARE MODE	The IN EXCLUSIVE MODE keyword phrase releases a regular Caché lock. The IN SHARE MODE keyword phrase releases a shared lock at the Caché level.
IMMEDIATE	<i>Optional</i> — If not specified, Caché releases the lock at the end of the current transaction. If specified, Caché releases the lock immediately.

Description

The **UNLOCK** command unlocks an SQL table that was locked by the **LOCK** command. This table must be an existing table for which you have the necessary privileges. If *tablename* is a temporary table, the command completes successfully, but performs no operation. If *tablename* is a view, the command fails with an SQLCODE -400 error.

UNLOCK and **UNLOCK TABLE** are synonymous.

The **UNLOCK** command reverses the **LOCK** operation. The **UNLOCK** command completes successfully even when no lock is held. You can use **LOCK** to lock a table multiple times; you must explicitly **UNLOCK** the table as many times as it was explicitly locked.

Privileges

The **UNLOCK** command is a privileged operation. Prior to using **UNLOCK IN SHARE MODE** it is necessary for your process to have SELECT privilege for the specified table. Prior to using **UNLOCK IN EXCLUSIVE MODE** it is necessary for your process to have INSERT, UPDATE, or DELETE privilege for the specified table. For IN EXCLUSIVE MODE, the INSERT or UPDATE privilege must be on at least one field of the table. Failing to hold sufficient privileges results in an SQLCODE -99 error (Privilege Violation). You can determine if the current user has the necessary privileges by invoking the [%CHECKPRIV](#) command. You can determine if a specified user has the necessary table-level privileges by invoking the [\\$SYSTEM.SQL.CheckPriv\(\)](#) method. For privilege assignment, refer to the [GRANT](#) command.

Nonexistent Table

If you try to unlock a nonexistent table, **UNLOCK** fails with a compile error, and the message SQLCODE=-30 : Table 'SQLUser.mytable' not found.

Examples

The following embedded SQL examples create a table, lock it and then unlock it:

```
NEW SQLCODE,%msg
&sql(CREATE TABLE mytest (
    ID NUMBER(12,0) NOT NULL,
    CREATE_DATE DATE DEFAULT CURRENT_TIMESTAMP(2),
    WORK_START DATE DEFAULT SYSDATE) )
IF SQLCODE=0 { WRITE !,"Table created" }
ELSE { WRITE !,"CREATE TABLE error: ",SQLCODE
    QUIT }
```

```
NEW SQLCODE,%msg
&sql(LOCK mytest IN EXCLUSIVE MODE)
IF SQLCODE=0 { WRITE !,"Table locked" }
ELSEIF SQLCODE=-110 { WRITE !,"Table is locked by another process",!,%msg }
ELSE { WRITE !,"Unexpected LOCK error: ",SQLCODE,!,%msg }
&sql(UNLOCK mytest IN EXCLUSIVE MODE)
IF SQLCODE=0 { WRITE !,"Table unlocked" }
ELSE { WRITE !,"Unexpected UNLOCK error: ",SQLCODE,!,%msg }
```

See Also

- [LOCK](#)
- [INSERT UPDATE DELETE](#)
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

UPDATE

Sets new values for specified columns in a specified table.

```
UPDATE [%NOFPLAN] [restriction] table-ref [[AS] t-alias]
    value-assignment-statement
    [FROM [optimize-option] table-ref1 [[AS] t-alias]
        {,table-ref2 [[AS] t-alias]} ]
    [WHERE condition-expression]

UPDATE [restriction] table-ref [[AS] t-alias]
    value-assignment-statement
    [WHERE CURRENT OF cursor]

value-assignment-statement ::=
    SET column1 = scalar-expression1 {,column2 = scalar-expression2} ... |
    [ (column1{,column2} ...) ] VALUES (scalar-expression1 {,scalar-expression2} ...)
    |
    VALUES :array()
```

Arguments

<code>%NOFPLAN</code>	<i>Optional</i> — The <code>%NOFPLAN</code> keyword specifies that Caché will ignore the frozen plan (if any) for this operation and generate a new query plan. The frozen plan is retained, but not used. For further details, refer to Frozen Plans in <i>Caché SQL Optimization Guide</i> .
<i>restriction</i>	<i>Optional</i> — One or more of the following keywords, separated by spaces: <code>%NOLOCK</code> , <code>%NOCHECK</code> , <code>%NOINDEX</code> , <code>%NOTRIGGER</code> .
<i>table-ref</i>	The name of an existing table where data is to be updated. You can also specify a view through which to perform the update on a table. You cannot specify a table-valued function or JOIN syntax in this argument. A table name (or view name) can be qualified (<code>schema.table</code>), or unqualified (<code>table</code>). An unqualified name is matched to its schema using either a schema search path (if provided) or the system-wide default schema name .
<code>AS t-alias</code>	<i>Optional</i> — An alias for a <i>table-ref</i> (table or view) name. An alias must be a valid identifier . The AS keyword is optional.

FROM <i>table-ref1</i>	<p><i>Optional</i> — A FROM clause used to specify the table or tables used to determine which rows are to be updated.</p> <p>Multiple tables can be specified as a comma-separated list or associated with ANSI join keywords. Any combination of tables or views can be specified. If you specify a comma between two <i>table-refs</i> here, Caché performs a CROSS JOIN on the tables and retrieves data from the results table of the JOIN operation. If you specify ANSI join keywords between two <i>table-refs</i> here, Caché performs the specified join operation. For further details, refer to the JOIN page of this manual.</p> <p>You can optionally specify one or more <i>optimize-option</i> keywords to optimize query execution. The available options are: %ALLINDEX, %FIRSTTABLE <i>tablename</i>, %FULL, %INORDER, %IGNOR-EINDICES, %NOFLATTEN, %NOMERGE, %NOSVSO, %NOTOPOPT, %NOUNIONOROPT, and %STARTTABLE. See FROM clause for more details.</p>
WHERE <i>condition-expression</i>	<p><i>Optional</i> — Specifies one or more boolean predicates used to determine which rows are to be updated. If a WHERE clause (or a WHERE CURRENT OF clause) is not supplied, UPDATE updates all the rows in the table. For further details, see WHERE.</p>
WHERE CURRENT OF <i>cursor</i>	<p><i>Optional: Embedded SQL only</i> — Specifies that the UPDATE operation updates the record at the current position of <i>cursor</i>. You can specify a WHERE CURRENT OF clause or a WHERE clause, but not both. For further details, see WHERE CURRENT OF.</p>
<i>column</i>	<p><i>Optional</i> — The name of an existing column. Multiple column names are specified as a comma-separated list. If omitted, all columns are updated.</p>
<i>scalar-expression</i>	<p>A column data value expressed as a scalar expression. Multiple data values are specified as a comma-separated list with each data value corresponding in sequence to a column.</p>
:array()	<p><i>Embedded SQL only</i> — An array of values specified as a host variable. The lowest subscript level of the array must be unspecified. Thus :myupdates(), :myupdates(5,), and :myupdates(1,1,) are all valid specifications.</p>

Description

An **UPDATE** command changes existing values for columns in a table. You can update data in a table directly, update through a **view**, or update using a subquery enclosed in parentheses. Updating through a view is subject to requirements and restrictions, as described in **CREATE VIEW**.

The **UPDATE** command provides one or more new column values to one or more existing base table rows that contain those columns. Assignment of data values to columns is done using a *value-assignment-statement*. By default, a *value-assignment-statement* updates all rows in the table.

More commonly, an **UPDATE** specifies the updating of a specific row (or rows) based on a *condition-expression*. By default, an **UPDATE** operation goes through all of the rows of a table and updates all rows that satisfy the *condition-expression*. If no rows satisfy the *condition-expression*, **UPDATE** completes successfully and sets SQLCODE=100 (No more data).

You can specify a **WHERE** clause or a **WHERE CURRENT OF** clause (but not both). If the **WHERE CURRENT OF** clause is used, **UPDATE** updates the record at the current position of the cursor. For details on positioned operations, see [WHERE CURRENT OF](#).

The **UPDATE** operation sets the **%ROWCOUNT** local variable to the number of updated rows, and the **%ROWID** local variable to the **RowID** value of the last row updated.

By default, the **UPDATE** operation is an all-or-nothing event. Either all specified rows and columns are updated, or none are.

INSERT OR UPDATE

The **INSERT OR UPDATE** statement is a variant of the **INSERT** statement, the performs both insert and update operations. First it attempts to perform an insert operation. If the insert request fails due to a **UNIQUE KEY** violation (for the field(s) of some unique key, there exists a row that already has the same value(s) as the row specified for the insert), then it automatically turns into an update request for that row, and **INSERT OR UPDATE** uses the specified field values to update the existing row.

Privileges

To perform an update, you must either have table-level **UPDATE** privilege for the specified table (or view) or column-level **UPDATE** privilege for the specified column(s). When updating all fields in a row, note that column-level privileges cover all table columns named in the **GRANT** command; table-level privileges cover all table columns, including those added after the privilege was assigned. Failing to have the necessary privileges results in an **SQLCODE -99** error (Privilege Violation). You can determine if the current user has **UPDATE** privilege by invoking the **%CHECKPRIV** command. You can determine if a specified user has table-level **UPDATE** privilege by invoking the **\$SYSTEM.SQL.CheckPriv()** method. For privilege assignment, refer to the [GRANT](#) command.

When a property is defined as **ReadOnly**, the corresponding table field is also defined as **ReadOnly**. A **ReadOnly** field may only be assigned a value using [InitialExpression](#) or [SqlComputed](#). Attempting to update a value (even a **NULL** value) for a field for which you have column-level **ReadOnly** (**SELECT** or **REFERENCES**) privilege results in an **SQLCODE -138** error: Cannot INSERT/UPDATE a value for a read only field.

You must have **SELECT** privilege for fields in a **WHERE** clause, whether or not those fields are to be updated. You must have both **SELECT** and **UPDATE** privileges for those fields if they are included in the update field list. In the following example, the **Name** field must have (at least) column-level **SELECT** privilege:

```
UPDATE Sample.Employee (Salary) VALUES (1000000) WHERE Name='Smith, John'
```

In the above example, the **Salary** field requires only column-level **UPDATE** privilege.

Value Assignment

You can assign new values to specified columns in a variety of ways.

- Using the **SET** keyword, specify one or more column = scalar-expression pairs as a comma-separated list. For example:

```
SET StatusDate='05/12/06',Status='Purged'
```

- Using the **VALUES** keyword, specify a list of columns equated to a corresponding scalar-expressions list. For example:

```
(StatusDate,Status) VALUES ('05/12/06','Purged')
```

When assigning scalar-expression values to a column list, there must be a scalar-expression for each specified column.

- Using the **VALUES** keyword without a column list, specify a list of scalar-expressions that implicitly correspond to the columns of the row in column order. The following example specifies all of the columns in the table, specifying a literal value to update the **Address** column:

```
VALUES (Name,DOB,'22 Main St. Anytown MA 12345',SSN)
```

When assigning values to an implicit column list, you must supply a value for every updateable field, in the order that the columns are defined in the DDL. (You do not specify the non-updateable RowID column.) These values can either be a literal to specify a new value, or the field name to specify the existing value. You cannot specify placeholder commas or omit trailing fields.

- Using the VALUES keyword without a column list, specify a subscripted array in which the numeric subscripts correspond to the column numbers, including in your column count the non-updateable RowID as column number 1. For example:

```
VALUES :myarray()
```

This value assignment can only be performed from [Embedded SQL](#) using a host variable. Unlike all other value assignments, this usage allows you to delay specifying which columns are to be updated until runtime (by populating the array at runtime). All other types of update require that the columns to be updated must be specified at compile time. For further details, see “[Host Variable as a Subscripted Array](#)” in the “Using Embedded SQL” chapter of *Using Caché SQL*.

For program examples demonstrating each of these types of UPDATE, refer to the [Examples section](#), below.

DISPLAY to LOGICAL Data Conversion

Data is stored in LOGICAL mode format. For example, a date is stored as an integer count of days. Data supplied in an UPDATE operation that is not in LOGICAL mode format must be converted to LOGICAL mode format. Compiled SQL supports automatic conversion of UPDATE data values from DISPLAY or ODBC format to LOGICAL format. Automatic conversion of UPDATE data requires two factors: when compiled, the SQL must specify RUNTIME mode; when executed, the SQL must execute in a LOGICAL mode environment.

- In [Embedded SQL](#), if you specify #SQLCompile Select=runtime, Caché will compile the SQL statement with code that converts data values from a display format to LOGICAL mode storage format. Caché performs this mode conversion both for single values and for arrays of values. For further details, see [#SQLCompile Select](#) in the “ObjectScript Macros and the Macro Preprocessor” chapter of *Using Caché ObjectScript*.
- In an SQL CREATE FUNCTION, CREATE METHOD, or CREATE PROCEDURE statement, if you specify SELECTMODE RUNTIME, Caché will compile the SQL statement with code that converts data values from a display format to LOGICAL mode storage format.

The UPDATE data may be in any format: DISPLAY format (for example, 6/17/2011), ODBC format (for example, 2011-06-17), or LOGICAL format (for example, 62259). The data is stored in LOGICAL format if the SQL execution environment is in LOGICAL mode. This is the default mode for all Caché SQL execution environments.

You can explicitly set the select mode to LOGICAL in SQL execution environments as follows:

- In an ObjectScript program or from the Terminal interface: invoke the `$$SYSTEM.SQL.SetSelectMode(0)` method.
- In [Dynamic SQL](#), specify `%SelectMode 0`.
- From the [SQL Shell](#), specify `SET SELECTMODE LOGICAL`.
- From the Management Portal select the **[System] > [SQL]** interface, then use the **Display Mode** drop-down list to specify Logical Mode.

SQLCODE Errors

By default, a multi-row UPDATE is an atomic operation. If one or more rows cannot be updated, the UPDATE operation fails and no rows are updated. Caché sets the SQLCODE variable, which indicates the success or failure of the UPDATE, and if the operation failed also sets %msg. To update a table, the update must meet all table, column name, and value requirements, as follows.

Tables:

- The table must exist in the current (or specified) namespace. If the specified table cannot be located, Caché issues an SQLCODE -30 error.
- The table cannot be defined as READONLY. Attempting to compile an **UPDATE** that references a read-only table results in an SQLCODE -115 error. Note that this error is now issued at compile time, rather than only occurring at execution time. See the description of READONLY objects in the [Other Options for Persistent Classes](#) chapter of *Using Caché Objects*.
- The table cannot be locked IN EXCLUSIVE MODE by another process. Attempting to update a locked table results in an SQLCODE -110 error, with a %msg such as the following: Unable to acquire lock for UPDATE of table 'Sample.Person' on row with RowID = '10'. Note that an SQLCODE -110 error occurs only when the **UPDATE** statement locates the first record to be updated, then cannot lock it within the timeout period.
- If the **UPDATE** specifies a non-existent field, an SQLCODE -29 is issued. To list all of the field names defined for a specified table, refer to [Column Names and Numbers](#) in the “Defining Tables” chapter of *Using Caché SQL*. If the field exists but none of the field values fulfill the **UPDATE** command’s WHERE clause, no rows are affected and SQLCODE 100 (end of data) is issued.
- If updating a table through a view, the view cannot be defined as WITH READ ONLY. Attempting to do so results in an SQLCODE -35 error. See the [CREATE VIEW](#) command for further details.

Column Names and Values:

- The update cannot include duplicate field names. Attempting an update that specifies two fields with the same name results in an SQLCODE -377 error.
- You cannot update a field that has been locked by another concurrent process. Attempting to do so results in an SQLCODE -110 error. This SQLCODE error can also occur if you are performing such a large number of updates that a <LOCKTABLEFULL> error occurs.
- You cannot update integer counter fields. These fields are non-modifiable. Attempting to do so generates the following errors: [RowID](#) field (SQLCODE -107); [IDENTITY](#) field (SQLCODE -107); [SERIAL](#) (%Library.Counter) field (SQLCODE -105); [ROWVERSION](#) field (SQLCODE -138). The field values for these fields are system-generated and not user-modifiable. Even when the user can insert an initial value for a counter field, the user cannot update the value.

The one exception is when adding a [SERIAL](#) (%Library.Counter) field to a table that has existing data. Existing records will have NULL for this added counter field. In this case, you can use **UPDATE** to change a NULL to an integer value. See the [ALTER TABLE](#) command for further details.

- You cannot update a field value if the update would violate the field’s uniqueness constraints. Attempting to update the value of a field (or group of fields) such that the update would violate a uniqueness constraint or a primary key constraint results in an SQLCODE -120 error. This error is returned if the field has a [UNIQUE data constraint](#), or if the [unique fields constraint](#) has been applied to a group of fields. The SQLCODE -120 %msg string includes both the field and the value that violate the uniqueness constraint. For example <Table 'Sample.MyTable', Constraint 'MYTABLE_UNIQUE3', Field(s) FullName="Molly Bloom"; failed unique check> or <Table 'Sample.MyTable', Constraint 'MYTABLE_PKEY2', Field(s) FullName="Molly Bloom"; failed unique check>. For details on listing a table’s unique value and primary key field constraints and the naming of constraints, refer to [Catalog Details: Constraints](#).
- You cannot update a field value if the update specifies a value that is not listed in its [VALUELIST](#) parameter. A property of a persistent class defined with a VALUELIST parameter can only accept as a valid value one of the values listed in VALUELIST, or be provided with no value (NULL). VALUELIST valid values are case-sensitive. Attempting to update with a data value that doesn’t match the VALUELIST values results in an SQLCODE -105 field value failed validation error.
- Numbers are inserted in [canonical form](#), but can be specified with leading and trailing zeros and multiple leading signs. However, in SQL, two consecutive minus signs are parsed as a single-line comment indicator. Therefore, attempting to specify a number with two consecutive leading minus signs results in an SQLCODE -12 error.

- When using a **WHERE CURRENT OF** clause, you cannot update a field using the current field value to generate an updated value. For example, `SET Salary=Salary+100` or `SET Name=UPPER(Name)`. Attempting to do so results in an SQLCODE -69 error: `SET <field> = <value expression> not allowed with WHERE CURRENT OF <cursor>`.
- You cannot update a **SERIAL** data type field unless its currently has no data value (NULL), or has a value of 0. Attempting to do so results in an SQLCODE -105 error.
- If updating one of the specified rows would violate foreign key referential integrity (and **%NOCHECK** is not specified), the **UPDATE** fails to update any rows and instead issues an SQLCODE -124 error. This does not apply if the foreign key was defined with the **NOCHECK** keyword.
- You cannot update a non-stream field with stream data. This results in an SQLCODE -303 error, as described below.

List Structures

Caché supports the list structure data type `%List` (data type class `%Library.List`). This is a compressed binary format, which does not map to a corresponding native data type for Caché SQL. It corresponds to data type **VARBINARY** with a default **MAXLEN** of 32749. For this reason, **Dynamic SQL** cannot use **UPDATE** or **INSERT** to set a property value of type `%List`. For further details, refer to the [Data Types](#) reference page in this manual.

Stream Values

You cannot use a single **UPDATE** to modify multiple rows that contain a stream value field. Stream data fields must be updated one row at a time.

You can update a Stream field with a literal value, or with an object reference (oref) to an existing stream object. Caché opens this object and copies its contents into the stream field you wish to update.

You cannot update a non-Stream field with the contents of a Stream field. This results in an SQLCODE -303 error: “No implicit conversion of Stream value to non-Stream field in UPDATE assignment is supported”. To update a string field with Stream data, you must first use the **SUBSTRING** function to convert the first *n* characters of the Stream data to a string, as shown in the following example:

```
UPDATE MyTable
SET MyStringField=SUBSTRING(MyStreamField,1,2000)
```

Computed Fields

A field defined with **COMPUTECODE** may recompute its value as part of the **UPDATE** operation, as follows:

- **COMPUTECODE**: value is computed and stored upon **INSERT**, value is not changed upon **UPDATE**.
- **COMPUTECODE** with **COMPUTEONCHANGE**: value is computed and stored upon **INSERT**, is recomputed and stored upon **UPDATE**.
- **COMPUTECODE** with **DEFAULT** and **COMPUTEONCHANGE**: default value is stored upon **INSERT**, value is computed and stored upon **UPDATE**.
- **COMPUTECODE** with **CALCULATED** or **TRANSIENT**: you cannot **UPDATE** a value for this field because no value is stored. The value is computed when queried. However, if you attempt to update a value in a calculated field, Caché performs validation on the supplied value and issues an error if the value is invalid. If the value is valid, Caché performs no update operation, issues no SQLCODE error, and increments **ROWCOUNT**.

However, a **COMPUTEONCHANGE** computed field is not recomputed when the **UPDATE** operation new field value is the same as the existing field value.

In most cases, you define a computed field as read-only. This prevents an update operation directly changing a value that is intended to be the result of a computation involving other field values. In this case, attempting to use **UPDATE** to overwrite the value of a computed field results in an SQLCODE -138 error.

However, you may wish to revise a computed field value to reflect an update to one (or more) of its source field values. You can do this by using an update trigger that recomputes the computed field value after you have updated a specified source field. For example, an update to the Salary data field might trip a trigger that recalculates the Bonus computed field. This update trigger recalculates Bonus and completes successfully, even when Bonus is a read-only field. See the [CREATE TRIGGER](#) statement.

FROM Clause

An **UPDATE** command may have no FROM keyword. It may simply specify the table (or view) to update, and select which rows to update using a WHERE clause.

However, you can also include an optional **FROM** clause after the *value-assignment-statement*. This FROM clause specifies one or more tables used to determine which records are to be updated. The FROM clause is commonly, but not always, used with a WHERE clause involving multiple tables. A FROM clause can be complex, and can include ANSI [join syntax](#). Any syntax supported in a **SELECT FROM** clause is permitted in an **UPDATE FROM** clause. This **UPDATE FROM** clause provides functionality compatibility with Transact-SQL.

The following example shows how this FROM clauses might be used. It updates those records from the Employees table where the same EmpId is also found in the Retirees table:

```
UPDATE Employees AS Emp
  SET retired='Yes'
  FROM Retirees AS Rt
 WHERE Emp.EmpId = Rt.EmpId
```

If the **UPDATE table-ref** and the FROM clause make reference to the same table, these references may either be to the same table, or to a join of two instances of the table. This depends on how table aliases are used:

- If neither table reference has an alias, both reference the same table:

```
UPDATE table1 value-assignment FROM table1,table2 /* join of 2 tables */
```

- If both table references have the same alias, both reference the same table:

```
UPDATE table1 AS x value-assignment FROM table1 AS x,table2 /* join of 2 tables */
```

- If both table references have aliases, and the aliases are different, Caché performs a join of two instances of the table:

```
UPDATE table1 AS x value-assignment FROM table1 AS y,table2 /* join of 3 tables */
```

- If the first table reference has an alias, and the second does not, Caché performs a join of two instances of the table:

```
UPDATE table1 AS x value-assignment FROM table1,table2 /* join of 3 tables */
```

- If the first table reference does not have an alias, and the second has a single reference to the table with an alias, both reference the same table, and this table has the specified alias:

```
UPDATE table1 value-assignment FROM table1 AS x,table2 /* join of 2 tables */
```

- If the first table reference does not have an alias, and the second has more than one reference to the table, Caché considers each aliased instance a separate table and performs a join on these tables:

```
UPDATE table1 value-assignment FROM table1,table1 AS x,table2 /* join of 3 tables */
UPDATE table1 value-assignment FROM table1 AS x,table1 AS y,table2 /* join of 4 tables */
```

Restriction Arguments

To use a *restriction* argument, you must have the corresponding *admin-privilege* for the current namespace. Refer to [GRANT](#) for further details.

Specifying *restriction* argument(s) restricts processing as follows:

- **%NOCHECK** — foreign key referential integrity checking is not performed. Column data validation for data type, maximum length, data constraints, and other validation criteria is also not performed. The **WITH CHECK OPTION** validation for a view is not performed when performing an **UPDATE** through a view.

Note: Because use of **%NOCHECK** can result in invalid data, this restriction argument should only be used when performing bulk inserts or updates from a reliable data source.

- **%NOLOCK** — the row is not locked upon **UPDATE**. This should only be used when a single user/process is updating the database.
- **%NOINDEX** — the index maps are not set during **UPDATE** processing.
- **%NOTRIGGER** — the base table triggers are not pulled (executed) during **UPDATE** processing. Neither **BEFORE** nor **AFTER** triggers are executed.

You can specify multiple *restriction* arguments in any order. Multiple arguments are separated by spaces.

Referential Integrity

If you do not specify **%NOCHECK**, Caché uses the system configuration setting to determine whether to perform foreign key referential integrity checking. You can set the system default as follows:

- The **\$SYSTEM.SQL.SetFilerRefIntegrity()** method call.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Perform Referential Integrity Checks on Foreign Keys for INSERT, UPDATE, and DELETE**. The default is “Yes”. If you change this setting, any new process started after changing it will have the new setting.

This setting does not apply to foreign keys that have been defined with the **NOCHECK** keyword.

During an **UPDATE** operation, for every foreign key reference which has a field value being updated, a shared lock is acquired on both the old (pre-update) referenced row and the new (post-update) referenced row in the referenced table(s). These rows are locked while performing referential integrity checking and updating the row. The lock is then released (it is not held until the end of the transaction). This ensures that the referenced row is not changed between the referential integrity check and the completion of the update operation. Locking the old row ensures that the referenced row is not changed before a potential rollback of the **UPDATE**. Locking the new row ensures that the referenced row is not changed between the referential integrity checking and the completion of the update operation.

If an **UPDATE** operation with **%NOLOCK** is performed on a [foreign key field defined with CASCADE, SET NULL, or SET DEFAULT](#), the corresponding referential action changing the foreign key table is also performed with **%NOLOCK**.

Atomicity

By default, **UPDATE**, **INSERT**, **DELETE**, and **TRUNCATE TABLE** are atomic operations. An **UPDATE** either completes successfully or the whole operation is rolled back. If any of the specified rows cannot be updated, none of the specified rows are updated and the database reverts to its state before issuing the **UPDATE**.

You can modify this default for the current process within SQL by invoking [SET TRANSACTION %COMMITMODE](#). You can modify this default for the current process in ObjectScript by invoking the **SetAutoCommit()** method. The following options are available:

- **IMPLICIT** or 1 (autocommit on) — The default behavior, as described above. Each **UPDATE** constitutes a separate transaction.
- **EXPLICIT** or 2 (autocommit off) — If no transaction is in progress, an **UPDATE** automatically initiates a transaction, but you must explicitly **COMMIT** or **ROLLBACK** to end the transaction. In **EXPLICIT** mode the number of database operations per transaction is user-defined.
- **NONE** or 0 (no auto transaction) — No transaction is initiated when you invoke **UPDATE**. A failed **UPDATE** operation can leave the database in an inconsistent state, with some of the specified rows updated and some not updated. To

provide transaction support in this mode you must use **START TRANSACTION** to initiate the transaction and **COMMIT** or **ROLLBACK** to end the transaction.

You can determine the atomicity setting for the current process using the `GetAutoCommit()` method, as shown in the following ObjectScript example:

```
DO $SYSTEM.SQL.SetAutoCommit($RANDOM(3))
SET x=$SYSTEM.SQL.GetAutoCommit()
IF x=1 {
    WRITE "Default atomicity behavior",!
    WRITE "automatic commit or rollback" }
ELSEIF x=0 {
    WRITE "No transaction initiated, no atomicity:",!
    WRITE "failed DELETE can leave database inconsistent",!
    WRITE "rollback is not supported" }
ELSE { WRITE "Explicit commit or rollback required" }
```

Transaction Locking

If you do not specify `%NOLOCK`, the system automatically performs standard record locking on **INSERT**, **UPDATE**, and **DELETE** operations. Each affected record (row) is locked for the duration of the current transaction.

The default lock threshold is 1000 locks per table. This means that if you update more than 1000 records from a table during a transaction, the lock threshold is reached and Caché automatically escalates the locking level from record locks to a table lock. This permits large-scale updates during a transaction without overflowing the lock table.

Caché applies one of the two following lock escalation strategies:

- “E”-type lock escalation: Caché uses this type of lock escalation if the following are true: (1) the class uses `%CacheStorage` (you can determine this from the [Catalog Details](#) in the Management Portal SQL schema display). (2) the class either does not specify an IDKey index, or specifies a single-property IDKey index. “E”-type lock escalation is described in the [LOCK](#) command in the *Caché ObjectScript Reference*.
- Traditional SQL lock escalation: The most likely reason why a class would not use “E”-type lock escalation is the presence of a multi-property IDKey index. In this case, each `%Save` increments the lock counter. This means if you do 1001 saves of a single object within a transaction, Caché will attempt to escalate the lock.

For both lock escalation strategies, you can determine the current system-wide lock threshold value using the `$$SYSTEM.SQL.GetLockThreshold()` method. The default is 1000. This system-wide lock threshold value is configurable:

- Using the `$$SYSTEM.SQL.SetLockThreshold()` method.
- Using the Management Portal. Go to **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Lock Threshold**. The default is 1000 locks. If you change this setting, any new process started after changing it will have the new setting.

You must have `USE` permission on the `%Admin Manage Resource` to change the lock threshold. Caché immediately applies any change made to the lock threshold value to all current processes.

One potential consequence of automatic lock escalation is a deadlock situation that might occur when an attempt to escalate to a table lock conflicts with another process holding a record lock in that table. There are several possible strategies to avoid this: (1) increase the lock escalation threshold so that lock escalation is unlikely to occur within a transaction. (2) substantially lower the lock escalation threshold so that lock escalation occurs almost immediately, thus decreasing the opportunity for other processes to lock a record in the same table. (3) apply a table lock for the duration of the transaction and do not perform record locks. This can be done at the start of the transaction by specifying `LOCK TABLE`, then `UNLOCK TABLE` (without the `IMMEDIATE` keyword, so that the table lock persists until the end of the transaction), then perform updates with the `%NOLOCK` option.

Automatic lock escalation is intended to prevent overflow of the lock table. However, if you perform such a large number of updates that a `<LOCKTABLEFULL>` error occurs, **UPDATE** issues an `SQLCODE -110` error.

For further details on transaction locking refer to [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*.

Row-Level Security

Caché row-level security permits **UPDATE** to modify any row that security permits it to access. It allows you to update a row even if the update creates a row that security will not permit you to subsequently access. To ensure that an update does not prevent you from subsequent **SELECT** access to the row, it is recommended that you perform the **UPDATE** through a view that has a **WITH CHECK OPTION**. For further details, refer to [CREATE VIEW](#).

ROWVERSION Counter Increment

If a table has a field of data type [ROWVERSION](#), performing an update on a row automatically updates the integer value of this field. The **ROWVERSION** field takes the next sequential integer from the namespace-wide row version counter. Attempting to specify an update value to a **ROWVERSION** field results in an **SQLCODE -138** error.

SERIAL (%Counter) Counter Increment

An **UPDATE** operation has no effect on [SERIAL \(%Library.Counter\)](#) counter field values. However, an update performed using [INSERT OR UPDATE](#) causes a skip in integer sequence for subsequent insert operations for a **SERIAL** field. Refer to [INSERT OR UPDATE](#) for further details.

Examples

The examples in this section update the `SQLUser.MyStudents` table. The following example creates the `SQLUser.MyStudents` table and populates it with data. Because repeated execution of this example would accumulate records with duplicate data, it uses **TRUNCATE TABLE** to remove old data before invoking **INSERT**. Execute this example before invoking the **UPDATE** examples:

```
CreateStudentTable
  ZNSPACE "Samples"
  SET stuDDL=5
  SET stuDDL(1)="CREATE TABLE SQLUser.MyStudents ("
  SET stuDDL(2)="StudentName VARCHAR(32),StudentDOB DATE,"
  SET stuDDL(3)="StudentAge INTEGER COMPUTECODE {SET {StudentAge}="
  SET stuDDL(4)="$PIECE(($PIECE($H,"",",",1)-{StudentDOB})/365,".",",1)} CALCULATED,"
  SET stuDDL(5)="Q1Grade CHAR,Q2Grade CHAR,Q3Grade CHAR,FinalGrade VARCHAR(2))"
  SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
  SET qStatus = tStatement.%Prepare(.stuDDL)
  IF qStatus'=1 {WRITE "DDL %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rtn = tStatement.%Execute()
  IF rtn.%SQLCODE=0 {WRITE !,"Table Create successful"}
  ELSEIF rtn.%SQLCODE=-201 {WRITE "Table already exists, SQLCODE=",rtn.%SQLCODE,!}
  ELSE {WRITE !,"table create failed, SQLCODE=",rtn.%SQLCODE,!
        WRITE rtn.%Message,! }
RemoveOldData
  SET clearit="TRUNCATE TABLE SQLUser.MyStudents"
  SET qStatus = tStatement.%Prepare(clearit)
  IF qStatus'=1 {WRITE "Truncate %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET truncrtn = tStatement.%Execute()
  IF truncrtn.%SQLCODE=0 {WRITE !,"Table old data removed",!}
  ELSEIF truncrtn.%SQLCODE=100 {WRITE !,"no data to be removed",!}
  ELSE {WRITE !,"truncate failed, SQLCODE=",truncrtn.%SQLCODE," ",truncrtn.%Message,! }
PopulateStudentTable
  SET studentpop=2
  SET studentpop(1)="INSERT INTO SQLUser.MyStudents (StudentName,StudentDOB) "
  SET studentpop(2)="SELECT Name,DOB FROM Sample.Person WHERE Age <= '21'"
  SET qStatus = tStatement.%Prepare(.studentpop)
  IF qStatus'=1 {WRITE "Populate %Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET poprtn = tStatement.%Execute()
  IF poprtn.%SQLCODE=0 {WRITE !,"Table Populate successful",!
                        WRITE poprtn.%ROWCOUNT," rows inserted"}
  ELSE {WRITE !,"table populate failed, SQLCODE=",poprtn.%SQLCODE,!
        WRITE poprtn.%Message }
```

You can use the following query to display the results of these examples:

```
SELECT %ID,* FROM SQLUser.MyStudents ORDER BY StudentAge,%ID
```

Some of the following **UPDATE** examples depend on field values set by other **UPDATE** examples; they should be run in the order specified.

In the following [Dynamic SQL](#) example, a `SET field=value UPDATE` modifies a specified field in selected records. In the `MyStudents` table, children under the age of 7 are not given grades:

```
ZNSPACE "Samples"
  SET studentupdate=3
  SET studentupdate(1)="UPDATE SQLUser.MyStudents "
  SET studentupdate(2)="SET FinalGrade='NA' "
  SET studentupdate(3)="WHERE StudentAge <= 6"
SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
SET qStatus = tStatement.%Prepare(.studentupdate)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET uprtn = tStatement.%Execute()
IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                    WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
ELSE {WRITE !,"Table update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message }
```

In the following [cursor-based Embedded SQL](#) example, a `SET field1=value1, field2=value2 UPDATE` modifies several fields in selected records. In the `MyStudents` table, it updates specified student records with Q1 and Q2 grades:

```
#SQLCompile Path=Sample
NEW %ROWCOUNT,%ROWID
&sql(DECLARE StuCursor CURSOR FOR
      SELECT * FROM MyStudents
      WHERE %ID IN(10,12,14,16,18,20,22,24) AND StudentAge > 6)
&sql(OPEN StuCursor)
  QUIT:(SQLCODE'=0)
FOR { &sql(FETCH StuCursor)
     QUIT:SQLCODE
     &sql(Update MyStudents SET Q1Grade='A',Q2Grade='A'
         WHERE CURRENT OF StuCursor)
  IF SQLCODE=0 {
  WRITE !,"Table Update successful"
  WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
  ELSE {
  WRITE !,"Table Update failed, SQLCODE=",SQLCODE }
  }
&sql(CLOSE StuCursor)
```

In the following [Dynamic SQL](#) example, a `field-list VALUES value-list UPDATE` modifies the values of several fields in selected records. In the `MyStudents` table, children who don't receive a final grade also don't receive quarterly grades:

```
ZNSPACE "Samples"
  SET studentupdate=3
  SET studentupdate(1)="UPDATE SQLUser.MyStudents "
  SET studentupdate(2)="(Q1Grade,Q2Grade,Q3Grade) VALUES ('x','x','x') "
  SET studentupdate(3)="WHERE FinalGrade='NA'"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.studentupdate)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET uprtn = tStatement.%Execute()
IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                    WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message, ! }
```

In the following [Dynamic SQL](#) example, a `VALUES value-list UPDATE` modifies all the field values in selected records. Note that this syntax requires that you specify a value for every field in the record. In the `MyStudents` table, several children have been withdrawn from school. Their record IDs and names are retained, with the word `WITHDRAWN` appended to the name; all other data is removed and the `DOB` field is used for the withdrawal date:

```
ZNSPACE "Samples"
SET studentupdate=4
SET studentupdate(1)="UPDATE SQLUser.MyStudents "
SET studentupdate(2)="VALUES (StudentName||' WITHDRAWN',"
SET studentupdate(3)="$PIECE($HOROLOG,' ',1),00,'-', '-','-', 'XX') "
SET studentupdate(4)="WHERE %ID IN(7,10,22)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.studentupdate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET uprtn = tStatement.%Execute()
IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                    WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message,! }
```

In the following [Dynamic SQL](#) example, a subquery **UPDATE** uses a subquery to select records. It then modifies these records using `SET field=value` syntax. Because of the way that `StudentAge` is calculated from date of birth in `SQLUser.MyStudents`, anyone less than a year old has a calculated age of <Null>, and anyone whose date of birth has been nulled has a very high calculated age. Here the `StudentName` field is flagged for future confirmation of the date of birth:

```
ZNSPACE "Samples"
SET studentupdate=3
SET studentupdate(1)="UPDATE (SELECT StudentName FROM SQLUser.MyStudents "
SET studentupdate(2)="WHERE StudentAge IS NULL OR StudentAge > 21) "
SET studentupdate(3)="SET StudentName = StudentName||' *** CHECK DOB' "
SET tStatement = ##class(%SQL.Statement).%New(0,"Sample")
SET qStatus = tStatement.%Prepare(.studentupdate)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET uprtn = tStatement.%Execute()
IF uprtn.%SQLCODE=0 {WRITE !,"Table Update successful"
                    WRITE !,"Rows updated=",uprtn.%ROWCOUNT," Final RowID=",uprtn.%ROWID}
ELSE {WRITE !,"Table Update failed, SQLCODE=",uprtn.%SQLCODE," ",uprtn.%Message,! }
```

In the following [Embedded SQL](#) example, a `VALUES :array()` **UPDATE** modifies the field values specified by column number in the array in selected records. A `VALUES :array()` update can only be done in Embedded SQL. Note that this syntax requires that you specify each value by DDL column number (including in your column count the `RowID` column (column 1) but supplying no value to this non-modifiable field). In the `MyStudents` table, children between 4 and 6 (inclusive) are given a 'P' (for 'Present') in their `Q1Grade` (column 5) and `Q2Grade` (column 6) fields. All other record data remains unchanged:

```
ZNSPACE "Samples"
SET array(5)="P"
SET array(6)="P"
&sql(UPDATE SQLUser.MyStudents VALUES :array()
     WHERE FinalGrade='NA' AND StudentAge > 3)
IF SQLCODE=0 {WRITE "Table Update successful",!
             WRITE "Rows updated=",%ROWCOUNT," Final RowID=",%ROWID }
ELSE {WRITE "Table Update failed, SQLCODE=",SQLCODE,! }
```

See Also

- [INSERT](#)
- [INSERT OR UPDATE](#)
- [DELETE](#)
- [SELECT](#)
- [VALUES](#)
- [FROM](#)
- [WHERE](#)
- [WHERE CURRENT OF](#)
- [CREATE TABLE](#)
- [CREATE VIEW](#)
- “[Modifying the Database](#)” chapter in *Using Caché SQL*

- [“Defining Tables”](#) chapter in *Using Caché SQL*
- [“Defining Views”](#) chapter of *Using Caché SQL*
- [Transaction Processing](#) in the “Modifying the Database” chapter of *Using Caché SQL*
- [SQL configuration settings](#) described in *Caché Advanced Configuration Settings Reference*.
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

USE DATABASE

Sets the current namespace and database.

```
USE [DATABASE] dbname
```

Arguments

<i>dbname</i>	The namespace and corresponding database to be used by the current process as the current namespace.
---------------	--

Description

The **USE DATABASE** command switches the current process to the specified namespace and its associated database. This allows you to change namespaces within SQL. The DATABASE keyword is optional.

The specified *dbname* is the name of the desired namespace and corresponding directory that contains the database files. Specify *dbname* as an [identifier](#). Namespace names are not case-sensitive. For further information on using namespaces, see [Namespaces and Databases](#) in the *Caché Programming Orientation Guide*.

Because USER is an [SQL Reserved Word](#), you must use a [delimited identifier](#) to specify the USER namespace, as shown in the following SQL Shell example:

```
USER>>USE DATABASE Samples
SAMPLES>>USE DATABASE "User"
USER>>
```

If the specified *dbname* does not exist, Caché issues an SQLCODE -400 error.

The **USE DATABASE** command is a privileged operation. Prior to using **USE DATABASE**, it is necessary to be logged in as a user with appropriate privileges. Failing to do so results in an SQLCODE -99 error (Privilege Violation).

Use the **\$\$SYSTEM.Security.Login()** method to assign a user with appropriate privileges:

```
DO $$SYSTEM.Security.Login( "_SYSTEM", "SYS" )
&sql(      )
```

You must have the **%service_Login:Use** privilege to invoke the **\$\$SYSTEM.Security.Login** method. For further information, refer to **%SYSTEM.Security** in the *InterSystems Class Reference*.

You can also switch to a different namespace using the ObjectScript [ZNSPACE](#) command, or the [SET \\$NAMESPACE](#) statement.

Executing via xDBC

When the **USE DATABASE** command is executed via xDBC, the server process performs a simulated connection reset. Data structures used by the server process are cleaned up. However, commit mode is not changed. The Read Committed setting is not changed either. If a transaction is in process, the transaction simply continues and is not committed or rolled back.

See Also

- [CREATE DATABASE](#) command
- [DROP DATABASE](#) command

VALUES

An INSERT/UPDATE clause that specifies data values for use in fields.

```
(field1{,fieldn})
VALUES (value1{,valuen})
```

Arguments

<i>field</i>	A field name or a comma-separated list of field names.
<i>value</i>	A value or comma-separated list of values. Each value is assigned to the corresponding field.

Description

The **VALUES** clause is used in an **INSERT** or **UPDATE** statement to specify the data values to insert into the fields.

Typically:

- **INSERT** queries use the following syntax:

```
INSERT INTO tablename (fieldname,fieldname,...)
VALUES (value,...)
```

- **UPDATE** queries use the following syntax:

```
UPDATE tablename (fieldname,fieldname,...)
VALUES (value,...)
```

The elements in the **VALUES** clause correspond in sequence to the fields specified after the table name. Note if there is only one value element specified in the **VALUES** clause, it is not necessary to enclose the element in parentheses.

The following embedded SQL example shows an **INSERT** statement that adds a single row to the "Employee" table:

```
&sql(INSERT INTO Employee (Name,SocSec,Telephone)
VALUES ("Boswell",333448888,"546-7989"))
```

```
&sql(INSERT INTO Employee (Name,SocSec,Telephone)
VALUES ('Boswell',333448888,'546-7989'))
```

INSERT and **UPDATE** queries can use a **VALUES** clause without requiring you to explicitly specify a list of field names after the table name. In order to omit the list of field names after the table name, your query must meet the following two criteria:

- The number of values specified in the **VALUES** clause is the same as the number of fields in the table (exclusive of the ID field).
- The values in the **VALUES** clause are listed in order of the internal column numbers of the fields, beginning with column 2. Column 1 is always reserved for the system-generated ID field, and is not specified in a **VALUES** clause.

For example, the query:

```
INSERT INTO Sample.Person VALUES (5,'John')
```

is equivalent to the query:

```
INSERT INTO Sample.Person (Age,Name) VALUES (5,'John')
```

if the table "Sample.Person" has exactly two user-defined fields.

In this example, the value 5 is assigned to the field with the lower column number, and the value "John" is assigned to the other field.

A **VALUES** clause can specify an element of an array, as is the following embedded SQL example:

```
&sql( UPDATE Person(Tel)
      VALUES :per('tel',)
      WHERE ID = :id )
```

An **UPDATE** query can also reference an array with unspecified last subscript. Whereas **INSERT** uses the presence and absence of array elements to assign values and default values to a newly created row, **UPDATE** uses the presence of an array element to indicate that the corresponding field should be updated. For example, consider the following array for a table with six columns:

```
emp("profile",2)="Smith"
emp("profile",3)=2
emp("profile",3,1)="1441 Main St."
emp("profile",3,2)="Cableton, IL 60433"
emp("profile",5)=NULL
emp("profile",7)=25
emp("profile","next")="F"
```

Column 1 is always reserved for the ID field, and is not user-specified. The inserted "Employee" row has Column 2, "Name", set to "Smith"; Column 3, "Address", set to a two-line value; Column 4, "Department", is not specified, and is thus set to the default, and Column 5, "Location", is set to NULL. The default value for "Location" is not used since the corresponding array element is defined with a null value. The array elements "7" and "next" do not correspond to column numbers in the "Employee" table, therefore the query ignores them. Here's the **UPDATE** statement that uses this array:

```
&sql(UPDATE Employee
      VALUES :emp('profile',)
      WHERE Employee = 379)
```

Given the above definitions and array values, this statement will update the values of the "Name", "Address", and "Location" fields of the "Employee" row for which Row ID = 379.

However, omitting the subscript entirely results in an SQLCODE -54 error: Array designator (last subscript omitted) expected after **VALUES**.

You may also use an array reference with an **UPDATE** query that targets multiple rows, for example:

```
&sql(UPDATE Employee
      VALUES :emp('profile',)
      WHERE Type = 'PART-TIME')
```

A **VALUES** clause variable cannot use dot syntax. Therefore, the following embedded SQL example is correct:

```
SET sname = state.Name
&sql(INSERT INTO StateTbl VALUES :sname)
```

The following is not correct:

```
&sql(INSERT INTO State VALUES :state.Name)
```

NULL and empty string values are different. For further details, see [NULL](#). For backward compatibility, all empty string ("") values in older existing data are considered as NULL values. In new data, empty strings are stored in the data as \$CHAR(0). Through SQL, NULL is referenced as 'NULL'. For example:

```
INSERT INTO Sample.Person
(SSN,Name,Home_City) VALUES ('123-45-6789', 'Doe,John', NULL)
```

Through SQL, empty string is referenced as " (two single quotes). For example:

```
INSERT INTO Sample.Person
(SSN,Name,Home_City) VALUES ('123-45-6789', 'Doe,John', '')
```

You cannot insert a NULL value for the ID field.

Examples

The following embedded SQL example inserts a record for “Doe,John” into the Sample.Person table. It then selects this record, and then deletes this record. A second **SELECT** confirms the deletion.

```

SET x="Doe,John",y="123-45-6789",z="Metropolis"
SET (a,b,c,d,e)=0
NEW SQLCODE,%ROWCOUNT,%ROWID
&sql(INSERT INTO Sample.Person
(Name,SSN,Home_City) VALUES (:x,:y,:z))
IF SQLCODE'=0 {
  WRITE !,"INSERT Error code ",SQLCODE
  QUIT }
&sql(SELECT Name,SSN,Home_City
      INTO :a,:b,:c
      FROM Sample.Person
      WHERE Name =:x)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"After INSERT:"
  WRITE !,"Name=",a," SSN=",b," City=",c
  WRITE !,"SQL code=",SQLCODE," Number of rows=",%ROWCOUNT }
&sql(DELETE FROM Sample.Person
      WHERE Name=:x)
&sql(SELECT Name,SSN
      INTO :d,:e
      FROM Sample.Person
      WHERE Name='Doe,John')
IF SQLCODE <0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"After DELETE:"
  WRITE !,"Name=",d," SSN=",e
  WRITE !,"SQL code=",SQLCODE," Number of rows=",%ROWCOUNT }

```

See Also

- [INSERT](#)
- [UPDATE](#)
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

WHERE

A **SELECT** clause that specifies one or more restrictive conditions.

```
SELECT fields
FROM table
WHERE condition-expression
```

Arguments

<i>condition-expression</i>	An expression consisting of one or more boolean predicates governing which data values are to be retrieved.
-----------------------------	---

Description

The optional **WHERE** clause can be used for the following purposes:

- To specify predicates that restrict which data values are to be returned.
- To specify an explicit join between two tables.
- To specify an implicit join between the base table and a field in another table.

The **WHERE** clause is most commonly used to specify one or more [predicates](#) that are used to restrict the data (filter out rows) retrieved by a **SELECT** query or subquery. You can also use a **WHERE** clause in an **UPDATE** command, **DELETE** command, or in a result set **SELECT** in an **INSERT** (or **INSERT OR UPDATE**) command.

The **WHERE** clause qualifies or disqualifies specific rows from the query selection. The rows that qualify are those for which the *condition-expression* is true. The *condition-expression* can be one or more logical tests (predicates). Multiple predicates can be linked by the AND and OR logical operators. See “[Predicates and Logical Operators](#)” for further details and restrictions.

If a predicate includes division and there are any values in the database that could produce a divisor with a value of zero or a NULL value, you cannot rely on order of evaluation to avoid division by zero. Instead, use a **CASE** statement to suppress the risk.

A **WHERE** clause can specify a *condition-expression* that includes a subquery. The subquery must be enclosed in parentheses.

A **WHERE** clause can specify an explicit join between two tables using the = (inner join), =* (left outer join), and *= (right outer join) symbolic join operators. For further details, refer to the [JOIN](#) page of this manual.

A **WHERE** clause can specify an implicit join between the base table and a field from another table using the arrow syntax (→) operator. For further details, refer to [Implicit Joins](#) in *Using Caché SQL*.

Specifying a Field

The simplest form of a **WHERE** clause specifies a predicate comparing a field to a value, such as `WHERE Age > 21`. Valid field values include the following: A column name (`WHERE Age > 21`); an %ID, %TABLENAME, or %CLASS-NAME; a scalar function specifying a column name (`WHERE ROUND(Age, -1) = 60`), a collation function specifying a column name (`WHERE %SQLUPPER(Name) %STARTSWITH ' AB'`).

Because the name of the [RowID field](#) can change when a table is re-compiled, a **WHERE** clause should avoid referring to the RowID by name (for example, `WHERE ID=22`). Instead, refer to the RowID using the %ID pseudo-column name (for example, `WHERE %ID=22`).

You cannot specify a field by column number.

You cannot specify a field by column alias; attempting to do so generates an SQLCODE -29 error. However, you can use a subquery to define a column alias, then use this alias in the **WHERE** clause. For example:

```
SELECT Interns FROM
  (SELECT Name AS Interns FROM Sample.Employee WHERE Age<21)
WHERE Interns %STARTSWITH 'A'
```

You cannot specify an aggregate field; attempting to do so generates an SQLCODE -19 error. However, you can supply an aggregate function value to a **WHERE** clause by using a subquery. For example:

```
SELECT Name, Age, AvgAge
FROM (SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person)
WHERE Age < AvgAge
ORDER BY Age
```

Integers and Strings

If a field defined as integer data type is compared to a numeric value, the numeric value is converted to [canonical form](#) before performing the comparison. For example, `WHERE Age=007.00` parses as `WHERE Age=7`. This conversion occurs in all modes.

If a field defined as integer data type is compared to a string value in Display mode, the string is parsed as a numeric value. For instance, an empty string (`''`), like any non-numeric string, is parsed as the number 0. This parsing follows ObjectScript rules for handling strings as numbers. For example, `WHERE Age='twenty'` parses as `WHERE Age=0`; `WHERE Age='20something'` parses as `WHERE Age=20`. For further details, refer to [Strings as Numbers](#) in the “Data Types and Values” chapter of *Using Caché ObjectScript*. SQL only performs this parsing in Display mode; in Logical or ODBC mode comparing an integer to a string value returns null.

To compare a string field with a string containing a single quote, double the single quote. For example, `WHERE Name %STARTSWITH 'O'''` returns O’Neil and O’Connor, but not Obama.

Date and Time

In Caché SQL dates and times are compared and stored using a Logical Mode internal representation. They can be returned in Logical mode, Display Mode, or ODBC mode. For example, the date September 28, 1944 is represented as: Logical mode 37891, Display mode 09/28/1944, ODBC mode 1944-09-28. When specifying a date or time in a *condition-expression* an error can occur due to a mismatch of SQL mode and date or time format, or due to an invalid date or time value.

A **WHERE** clause *condition-expression* must use the date or time format that corresponds to the current mode. For example, when in Logical mode, to return records with a date of birth in 2005, the **WHERE** clause would appear as follows: `WHERE DOB BETWEEN 59901 AND 60265`. When in Display mode, the same **WHERE** clause would appear as follows: `WHERE DOB BETWEEN '01/01/2005' AND '12/31/2005'`.

Failing to match the *condition-expression* date or time format to the display mode results in an error:

- In Display mode or ODBC mode, specifying date data in the incorrect format generates an SQLCODE -146 error. Specifying time data in the incorrect format generates an SQLCODE -147 error.
- In Logical mode, specifying date or time data in the incorrect format does not generate an error, but either returns no data or returns unintended data. This is because Logical mode does not parse a date or time in Display or ODBC format as a date or time value. The following **WHERE** clause, when executed in Logical mode, returns unintended data: `WHERE DOB BETWEEN 37500 AND 38000 AND DOB <> '1944-09-28'` returns a range of DOB values, including DOB=37891 (September 28, 1944), which the `<>` predicate was attempting to omit.

An invalid date or time value also generates an SQLCODE -146 or -147 error. An invalid date is one that you can specify in Display mode/ODBC mode, but Caché cannot convert into a Logical mode equivalent. For example, in ODBC mode the following generates an SQLCODE -146 error: `WHERE DOB > '1830-01-01'` because Caché cannot process a date value prior to December 31, 1840. The following in ODBC mode also generates an SQLCODE -146 error: `WHERE DOB BETWEEN '2005-01-01' AND '2005-02-29'`, because 2005 is not a leap year.

When in Logical mode, a Display mode or ODBC mode value is not parsed as a date or time value, and therefore its value is not validated. For this reason, in Logical mode a **WHERE** clause such as `WHERE DOB > '1830-01-01'` does not return an error.

Stream Fields

In most situations, you cannot use a stream field in a **WHERE** clause predicate. Doing so results in an SQLCODE -313 error. However, the following uses of stream fields are allowed in a **WHERE** clause:

- **Stream null testing:** you can specify the predicate `streamfield IS NULL` or `streamfield IS NOT NULL`.
- **Stream length testing:** you can specify a `CHARACTER_LENGTH(streamfield)`, `CHAR_LENGTH(streamfield)`, or `DATALength(streamfield)` function in a **WHERE** clause predicate.
- **Stream substring testing:** you can specify a `SUBSTRING(streamfield, start, length)` function in a **WHERE** clause predicate.

List Structures

Caché supports the list structure data type `%List` (data type class `%Library.List`). This is a compressed binary format, which does not map to a corresponding native data type for Caché SQL. It corresponds to data type `VARBINARY` with a default `MAXLEN` of 32749. For this reason, **Dynamic SQL** cannot use `%List` data in a **WHERE** clause comparison. For further details, refer to the [Data Types](#) reference page in this manual.

To reference structured list data, use the **%INLIST** predicate or the **FOR SOME %ELEMENT** predicate.

To use the data values of a list field in a *condition-expression*, you can use **%EXTERNAL** to compare the list values to a predicate. For example, to return all records in which the `FavoriteColors` list field value consists of the single element 'Red':

```
SELECT Name, FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors)='Red'
```

When **%EXTERNAL** converts a list to `DISPLAY` format, the displayed list items appear to be separated by a blank space. This “space” is actually the two non-display characters `CHAR(13)` and `CHAR(10)`. To use a *condition-expression* against more than one element in the list, you must specify these characters. For example, to return all records in which the `FavoriteColors` list field value consists of the two elements 'Orange' and 'Black' (in that order):

```
SELECT Name, FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors)='Orange' || CHAR(13) || CHAR(10) || 'Black'
```

Variables

A **WHERE** clause predicate can specify:

A **%TABLENAME**, or **%CLASSNAME pseudo-field variable** keyword. **%TABLENAME** returns the current table name. **%CLASSNAME** returns the name of the class corresponding to the current table. If the query references multiple tables, you can prefix the keyword with a table alias. For example, `t1.%TABLENAME`.

One or more of the following ObjectScript special variables (or their abbreviations): **\$HOROLOG**, **\$JOB**, **\$NAMESPACE**, **\$TLEVEL**, **\$USERNAME**, **\$ZHOROLOG**, **\$ZJOB**, **\$ZNSPACE**, **\$ZPI**, **\$ZTIMESTAMP**, **\$ZTIMEZONE**, **\$ZVERSION**.

List of Predicates

The SQL predicates fall into the following categories:

- [Equality Comparison Predicates](#)
- [BETWEEN Predicate](#)
- [IN and %INLIST Predicates](#)
- [Substring Predicates: %STARTSWITH, %CONTAINS, and %CONTAINSTERM](#)

- [NULL Predicate](#)
- [EXISTS Predicate](#)
- [FOR SOME Predicate](#)
- [FOR SOME %ELEMENT Predicate](#)
- [LIKE, %MATCHES, and %PATTERN Predicates](#)
- [%INSET and %FIND Predicates](#)

Predicate Case-Sensitivity

A predicate uses the [collation type](#) defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

The **%INLIST**, **Contains** operator (`()`), **%MATCHES**, and **%PATTERN** predicates do not use the field’s default collation. They always uses EXACT collation, which is case-sensitive.

A predicate comparison of two literal strings is always case-sensitive.

Predicate Conditions and %NOINDEX

You can preface a predicate condition with the **%NOINDEX** keyword to prevent the query optimizer using an index on that condition. This is most useful when specifying a range condition that is satisfied by the vast majority of the rows. For example, `WHERE %NOINDEX Age >= 1`. For further details, refer to [Index Optimization Options](#) in the *Caché SQL Optimization Guide*.

Predicate Condition on Outlier Value

If the **WHERE** clause in a Dynamic SQL query selects on a non-null outlier value, you can significantly improve performance by enclosing the outlier value literal in double parentheses. These double parentheses cause Dynamic SQL to use the outlier selectivity when optimizing. For example, if your business is located in Massachusetts (MA), a large percentage of your employees will reside in Massachusetts. For the Employees table Home_State field, 'MA' is the outlier value. To optimally select for this value, you should specify `WHERE Home_State=(('MA'))`.

This syntax should not be used in Embedded SQL or in a view definition. In Embedded SQL or a view definition, the outlier selectivity is always used and requires no special coding.

A **WHERE** clause in a Dynamic SQL query automatically optimizes for a null outlier value. For example, a clause such as `WHERE FavoriteColors IS NULL`. No special coding is required for IS NULL and IS NOT NULL predicates when NULL is the outlier value.

Outlier selectivity is determined by running the [Tune Table](#) utility. For further details, refer to [Outlier Optimization](#) in the “Optimizing Tables” chapter of the *Caché SQL Optimization Guide*.

Equality Comparison Predicates

The following are the available equality comparison predicates:

Table B-2: SQL Equality Comparison Predicates

Predicate	Operation
=	Equals
<>	Does not equal
!=	Does not equal
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

For example:

```
SELECT Name, Age FROM Sample.Person
WHERE Age < 21
```

SQL defines comparison operations in terms of collation: the order in which values are sorted. Two values are equal if they collate in exactly the same way. A value is greater than another value if it collates after the second value. String [field collation](#) takes the field's default collation. The Caché default collation is not case-sensitive. Thus, a comparison of two string field values or a comparison of a string field value with a string literal is (by default) not case-sensitive. For example, if `Home_State` field values are uppercase two-letter strings:

Expression	Value
'MA' = Home_State	TRUE for values MA.
'ma' = Home_State	TRUE for values MA.
'VA' < Home_State	TRUE for values VT, WA, WI, WV, WY.
'ar' >= Home_State	TRUE for values AK, AL, AR.

Note, however, that a comparison of two literal strings *is* case-sensitive: `WHERE 'ma' = 'MA'` is always FALSE.

BETWEEN Predicate

The **BETWEEN** comparison operator allows you to select those data values that are in the range specified by the syntax `BETWEEN lowval AND highval`. This range is inclusive of the *lowval* and *highval* values themselves. This is equivalent to a paired greater than or equal to operator and a less than or equal to operator. This comparison is shown in the following example:

```
SELECT Name, Age FROM Sample.Person
WHERE Age BETWEEN 18 AND 21
```

This returns all the records in the `Sample.Person` table with an `Age` value between 18 and 21, inclusive of those values. Note that you must specify the `BETWEEN` values in ascending order; a predicate such as `BETWEEN 21 AND 18` would return no records.

Like most predicates, `BETWEEN` can be inverted using the `NOT` logical operator, as shown in the following example:

```
SELECT Name, Age FROM Sample.Person
WHERE Age NOT BETWEEN 20 AND 55
ORDER BY Age
```

This returns all the records in the Sample.Person table with an Age value less than 20 or greater than 55, exclusive of those values.

BETWEEN is commonly used for a range of numeric values, which collate in numeric order. However, **BETWEEN** can be used for a collation sequence range of values of any data type.

BETWEEN uses the same collation type as the column it is matching against. By default, string data types collate as not case-sensitive.

For further details, refer to the [BETWEEN](#) predicate reference page in this manual.

IN and %INLIST Predicates

The **IN** predicate is used for matching a value to an unstructured series of items. It has the following syntax:

```
WHERE field IN (item1,item2[,...])
```

Collation applies to the IN comparison as it applies to an equality test. IN uses the field's default collation. By default, comparisons with field string values are not case-sensitive.

The **%INLIST** predicate is a Caché extension for matching a value to the elements of a Caché list structure. It has the following syntax:

```
WHERE item %INLIST listfield
```

%INLIST uses EXACT collation. Therefore, by default, **%INLIST** string comparisons are case-sensitive.

With either predicate you can perform equality comparisons and subquery comparisons.

For further details, refer to the [IN](#) and [%INLIST](#) predicate reference pages in this manual.

Substring Predicates

You can use the following to compare a field value to a substring:

Table B-3: SQL Substring Predicates

Predicate	Operation
%STARTSWITH	The value must start with the specified substring.
[Contains operator. The value must contain the specified substring.
%CONTAINS %CONTAINSTERM	The value must contain all of the specified substrings. Comparison is word-aware, using stemming and other algorithms.

%STARTSWITH Predicate

The Caché **%STARTSWITH** comparison operator permits you to perform partial matching on the initial characters of a string or numeric. The following example uses **%STARTSWITH** to select those records in which the Name value begins with "S":

```
SELECT Name, Age FROM Sample.Person
WHERE Name %STARTSWITH 'S'
```

Like other string field comparisons, **%STARTSWITH** comparisons use the field's default collation. By default, string fields are not case-sensitive. For example:

```
SELECT Name, Home_City, Home_State FROM Sample.Person
WHERE Home_City %STARTSWITH Home_State
```

For further details, refer to the [%STARTSWITH](#) predicate reference page in this manual.

Contains Operator (I)

The Contains operator is the open bracket symbol: [. It permits you to match a substring (string or numeric) to any part of a field value. The comparison is always case-sensitive. The following example uses the Contains operator to select those records in which the Name value contains a “S”:

```
SELECT Name, Age FROM Sample.Person
WHERE Name [ 'S'
```

%CONTAINS and %CONTAINSTERM Predicates

The **%CONTAINS** and **%CONTAINSTERM** predicates compare a column value to one or more substrings. If multiple substrings are specified, the column value must contain all of the specified substrings.

These comparison predicates are far more sophisticated than the Contains operator. They permit text-aware searching, such as handling word stemming, multiple-word phrases, and indexing on only significant words.

To use **%CONTAINS** or **%CONTAINSTERM**, the column value operand must be of type **%Text**. To do this, you must use Studio to change the property type from **%String** to **%Text**.

The **%CONTAINS** and **%CONTAINSTERM** predicates are Collection Predicates.

For further details, refer to the [%CONTAINS](#) and [%CONTAINSTERM](#) reference pages in this manual.

NULL Predicate

This detects undefined values. You can detect all null values, or all non-null values. The NULL predicate has the following syntax:

```
WHERE field IS [NOT] NULL
```

NULL predicate conditions are one of the few predicates that can be used on stream fields in a WHERE clause.

For further details, refer to the [NULL](#) predicate reference page in this manual.

EXISTS Predicate

This operates with subqueries to test whether a subquery evaluates to the empty set.

```
SELECT t1.disease FROM illness_tab t1 WHERE EXISTS
  (SELECT t2.disease FROM disease_registry t2
   WHERE t1.disease = t2.disease
   HAVING COUNT(t2.disease) > 100)
```

For further details, refer to the [EXISTS](#) predicate reference page in this manual.

FOR SOME Predicate

The FOR SOME predicate of the WHERE clause can be used to determine whether or not to return any records based on a condition test of one or more field values. This predicate has the following syntax:

```
FOR SOME (table [AS t-alias]) (fieldcondition)
```

FOR SOME specifies that *fieldcondition* must evaluate to true; at least one of the field values must match the specified condition. *table* can be a single table or a comma-separated list of tables, and each table can optionally take a table alias. *fieldcondition* specifies one or more conditions for one or more fields within the specified *table*. Both the *table* argument and the *fieldcondition* argument must be delimited by parentheses.

The following example shows the use of the FOR SOME predicate to determine whether to return a result set:

```
SELECT Name, Age AS AgeWithWorkers
FROM Sample.Person
WHERE FOR SOME (Sample.Person) (Age < 65)
ORDER BY Age
```

In the above example, if at least one field contains an Age value less than the specified age, all of the records are returned. Otherwise, no records are returned.

For further details, refer to the [FOR SOME](#) predicate reference page in this manual.

FOR SOME %ELEMENT Predicate

The **FOR SOME %ELEMENT** predicate of the WHERE clause has the following syntax:

```
FOR SOME %ELEMENT(field) [AS e-alias] (predicate)
```

The **FOR SOME %ELEMENT** predicate matches the elements in *field* with the specified *predicate* clause value. The **SOME** keyword specifies that at least one of the elements in *field* must satisfy the specified *predicate* condition. The *predicate* can contain the **%VALUE** or **%KEY** keyword.

The **FOR SOME %ELEMENT** predicate is a Collection Predicate.

For further details, refer to the [FOR SOME %ELEMENT](#) predicate reference page in this manual.

LIKE, %MATCHES, and %PATTERN Predicates

These three predicates allow you to perform pattern matching.

- **LIKE** allows you to pattern match using literals and wildcards. Use **LIKE** when you wish to return data values that contain a known substring of literal characters, or contain several known substrings in a known sequence. **LIKE** uses the collation of its target for letter case comparisons.
- **%MATCHES** allows you to pattern match using literals, wildcards, and lists and ranges. Use **%MATCHES** when you wish to return data values that contain a known substring of literal characters, or contain one or more literal characters that fall within a list or range of possible characters, or contain several such substrings in a known sequence. **%MATCHES** uses **EXACT** collation for letter case comparisons.
- **%PATTERN** allows you to specify a pattern of character types. For example, '1U4L1', '.A' (1 uppercase letter, 4 lowercase letters, one literal comma, followed by any number of letter characters of either case). Use **%PATTERN** when you wish to return data values that contain a known sequence of character types. **%PATTERN** can specify known literal characters, but is especially useful when the data value is unimportant, but the character type format of those values is significant.

To perform a comparison with the first characters of a string, use the **%STARTSWITH** predicate.

Predicates and Logical Operators

Multiple predicates can be associated using the **AND** and **OR** logical operators. Multiple predicates can be grouped using parentheses. Because Caché optimizes execution of the **WHERE** clause using defined indices and other optimizations, the order of evaluation of predicates linked by **AND** and **OR** logical operators cannot be predicted. For this reason, the order in which you specify multiple predicates has little or no effect on performance. If strict left-to-right evaluation of predicates is desired, you can use a **CASE** statement.

Note: The OR logical operator cannot be used to associate a Collection Predicate that references a table field with a predicate that references a field in a different table. For example,

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple')  
OR t2.Age < 65
```

The Collection Predicates are [FOR SOME %ELEMENT](#), [%CONTAINS](#), and [%CONTAINSTERM](#). Because this restriction depends on how the optimizer uses indices, SQL may only enforce this restriction when indices are added to a table. It is strongly suggested that this type of logic be avoided in all queries.

For further details, refer to “[Logical Operators](#)” in the “Language Elements” chapter of *Using Caché SQL*.

See Also

- [SELECT](#) statement
- [HAVING](#) clause
- [Overview of Predicates](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

WHERE CURRENT OF

An UPDATE/DELETE clause that specifies the current row using a cursor.

```
WHERE CURRENT OF cursor
```

Arguments

<i>cursor</i>	Specifies that the operation is done at the current position of <i>cursor</i> , which is a cursor that points to the table.
---------------	---

Description

The **WHERE CURRENT OF** clause can be used in a [cursor-based Embedded SQL UPDATE](#) or **DELETE** statement to specify the cursor positioned on the record to be updated or deleted. For example:

```
&sql(DELETE FROM Sample.Employees WHERE CURRENT OF EmployeeCursor)
```

which deletes the row that the last **FETCH** command obtained from the "EmployeeCursor" cursor.

An Embedded SQL **UPDATE** or **DELETE** can use a [WHERE](#) clause (with no cursor), or a **WHERE CURRENT OF** with a declared cursor, but not both. If you specify an **UPDATE** or **DELETE** with neither **WHERE** nor **WHERE CURRENT OF**, all of the records in the table are updated or deleted.

UPDATE Restriction

When using a **WHERE CURRENT OF** clause, you cannot update a field using the current field value to generate an updated value. For example, `SET Salary=Salary+100` or `SET Name=UPPER(Name)`. Attempting to do so results in an SQLCODE -69 error: SET <field> = <value expression> not allowed with WHERE CURRENT OF <cursor>.

Examples

The following Embedded SQL example shows an **UPDATE** operation using **WHERE CURRENT OF**:

```
NEW %ROWCOUNT,%ROWID
&sql(DECLARE WPCursor CURSOR FOR
      SELECT Lang FROM SQLUser.WordPairs
      WHERE Lang='Sp' )
&sql(OPEN WPCursor)
      QUIT:(SQLCODE'=0)
FOR { &sql(FETCH WPCursor)
      QUIT:SQLCODE
      &sql(UPDATE SQLUser.WordPairs SET Lang='Es'
          WHERE CURRENT OF WPCursor)
      IF SQLCODE=0 {
      WRITE !,"Update succeeded"
      WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
      ELSE {
      WRITE !,"Update failed, SQLCODE=",SQLCODE }
      }
&sql(CLOSE WPCursor)
```

The following Embedded SQL example shows a **DELETE** operation using **WHERE CURRENT OF**:

```
NEW %ROWCOUNT,%ROWID
&sql(DECLARE WPCursor CURSOR FOR
      SELECT Lang FROM SQLUser.WordPairs
      WHERE Lang='En')
&sql(OPEN WPCursor)
      QUIT:(SQLCODE'=0)
FOR { &sql(FETCH WPCursor)
      QUIT:SQLCODE
      &sql(DELETE FROM SQLUser.WordPairs
          WHERE CURRENT OF WPCursor)
      IF SQLCODE=0 {
        WRITE !,"Delete succeeded"
        WRITE !,"Row count=",%ROWCOUNT," RowID=",%ROWID }
      ELSE {
        WRITE !,"Delete failed, SQLCODE=",SQLCODE }
      }
&sql(CLOSE WPCursor)
```

See Also

- [DECLARE, OPEN, FETCH, CLOSE](#)
- [DELETE, UPDATE, INSERT OR UPDATE](#)
- [SQL Cursors](#) in the “Using Embedded SQL” chapter of *Using Caché SQL*
- [SQLCODE error messages](#) listed in the *Caché Error Reference*

SQL Predicate Conditions

Overview of Predicates

Describes logical conditions that evaluate to either true or false.

Use of Predicates

A predicate is a condition expression that evaluates to a boolean value, either true or false.

Predicates can be used as follows:

- In a **SELECT** statement's **WHERE** clause or **HAVING** clause to determine which rows are relevant to a particular query. Note that not all predicates can be used in a **HAVING** clause.
- In a **JOIN** operation's **ON** clause to determine which rows are relevant to the join operation.
- In an **UPDATE** or **DELETE** statement's **WHERE** clause, to determine which rows are to be modified.
- In a **WHERE CURRENT OF** statement's **AND** clause.
- In a **CREATE TRIGGER** statement's **WHEN** clause to determine when to apply triggered action code.

List of Predicates

Every predicate contains one or more comparison operators, either symbols or keyword clauses. Caché SQL supports the following comparison operators:

Comparison Operator	Description
= (equals) <> (does not equal) != (does not equal) > (is greater than) >= (is greater than or equal to) < (is less than) <= (is less than or equal to)	Equality comparison conditions. Can be used for numeric comparisons or string collation order comparisons. For numeric comparisons, an empty string value ("") is evaluated as 0. A NULL in any equality comparison always returns the empty set; use the IS NULL predicate instead. See Relational Operators in <i>Using Caché SQL</i> .
IS [NOT] NULL	Tests whether a field has undefined (NULL) values. See IS NULL .
IS [NOT] JSON	Tests whether a value is a JSON formatted string or an oref to a JSON array or a JSON object. See IS JSON .
EXISTS (subquery)	Uses a subquery to test a specified table for existence of one or more rows. See EXISTS .
BETWEEN x AND y	A BETWEEN condition uses >= and <= comparison conditions together. Match must be between two specified range limit values (inclusive). See BETWEEN .
IN (item1,item2[...],itemn) IN (subquery)	An equality condition that matches a field value to any of the items in a comma-separated list, or any of the items returned by a subquery. See IN .

Comparison Operator	Description
%INLIST listfield	An equality condition that matches a field value to any of the elements in a %List structured list. See %INLIST .
[Contains operator . Match must contain the specified string. The Contains operator uses EXACT collation, and is therefore case-sensitive. Must specify value in Logical format.
]	Follows operator . Match must appear after the specified item in collation sequence. Must specify value in Logical format.
%STARTSWITH string	Match must begin with the specified string. See %STARTSWITH .
%CONTAINS %CONTAINSTERM	Match word-aware strings with complex text analysis. Match must contain all of the specified single-word or multi-word strings. WHERE clause only; cannot be used in a HAVING clause. See %CONTAINS and %CONTAINSTERM .
FOR SOME	A boolean comparison condition. The FOR SOME condition must be true for at least one data value of the specified field. See FOR SOME .
FOR SOME %ELEMENT	A list element comparison condition with a %VALUE or %KEY predicate clause. %VALUE must match the value of at least one element of the list. %KEY must be less than or equal to the number of elements in the list. %VALUE and %KEY clauses can use any of the other comparison operators. See FOR SOME %ELEMENT .
LIKE	A pattern match condition using literals and wildcards. Use LIKE when you wish to return data values that contain a known substring of literal characters, or contain several known substrings in a known sequence. LIKE uses the collation of its target for letter case comparisons. (Contrast with the Contains operator, which uses EXACT collation.) See LIKE .
%MATCHES	A pattern match condition using literals, wildcards, and lists and ranges. Use %MATCHES when you wish to return data values that contain a known substring of literal characters, or contain one or more literal characters that fall within a list or range of possible characters, or contain several such substrings in a known sequence. %MATCHES uses EXACT collation for letter case comparisons. See %MATCHES .

Comparison Operator	Description
%PATTERN	A pattern match condition using character types. For example, '1U4L1', '.A' (1 uppercase letter, 4 lowercase letters, one literal comma, followed by any number of letter characters of either case). Use %PATTERN when you wish to return data values that contain a known sequence of character types. %PATTERN can specify known literal characters, but is especially useful when the data value is unimportant, but the character type format of those values is significant. See %PATTERN .
ALL ANY SOME	A quantified-comparison condition. See ALL , ANY , and SOME .
%INSET %FIND	Field value comparison conditions that enable filtering of RowID field values using an abstract, programmatically specified temp-file or bitmap index. %INSET supports simple comparisons. %FIND supports comparisons involving a bitmap index.

NULL

A NULL is the absence of any value. By definition, it fails all boolean tests: no value is equal to NULL, no value is unequal to NULL, no value is greater than or less than NULL. Even NULL=NULL fails as a predicate. Because the IN predicate is a series of OR'ed equality tests, it is not meaningful to specify NULL in the IN value list. Therefore, specifying any predicate condition eliminates any instances of that field that are NULL. The only way to include NULL fields in the result set from a predicate condition is to use the IS NULL predicate. This is shown in the following example:

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors = $LISTBUILD('Red') OR FavoriteColors IS NULL
```

Collation

A predicate uses the [collation type](#) defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

If you specify a collation type in a query, you must specify it on both sides of the comparison. Specifying a collation type can affect index usage; for further details, refer to [Index Collation](#) in the “Defining and Building Indices” chapter of the *Caché SQL Optimization Guide*.

Certain predicate comparisons can involve substrings embedded within a string: the Contains operator (()), the **%MATCHES** predicate, and the **%PATTERN** predicate. These predicates always uses EXACT collation, and are therefore always case-sensitive. Because some collations append a blank space to a string, these predicates could not perform their function if they followed the field's default collation. However, the **LIKE** predicate can use wildcards to match substrings embedded within a string. **LIKE** uses the field's default collation, which by default is not case-sensitive.

Compound Predicates

A predicate is the simplest version of a condition expression; a condition expression can consist of one or more predicates. You can link multiple predicates together with the AND and OR [logical operators](#). You can invert the sense of a predicate by placing the NOT unary operator before the predicate. The NOT unary operator only affects the predicate that immediately

follows it. Predicates are evaluated in strict left-to-right order. You can use parentheses to group predicates. You can place a NOT unary operator before the opening parentheses to invert the sense of a group of predicates. Spaces are not required before or after parentheses, or between parentheses and logical operators.

The **IN** and **%INLIST** predicates are functionally equivalent to multiple OR equality predicates. The following examples are equivalent:

```
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State='MA' OR Home_State='VT' OR Home_State='NH' "
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

```
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State IN('MA','VT','NH') "
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

```
SET list=$LISTBUILD("MA","VT","NH")
SET q1="SELECT Name,Home_State FROM Sample.Person "
SET q2="WHERE Home_State %INLIST(?)"
SET myquery=q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(list)
DO rset.%Display()
```

The **FOR SOME %ELEMENT** predicate can contain logical operators, as well as be linked to other predicates using logical operators. This is shown in the following example:

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red' OR %Value='White'
OR %Value %STARTSWITH 'B')
AND (Name BETWEEN 'A' AND 'F' OR Name %STARTSWITH 'S')
ORDER BY Name
```

Note the parentheses around (Name BETWEEN 'A' AND 'F' OR Name %STARTSWITH 'S'); without these grouping parentheses, the **FOR SOME %ELEMENT** condition would not apply to Name %STARTSWITH 'S'.

Collection Predicates with OR

FOR SOME %ELEMENT, **%CONTAINS**, and **%CONTAINSTERM** are Collection Predicates. The use of these predicates with the OR logical operator is restricted, as follows. The OR logical operator cannot be used to associate a Collection Predicate that references a table field with a predicate that a references a field in a different table. For example,

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple')
OR t2.Age < 65
```

Because this restriction depends on how the optimizer uses indices, SQL may only enforce this restriction when indices are added to a table. It is strongly suggested that this type of logic be avoided in all queries.

Predicates and %SelectMode

All predicates perform their comparisons using Logical (internal storage) data values. However, some predicates can perform format mode conversion on the predicate value(s), converting it from ODBC or Display format to Logical format. Other predicates cannot perform format mode conversion, and therefore must always specify the predicate value in Logical format.

Predicates that perform format mode conversion determine whether conversion is required from the data type (such as DATE or %List) of the matching field and determine the type of conversion from the [%SelectMode setting](#). If %SelectMode

is set to a value other than Logical format (such as %SelectMode=ODBC or %SelectMode=Display) the predicate value(s) must be specified in the correct ODBC or Display format.

- Equality predicate perform format mode conversion. Caché converts the predicate value to Logical format, then matches it with the field values. If %SelectMode is set to a mode other than Logical format, the predicate value(s) must be specified in the %SelectMode format (ODBC or Display) for data types whose display value differs from the Logical storage value. For example, dates, times, and %List-formatted strings. Because Caché automatically performs this format conversion, specifying this type of predicate value in Logical format commonly results in an SQLCODE error. For example, SQLCODE -146 “Unable to convert date input to a valid logical date value” (Caché assumes the supplied Logical value is an ODBC or Display value and attempts to convert it to a Logical value — which doesn’t succeed.) Affected predicates include =, <, >, BETWEEN, and IN.
- Pattern predicates cannot perform format mode conversion, because Caché cannot meaningfully convert the predicate value. Therefore, the predicate value must be specified in Logical format, regardless of the %SelectMode setting. Specifying predicate value(s) in ODBC or Display format commonly results in no data matches or unintended data matches. Affected predicates include %INLIST, LIKE, %MATCHES, %PATTERN, %STARTSWITH, [(the Contains operator), and] (the Follows operator).

You can use the %INTERNAL, %EXTERNAL, or %ODBCOUT format-transform functions to transform the field that the predicate operates upon. This allows you to specify the predicate value in another format. For example, WHERE %ODBCOut (DOB) %STARTSWITH '1955-'. However, specifying a format-transform function on a matching field prevents the use of an index for the field. This can have a significant negative effect upon performance.

In the following Dynamic SQL example, the **BETWEEN** predicate (an equality predicate) must specify dates in %SelectMode=1 (ODBC) format:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB BETWEEN '1950-01-01' AND '1960-01-01'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

In the following Dynamic SQL examples, the %STARTSWITH predicate (a pattern predicate) cannot perform format mode conversion. The first example attempts to specify a %STARTSWITH for dates in the %SelectMode=ODBC format for years in the 1950s. However, because the table does not contain birth dates that begin with \$HOROLOG 195 (dates in the year 1894), no rows are selected:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %STARTSWITH '195'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following example uses the %ODBCOut format-transform function on the matching DOB field so that %STARTSWITH can be used to select for years in the 1950s in ODBC format. However, note that this usage prevents the use of an index on the DOB field.

```

ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOut(DOB) %STARTSWITH '195'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"

```

In the following example the **%STARTSWITH** predicate specifies a **%STARTSWITH** for dates in Logical (internal) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 (\$HOROLOG 41000) through December 28 1955 (\$HOROLOG 41999)) are selected. The DOB field index is used:

```

ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %STARTSWITH '41'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"

```

Suppress Literal Substitution

You can [suppress literal substitution](#) during compile pre-parsing by enclosing the predicate argument in double parentheses. For example, `LIKE (('abc%'))`. This may improve query performance by improving overall selectivity and/or subscript bounding selectivity. However, it should be avoided when the same query is called multiple times with different values, as it will result in the creation of a separate cached query for each query call.

Example

The following example uses a variety of conditions in the **WHERE** clause of a query:

```

SELECT PurchaseOrder FROM MyTable
  WHERE OrderTotal >= 1000
  AND ItemName %STARTSWITH :partname
  AND AnnualOrders BETWEEN 50000 AND 100000
  AND City LIKE 'Ch%'
  AND CustomerNumber IN
    (SELECT CustNum FROM TheTop100
     WHERE TheTop100.City='Boston')
  AND :minorder > SOME
    (SELECT OrderTotal FROM Orders
     WHERE Orders.Customer = :cust)

```

See Also

- [SELECT](#) statement, [HAVING](#) clause, [WHERE](#) clause
- [CREATE TRIGGER](#)

ALL

Matches a value with all corresponding values from a subquery.

```
scalar-expression comparison-operator ALL (subquery)
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with the result set generated by the <i>subquery</i> .
<i>comparison-operator</i>	One of the following comparison operators: = (equal to), <> or != (not equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), [(contains), or] (follows).
<i>subquery</i>	A subquery, enclosed in parentheses, which returns a result set from a single column that is used for the comparison with <i>scalar-expression</i> .

Description

The **ALL** keyword works in conjunction with a comparison operator to create a [predicate](#) (a quantified comparison condition) that is true if the value of a scalar expression matches *all* of the corresponding values retrieved by the [subquery](#). The **ALL** predicate compares a single *scalar-expression* item with a single subquery **SELECT** item. A subquery with more than one select item generates an SQLCODE -10 error.

ALL can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

Where applicable, the system automatically applies Set-Valued Subquery Optimization (SVSO) to an **ALL** subquery. For details on this optimization, and using the %NOSVSO keyword to override it, refer to “Query Optimization Options” on the [FROM clause](#) reference page.

Examples

The following example selects those ages in the Person database that are less than all of the ages in the Employee database:

```
SELECT DISTINCT Age FROM Sample.Person
WHERE Age < ALL
  (SELECT Age FROM Sample.Employee)
ORDER BY Age
```

The following example selects those names in the Person database that are longer or shorter than all of the names in the Employee database:

```
SELECT $LENGTH(Name) AS NameLength,Name FROM Sample.Person
WHERE $LENGTH(Name) > ALL
  (SELECT $LENGTH(Name) FROM Sample.Employee)
OR $LENGTH(Name) < ALL
  (SELECT $LENGTH(Name) FROM Sample.Employee)
```

The following example returns a list of states west of the Mississippi River, all of which states do not contain an employee with the title of Manager or Director:

```
SELECT DISTINCT State
FROM Sample.USZipCode
WHERE Longitude < -93
  AND State != ALL
    (SELECT Home_State FROM Sample.Employee
     WHERE Title [ 'Manager' OR Title [ 'Director' ])
ORDER BY State
```

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [ANY SOME](#)
- [Overview of Predicates](#)

ANY

Matches a value with at least one matching value from a subquery.

```
scalar-expression comparison-operator ANY (subquery)
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with the result set generated by <i>subquery</i> .
<i>comparison-operator</i>	One of the following comparison operators: = (equal to), <> or != (not equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), [(contains), or] (follows).
<i>subquery</i>	A subquery, enclosed in parentheses, which returns a result set that is used for the comparison with <i>scalar-expression</i> .

Description

The **ANY** keyword works in conjunction with a comparison operator to create a [predicate](#) (a quantified comparison condition) that is true if the value of a scalar expression matches one or more of the corresponding values retrieved by the [subquery](#). The **ANY** predicate compares a single *scalar-expression* item with a single subquery **SELECT** item. A subquery with more than one select item generates an SQLCODE -10 error.

Note: The **ANY** and **SOME** keywords are synonyms.

ANY can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

Where applicable, the system automatically applies Set-Valued Subquery Optimization (SVSO) to an **ANY** subquery. For details on this optimization, and using the %NOSVSO keyword to override it, refer to “Query Optimization Options” on the [FROM clause](#) reference page.

Example

The following example selects those employees with salaries greater than \$75,000 that live in any of the states west of the Mississippi River:

```
SELECT Name,Salary,Home_State FROM Sample.Employee
WHERE Salary > 75000
AND Home_State = ANY
  (SELECT State FROM Sample.USZipCode
   WHERE Longitude < -93)
ORDER BY Home_State
```

See Also

- [SELECT statement HAVING clause WHERE clause](#)
- [ALL SOME](#)
- [Overview of Predicates](#)

BETWEEN

Matches a value to a range of values.

```
scalar-expression BETWEEN lowval AND highval
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with the range of values between <i>lowval</i> and <i>highval</i> (inclusive).
<i>lowval</i>	Expression that resolves to the low collation sequence value specifying the beginning of a range of values to match with each value in <i>scalar-expression</i> .
<i>highval</i>	Expression that resolves to the high collation sequence value specifying the end of a range of values to match with each value in <i>scalar-expression</i> .

Description

The **BETWEEN** predicate allows you to select those data values that are in the range specified by *lowval* and *highval*. This range is inclusive of the *lowval* and *highval* values themselves. This is equivalent to a paired greater than or equal to operator and a less than or equal to operator. This comparison is shown in the following example:

```
SELECT Name, Age FROM Sample.Person
WHERE Age BETWEEN 18 AND 21
ORDER BY Age
```

This returns all the records in the Sample.Person table with an Age value between 18 and 21, inclusive of those values. Note that you must specify the **BETWEEN** values in ascending order; a predicate such as `BETWEEN 21 AND 18` would return the null string. If none of the scalar expression values fall within the specified range, **BETWEEN** returns the null string.

Like most predicates, **BETWEEN** can be inverted using the NOT logical operator. Neither **BETWEEN** nor **NOT BETWEEN** can be used to return NULL fields. To return NULL fields use **IS NULL**. **NOT BETWEEN** is shown in the following example:

```
SELECT Name, Age FROM Sample.Person
WHERE Age NOT BETWEEN 20 AND 55
ORDER BY Age
```

This returns all the records in the Sample.Person table with an Age value less than 20 or greater than 55, exclusive of those values.

BETWEEN can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

Collation Types

BETWEEN is commonly used for a range of numeric values, which collate in numeric order. However, **BETWEEN** can be used for a collation sequence range of values of any data type.

BETWEEN uses the same collation type as the column it is matching against. By default, string data types collate as SQLUPPER, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

If your query assigns a different collation type to the column, you must also apply this collation type to the **BETWEEN** *substring*. This is shown in the following examples:

In the following example, **BETWEEN** uses the fields' default letter case collation, **SQLUPPER**, which is not case-sensitive. It returns records where Name is higher in alphabetical order than Home_State, and Home_State is higher in alphabetical order than Home_City:

```
SELECT Name,Home_State,Home_City
FROM Sample.Person
WHERE Home_State BETWEEN Name AND Home_City
ORDER BY Home_State
```

In the following example, **BETWEEN** string comparisons are not case-sensitive, because the Home_State field is defined as **SQLUPPER**. This means that the *lowval* and *highval* are functionally identical, selecting 'MA' in any lettercase:

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State
    BETWEEN 'MA' AND 'Ma'
ORDER BY Home_State
```

In the following example, the **%SQLSTRING** collation function causes **BETWEEN** string comparisons to be case-sensitive. It selects those records with Home_State values of 'MA' through 'Ma', which in this data set includes 'MA', 'MD', 'ME', 'MO', 'MS', and 'MT':

```
SELECT Name,Home_State FROM Sample.Person
WHERE %SQLSTRING(Home_State)
    BETWEEN %SQLSTRING('MA') AND %SQLSTRING('Ma')
ORDER BY Home_State
```

In the following example, the **BETWEEN** string comparison is not case-sensitive and ignores blank spaces and punctuation marks:

```
SELECT Name FROM Sample.Person
WHERE %STRING(Name) BETWEEN %STRING('OA') AND %STRING('OZ')
ORDER BY Name
```

Using **%STRING**, this example can select Odem, O'Donnell, and Olsen. Without the **%STRING** collation type, O'Donnell would not be selected.

Refer to **%SQLUPPER** for further information on case transformation functions.

The following example shows **BETWEEN** used in an **INNER JOIN** operation **ON** clause. It is performing a string comparison which is not case-sensitive:

```
SELECT P.Name AS PersonName,E.Name AS EmpName
FROM Sample.Person AS P INNER JOIN Sample.Employee AS E
ON P.Name BETWEEN 'an' AND 'ch' AND P.Name=E.Name
```

%SelectMode

If **%SelectMode** is set to a value other than Logical format, the **BETWEEN** predicate values must be specified in the **%SelectMode** format (ODBC or Display). This applies mainly to dates, times, and Caché format lists (%List). Specifying predicate value(s) in Logical format commonly results in an SQLCODE error. For example, SQLCODE -146 "Unable to convert date input to a valid logical date value".

In the following Dynamic SQL example, the **BETWEEN** predicate must specify dates in **%SelectMode=1** (ODBC) format:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB BETWEEN '1950-01-01' AND '1960-01-01'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [Overview of Predicates](#)
- “[Collation](#)” chapter in *Using Caché SQL*

%CONTAINS

Matches a value to one or more phrases using word-aware matching.

```
scalar-expression %CONTAINS(word[,word...])
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with one or more <i>word</i> strings. Must be of data type %Text.
<i>word</i>	An alphabetic string or comma-separated list of alphabetic strings to match with values in <i>scalar-expression</i> . A <i>word</i> should be a complete word, or a phrase consisting of several complete words. The word or phrase should be delimited with single quotes.

Description

The **%CONTAINS** predicate allows you to select those data values that match the string or strings specified in *word*. This comparison operation is word-aware; it is not a simple string match operation. If you specify more than one *word*, *scalar-expression* must contain all of the specified word strings. Word strings may be presented in any order. If *word* does not match any of the scalar expression data values, **%CONTAINS** returns the null string.

%CONTAINS is a collection predicate. It can only be used in the **WHERE** clause of a **SELECT** statement. **%CONTAINS** cannot be used as a predicate that selects fields for a **JOIN** operation.

%CONTAINS can be used on a %Text string or a character stream field.

To use **%CONTAINS** on a string, change the %String property to %Text, and set LANGUAGECLASS and MAXLEN [property parameters](#). For example:

```
Property MySentences As %Text(LANGUAGECLASS = "%Text.English",MAXLEN = 1000);
```

Specifying a MAXLEN value (in bytes) is required for %Text properties.

To use **%CONTAINS** to search a [character stream field](#), the stream field must be defined as type %Stream.GlobalCharacterSearchable. For example:

```
Property MyTextStream As %Stream.GlobalCharacterSearchable(LANGUAGECLASS = "%Text.English");
```

The **%CONTAINS** predicate is one of the few predicates that can be used on a [stream field](#) in a **WHERE** clause.

The available languages are English, French, German, Italian, Japanese, Portuguese, and Spanish. See the %Text package class documentation (in %SYS) in the *InterSystems Class Reference* for further details.

The system generates an SQLCODE -309 error if *scalar-expression* is neither data type %Text nor %Stream.GlobalCharacterSearchable.

The system generates an SQLCODE -472 error if *scalar-expression* is not a collection-valued field (property).

Word-Aware Matching

A *word* should always be a complete word or sequence of words. When the *word* argument(s) are single words, Caché uses language analysis rules, including punctuation analysis and stemming rules, to match only the specified word. For example, the *word* argument “set” would match the word “set” or “Set” (not case-sensitive), and also stem forms such as “sets” and “setting”. However, it would *not* match words such as “setscrew,” “settle,” or “Seth,” even though these words contain the specified string.

Stemming rules provide for matching between any two forms of the word stem. Stemming is performed on both the search term(s) and the searched text. For example, you can specify `%CONTAINS('jumping')` and match the word “jumps” in the text.

This word-aware matching is fundamentally different from the character-by-character string matching performed by the SQL [Contains operator](#) (`()`).

Multiple-Word Phrases

The `%CONTAINS` keyword operator can search for multiple-word *word* strings, such as 'United States of America'. How these searches are conducted depends upon the setting of the `NGRAMLEN` property for the class. By default, `NGRAMLEN=1`, which means that `%CONTAINS` treats a *word* argument as a single word, regardless of how many words it actually contains.

If `NGRAMLEN` is equal or greater than the number of words in a *word* phrase, the phrase is treated as a word-aware test. That is, comparisons are *not* case-sensitive, and comparison is word-by-word. Thus 'School of Art' would match 'School of Art', or 'school of art', but not 'School of Arthur Murray'.

If `NGRAMLEN` is less than the number of words in a *word* phrase, Caché consults the dictionary of the `LANGUAGECLASS` to select the least-frequently-occurring term of length `NGRAMLEN` (or less). For example, when `NGRAMLEN=2` you may still use predicates such as `myDoc %CONTAINS('big black dog')`. In this case, the least-frequently-occurring term of length `NGRAMLEN` might be 'big black'. Caché then retrieves the `myDoc` property (typically from the [master map](#)), and the document is re-parsed to determine if the post-stemming representation of the query pattern appeared in the post-stemming representation of `myDoc`. This re-parsing is expensive. In the case of the Japanese language, which does not require case conversion, some of the post-processing done by `%CONTAINS` is unnecessary overhead.

Contains Analysis

`%CONTAINS` matching is governed by the class parameters of the `%Text.Text` system class, found in the `%SYS` namespace. These parameters allow you to specify, among other things, whether comparison is to be case-sensitive or not case-sensitive, and the treatment of numbers, punctuation characters, and multi-word phrases.

Caché can use specific language analysis rules, including common word analysis (“noise word” lists) and stemming rules, to determine similarity. The available languages are English, French, German, Italian, Japanese, Portuguese, and Spanish.

For a much more detailed treatment of `%CONTAINS` and `%Text`, refer to the `%Text` package class documentation in the *InterSystems Class Reference*.

%CONTAINS and %CONTAINSTERM

The `%CONTAINS` and `%CONTAINSTERM` predicates perform the same word-aware comparison for their supplied *word* arguments. They differ in their requirements for multiple-word phrases:

- A `%CONTAINSTERM` argument cannot contain noise words. A `%CONTAINS` argument can contain noise words.

The `%CONTAINSTERM` operator gives superior performance for certain types of comparisons, especially very large search sets. `%CONTAINSTERM` is preferable when performing comparisons in Japanese.

A `%CONTAINS` comparison is case-sensitive, unless the `CASEINSENSITIVE` class property is specified.

`%CONTAINSTERM` comparisons use the [collation type](#) defined for the *scalar-expression*. By default, string data type fields are defined with `SQLUPPER` collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#). Because `%CONTAINSTERM` uses the field collation type, the `CASEINSENSITIVE` class property is not applicable to the `%CONTAINSTERM` comparison. Collation can optionally be set to `%EXACT` for case-sensitive operations.

Use of `%CONTAINS` in a search of text represented as a [stream field](#) requires that the stemmed, noiseword-filtered text be converted to a string. If such a stream is longer than the maximum length of a string, then applications should use `%CONTAINSTERM` instead of `%CONTAINS`, as there is no limitation on size of a stream when `%CONTAINSTERM`

is in use. For information on the maximum length of a string, see the section “[Support for Long String Operations](#)” in the chapter “[Server Configuration Options](#)” in the *Caché Programming Orientation Guide*.

For further details, refer to the [%CONTAINSTERM](#) predicate.

Collection Predicates

%CONTAINS is a collection predicate. It can be used in most contexts where a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual. It is subject to the following restrictions:

- You cannot use **%CONTAINS** in a HAVING clause.
- You cannot use **%CONTAINS** as a predicate that selects fields for a **JOIN** operation.
- You cannot associate **%CONTAINS** with another predicate condition using the OR logical operator if the two predicates reference fields in different tables. For example:

```
WHERE t1.text %CONTAINS('Continental United States') OR t2.Timezone BETWEEN 5 AND 8
```

Because this restriction depends on how the optimizer uses indices, SQL may only enforce this restriction when indices are added to a table. It is strongly suggested that this type of logic be avoided in all queries.

iKnow and iFind

The Caché [iKnow text analysis tool](#) and [iFind text search tool](#) also provide word-aware analysis. These facilities are entirely separate from %Text classes. They provide a substantially different and significantly more sophisticated form of textual analysis.

Examples

The following Embedded SQL example performs a **%CONTAINS** comparison with a literal phrase:

```
&sql(SELECT name,stats
      INTO :badname, :badstat
      FROM Sample.Employee
      WHERE status %CONTAINS('an invalid value'))
```

The following Embedded SQL example uses a host variable to perform a **%CONTAINS** comparison:

```
SET text="invalid"
&sql(SELECT name,stats
      INTO :badname, :badstat
      FROM Sample.Employee
      WHERE status %CONTAINS(:text))
```

Other Equivalence Comparisons

You can perform other types of equivalence comparisons by using string comparison operators. These include the following:

- An equivalence comparison on the initial character(s) of a string using **%STARTSWITH**.
- An equivalence comparison on the entire string, using the equal sign operator:

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State = 'VT'
```

This example selects any record that contains the Home_State field value “VT”. By default, this string comparison is not case-sensitive.

- A non-equivalence comparison on the entire string, using the does not equal operator:

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State <> 'MA'
ORDER BY Home_State
```

This example selects all records that where the Home_State field value *is not* equal to “MA”. By default, this string comparison is not case-sensitive.

- An equivalence comparison on the entire string to multiple values, using the IN keyword operator:

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State IN ('VT','MA','NH','ME')
ORDER BY Home_State
```

This example selects any record that contains any of the specified Home_State field values. By default, this string comparison is not case-sensitive.

- An equivalence comparison on the entire string to a value pattern, using the %PATTERN keyword operator:

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State %PATTERN '1U1"C"'
ORDER BY Home_State
```

This example selects any record that contains a Home_State field value that matches the pattern of 1U (one uppercase letter) followed by 1"C" (one literal letter “C”). This pattern would be fulfilled by the Home_State abbreviations “NC” or “SC”.

- An equivalence comparison of a substring to a value, using the contains operator:

```
SELECT Name FROM Sample.Person
WHERE Name [ 'y'
```

This example selects all Name records that contain the lowercase letter “y”. By default, this string comparison *is* case-sensitive.

- An equivalence comparison of a substring with one or more wildcards to a value, using the LIKE keyword operator:

```
SELECT Name FROM Sample.Person
WHERE Name LIKE '_a%'
```

This example selects all Name records that contain the letter “a” as the second letter. This string comparison uses the Name collation type to determine whether the comparison is case-sensitive or not case-sensitive.

For further details on these and other comparison conditional predicates, refer to the [WHERE](#) clause.

See Also

- [SELECT](#) statement, [WHERE](#) clause
- [%CONTAINSTERM](#) predicate
- [%PATTERN](#)
- [%SIMILARITY](#) function
- [%SQLUPPER](#)
- [%INTERNAL](#)
- [Overview of Predicates](#)
- “[Queries Invoking Free-text Search](#)” in the “Querying the Database” chapter of *Using Caché SQL*

%CONTAINSTERM

Matches a value to one or more words using word-aware matching.

```
scalar-expression %CONTAINSTERM(word[ ,word... ])
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with one or more <i>word</i> strings.
<i>word</i>	An alphabetic string or comma-separated list of alphabetic strings to match with values in <i>scalar-expression</i> . A <i>word</i> must be a complete word, or a multiple-word phrase (with the words separated by spaces). The word or phrase should be delimited with single quotes. See below for restrictions on multiple-word phrases.

Description

The **%CONTAINSTERM** predicate allows you to select those data values that match the word or words specified in *word*. This comparison operation is word-aware; it is not a simple string match operation. If you specify more than one *word*, *scalar-expression* must contain all of the specified word strings. Word strings may be presented in any order. If *word* does not match any of the scalar expression data values, **%CONTAINSTERM** returns the null string.

%CONTAINSTERM is a collection predicate. It can only be used in the **WHERE** clause of a **SELECT** statement.

%CONTAINSTERM can be used on a %Text string or a character stream field.

To use **%CONTAINSTERM** on a string, change the %String property to %Text, and set LANGUAGECLASS and MAXLEN [property parameters](#). For example:

```
Property MySentences As %Text(LANGUAGECLASS = "%Text.English",MAXLEN = 1000);
```

Specifying a MAXLEN value (in bytes) is required for %Text properties.

To use **%CONTAINSTERM** to search a [character stream field](#), the stream field must be defined as type %Stream.GlobalCharacterSearchable. For example:

```
Property MyTextStream As %Stream.GlobalCharacterSearchable(LANGUAGECLASS = "%Text.English");
```

The **%CONTAINSTERM** predicate is one of the few predicates that can be used on a [stream field](#) in a **WHERE** clause.

The available languages are English, French, German, Italian, Japanese, Portuguese, and Spanish. See the %Text package class documentation (in %SYS) in the *InterSystems Class Reference* for further details.

The system generates an SQLCODE -309 error if *scalar-expression* is neither data type %Text nor %Stream.GlobalCharacterSearchable.

The system generates an SQLCODE -472 error if *scalar-expression* is not a collection-valued field (property).

%CONTAINS and %CONTAINSTERM

The **%CONTAINS** and **%CONTAINSTERM** predicates perform the same word-aware comparison for their supplied *word* arguments. They differ in their requirements for multiple-word phrases:

- A **%CONTAINSTERM** argument cannot contain noise words. A **%CONTAINS** argument can contain noise words.
- Every **%CONTAINSTERM** argument phrase must be NGRAMLEN words or less in length. A **%CONTAINS** argument phrase can be any number of words in length.

The **%CONTAINSTERM** operator gives superior performance for certain types of comparisons, especially very large search sets. **%CONTAINSTERM** is preferable when performing comparisons in Japanese.

%CONTAINSTERM comparisons use the [collation type](#) of the *scalar-expression*, and are generally not case-sensitive. For this reason, the CASEINSENSITIVE class property is not applicable to **%CONTAINSTERM** comparisons. Collation can optionally be set to %EXACT for case-sensitive operations.

Word-Aware Matching

Caché uses language analysis rules, including punctuation analysis and stemming rules, to match only the specified word. For example, the *word* argument “set” would match the word “set” or “Set” (not case-sensitive), and also stem forms such as “sets” and “setting”. However, it would *not* match words such as “setscrew,” “settle,” or “Seth,” even though these words contain the specified string.

Stemming rules provide for matching between any two forms of the word stem. Stemming is performed on both the search term(s) and the searched text. For example, you can specify %CONTAINSTERM(' jumping') and match the word “jumps” in the text.

This word-aware matching is fundamentally different from the character-by-character string matching performed by the SQL Contains operator (I).

For further details on contains comparison, refer to the [%CONTAINS](#) operator.

Collection Predicates

%CONTAINSTERM is a collection predicate. It can be used in most contexts where a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual. It is subject to the following restrictions:

- You cannot use **%CONTAINSTERM** in a HAVING clause.
- You cannot use **%CONTAINSTERM** as a predicate that selects fields for a JOIN operation.
- You cannot associate **%CONTAINSTERM** with another predicate condition using the OR logical operator if the two predicates reference fields in different tables. For example:

```
WHERE t1.text %CONTAINSTERM('Continental United States') OR t2.Timezone BETWEEN 5 AND 8
```

Because this restriction depends on how the optimizer uses indices, SQL may only enforce this restriction when indices are added to a table. It is strongly suggested that this type of logic be avoided in all queries.

iKnow and iFind

The Caché [iKnow text analysis tool](#) and [iFind text search tool](#) also provide word-aware analysis. These facilities are entirely separate from %Text classes. They provide a substantially different and significantly more sophisticated form of textual analysis.

Examples

The following Embedded SQL example performs a **%CONTAINSTERM** comparison with a literal phrase:

```
&sql(SELECT name,stats
      INTO :badname, :badstat
      FROM Sample.Employee
      WHERE status %CONTAINSTERM('invalid value'))
```

The following Embedded SQL example uses a host variable to perform a **%CONTAINSTERM** comparison:

```
SET text="invalid"
&sql(SELECT name,stats
      INTO :badname, :badstat
      FROM Sample.Employee
      WHERE status %CONTAINSTERM(:text))
```

See Also

- [SELECT](#) statement [WHERE](#) clause
- [%CONTAINS](#) predicate
- [%SIMILARITY](#) function
- [Overview of Predicates](#)
- [“Queries Invoking Free-text Search”](#) in the “Querying the Database” chapter of *Using Caché SQL*

EXISTS

Checks a table for the existence of at least one corresponding row.

```
EXISTS select-statement
```

Arguments

<i>select-statement</i>	A simple query , usually containing a condition expression .
-------------------------	--

Description

The **EXISTS** predicate tests a specified table, typically for existence of at least a row. Since the **SELECT** statement following the **EXISTS** is being checked for containing something, the clause is often of the form:

```
EXISTS (SELECT... FROM... WHERE...)
```

where a typical statement might be:

```
SELECT name
  FROM Table_A
 WHERE EXISTS
   (SELECT *
    FROM Table_B
    WHERE Table_B.Number = Table_A.Number)
```

In this example, the predicate tests for the existence of one or more rows specified by the subquery.

Note that the test must occur on a **SELECT** statement (not on a **UNION**).

The **NOT EXISTS** clause tests for the non-existence of a row in a table, as shown in the following example:

```
SELECT EmployeeName, Age
  FROM Employees
 WHERE NOT EXISTS (SELECT * FROM BonusTable
 WHERE NOT (BonusTable.Result = 'Positive'
 AND Employees.EmployeeNum = BonusTable.EmployeeNum))
```

EXISTS can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

Where applicable, the system automatically applies Set-Valued Subquery Optimization (SVSO) to an **EXISTS** or **NOT EXISTS** subquery. For details on this optimization, and using the %NOSVSO keyword to override it, refer to “Query Optimization Options” on the [FROM clause](#) reference page.

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [Overview of Predicates](#)

%FIND

Matches a value to a set of generated values with bitmap chunks iteration.

```
scalar-expression %FIND valueset [SIZE ((nn))]
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly the RowId field of a table) whose values are being compared with <i>valueset</i> .
<i>valueset</i>	An object reference (oref) to a user-defined object that implements bitmap chunks iteration methods and the ContainsItem() method. This method takes a set of data values and returns a boolean when there is a match with a value in <i>scalar-expression</i> .
SIZE ((nn))	<i>Optional</i> — An order-of-magnitude integer (10, 100, 1000, etc.) used for query optimization.

Description

The **%FIND** predicate allows you to filter a result set by selecting those data values that match the values specified in *valueset*, iterating through values in a sequence of bitmap chunks. This match is successful when a *scalar-expression* value matches a value in *valueset*. If the *valueset* values do not match any of the scalar expression values, **%FIND** returns the null string. This match is always performed on the logical (internal storage) data value, regardless of the display mode.

%FIND, like the other comparison conditions, is used in the **WHERE** clause or the **HAVING** clause of a **SELECT** statement.

%FIND enables filtering of field values using an abstract, programmatically specified set of matching values. Specifically, it enables filtering of RowId field values using an abstract, programmatically specified bitmap, where *valueset* behaves similar to the subscript layer of a bitmap index.

The user-defined class is derived from the abstract class %SQL.AbstractFind. this abstract class defines the **ContainsItem()** boolean method. The **ContainsItem()** method matches the *scalar-expression* values to the *valueset* values.

Iteration through values in a sequence of bitmap chunks is performed using the following three methods:

- **GetChunk(c)**, which returns the bitmap chunk with chunk number *c*.
- **NextChunk(.c)**, which returns the first bitmap chunk with chunk number $> c$.
- **PreviousChunk(.c)**, which returns the first bitmap chunk with chunk number $< c$.

Refer to %SQL.AbstractFind for further details concerning these four methods.

Collation Types

%FIND uses the same [collation type](#) as the column it is matched against. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#). If you assign a different collation type to the column, you must also apply this collation type to the **%FIND** *substring*. Refer to [%SQLUPPER](#) for further information on case transformation functions.

SIZE Clause

The optional **%FIND** SIZE clause provides the integer *nn*, which specifies an order-of-magnitude estimate of the number of values in *valueset*. Caché uses this order-of-magnitude estimate to determine the optimal query plan. Specify *nn* as one

of the following literals: 10, 100, 1000, 10000, etc. Because *nm* must be available as a constant value at compile time, it must be specified as a literal in all SQL code. Note that nesting parentheses must be specified as shown for all SQL, with the exception of Embedded SQL.

%FIND and %INSET Compared

- **%INSET** is the simplest and most general interface. It supports the **ContainsItem()** method.
- **%FIND** supports iteration over bitmap chunks using a [bitmap index](#). It emulates the functionality of the ObjectScript [\\$ORDER](#) function, supporting **NextChunk()**, **PreviousChunk()**, and **GetChunk()** iteration methods, as well as the **ContainsItem()** method.
-

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [%INSET](#) predicate
- [Overview of Predicates](#)
- [SEARCH_INDEX](#) function

FOR SOME

Determines whether to return a record based on a condition test of field values.

```
FOR SOME (table [AS t-alias]) (fieldcondition)
```

Arguments

<i>table</i>	<i>table</i> can be a single table or a comma-separated list of tables. The enclosing parentheses are mandatory.
AS <i>t-alias</i>	<i>Optional</i> — An alias for the preceding <i>table</i> name. An alias must be a valid identifier ; it can be a delimited identifier. For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> . The AS keyword is optional.
<i>fieldcondition</i>	<i>fieldcondition</i> specifies one or more condition expressions referencing one or more fields. The <i>fieldcondition</i> is enclosed with parentheses. You can specify multiple condition expressions within <i>fieldcondition</i> using AND (&) and OR (!) logical operators.

Description

The **FOR SOME** predicate allows you to determine whether or not to return a record based on a boolean condition test of the values of one or more fields in a table. If *fieldcondition* evaluates as true, the record is returned. If *fieldcondition* evaluates as false, the record is not returned.

FOR SOME can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

At Caché release 2015.1 and subsequent, delimiting parentheses are mandatory for the *table* (and its optional *t-alias*) argument. Delimiting parentheses are also mandatory for the *fieldcondition* argument. Whitespace is permitted, but not required, between these two sets of parentheses.

Commonly, **FOR SOME** is used to determine whether to return a record from a table based on the contents of a record in another table. **FOR SOME** can also be used to determine whether to return a record from a table based on the contents of a record in the same table. In this latter case, either all records are returned or no records are returned.

In the following example, **FOR SOME** returns all records in the Sample.Person table in which its Name field value matches the Name field value in the Sample.Employee table:

```
SELECT Name, COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e)(e.Name=p.Name)
ORDER BY Name
```

In the following example, **FOR SOME** returns records in the Sample.Person table based on a boolean test of the *same* table. This program returns all Sample.Person records if at least one record has an Age value greater than 65. Otherwise, it returns no records. Because at least one record in Sample.Person has an Age field value greater than 65, all Sample.Person records are returned:

```
SELECT Name, Age, COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Person)(Age>65)
ORDER BY Age
```

Like most predicates, **FOR SOME** can be inverted using the NOT logical operator, as shown in the following example:

```
SELECT Name, Age, COUNT(Name) AS NameCount
FROM Sample.Person
WHERE NOT FOR SOME (Sample.Person) (Age>65)
ORDER BY Age
```

Compound Conditions

A *fieldcondition* can contain more than one condition expression. The set of conditions is enclosed in parentheses. Multiple conditions are specified with the logical operators AND and OR, which can also be specified using the & and ! symbols. A logical operator may be followed by the NOT unary operator. By default, conditions are evaluated in left-to-right order. You can specify a different order of evaluation by grouping multiple conditions using parentheses.

```
SELECT Name, COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e) (e.Name=p.Name AND p.Name %STARTSWITH 'A')
ORDER BY Name
```

```
SELECT Name, COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e) (e.Name=p.Name OR p.Name %STARTSWITH 'A')
ORDER BY Name
```

In the following example, **FOR SOME** returns all records in the Sample.Person table in which its Name field value matches the Name field value in the Sample.Employee table, and their residence (Home_State) is in the same state as their office (Office_State):

```
SELECT Name, Home_State, COUNT(Name) AS NameCount
FROM Sample.Person AS p
WHERE FOR SOME (Sample.Employee AS e) (p.Name=e.Name AND p.Home_State=e.Office_State)
ORDER BY Name
```

Multiple Tables

You can specify multiple tables as a comma-separated list before the *fieldcondition*. The condition that determines whether to return records may reference the table from which data is being selected, or may reference field values in another table. Table aliases are usually required to associate each specified field with its table.

In the following example, all records are returned if there is at least one Name in the Sample.Person table that is also found in the Sample.Employee table. Because this condition is true for at least one record, all Sample.Person records are returned:

```
SELECT Name AS PersonName, Age, COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Employee AS e, Sample.Person AS p) (e.Name=p.Name)
ORDER BY Name
```

In the following example, all records are returned if there is at least one Name in the Sample.Person table that is also found in the Sample.Company table. Because names of persons and names of companies (in this data set) are never the same, this condition is not true for any record. Therefore, no Sample.Person records are returned:

```
SELECT Name AS PersonName, Age, COUNT(Name) AS NameCount
FROM Sample.Person
WHERE FOR SOME (Sample.Company AS c, Sample.Person AS p) (c.Name=p.Name)
ORDER BY Name
```

See Also

- [SELECT](#) statement, [HAVING](#) clause, [WHERE](#) clause
- [Overview of Predicates](#)
- [FOR SOME %ELEMENT](#) predicate

FOR SOME %ELEMENT

Matches list element values or the number of list elements with a predicate.

```
FOR SOME %ELEMENT(field) [[AS] e-alias] (predicate)
```

Arguments

<i>field</i>	A scalar expression (most commonly a data column) whose elements are being compared with <i>predicate</i> .
AS <i>e-alias</i>	<i>Optional</i> — An element alias used to qualify %KEY or %VALUE within the <i>predicate</i> . Commonly, this alias is used when the <i>predicate</i> contains a nested FOR SOME %ELEMENT condition. The alias must be a valid identifier . For further details see the “Identifiers” chapter of <i>Using Caché SQL</i> . The AS keyword is optional.
(<i>predicate</i>)	A predicate condition, enclosed in parentheses. Within this condition use %VALUE and/or %KEY to determine what the condition is matching. %VALUE matches the element value (%VALUE='Red'). %KEY matches the minimum number of elements (%KEY=2). Within this condition, %VALUE and %KEY may be optionally qualified if you have specified an <i>e-alias</i> . This predicate can consist of multiple condition expressions with AND and OR logical operators.

Description

The **FOR SOME %ELEMENT** predicate matches the list elements in *field* with the specified *predicate*. The **SOME** keyword specifies that at least one of the elements in the *field* must satisfy the specified *predicate* clause.

The *predicate* clause must contain either the %VALUE or the %KEY keyword, followed by a predicate condition. These keywords are not case-sensitive.

The use of %VALUE and %KEY is explained in the following examples:

- (%VALUE='Red') matches all *field* values that contain the value Red as one of their list elements. The *field* may only contain the single element Red, or it may contain multiple elements, one of which is the element Red.
- (%KEY=2) matches all *field* values that contain at least 2 elements. The *field* may contain exactly two elements or it may contain more than two elements. The %KEY value must be a positive integer. (%KEY=0) does not match any *field* values.

FOR SOME %ELEMENT cannot be used to match a *field* that is NULL.

The *predicate* clause can use any [predicate condition](#), not just the equality condition. The following are some examples of *predicate* clauses:

```
(%VALUE='Red')
(%VALUE > 21)
(%VALUE %STARTSWITH 'R')
(%VALUE [ 'e' )
(%VALUE IN ('Red','Blue'))
(%VALUE IS NOT NULL)
(%KEY=3)
(%KEY > 1)
(%KEY IS NOT NULL)
```

For performance reasons, the predicate %STARTSWITH 'abc' is preferable to the equivalent predicate LIKE 'abc%'.

You can specify multiple predicate conditions using AND, OR, and NOT [logical operators](#). Caché applies the combined predicate conditions to each element. Therefore, it is not meaningful to apply two %VALUE or two %KEY predicates using an AND test.

For example, using **FOR SOME %ELEMENT** to match a field containing the values Red, Green, Red Green, Black Red, Green Yellow Red, Green Black, Yellow, or Black Yellow:

- (`%VALUE='Red'`) matches any field containing the element Red: Red, Red Green, Black Red, and Red Yellow Green.
- (`%VALUE='Red' OR %VALUE='Green'`) matches any field containing either element (or both, in any order): Red, Green, Red Green, Black Red, Green Yellow Red, Green Black. This is functionally identical to (`%VALUE IN('Red','Green')`).
- (`%VALUE='Red' AND %VALUE='Green'`) matches no field values because it matches each element against both Red and Green, and no element can have the value Red and the value Green. This predicate does *not* match the two-element value Red Green.
- (`%VALUE='Red' AND %KEY=2`) matches Red Green, Black Red, Green Yellow Red.
- (`%VALUE='Red' OR %KEY=2`) matches Red, Red Green, Black Red, Green Yellow Red, Green Black, Black Yellow.

FOR SOME %ELEMENT is a collection predicate. It can be used in most contexts where a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual. It is subject to the following restrictions:

- You cannot use **FOR SOME %ELEMENT** in a HAVING clause.
- You cannot use **FOR SOME %ELEMENT** as a predicate that selects fields for a **JOIN** operation.
- You cannot associate **FOR SOME %ELEMENT** with another predicate condition using the OR logical operator if the two predicates reference fields in different tables. For example:

```
WHERE FOR SOME %ELEMENT(t1.FavoriteColors) (%VALUE='purple') OR t2.Age < 65
```

Because this restriction depends on how the optimizer uses indices, SQL may only enforce this restriction when indices are added to a table. It is strongly suggested that this type of logic be avoided in all queries.

Collection Index

An important use of **FOR SOME %ELEMENT** is to select elements using a collection index. If the appropriate KEYS or ELEMENTS index is defined for *field*, Caché uses this index rather than directly referencing the field value elements.

If the following collection index is defined:

```
INDEX fcIDX1 ON FavoriteColors(ELEMENTS);
```

The following query uses this index:

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red')
```

If the following collection index is defined:

```
INDEX fcIDX2 ON FavoriteColors(KEYS) [ Type = bitmap ];
```

The following query uses this index:

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%KEY=2)
```

For further details on **FOR SOME %ELEMENT** with collection indices, refer to [Collection Indexing and Querying Collections through SQL](#) in the “Querying the Database” chapter of *Using Caché SQL*.

Examples

The following example uses **FOR SOME %ELEMENT** to return those rows where the FavoriteColors list contains the element 'Red':

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE='Red')
```

In the following example, the %VALUE predicate contains an IN statement specifying a comma-separated list. This example returns those rows where the FavoriteColors list contains either the element 'Red' or the element 'Blue' (or both):

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) (%VALUE IN ('Red','Blue'))
```

The following example uses a predicate clause with two Contains operators ([]). It returns those rows where the FavoriteColors list has an element that contains a lowercase 'l' and a lowercase 'e' (the contains operator is case-sensitive). In this case, the elements 'Blue', 'Yellow', and 'Purple':

```
SELECT Name,FavoriteColors AS Preferences
FROM Sample.Person
WHERE FOR SOME %ELEMENT(FavoriteColors) AS fc (fc.%VALUE [ 'l' AND fc.%VALUE [ 'e' ])
```

This example also demonstrates how an element alias (*e-alias*) is used.

The following Dynamic SQL example uses %KEY to return rows based on the number of elements in FavoriteColors. The first **%Execute()** sets %KEY=1, returning all rows that have one or more FavoriteColors elements. The second **%Execute()** sets %KEY=2, returning all rows that have two or more FavoriteColors elements:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT %ID,Name,FavoriteColors FROM Sample.Person "
SET q2 = "WHERE FOR SOME %ELEMENT(FavoriteColors) (%KEY=?) "
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(1)
DO rset.%Display()
WRITE !,"End of data %KEY 1",!!
SET rset = tStatement.%Execute(2)
DO rset.%Display()
WRITE !,"End of data %KEY 2"
```

See Also

- [SELECT statement, WHERE clause](#)
- [Overview of Predicates](#)
- [FOR SOME predicate](#)

IN

Matches a value to items in an unstructured comma-separated list.

```
scalar-expression IN (item1,item2[,...])
scalar-expression IN (subquery)
```

Description

The **IN** predicate is used for matching a value to an unstructured series of items. Typically, it compares column data values to a comma-separated list of values. **IN** can perform [equality comparisons](#) and [subquery comparisons](#).

Like most predicates, **IN** can be inverted using the NOT logical operator. Neither **IN** nor **NOT IN** can be used to return NULL fields. To return NULL fields use [IS NULL](#).

IN can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

Equality Comparison

The **IN** predicate can serve as shorthand for the use of multiple equality comparisons linked together with the OR operator. For instance:

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State IN ('ME','NH','VT','MA','RI','CT')
```

evaluates true if Home_State equals any of the values in the comma-separated list. The listed items can be constants or expressions.

IN comparisons use the [collation type](#) defined for the *scalar-expression*, regardless of the collation type of the individual *items*. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

The following two examples show that collation matching is based on the *scalar-expression* collation. The Home_State field is defined with SQLUPPER (not case-sensitive) collation. Therefore, the following example returns NH Home_State values:

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State IN ('ME','nH','VT')
```

The following example does not return NH Home_State values:

```
SELECT Name, Home_State FROM Sample.Person
WHERE %EXACT(Home_State) IN ('ME','nH','VT')
```

It is not meaningful to include NULL in the list of values. NULL is the absence of a value, and therefore fails all equality tests. Specifying an **IN** predicate (or any other predicate) eliminates any instances of the specified field that are NULL. This is shown in the following *incorrect* (but executable) example:

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors IN ($LISTBUILD('Red'),$LISTBUILD('Blue'),NULL)
/* NULL here is meaningless. No FavoriteColor NULL fields returned */
```

The only way to include a field with NULL in the predicate result set is to specify the **IS NULL** predicate, as shown in the following example:

```
SELECT FavoriteColors FROM Sample.Person
WHERE FavoriteColors IN ($LISTBUILD('Red'),$LISTBUILD('Blue')) OR FavoriteColors IS NULL
```

When dates or times are used for **IN** predicate equality comparisons, the appropriate data type conversions are automatically performed. If the **WHERE** field is type **TimeStamp**, values of type **Date** or **Time** are converted to **TimeStamp**. If the **WHERE** field is type **Date**, values of type **TimeStamp** or **String** are converted to **Date**. If the **WHERE** field is type **Time**, values of type **TimeStamp** or **String** are converted to **Time**.

The following examples both perform the same equality comparisons and return the same data. The **DOB** field is of data type **Date**:

```
SELECT Name,DOB FROM Sample.Person
WHERE DOB IN ({d '1951-02-02'},{d '1987-02-28'})
```

```
SELECT Name,DOB FROM Sample.Person
WHERE DOB IN ({ts '1951-02-02 02:37:00'},{ts '1987-02-28 16:58:10'})
```

For further details refer to [Date and Time Constructs](#).

%SelectMode

If **%SelectMode** is set to a value other than **Logical** format, the **IN** predicate values must be specified in the **%SelectMode** format (**ODBC** or **Display**). This applies mainly to dates, times, and **Caché** format lists (**%List**). Specifying predicate values in **Logical** format commonly results in an **SQLCODE** error. For example, **SQLCODE -146** “Unable to convert date input to a valid logical date value”.

In the following **Dynamic SQL** example, the **IN** predicate must specify dates in **%SelectMode=1** (**ODBC**) format:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB IN('1956-03-05','1956-04-08','1956-04-18','1956-12-08')"
```

```
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus='1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Subquery Comparison

You can use the **IN** predicate with a subquery to test whether a column value (or any other expression) equals any of the subquery row values. For example:

```
SELECT Name,Home_State FROM Sample.Person
WHERE Name IN
  (SELECT Name FROM Sample.Employee
   HAVING Salary < 50000)
```

Note that the subquery must have exactly one select-item in the **SELECT** list.

The following example uses an **IN** subquery to return the **Employee** states that are not **Vendor** states:

```
SELECT Home_State
FROM Sample.Employee
WHERE Home_State NOT IN (SELECT Address_State FROM Sample.Vendor)
GROUP BY Home_State
```

The following example matches a collation function expression to an **IN** predicate with a subquery:

```
SELECT Name,Id FROM Sample.Person
WHERE %EXACT(Spouse) NOT IN
  (SELECT Id FROM Sample.Person
   WHERE Age < 65)
```

An **IN** cannot specify both a subquery and a comma-separated list of literal values.

Literal Substitution Override

You can override literal substitution during compile pre-parsing by enclosing each **IN** predicate argument with parentheses. For example, `WHERE Home_State IN (('ME'), ('NH'), ('VT'), ('MA'), ('RI'), ('CT'))`. This may improve query performance by improving overall selectivity and/or subscript bounding selectivity. However, it should be avoided when the same query is called multiple times with different values, as it will result in the creation of a separate cached query for each query call. For further details, refer to [Literal Substitution](#) in the “Cached Queries” chapter of the *Caché SQL Optimization Guide*.

IN and %INLIST

Both the **IN** and **%INLIST** predicates can be used to supply multiple values to use for OR equality comparisons. The **%INLIST** predicate is used for matching a value to the elements of a %List structure. In Dynamic SQL you can supply the **%INLIST** predicate values as a single host variable. You must supply the **IN** predicate values as individual host variables. Therefore, changing the number of **IN** predicate values results in the creation of a separate cached query. **%INLIST** takes a single predicate value, a %List with multiple elements; changing the number of %List elements does not result in the creation of a separate cached query. **%INLIST** also provides an order-of-magnitude **SIZE** argument that SQL uses to optimize performance. For these reasons it is often advantageous to use `%INLIST($LISTFROMSTRING(val))` rather than `IN(val1, val2, val3, .. valn)`.

%INLIST can perform equality comparisons; it cannot perform subquery comparisons.

For further details, refer to [%INLIST](#).

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [%INLIST](#) predicate
- [Overview of Predicates](#)

%INLIST

Matches a value to the elements in a %List structured list.

```
scalar-expression %INLIST list [SIZE ((nn))]
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with <i>list</i> elements.
<i>list</i>	A %List structure containing one or more elements.
SIZE ((<i>nn</i>))	<i>Optional</i> — An integer specifying an order-of-magnitude estimate of the number of elements in <i>list</i> . Must be specified as a literal with one of the following values: 10, 100, 1000, 10000, and so forth.

Description

The **%INLIST** predicate is a Caché extension for matching the values of a field with the elements of a list structure. Both **%INLIST** and **IN** allow you to perform such equality comparisons with multiple specified values. **%INLIST** specifies these multiple values as the elements of a single *list* argument. Therefore, **%INLIST** allows you to vary the number of values to match without creating a separate cached query.

The optional **%INLIST** SIZE clause provides the integer *nn*, which specifies an order-of-magnitude estimate of the number of list elements in *list*. Caché uses this order-of-magnitude estimate to determine the optimal query plan. Because the same cached query is used regardless of the number of elements in *list*, specifying SIZE allows you to create a cached query optimized for the anticipated approximate number of elements in *list*. Changing the SIZE literal creates a separate cached query. Specify *nn* as one of the following literals: 10, 100, 1000, 10000, etc. Because *nn* must be available as a constant value at compile time, it must be specified as a literal in all SQL code. Note that double parentheses must be specified as shown for all compiled SQL (Dynamic SQL). Double parentheses are not used with Embedded SQL. For further details, refer to the “[Cached Queries](#)” chapter in *Caché SQL Optimization Guide*.

%INLIST performs an equality comparison with each of the elements of *list*. **%INLIST** comparisons use the [collation type](#) defined for the *scalar-expression*. Therefore, comparisons of *list* elements may be case-sensitive or not case-sensitive, depending on the collation of *scalar-expression*. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “[Collation](#)” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

It is not meaningful to specify NULL as a comparison value. NULL is the absence of a value, and therefore fails all equality tests. Specifying an **%INLIST** predicate (or any other predicate) eliminates any instances of the specified field that are NULL. You must specify the **IS NULL** predicate to include fields with NULL in the predicate result set.

Like most predicates, **%INLIST** can be inverted using the NOT logical operator. Neither **%INLIST** nor **NOT %INLIST** can be used to return NULL fields. To return NULL fields use **IS NULL**.

If the match expression is not in %List format, **%INLIST** generates an SQLCODE -400 error. For example, if the class is a MultiValue class (specified with %MV.Adaptor), or the SqlListType of the collection property is DELIMITED, the logical value of the list field is not in %List format. For further details on list structures, see the SQL [\\$LIST](#) function.

%INLIST can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

For matching a value to an unstructured series of items, such as a comma-separated list of values, use the **IN** predicate. **IN** can perform equality comparisons and subquery comparisons.

%SelectMode

The **%INLIST** predicate does not use the current **%SelectMode** setting. The elements of *list* should be specified in Logical format, regardless of the **%SelectMode** setting. Attempting to specify *list* elements in ODBC format or Display format commonly results in no data matches or unintended data matches.

You can use the **%EXTERNAL** or **%ODBCOUT** format-transform functions to transform the *scalar-expression* field that the predicate operates upon. This allows you to specify the *list* elements in Display format or ODBC format. However, using a format-transform function prevents the use of the index for the field, and can thus have a significant performance impact.

In the following Dynamic SQL example, the **%INLIST** predicate specifies a list containing date value elements for the year 1978 in Logical format, not in **%SelectMode=1** (ODBC) format. Dates that correspond to these \$HOROLOG format dates are selected:

```
ZNSPACE "SAMPLES"
SET bday=$LISTBUILD(50039)
FOR i=50039:1:50403 {SET bday=bday_$LISTBUILD(i) }
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %INLIST ?"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(bday)
DO rset.%Display()
```

The following Dynamic SQL example uses the **%ODBCOUT** format-transform function to transform the DOB field matched by the predicate. This allows you to specify the **%INLIST** list elements in ODBC format. However, specifying the format-transform function prevents the use of an index for DOB field values:

```
ZNSPACE "SAMPLES"
SET births=$LISTBUILD("1978-01-15","1978-08-22","1978-10-01")
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %INLIST ?"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(births)
DO rset.%Display()
```

%INLIST and IN

Both the **%INLIST** and **IN** predicates can be used to supply multiple values to use for equality comparisons. The following Dynamic SQL examples return the same results:

```
ZNSPACE "SAMPLES"
SET states=$LISTBUILD("VT","NH","ME")
SET myquery = "SELECT Name,Home_State FROM Sample.Person WHERE Home_State %INLIST ?"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(states)
DO rset.%Display()
```

```
ZNSPACE "SAMPLES"
SET s1="VT"
SET s2="NH"
SET s3="ME"
SET myquery = "SELECT Name,Home_State FROM Sample.Person WHERE Home_State IN(?,?,?)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(s1,s2,s3)
DO rset.%Display()
```

However, in Dynamic SQL you can supply the **%INLIST** predicate values as a single host variable; you must supply the **IN** predicate values as individual host variables. Therefore, changing the number of **IN** predicate values results in the creation

of a separate cached query. Changing the number of **%INLIST** predicate values does not result in the creation of a separate cached query. For further details, refer to the “[Cached Queries](#)” chapter in *Caché SQL Optimization Guide*.

Examples

The following example matches Home_State column values to the elements of a structured list of northern New England states:

```
ZNSPACE "SAMPLES"
SET states=$LISTBUILD("VT","NH","ME")
SET myquery="SELECT Name,Home_State FROM Sample.Person "_
    "WHERE Home_State %INLIST ?"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(states)
DO rset.%Display()
```

The following two examples show that collation matching is based on the *scalar-expression* collation. The Home_State field is defined with SQLUPPER collation which is not case-sensitive. The *list* in these examples specifies New Hampshire as “nH”, rather than “NH”. The first example returns NH Home_State values, the second example does not return NH Home_State values:

```
ZNSPACE "SAMPLES"
SET states=$LISTBUILD("VT","nH","ME")
SET myquery="SELECT Name,Home_State FROM Sample.Person "_
    "WHERE Home_State %INLIST ?"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(states)
DO rset.%Display()
```

```
ZNSPACE "SAMPLES"
SET states=$LISTBUILD("VT","nH","ME")
SET myquery="SELECT Name,Home_State FROM Sample.Person "_
    "WHERE %EXACT(Home_State) %INLIST ?"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(states)
DO rset.%Display()
```

The following example creates a cached query with a SIZE literal of 10. Specifying SIZE 10 is optimal for this query, because 10 corresponds in order-of-magnitude to the actual number of elements in the list. Changing the number of elements in the list does not create a separate cached query. Changing the SIZE literal does create a separate cached query:

```
ZNSPACE "SAMPLES"
SET states=$LISTBUILD("VT","NH","ME")
SET myquery="SELECT Name,Home_State FROM Sample.Person "_
    "WHERE Home_State %INLIST ? SIZE ((10))"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(states)
DO rset.%Display()
```

```
ZNSPACE "SAMPLES"
SET states=$LISTBUILD("VT","nH","ME")
SET myquery="SELECT Name,Home_State FROM Sample.Person "_
    "WHERE Home_State %INLIST ?"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(states)
DO rset.%Display()
```

The following example creates a cached query with a SIZE literal of 10. Specifying SIZE 10 is optimal for this query, because 10 corresponds in order-of-magnitude to the actual number of elements in the list. Changing the number of elements in the list does not create a separate cached query. Changing the SIZE literal does create a separate cached query:

```
ZNSPACE "SAMPLES"
SET states=$LISTBUILD("VT","NH","ME")
SET myquery="SELECT Name,Home_State FROM Sample.Person "_
           "WHERE Home_State %INLIST ? SIZE ((10))"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
           IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(states)
DO rset.%Display()
```

See Also

- [SELECT](#) statement, [HAVING](#) clause, [WHERE](#) clause
- [\\$LISTBUILD](#) function
- [IN](#) predicate
- [Overview of Predicates](#)

%INSET

Matches a value to a set of generated values.

```
scalar-expression %INSET valueset [SIZE ((nn))]
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly the RowId field of a table) whose values are being compared with <i>valueset</i> .
<i>valueset</i>	An object reference (oref) to a user-defined object that implements a ContainsItem() method. This method takes a set of data values and returns a boolean when there is a match with a value in <i>scalar-expression</i> .
SIZE ((<i>nn</i>))	<i>Optional</i> — An order-of-magnitude integer (10, 100, 1000, etc.) used for query optimization.

Description

The **%INSET** predicate allows you to filter a result set by selecting those data values that match the values specified in *valueset*. This match is successful when a *scalar-expression* value matches a value in *valueset*. If the *valueset* values do not match any of the scalar expression values, **%INSET** returns the null string. This match is always performed on the logical (internal storage) data value, regardless of the display mode.

%INSET, like the other comparison conditions, is used in the **WHERE** clause or the **HAVING** clause of a **SELECT** statement.

%INSET enables filtering of field values using an abstract, programmatically specified set of matching values. Specifically, it enables filtering of RowId field values using an abstract, programmatically specified temp-file or bitmap index, where *valueset* behaves similar to the lowest subscript layer of a bitmap index or a regular index.

The user-defined class is derived from the abstract class %SQL.AbstractFind. this abstract class defines the **ContainsItem()** method, which is the only method supported by **%INSET**. The **ContainsItem()** method returns the *valueset*. Refer to %SQL.AbstractFind for further details.

Collation Types

%INSET uses the same [collation type](#) as the column it is matched against. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#). If you assign a different collation type to the column, you must also apply this collation type to the **%INSET** *substring*. Refer to [%SQLUPPER](#) for further information on case transformation functions.

SIZE Clause

The optional **%INSET** SIZE clause provides the integer *nn*, which specifies an order-of-magnitude estimate of the number of values in *valueset*. Caché uses this order-of-magnitude estimate to determine the optimal query plan. Specify *nn* as one of the following literals: 10, 100, 1000, 10000, etc. Because *nn* must be available as a constant value at compile time, it must be specified as a literal in all SQL code. Note that nesting parentheses must be specified as shown for all SQL, with the exception of Embedded SQL.

%INSET and %FIND Compared

- **%INSET** is the simplest and most general interface. It supports the **ContainsItem()** method.

- **%FIND** supports iteration over bitmap chunks using a [bitmap index](#). It emulates the functionality of the ObjectScript [\\$ORDER](#) function, supporting **NextChunk()**, **PreviousChunk()**, and **GetChunk()** iteration methods, as well as the **ContainsItem()** method.

-

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [%FIND](#) predicate
- [Overview of Predicates](#)

IS JSON

Determines if a data value is in JSON format.

```
scalar-expression IS [NOT] JSON [keyword]
```

Arguments

<i>scalar-expression</i>	A scalar expression that is being checked for JSON formatting.
<i>keyword</i>	<i>Optional</i> — One of the following: VALUE, SCALAR, ARRAY, or OBJECT. The default is VALUE.

Description

The **IS JSON** predicate determines if a data value is in JSON format. The following example determines if the predicate is a properly-formatted JSON string, either a JSON object or a JSON array:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT TOP 5 Name FROM Sample.Person "
SET q2 = "WHERE '{"name":"'Fred"', "spouse":"'Wilma"'}' IS JSON"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

IS JSON (with or without the optional VALUE keyword) returns true for any JSON array or JSON object. This includes an empty JSON array ' [] ' or an empty JSON object ' {} '.

The VALUE keyword and the SCALAR keyword are synonyms.

IS JSON ARRAY returns true for a JSON array ref. **IS JSON OBJECT** returns true for a JSON object ref. This is shown in the following examples:

```
ZNSPACE "SAMPLES"
SET jarray=[1,2,3,5,8,13,21,34]
WRITE "JSON array: ", jarray,!
SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON ARRAY"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(jarray)
DO rset.%Display()
```

```
ZNSPACE "SAMPLES"
SET jarray=[1,2,3,5,8,13,21,34]
WRITE "JSON array: ", jarray,!
SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON OBJECT"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(jarray)
DO rset.%Display()
```

```
ZNSPACE "SAMPLES"
SET jobj={"name":"Fred","spouse":"Wilma"}
WRITE "JSON object: ", jobj,!
SET myquery = "SELECT TOP 5 Name FROM Sample.Person WHERE ? IS JSON OBJECT"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(jobj)
DO rset.%Display()
```

For further details, refer to the ObjectScript **SET** command subsection [“JSON Object and JSON Array”](#).

The **IS NOT JSON** predicate is one of the few predicates that can be used on a [stream field](#) in a **WHERE** clause. Its behavior is the same as **IS NOT NULL**. This is shown in the following example:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Title,%OBJECT(Picture) AS PhotoOref FROM Sample.Employee "
SET q2 = "WHERE Picture IS NOT JSON"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

IS JSON can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

See Also

- [SELECT](#) statement, [HAVING](#) clause, [WHERE](#) clause
- [JSON_ARRAY](#), [JSON_OBJECT](#) functions
- [JSON_ARRAYAGG](#) aggregate function
- [Overview of Predicates](#)

IS NULL

Determines if a data value is NULL.

```
scalar-expression IS [NOT] NULL
```

Description

The **IS NULL** predicate detects undefined values. You can detect all null values, or all non-null values:

```
SELECT Name, FavoriteColors FROM Sample.Person  
WHERE FavoriteColors IS NULL
```

```
SELECT Name, FavoriteColors FROM Sample.Person  
WHERE FavoriteColors IS NOT NULL
```

The **IS NULL** / **IS NOT NULL** predicate is one of the few predicates that can be used on a [stream field](#) in a **WHERE** clause. This is shown in the following example:

```
SELECT Title,%OBJECT(Picture) AS PhotoOref FROM Sample.Employee  
WHERE Picture IS NOT NULL
```

IS NULL can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

The **IS NULL** predicate should not be confused with the SQL [ISNULL](#) function.

See Also

- [SELECT](#) statement, [HAVING](#) clause, [WHERE](#) clause
- [Overview of Predicates](#)

LIKE

Matches a value with a pattern string containing literals and wildcards.

```
scalar-expression LIKE pattern [ESCAPE char]
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with <i>pattern</i> .
<i>pattern</i>	A quoted string representing the pattern of characters to match with each value in <i>scalar-expression</i> . The <i>pattern</i> string can contain literal characters, and the underscore (_) and percent (%) wildcard characters.
ESCAPE <i>char</i>	<i>Optional</i> — A string containing a single character. This <i>char</i> character can be used in <i>pattern</i> to specify that the character immediately following it is to be treated as a literal.

Description

The **LIKE** predicate allows you to select those data values that match the character or characters specified in *pattern*. The *pattern* may contain wildcard characters. If *pattern* does not match any of the scalar expression values, **LIKE** returns the null string.

LIKE can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

The **LIKE** predicate supports the following wildcards:

Table C–1: LIKE Wildcard Characters

Character	Matches
_	Any single character.
%	Any sequence of 0 or more characters. (In accordance with the SQL standard, NULL is <i>not</i> considered a sequence of 0 characters, and is thus not selected by this wildcard.)

In Dynamic SQL or Embedded SQL, a *pattern* can represent wildcard characters and input parameters or input host variables as concatenated strings, as shown in the [Examples](#) section.

Collation Types

The *pattern* string uses the same [collation type](#) as the column it is matching against. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

If **LIKE** is applied against a field with the SQLUPPER default collation type, the **LIKE** clause returns matches that ignore letter case. You can use the SQLSTRING collation type to perform a **LIKE** string comparison that is case-sensitive.

The following example returns all names that contain the substring “Ro”. Because **LIKE** is not case-sensitive, **LIKE** `'%Ro%'` returns Robert, Rogers, deRocca, LaRonga, Brown, Mastroni, and so forth:

```
SELECT Name FROM Sample.Person
WHERE Name LIKE '%Ro%'
```

Compare this to the [Contains operator](#) (`()`), which uses EXACT (case-sensitive) collation:

```
SELECT Name FROM Sample.Person
WHERE Name [ 'Ro'
```

By using the `%SQLSTRING` collation type, you can use **LIKE** to return only those names that contain the case-sensitive substring “Ro”. It would not return Mastroni or Brown:

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING(Name) LIKE '%Ro%'
```

In the above example, the leading space that `%SQLSTRING` appended to Name values was handled by the `%` wildcard. A more robust example would specify the collation type on both sides of the predicate:

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING(Name) LIKE %SQLSTRING('%Ro%')
```

Do not use the `%ALPHAUP` or `%STRING` functions with *pattern*, because these functions remove the punctuation marks that *pattern* uses as wildcard characters.

Refer to `%SQLUPPER` for further information on case transformation functions.

All Values, Empty String Values, and NULL

If the *pattern* value is percent (`%`), **LIKE** selects all values for the specified field, including empty string values:

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE '%'
```

It does not select fields that are NULL.

Specifying a *pattern* value of empty string returns empty string values.

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE ''
```

Specifying a *pattern* value of NULL is not a meaningful operation. It completes successfully, but returns no values.

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE NULL
```

Like most predicates, **LIKE** can be inverted using the NOT logical operator. Neither **LIKE** nor **NOT LIKE** can be used to return NULL fields. To return NULL fields use [IS NULL](#).

ESCAPE Clause

ESCAPE permits the use of a wildcard character as a literal character within *pattern*. ESCAPE *char*, if provided and if it is a single character, indicates that any character directly following it in *pattern* is to be understood as a literal character, rather than a wildcard or formatting character. The following example shows the use of ESCAPE to return values that contain the string `'_SYS'`:

```
SELECT * FROM MyTable
WHERE symbol_field LIKE '%\_SYS%' ESCAPE '\'
```

%SelectMode

The **LIKE** predicate does not use the current `%SelectMode` setting. A *pattern* should be specified in Logical format, regardless of the `%SelectMode` setting. Attempting to specify a *pattern* in ODBC format or Display format commonly results in no data matches or unintended data matches.

You can use the `%EXTERNAL` or `%ODBCOUT` format-transform functions to transform the *scalar-expression* field that the predicate operates upon. This allows you to specify the *pattern* in Display format or ODBC format. However, using a format-transform function prevents the use of the index for the field, and can thus have a significant performance impact.

In the following Dynamic SQL example, the **LIKE** predicate specifies the date *pattern* in Logical format, not in %Select-Mode=1 (ODBC) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 (\$HOROLOG 41000) through December 28 1955 (\$HOROLOG 41999)) are selected:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB LIKE '41%"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example uses the %ODBCOUT format-transform function to transform the DOB field matched by the predicate. This allows you to specify the **LIKE** *pattern* in ODBC format. It selects rows with DOB field ODBC values beginning with 195 (dates within the range of years 1950 through 1959). However, specifying the format-transform function prevents the use of an index for DOB field values:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) LIKE '195%"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Literal Substitution Override

You can override literal substitution during compile pre-parsing by enclosing the **LIKE** predicate argument with double parentheses. For example, WHERE Name LIKE (('Mc%')) or WHERE Name LIKE (('son%')). This may improve query performance by improving overall selectivity and/or subscript bounding selectivity. However, it should be avoided when the same query is called multiple times with different values, as it will result in the creation of a separate cached query for each query call.

Examples

The following example uses the **WHERE** clause to select Name values that contain “son”, including those that begin or end with “son”. By default, **LIKE** string comparisons are not case-sensitive:

```
SELECT %ID,Name FROM Sample.Person
WHERE Name LIKE '%son%'
```

The following Embedded SQL example returns the same result set as the previous example. Note how the input host variable (:subname) is specified in the **LIKE** *pattern* using the concatenation operator:

```
ZNSPACE "SAMPLES"
SET subname="son"
&sql(DECLARE C1 CURSOR FOR SELECT %ID,Name INTO :id,:nameout FROM Sample.Person
WHERE Name LIKE '%'_:subname_')
&sql(OPEN C1)
QUIT:(SQLCODE'=0)
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
WRITE id," ",nameout,!
&sql(FETCH C1) }
&sql(CLOSE C1)
```

The following Dynamic SQL example returns the same result set as the previous example. Note how the input parameter (?) is specified in the **LIKE** *pattern* using the concatenation operator:

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT %ID,Name FROM Sample.Person WHERE Name LIKE '%'_?_'%"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("son")
DO rset.%Display()
```

The following example uses the **WHERE** clause to select FavoriteColors values that contain “blue”. The FavoriteColors field is a %List field; the % wildcards handle the %List formatting characters:

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors LIKE '%blue%'
```

The following example uses a **HAVING** clause to select records for people whose age starts with a 1 followed by a single character. It displays the average for all ages and the average for the ages selected by the **HAVING** clause. It orders the results by age. All returned values have ages from 10 through 19.

```
SELECT Name,
       Age,
       AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS AvgTeen
FROM Sample.Person
HAVING Age LIKE '1_'
ORDER BY Age
```

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [%MATCHES](#) predicate
- [%PATTERN](#) predicate
- [Overview of Predicates](#)

%MATCHES

Matches a value with a pattern string containing literals, wildcards, and ranges.

```
scalar-expression %MATCHES pattern [ESCAPE char]
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with <i>pattern</i> .
<i>pattern</i>	A quoted string representing the pattern of characters to match with each value in <i>scalar-expression</i> . The <i>pattern</i> string can contain literal characters, the question mark (?) and asterisk (*) wildcard characters, square brackets used to specify allowed values, and the backslash (\) used to specify that the character immediately following it is to be treated as a literal. The <i>pattern</i> can also be the empty string or NULL, though it does not match or return NULL items.
ESCAPE <i>char</i>	<i>Optional</i> — A string containing a single character. This <i>char</i> character can be used in <i>pattern</i> to specify that the character immediately following it is to be treated as a literal. If not specified, the default escape character is backslash (\).

Description

The **%MATCHES** predicate is a Caché extension for matching a value to a pattern string. **%MATCHES** returns True or False for the match operation. The *pattern* string can consist of literal characters, wild card characters, and list or ranges of matching literals.

Pattern matches are case-sensitive. Pattern matching is based on the EXACT value of *scalar-expression*, not its collation value. Therefore, a **%MATCHES** operation is always case-sensitive, even when the [collation type](#) of *scalar-expression* is not case-sensitive.

%MATCHES supports the following *pattern* wildcards:

?	Matches any single character of any type.
*	Matches zero or more characters of any type.
[abc]	Matches any one of the characters specified in brackets.
[a-z]	Matches character within the range specified in brackets, inclusive of the specified characters.
[^A-Z] [^a-z] [^0-9]	These ranges match any characters <i>except</i> those specified in brackets. You can use this syntax to specify no uppercase letters, or no lowercase letters, or no numbers. Only the specified literal ranges shown are supported.
\	Treats the character following as a literal character, rather than as a wildcard. Backslash is the default escape character; you can specify another character as the escape character using the optional ESCAPE clause.

Like most predicates, **%MATCHES** can be inverted using the NOT operator: `item NOT %MATCHES pattern`. Neither **%MATCHES** nor **NOT %MATCHES** can be used to return NULL fields. To return NULL fields use [IS NULL](#).

The backslash (\) character is the default escape character. It can be used to specify that a wildcard character is to be used as a literal match at the specified pattern location. For example, to match a question mark as the first character of a string specify '\?*. To match a question mark as the fourth character of a string specify '???\?*. To match a question mark anywhere in a string specify '*\?*. To match a string that consists of only an asterisk character specify '*. To match a string that contains at least one asterisk character specify '***'. To match a backslash character anywhere in a string specify '**'.

%MATCHES can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

%MATCHES is supported for compatibility with Informix SQL.

%SelectMode

The **%MATCHES** predicate does not use the current **%SelectMode** setting. A *pattern* should be specified in Logical format, regardless of the **%SelectMode** setting. Attempting to specify a *pattern* in ODBC format or Display format commonly results in no data matches or unintended data matches.

You can use the **%EXTERNAL** or **%ODBCOUT** format-transform functions to transform the *scalar-expression* field that the predicate operates upon. This allows you to specify the *pattern* in Display format or ODBC format. However, using a format-transform function prevents the use of the index for the field, and can thus have a significant performance impact.

In the following Dynamic SQL example, the **%MATCHES** predicate specifies the date *pattern* in Logical format, not in **%SelectMode=1** (ODBC) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 (\$HOROLOG 41000) through December 28 1955 (\$HOROLOG 41999)) are selected:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %MATCHES '41*'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example uses the **%ODBCOUT** format-transform function to transform the DOB field matched by the predicate. This allows you to specify the **%MATCHES** *pattern* in ODBC format. It selects rows with DOB field ODBC values beginning with 195 (dates within the range of years 1950 through 1959). However, specifying the format-transform function prevents the use of an index for DOB field values:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %MATCHES '195*'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Examples

The following example returns all last names that begin with “A”:

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES 'A*'
```

The following example returns all first names that begin with “A”:

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '*,A*'
```

The following example returns all names that contain the letter “A” (in last name, first name, or middle initial):

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '*A*'
```

The following example returns all names that *do not* contain the letters “A”, “a”, “E” or “e”:

```
SELECT Name FROM Sample.Person
WHERE Name NOT %MATCHES '*[AaEe]*'
```

The following example returns all five-letter last names with first names that begin with “A” through “D”:

```
SELECT Name FROM Sample.Person
WHERE Name %MATCHES '?????,[A-D]*'
```

See Also

- [SELECT](#) statement, [HAVING](#) clause, [WHERE](#) clause
- [LIKE](#) predicate
- [%PATTERN](#) predicate
- [Overview of Predicates](#)

%PATTERN

Matches a value with a pattern string containing literals, wildcards, and character type codes.

```
scalar-expression %PATTERN pattern
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with <i>pattern</i> .
<i>pattern</i>	A quoted string representing the pattern of characters to match with each value in <i>scalar-expression</i> . The <i>pattern</i> string can contain literal characters enclosed in double quotes, letter codes that specify types of characters, and numbers and the period (.) character as wildcard characters.

Description

The **%PATTERN** predicate allows you to match a pattern of character type codes and literals to the data values supplied by *scalar-expression*. If *pattern* matches a complete scalar expression value, this value is returned. If *pattern* does not fully match any of the scalar expression values, **%PATTERN** returns the null string.

%PATTERN can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

%PATTERN uses the same pattern codes as the ObjectScript pattern match operator (the ? operator). A pattern consists of one or more pairs of a repetition count followed by a value. A repetition count can be an integer, a period (.) meaning “any number of characters”, or a range specified by using a combination of a period with integers. A value can be either a character type code letter or a literal string (specified in quotes).

Note that a pattern often consists of multiple repetition/value pairs, because the pattern must exactly match the entire data value. For this reason, many patterns end with the “.E” pair, which means that the rest of the data value can consist of any number of characters of any type.

A few simple examples of pattern match pairs:

- 1L means one (and only one) lowercase letter.
- 1"L" means one literal character “L”.
- 1"617" means one literal string “617”.
- .U means any number of uppercase letters.
- .E means any number of printable characters of any type.
- .3A means any number up to three (three or less) letters (either uppercase or lowercase).
- 3.N means three or more numeric digits.
- 3.6N means three to six (inclusive) numeric digits.

Pattern matches are case-sensitive. Pattern matching is based on the EXACT value of *scalar-expression*, not its collation value. Therefore, a literal letter specified in a **%PATTERN** operation is always matched case-sensitive, even when the [collation type](#) of *scalar-expression* is not case-sensitive.

In Dynamic SQL the SQL query is specified as an ObjectScript string, delimited by double quotes. For this reason, double quotes within a *pattern* string must be doubled. Thus the pattern for a US dollar amount: '1"\$1.N1"."2N' would be specified in Dynamic SQL as '1"\$1.N1"."2N'.

For further details on pattern codes, refer to [Pattern Matching](#) in the [Operators and Expressions](#) chapter of *Using Caché ObjectScript*.

%SelectMode

The **%PATTERN** predicate does not use the current **%SelectMode** setting. A *pattern* should be specified in Logical format, regardless of the **%SelectMode** setting. Attempting to specify a *pattern* in ODBC format or Display format commonly results in no data matches or unintended data matches.

You can use the **%EXTERNAL** or **%ODBCOUT** format-transform functions to transform the *scalar-expression* field that the predicate operates upon. This allows you to specify the *pattern* in Display format or ODBC format. However, using a format-transform function prevents the use of the index for the field, and can thus have a significant performance impact.

In the following Dynamic SQL example, the **%PATTERN** predicate specifies the date *pattern* in Logical format, not in **%SelectMode=1** (ODBC) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 (\$HOROLOG 41000) through December 28 1955 (\$HOROLOG 41999)) are selected:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %PATTERN '1"41"3N' "
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example uses the **%ODBCOUT** format-transform function to transform the DOB field matched by the predicate. This allows you to specify the **%PATTERN** *pattern* in ODBC format. It selects rows with DOB field ODBC values beginning with 195 (dates within the range of years 1950 through 1959). However, specifying the format-transform function prevents the use of an index for DOB field values:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE %ODBCOUT(DOB) %PATTERN '1"195"0.E' "
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Examples

The following example uses a **%PATTERN** operator in the **WHERE** clause to select Home_State values in which the first character is any uppercase letter and the second character is the letter "C":

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State %PATTERN 'lU1"C'
```

This example selects records with a Home_State of North Carolina (NC) or South Carolina (SC).

The following example uses a **%PATTERN** operator in the **WHERE** clause to select Name values that start with an uppercase letter followed by a lowercase letter.

```
SELECT Name FROM Sample.Person
WHERE Name %PATTERN 'lU1.L'E'
```

The pattern here translates as: 1U (one uppercase letter), followed 1L (one lowercase letter), followed by .E (any number of characters of any type). Note that this pattern would exclude names such as "JONES", O'Reilly" and "deGastyne".

The following example uses a **%PATTERN** operator in a **HAVING** clause to select records for people whose first name starts with the letters "Jo", and to return the count of records searched and records returned.

```
SELECT Name ,
       COUNT(Name) AS TotRecs ,
       COUNT(Name %AFTERHAVING) AS JoRecs
FROM Sample.Person
HAVING Name %PATTERN '1U.L1" , "1"Jo".E'
```

In this case, the Name field values are formatted as Lastname,Firstname and may contain an optional middle name or initial. To reflect this name format, the pattern here translates as: 1U (one uppercase letter), followed .L (any number of lowercase letters), followed by 1", " (one literal comma character), followed by 1"Jo" (one literal string with the value "Jo"), followed by .E (any number of characters of any type).

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [LIKE](#) predicate
- [%MATCHES](#) predicate
- [Overview of Predicates](#)

SOME

Matches a value with at least one matching value from a subquery.

```
scalar-expression comparison-operator SOME (subquery)
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with the result set generated by <i>subquery</i> .
<i>comparison-operator</i>	One of the following comparison operators: = (equal to), <> or != (not equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), [(contains), or] (follows).
<i>subquery</i>	A subquery, enclosed in parentheses, which returns a result set that is used for the comparison with <i>scalar-expression</i> .

Description

The **SOME** keyword works in conjunction with a comparison operator to create a [predicate](#) (a quantified comparison condition) that is true if the value of a scalar expression matches one or more of the corresponding values retrieved by the [subquery](#). The **SOME** predicate compares a single *scalar-expression* item with a single subquery **SELECT** item. A subquery with more than one select item generates an SQLCODE -10 error.

Note: The **SOME** and **ANY** keywords are synonyms.

SOME can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

Example

The following example selects those employees with salaries greater than \$75,000 that live in any of the states west of the Mississippi River:

```
SELECT Name,Salary,Home_State FROM Sample.Employee
WHERE Salary > 75000
AND Home_State = SOME
  (SELECT State FROM Sample.USZipCode
   WHERE Longitude < -93)
ORDER BY Home_State
```

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause
- [ALL ANY](#)
- [Overview of Predicates](#)

%STARTSWITH

Matches a value with a substring specifying initial characters.

```
scalar-expression %STARTSWITH substring
```

Arguments

<i>scalar-expression</i>	A scalar expression (most commonly a data column) whose values are being compared with <i>substring</i> .
<i>substring</i>	An expression that resolves to a string or a numeric containing the first character or characters to match with values in <i>scalar-expression</i> .

Description

The **%STARTSWITH** predicate allows you to select those data values that begin with the character or characters specified in *substring*. If *substring* does not match any of the scalar expression values, **%STARTSWITH** returns the null string. This match is always performed on the logical (internal storage) data value, regardless of the display mode.

%STARTSWITH can be used wherever a [predicate condition](#) can be specified, as described in the [Overview of Predicates](#) page of this manual.

The following example selects all names that begin with “M”:

```
SELECT Name FROM Sample.MyTest WHERE Name %STARTSWITH 'M'
```

You can use NOT to invert the sense of a predicate. The following example selects all names except those that begin with “M”:

```
SELECT Name FROM Sample.MyTest WHERE NOT Name %STARTSWITH 'M'
```

Collation Types

%STARTSWITH uses the same [collation type](#) as the field it is matched against. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

Note: **%STARTSWITH** may give unexpected results when applied to a field defined with collation type EXACT, UPPER, or ALPHAUP. For details, refer to [Legacy Collation Types](#) in *Using Caché SQL*.

In the following example, UpName is defined as SQLUPPER; the *substring* match is case-insensitive:

```
SELECT UpName FROM Sample.MyTest WHERE UpName %STARTSWITH 'mo'
```

If you assign a different collation type to the column in the WHERE clause, this collation type is matched to the literal value of the **%STARTSWITH** *substring*.

In the following example, UpName is defined as SQLUPPER; but the *substring* match is EXACT (case-sensitive):

```
SELECT UpName FROM Sample.MyTest WHERE %EXACT(UpName) %STARTSWITH 'mo'
```

Some collation functions append a space character to a field value. This can cause **%STARTSWITH** to match no values, unless you apply an equivalent collation function to the *substring*.

In the following example, ExactName is defined as EXACT; because the query applies %SQLUPPER to the *scalar-expression* the comparison now involves a string starting with an appended space character. This comparison would return no fields:

```
SELECT ExactName FROM Sample.MyTest WHERE %SQLUPPER(ExactName) %STARTSWITH 'Ra'
```

Therefore, you must append a space character to the *substring* as well. The following example applies a non-case-sensitive match to an EXACT field:

```
SELECT ExactName FROM Sample.MyTest WHERE %SQLUPPER(ExactName) %STARTSWITH %SQLUPPER('Ra')
```

A **%STARTSWITH** string comparison that is not case-sensitive and ignores blank spaces and punctuation marks (except commas):

```
SELECT Name FROM Sample.Person
WHERE %STRING(Name) %STARTSWITH %STRING(' od ')
```

Using %STRING, this example can select both O'Donnell and Odem.

Refer to [collation types](#) for further information on case transformation functions.

%SelectMode

The **%STARTSWITH** predicate cannot use the current **%SelectMode** setting. A *substring* must be specified in Logical format, regardless of the **%SelectMode** setting. Specifying predicate value(s) in ODBC or Display format commonly results in no data matches or unintended data matches. This applies mainly to dates, times, and Caché format lists (%List).

In the following Dynamic SQL example, the **%STARTSWITH** predicate must specify the date *substring* in Logical format, not in **%SelectMode=1** (ODBC) format. Rows with DOB Logical values beginning with 41 (dates from April 4 1953 (\$HOROLOG 41000) through December 28 1955 (\$HOROLOG 41999)) are selected:

```
ZNSPACE "SAMPLES"
SET q1 = "SELECT Name,DOB FROM Sample.Person "
SET q2 = "WHERE DOB %STARTSWITH '41%'"
SET myquery = q1_q2
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

List Fields

If *scalar-expression* is a list field, **%STARTSWITH** can use **%EXTERNAL** to compare the list values to *substring*. For example, to determine all records in which the FavoriteColors list field begins with 'B!':

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'B!'
```

When **%EXTERNAL** converts a list to DISPLAY format, the displayed list items appear to be separated by a blank space. This “space” is actually the two non-display characters CHAR(13) and CHAR(10). To use **%STARTSWITH** with more than one element in the list, you must specify these characters:

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Orange' || CHAR(13) || CHAR(10) || 'B'
```

Filtering Out NULLs

- If the *scalar-expression* is any non-null data value and the *substring* is an “empty” value, **%STARTSWITH** always returns the *scalar-expression*.
- If the *scalar-expression* is null and the *substring* is an “empty” value, **%STARTSWITH** does not return the *scalar-expression*.

An “empty” *substring* value can be any of the following: NULL, CHAR(0), the empty string (''), a string consisting of only blank spaces (' '), CHAR(32) the space character, and CHAR(9) the tab character. By default, **%STARTSWITH** uses all of these values for filtering out nulls.

To return *scalar-expression* values that consist of only whitespace characters, you must use **%EXACT** collation.

In all of the following examples, **%STARTSWITH** returns the same results. It restricts the result set to non-null FavoriteColors values:

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH NULL
```

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH ''
```

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH ' '
```

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE FavoriteColors %STARTSWITH CHAR(9)
```

Note that the **%EXTERNAL** collation type is not used for *scalar-expression* when filtering nulls from a list field.

%STARTSWITH NULL and empty string behavior differs with a compound *substring*, because of the definitions of NULL and empty string. When you concatenate a value with NULL, the result is NULL. When you concatenate a value with the empty string, the result is the value. This is shown in the following examples:

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'B' || NULL
/* Selects all non-null rows */
```

```
SELECT Name,FavoriteColors
FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'B' || ''
/* Selects all values that begin with B */
```

Leading and Trailing Blanks

In most cases, **%STARTSWITH** treats leading blanks the same as any other character. For example, **%STARTSWITH ' B'** can be used to select field values with exactly one leading blank followed by the letter B. However, a *substring* containing only blanks does not select for leading blanks; it selects for non-null values.

%STARTSWITH behavior with trailing blanks depends on the data type and collation type. **%STARTSWITH** ignores trailing blanks in a string *substring* defined as SQLUPPER. **%STARTSWITH** does not ignore trailing blanks in a numeric, date, or list *substring*.

In the following example, **%STARTSWITH** restricts the result set to names that begin with 'M'. Because Name is an SQLUPPER string data type, the trailing blanks in the *substring* are ignored:

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH 'M '
```

In the following example, **%STARTSWITH** eliminates all rows from the result set because the trailing blanks in the *substring* are not ignored for a numeric value:

```
SELECT Name,Age FROM Sample.Person
WHERE Age %STARTSWITH '6 '
```

In the following example, **%STARTSWITH** eliminates all rows from the result set because the trailing blank in the *substring* is not ignored for a list value:

```
SELECT Name,FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Blue '
```

However, in the following example, the result set consists of those list values that start with Blue followed by a list delimiter (which is displayed as a blank space); in other words, lists beginning with 'Blue' that contain more than one item:

```
SELECT Name, FavoriteColors FROM Sample.Person
WHERE %EXTERNAL(FavoriteColors) %STARTSWITH 'Blue' || CHAR(13) || CHAR(10)
```

Range of Subscripts

When *scalar-expression* is retrieved from a subscript, **%STARTSWITH** can be used as an index-limiting range condition, narrowing the range of *scalar-expression* subscript values that needs to be traversed. The logic is to start the subscript range with the given *substring* prefix value, and stop as soon as the subscript value no longer starts with *substring*.

National Collation Ambiguous Characters

In some national languages two characters or character combinations are considered first-pass collation equivalent. Commonly this is a character with or without an accent mark, such as in the Czech2 locale, in which CHAR(65) and CHAR(193) both collate as "A". **%STARTSWITH** recognizes these characters as equivalent.

The following example shows the first-pass collation for Czech2 CHAR(65) (A) and CHAR(193) (Á):

```
M
MA
MÁ
MAC
MÁC
MACX
MÁCX
MAD
MÁD
MB
```

It is important to note that you cannot know at query compile time which national collation would be used at run time. Therefore, **%STARTSWITH** subscript traversal code has to be written so that it will correctly satisfy any likely runtime situation.

Other Equivalence Comparisons

%STARTSWITH performs an equivalence comparison on the initial character(s) of a string. You can perform other types of equivalence comparisons by using string comparison operators. These include the following:

- An equivalence comparison on the entire string, using the equal sign operator:

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State = 'VT'
```

This example selects any record that contains the Home_State field value "VT". Because Home_State is defined as SQLUPPER, this string comparison is not case-sensitive.

- An non-equivalence comparison on the entire string, using the does not equal operator:

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State <> 'MA'
ORDER BY Home_State
```

This example selects all records that where the Home_State field value *is not* equal to "MA".

- An equivalence comparison on the entire string to multiple values, using the IN keyword operator:

```
SELECT Name, Home_State FROM Sample.Person
WHERE Home_State IN ('VT', 'MA', 'NH', 'ME')
ORDER BY Home_State
```

This example selects any record that contains any of the specified Home_State field values.

- An equivalence comparison on the entire string to a value pattern, using the **%PATTERN** keyword operator:

```
SELECT Name,Home_State FROM Sample.Person
WHERE Home_State %PATTERN '1U1"C"'
ORDER BY Home_State
```

This example selects any record that contains a Home_State field value that matches the pattern of 1U (one uppercase letter) followed by 1"C" (one literal letter "C"). This pattern would be fulfilled by the Home_State abbreviations "NC" or "SC".

- An equivalence comparison of a substring to a value, using the Contains operator:

```
SELECT Name FROM Sample.Person
WHERE Name [ 'y'
```

This example selects all Name records that contain the lowercase letter "y". By default, a Contains operator comparison is case-sensitive, even when the field is defined as not case-sensitive.

- A word-aware equivalence comparison of one or more substrings to a value, using the **%CONTAINS** or **%CONTAINSTERM** comparison operators. These operators can only be used on strings redefined with the %Text property.
- An equivalence comparison of a substring with one or more wildcards to a value, using the LIKE keyword operator:

```
SELECT Name FROM Sample.Person
WHERE Name LIKE '_a%'
```

This example selects all Name records that contain the letter "a" as the second letter. This string comparison uses the Name collation type to determine whether the comparison is case-sensitive or not case-sensitive.

For further details on these and other comparison conditional predicates, refer to the [WHERE](#) clause.

Examples

The following example uses the **WHERE** clause to select Name values that start with the letter "R" or "r". By default, **%STARTSWITH** string comparisons are not case-sensitive:

```
SELECT Name FROM Sample.Person
WHERE Name %STARTSWITH 'r'
```

The following example returns one record for each distinct Home_State name that begins with "M":

```
SELECT DISTINCT Home_State FROM Sample.Person
WHERE Home_State %STARTSWITH 'M'
ORDER BY Home_State
```

The following example uses a **HAVING** clause to select records for people whose age starts with a 2, displays the average for all ages and the average for the ages selected by the **HAVING** clause. It orders the results by age:

```
SELECT Name,
       Age,
       AVG(Age) AS AvgAge,
       AVG(Age %AFTERHAVING) AS Avg20
FROM Sample.Person
HAVING Age %STARTSWITH 2
ORDER BY Age
```

The following example performs a **%STARTSWITH** comparison with the internal date format value for the DOB (date of birth) field. In this case, it select all dates from 11/5/1988 (**\$H=54000**) through 08/1/1991 (**\$H=54999**):

```
SELECT Name,DOB
FROM Sample.Person
WHERE DOB %STARTSWITH 54
ORDER BY DOB
```

See Also

- [SELECT](#) statement [HAVING](#) clause [WHERE](#) clause

- [Overview of Predicates](#)
- “[Collation](#)” chapter in *Using Caché SQL*

SQL Aggregate Functions

Overview of Aggregate Functions

Functions that evaluate all of the values of a column and return a single aggregate value.

Supported Aggregate Functions

An aggregate function performs a task in relation to one or more values from a single column and returns a single value. The supported functions are:

- **SUM** — returns the sum of the values of a specified column.
- **AVG** — returns the average of the values of the specified column.
- **COUNT** — returns the number of rows in a table, or the number of non-null values in a specified column.
- **MAX** — returns the maximum value used within a specified column.
- **MIN** — returns the minimum value used within a specified column.
- **VARIANCE, VAR_SAMP, VAR_POP** — returns the statistical variance of the values of a specified column.
- **STDDEV, STDDEV_SAMP, STDDEV_POP** — returns the statistical standard deviation of the values of a specified column.
- **LIST** — returns all of the values used within a specified column as a comma-separated list.
- **%DLIST** — returns all of the values used within a specified column as elements in a Caché list structure.
- **XMLAGG** — returns all of the values used within a specified column as a concatenated string.
- **JSON_ARRAYAGG** — returns all of the values used within a specified column as a JSON format array.

Aggregate functions ignore fields that are NULL. For example, **LIST** and **%DLIST** do not include elements for rows in which the specified field is NULL. **COUNT** only counts non-null values of the specified field.

All aggregate functions support the optional **DISTINCT** keyword clause. This keyword limits the aggregate operation to only distinct (unique) field values. The default is to perform the aggregate operation on all non-NULL values, including duplicate values. The **MIN** and **MAX** aggregate functions support the **DISTINCT** keyword, although it perform no operation.

Aggregate functions support the full **DISTINCT** keyword clause syntax, including the optional `BY(item-list)` subclause. Refer to the [DISTINCT clause](#) for details.

The aggregate function `DISTINCT field1` clause ignores `field1` values that are NULL. This differs from the **DISTINCT** clause of the **SELECT** statement: a `SELECT DISTINCT` clause returns one row for the distinct NULL, just as it returns one row for each distinct field value. However, an aggregate function `DISTINCT BY(field2) field1` does not ignore the distinct NULL for `field2`. For example, if `FavoriteColors` has 50 distinct values and multiple NULLs, the number of **DISTINCT** rows returned is 51, the `COUNT(DISTINCT FavoriteColors)` is 50, and the `COUNT(DISTINCT BY(FavoriteColors) %ID)` is 51:

```
SELECT DISTINCT FavoriteColors,
       COUNT(DISTINCT FavoriteColors),
       COUNT(DISTINCT BY(FavoriteColors) %ID)
FROM Sample.Person
```

Aggregate functions (with the exception of **COUNT**) cannot be applied to a [stream field](#). Attempting to do so generates an SQLCODE -37 error. You can use **COUNT** to count stream field values, with some restrictions.

Using Aggregate Functions

An aggregate function can be used in:

- **SELECT list**, either as a listed *select-item* or in a subquery *select-item*.

- [HAVING clause](#).

An aggregate function can be specified in a `DISTINCT BY` clause without error, though this usage is not meaningful and always returns a single row.

An aggregate function *cannot* be used directly in:

- an `ORDER BY` clause. Attempting to do so generates an `SQLCODE -73` error. However, you can specify a [column alias](#) for an aggregate function in an `ORDER BY` clause.
- a `WHERE` clause. Attempting to do so generates an `SQLCODE -19` error.
- a `GROUP BY` clause. Attempting to do so generates an `SQLCODE -19` error.
- a `TOP` clause. Attempting to do so generates an `SQLCODE -1` error.
- a `JOIN`. Attempting to specify an aggregate in an `ON` clause generates an `SQLCODE -19` error. Attempting to specify an aggregate in a `USING` clause generates an `SQLCODE -1` error.

However, you can supply an aggregate function value to a `WHERE` or `HAVING` clause by using a subquery. For example, to use a `WHERE` clause to select `Age` values that are less than the average `Age` value, you can place the `AVG` aggregate function in a subquery:

```
SELECT Name, Age, AvgAge
FROM (SELECT Name, Age, AVG(Age) AS AvgAge FROM Sample.Person)
WHERE Age < AvgAge
ORDER BY Age
```

Combining Aggregates and Fields

Caché SQL allows you to specify an aggregate function with other `SELECT` items in a query. An aggregate such as `COUNT(*)` does not need to be in a separate query.

```
SELECT TOP 5 COUNT(*), Name, AVG(Age)
FROM Sample.Person
ORDER BY Name
```

When you specify an aggregate function and specify no field select items in the select list, Caché SQL returns one row. A `TOP` clause is ignored, unless it is `TOP 0` (return no rows):

```
SELECT TOP 7 AVG(Age), LIST(Age)
FROM Sample.Person
WHERE Age > 75
```

When you specify an aggregate function and specify one or more field select items in the select list, Caché SQL returns as many rows as required for the field item:

```
SELECT DISTINCT Age, AVG(Age), LIST(Age)
FROM Sample.Person
WHERE Age > 75
```

Column Names and Aliases

By default, the column name assigned to the results of an aggregate function is `Aggregate_n`, where the *n* number suffix is the column order number, as specified in the `SELECT` list. Thus, the following example creates column names `Aggregate_2` and `Aggregate_5`:

```
SELECT TOP 5 Home_State, COUNT(*), Name, Age, AVG(Age)
FROM Sample.Person
ORDER BY Name
```

To specify another column name (a column alias), use the `AS` keyword:

```
SELECT COUNT(*) AS PersonCount
FROM Sample.Person, Sample.Employee
```

You can use a column alias to specify an aggregate field in an **ORDER BY** clause. The following example lists people in the order that their ages diverge from the average age:

```
SELECT Name, Age,
       AVG(Age) AS AvgAge,
       ABS(Age - AVG(Age)) AS RelAge
FROM Sample.Person
ORDER BY RelAge
```

For further details on [column aliases](#), refer to the **SELECT** statement.

With **ORDER BY**

The **LIST**, **%DLIST**, **XMLAGG**, and **JSON_ARRAYAGG** functions combine the values of a table column from multiple rows into a single aggregate value. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within these aggregates. Under certain circumstances, the results of these aggregates may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

With **DISTINCT** and **GROUP BY**

A **SELECT DISTINCT** with a *select-item* aggregate function and a **GROUP BY** clause returns the same results as if the **DISTINCT** keyword were not present. To achieve the desired results, put the aggregate function in a subquery.

For example, you wish to return the number of distinct counts of persons in states (there are states with 4 people, there are states with 6 people, etc.). You would expect to achieve this result as follows:

```
SELECT DISTINCT COUNT(*) AS PersonCounts
FROM Sample.Person
GROUP BY Home_State
```

Instead, you get a person count for each state, the same as if the **DISTINCT** keyword were not present:

```
SELECT COUNT(*) AS PersonCounts
FROM Sample.Person
GROUP BY Home_State
```

To achieve your intended result, you need to use a subquery, as follows:

```
SELECT DISTINCT *
FROM (SELECT COUNT(*) AS PersonCounts FROM Sample.Person
      GROUP BY Home_State)
```

Row Counts

In a query that contains one or more aggregate functions the **%ROWCOUNT** value depends on the query:

- If the select-list does not contain any references to fields in the **FROM** clause table(s), other than fields supplied to aggregate functions, the query returns **%ROWCOUNT 1** when the query selects no rows. **COUNT** returns 0, other aggregate functions return **NULL**:

```
ZNSPACE "SAMPLES"
SET q1="SELECT COUNT(*) AS NumRows,COUNT(FavoriteColors) AS NumColors,"_
      "AVG(Age) AS AvgAge,MAX(Age) AS MaxAge FROM Sample.Person"
SET q2="SELECT COUNT(*) AS NumRows,COUNT(FavoriteColors) AS NumColors,"_
      "AVG(Age) AS AvgAge,MAX(Age) AS MaxAge FROM Sample.Person WHERE Name='ZZZ'"
SET tStatement = ##class(%SQL.Statement).%New()
QueryReturnsData
SET qStatus = tStatement.%Prepare(q1)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount with data:",rset.%ROWCOUNT,!
QueryReturnsNoData
SET qStatus = tStatement.%Prepare(q2)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount without data:",rset.%ROWCOUNT
```

The select-item list can include other items, so long as they do not reference fields in the FROM clause table(s):

```
ZNSPACE "SAMPLES"
SET q1="SELECT COUNT(*) AS NumRows,COUNT(Name) AS NumNames,"_
      "(SELECT Name FROM Sample.Company) AS SubQ,$LENGTH('this string'),%CLASSNAME "_
      "FROM Sample.Person WHERE Name='ZZZ'"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(q1)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount:",rset.%ROWCOUNT
```

However, if this type of query selects no rows and has a **GROUP BY** clause, it returns %ROWCOUNT 0.

- A query containing TOP 0 always returns %ROWCOUNT 0. Aggregate functions are not evaluated:

```
ZNSPACE "SAMPLES"
SET q1="SELECT TOP 0 COUNT(*) AS NumRows,COUNT(Name) AS NumNames,"_
      "AVG(Age) AS AvgAge FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(q1)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount:",rset.%ROWCOUNT
```

- A query containing aggregate functions and fields returns the %ROWCOUNT of rows returned. If the result set contains no rows, the query returns %ROWCOUNT 0, **COUNT** returns NULL:

```
ZNSPACE "SAMPLES"
SET q1="SELECT TOP 4 COUNT(*) AS NumRows,COUNT(Name) AS NumNames,"_
      "Name,AVG(Age) AS AvgAge FROM Sample.Person"
SET q2="SELECT TOP 4 COUNT(*) AS NumRows,COUNT(Name) AS NumNames,"_
      "Name,AVG(Age) AS AvgAge FROM Sample.Person WHERE Name='ZZZ'"
SET tStatement = ##class(%SQL.Statement).%New()
QueryReturnsData
SET qStatus = tStatement.%Prepare(q1)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount with data:",rset.%ROWCOUNT,!!
QueryReturnsNoData
SET qStatus = tStatement.%Prepare(q2)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount without data:",rset.%ROWCOUNT
```

Aggregates, Transactions, and Locking

Including an aggregate function in a query causes the query to return the current state of the data to all result set fields, including uncommitted changes to the data. Thus, an ISOLATION LEVEL READ COMMITTED setting is ignored for a query containing an aggregate function. The current state of uncommitted data is as follows:

- **INSERT** and **UPDATE**: the aggregate calculation *does* include the modified values, even though these modifications are not yet committed and may be rolled back.
- **DELETE** and **TRUNCATE TABLE**: the aggregate calculation *does not* include deleted rows, even though these deletions are not yet committed and may be rolled back.

Because aggregate functions usually involve data from a large number of rows, it is not acceptable to issue a transaction lock on all of the rows involved in an aggregate calculation. It is therefore possible that another user may be performing a transaction that modifies the data while an aggregate calculation is in process.

See Also

- [AVG](#), [COUNT](#), [%DLIST](#), [JSON_ARRAYAGG](#), [LIST](#), [MAX](#), [MIN](#), [STDDEV](#), [STDDEV_SAMP](#), [STDDEV_POP](#), [SUM](#), [VARIANCE](#), [VAR_SAMP](#), [VAR_POP](#), [XMLAGG](#) aggregate functions

- [SELECT](#) statement

AVG

An aggregate function that returns the average of the values of the specified column.

```
AVG([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Specifies that AVG return the average of all values for <i>expression</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that AVG calculate the average on only the unique instances of a value. DISTINCT can specify a <code>BY(col-list)</code> subclause, where <i>col-list</i> can be a single field or a comma-separated list of fields.
<i>expression</i>	Any valid expression. Usually the name of a column that contains the data values to be averaged.
<code>%FOREACH(col-list)</code>	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on <code>%FOREACH</code> .
<code>%AFTERHAVING</code>	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

The **AVG** [aggregate function](#) returns the average of the values of *expression*. Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query.

AVG can be used in a **SELECT** query or subquery that references either a table or a view. **AVG** can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

AVG cannot be used in a **WHERE** clause. **AVG** cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

AVG, like all aggregate functions, can take an optional **DISTINCT** clause. `AVG(DISTINCT col1)` averages only those `col1` field values that are distinct (unique). `AVG(DISTINCT BY(col2) col1)` averages only those `col1` field values in records where the `col2` values are distinct (unique). Note however that the distinct `col2` values may include a single `NULL` as a distinct value.

Data Values

AVG returns either `NUMERIC` data type values or `DOUBLE` data type values. If *expression* is data type `DOUBLE`, **AVG** returns `DOUBLE`; otherwise, it returns `NUMERIC`.

For non-`DOUBLE` *expression* values, **AVG** returns a double-precision floating point number. The precision of the value returned by **AVG** is 18. The scale of the returned value depends upon the precision and scale of *expression*: the scale of the value returned by **AVG** is equal to 18 minus the *expression* precision, plus the *expression* scale ($as=ap-ep+es$).

For `DOUBLE` *expression* values, the scale is 0.

AVG is normally applied to a field or expression that has a numeric value, such as a number field or a date field. By default, aggregate functions use Logical (internal) data values, rather than Display values. Because no type checking is performed, it is possible (though rarely meaningful) to invoke it for nonnumeric fields; **AVG** evaluates nonnumeric values, including the empty string ("), as zero (0). If *expression* is data type `VARCHAR`, the return value to ODBC or JDBC is of data type `DOUBLE`.

`NULL` values in data fields are ignored when deriving an **AVG** aggregate function value. If no rows are returned by the query, or the data field value for all rows returned is `NULL`, **AVG** returns `NULL`.

Averaging a Single Value

If all of the *expression* values supplied to **AVG** are the same, the resulting average depends on the number of accessed rows in the table (the divisor). For example, if all of the rows in the table have the same value for a specific column, the average value of that column is a calculated value, which may differ slightly from the value in the individual columns. To avoid this discrepancy, you can use the **DISTINCT** keyword.

The following example shows how a slight inequality can result from the calculation of an average. The first query does not reference table rows, so **AVG** calculates by dividing by 1. The second query references table rows, so **AVG** calculates by dividing by the number of rows in the table. The third query references table rows, but averages the **DISTINCT** values of a single value; in this case **AVG** calculates by dividing by 1.

```
SET pi=$ZPI
&sql(SELECT :pi,AVG(:pi) INTO :p,:av FROM Sample.Person)
WRITE p," the value of pi",!
WRITE av," avg of pi/1",!
&sql(SELECT Name,:pi,AVG(:pi) INTO :n,:p,:av FROM Sample.Person)
WRITE av," avg calculated using numRows",!
&sql(SELECT Name,:pi,AVG(DISTINCT :pi) INTO :n,:p,:av FROM Sample.Person)
WRITE av," avg of pi/1"
```

Optimization

SQL optimization of an **AVG** calculation can use a [bitslice index](#), if this index is defined for the field.

Changes Made During the Current Transaction

Like all aggregate functions, **AVG** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to [SET TRANSACTION](#) and [START TRANSACTION](#).

Examples

The following query lists the average salary for all employees in the `Sample.Employee` database. Because all rows returned by the query would have identical values for this average, this query only returns a single row, consisting of the average salary. For display purposes, this query concatenates a dollar sign to the value (using the `||` operator), and uses the `AS` clause to label the column:

```
SELECT '$' || AVG(Salary) AS AverageSalary
FROM Sample.Employee
```

The following query lists each state with the average salary for the employees in that state:

```
SELECT Home_State,'$' || AVG(Salary) AS AverageSalary
FROM Sample.Employee
GROUP BY Home_State
```

The following query lists the name and salary for those employees whose salary is greater than the average salary. It also lists the average salary for all employees; this value is the same for all rows returned by the query:

```
SELECT Name,Salary,
'$' || AVG(Salary) AS AverageAllSalary
FROM Sample.Employee
HAVING Salary>AVG(Salary)
ORDER BY Salary
```

The following query lists the name and salary for those employees whose salary is greater than the average salary. It also lists the average salary for those employees with above-average salaries; this value is the same for all rows returned by the query:

```
SELECT Name,Salary,
'$' || AVG(Salary %AFTERHAVING) AS AverageHighSalary
FROM Sample.Employee
HAVING Salary>AVG(Salary)
ORDER BY Salary
```

The following query lists those states containing more than three employees with the average salary of that state's employees, and the average salary of that state's employees earning more than \$20,000:

```
SELECT Home_State,
       '$' || AVG(Salary) AS AvgStateSalary,
       '$' || AVG(Salary %AFTERHAVING) AS AvgLargerSalaries
FROM Sample.Employee
GROUP BY Home_State
HAVING COUNT(*) > 3 AND Salary > 20000
ORDER BY Home_State
```

The following query uses several forms of the DISTINCT clause. The AVG(DISTINCT BY col-list examples may include an additional Age value in the average, because the BY clause can include a single NULL as a distinct value, if Home_City contains one or more NULLs:

```
SELECT AVG(Age) AS AveAge,AVG(ALL Age) AS Synonym,
       AVG(DISTINCT Age) AS AveDistAge,
       AVG(DISTINCT BY(Home_City) Age) AS AvgAgeDistCity,
       AVG(DISTINCT BY(Home_City,Home_State) Age) AS AvgAgeDistCityState
FROM Sample.Person
```

The following query uses both the %FOREACH and the %AFTERHAVING keywords. It returns a row for those states containing people whose names start with “A”, “M”, or “W” (HAVING clause and GROUP BY clause). Each state row contains the following values:

- LIST(Age %FOREACH(Home_State)): a list of the ages of all of the people in the state.
- AVG(Age %FOREACH(Home_State)): the average age of all of the people in the state.
- AVG(Age %AFTERHAVING): the average age of all of the people in the database that meet the HAVING clause criteria. (This number is the same for all rows.)
- LIST(Age %FOREACH(Home_State) %AFTERHAVING): a list of the ages of all of the people in the state that meet the HAVING clause criteria.
- AVG(Age %FOREACH(Home_State) %AFTERHAVING): the average age of all of the people in the state that meet the HAVING clause criteria.

```
SELECT Home_State,
       LIST(Age %FOREACH(Home_State)) AS StateAgeList,
       AVG(Age %FOREACH(Home_State)) AS StateAgeAvg,
       AVG(Age %AFTERHAVING ) AS AgeAvgHaving,
       LIST(Age %FOREACH(Home_State)%AFTERHAVING ) AS StateAgeListHaving,
       AVG(Age %FOREACH(Home_State)%AFTERHAVING ) AS StateAgeAvgHaving
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'A%' OR Name LIKE 'M%' OR Name LIKE 'W%'
ORDER BY Home_State
```

See Also

- [Aggregate Functions](#) overview
- [COUNT](#) aggregate function
- [SUM](#) aggregate function

COUNT

An aggregate function that returns the number of rows in a table or a specified column.

```
COUNT ( * )
```

```
COUNT([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

*	Specifies that all rows should be counted to return the total number of rows in the specified table. COUNT(*) takes no other arguments and cannot be used with the ALL or DISTINCT keywords. COUNT(*) does not take an <i>expression</i> argument, and does not use information about any particular column. COUNT(*) returns the number of rows in a specified table or view without eliminating duplicates. It counts each row separately, including rows that contain NULL values.
ALL	<i>Optional</i> — Specifies that COUNT return the count of all values for <i>expression</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that COUNT return the count of the distinct (unique) values for <i>expression</i> . Cannot be used with a stream field. DISTINCT can specify a BY(<i>col-list</i>) subclause, where <i>col-list</i> can be a single column name or a comma-separated list of column names.
<i>expression</i>	Any valid expression. Usually the name of a column that contains the data values to be counted.
%FOREACH(<i>col-list</i>)	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on %FOREACH. The <i>col-list</i> cannot contain a stream field.
%AFTERHAVING	<i>Optional</i> — Applies the condition found in the HAVING clause.

COUNT returns the INTEGER [data type](#).

Description

The **COUNT** [aggregate function](#) has two forms:

- **COUNT(*expression*)** returns the count of the number of values in *expression* as an integer. Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query. **COUNT(*expression*)** does not count NULL values. It can optionally count or not count duplicate field values. **COUNT** returns data type INTEGER with xDBC length 4, precision 18, and scale 0.
- **COUNT(*)** returns the count of the number of rows in the table as an integer. **COUNT(*)** counts all rows, regardless of the presence of duplicate field values or NULL values.

COUNT can be used in a **SELECT** query or subquery that references either a table or a view. **COUNT** can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

COUNT cannot be used in a **WHERE** clause. **COUNT** cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

COUNT(*expression*) like all aggregate functions, can take an optional **DISTINCT clause**. The DISTINCT clause counts only those columns having distinct (unique) values. COUNT DISTINCT does not count NULL as a distinct value. COUNT(DISTINCT BY(*col2*) *col1*) counts *col1* values for distinct *col2* values; however, the distinct *col2* values may include a single NULL as a distinct value.

The ALL keyword counts all non-NULL values, including all duplicates. ALL is the default behavior if no keyword is specified.

No Rows Returned

If no rows are selected, **COUNT** either returns 0 or NULL, depending on the query:

- **COUNT** returns 0 if the select-list does not contain any references to fields in the FROM clause table(s), other than fields supplied to aggregate functions. Only the **COUNT** aggregate function returns 0; other aggregate functions return NULL. The query returns a %ROWCOUNT of 1. This is shown in the following example:

```
SET myquery = 3
SET myquery(1) = "SELECT COUNT(*) AS Recs,COUNT(Name) AS People,"
SET myquery(2) = "AVG(Age) AS AvgAge,MAX(Age) AS MaxAge,CURRENT_TIMESTAMP AS Now"
SET myquery(3) = " FROM Sample.Employee WHERE Name %STARTSWITH 'ZZZ'"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount:",rset.%ROWCOUNT
```

- **COUNT** returns NULL if the select-list contains any direct reference to a field in a FROM clause table, or if TOP 0 is specified. The query returns a %ROWCOUNT of 0. The following example does not return a **COUNT** value because the %ROWCOUNT value is 0:

```
SET myquery = 2
SET myquery(1) = "SELECT COUNT(*) AS Recs,COUNT(Name) AS People,$LENGTH(Name) AS NameLen"
SET myquery(2) = " FROM Sample.Employee WHERE Name %STARTSWITH 'ZZZ'"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount:",rset.%ROWCOUNT
```

- **COUNT(*)** returns 1 if no table is specified. The query returns a %ROWCOUNT of 1. This is shown in the following example:

```
SET myquery = "SELECT COUNT(*) AS Recs"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"Rowcount:",rset.%ROWCOUNT
```

Stream Fields

You can use **COUNT(expression)** to count [stream field](#) values, with some restrictions. **COUNT(streamfield)** counts all non-NULL values. It does not check for duplicate values.

You cannot specify the **COUNT** function's **DISTINCT** keyword when *expression* is a stream field. Attempting to use a **DISTINCT** keyword with a stream field results in an SQLCODE -37 error.

You cannot specify a stream field in a %FOREACH *col-list*. Attempting to do so results in an SQLCODE -37 error.

The following example shows valid uses of the **COUNT** function, where Title is a string field and Notes and Picture are stream fields:

```
SELECT DISTINCT Title,COUNT(Notes),COUNT(Picture %FOREACH(Title))
FROM Sample.Employee
```

The following examples are *not* valid when Title is a string field and Notes and Picture are stream fields:

```
-- Invalid: DISTINCT keyword with stream field
SELECT Title,COUNT(DISTINCT Notes) FROM Sample.Employee
```

```
-- Invalid: %FOREACH col-list contains stream field
SELECT Title,COUNT(Notes %FOREACH(Picture))
FROM Sample.Employee
```

Privileges

To use **COUNT(*)** you must have table-level **SELECT** privilege for the specified table. To use **COUNT(column-name)** you must have column-level **SELECT** privilege for the specified column, or table-level **SELECT** privilege for the specified table. You can determine if the current user has **SELECT** privilege by invoking the **%CHECKPRIV** command. You can determine if a specified user has table-level **SELECT** privilege by invoking the **\$\$SYSTEM.SQL.CheckPriv()** method. For privilege assignment, refer to the **GRANT** command.

Performance

For optimal **COUNT** performance, you should define indices as follows:

- For **COUNT(*)**, define a **bitmap extent index**, if needed. This index may have been automatically defined when the table was created.
- For **COUNT(fieldname)**, define a **bitslice index** for the specified field.

Changes Made by Uncommitted Transactions

Like all aggregate functions, **COUNT** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level, as follows:

- **COUNT** counts inserted and updated records, even though those changes have not been committed and may be rolled back.
- **COUNT** *does not* count deleted records, even though those deletions have not been committed and may be rolled back.

For further details, refer to **SET TRANSACTION** and **START TRANSACTION**.

Examples

The following example returns the total number of rows in **Sample.Person**:

```
SELECT COUNT(*) AS TotalPersons
FROM Sample.Person
```

The following example returns the count of names, spouses, and favorite colors in **Sample.Person**. These counts differ because some **Spouse** and **FavoriteColors** fields have **NULL**; **COUNT** does not count nulls:

```
SELECT COUNT(Name) AS People,
       COUNT(Spouse) AS PeopleWithSpouses,
       COUNT(FavoriteColors) AS PeopleWithColorPref
FROM Sample.Person
```

The following example returns three values: the total number of rows, the total number of non-**NULL** values in the **FavoriteColors** field, and the total number of distinct non-**NULL** values in the **FavoriteColors** field:

```
SELECT COUNT(*) As TotalPersons,
       COUNT(FavoriteColors) AS WithColorPref,
       COUNT(DISTINCT FavoriteColors) AS ColorPrefs
FROM Sample.Person
```

The following example uses **COUNT DISTINCT** to return the count of distinct **FavoriteColors** values in **Sample.Person**. (**FavoriteColors** contains several data values and multiple **NULL**s.) This example also uses the **DISTINCT** clause to return one row for each distinct **FavoriteColors** value. The row count is one larger than the **COUNT(DISTINCT FavoriteColors)** count, because **DISTINCT** returns a row for a single **NULL** as a distinct value, but **COUNT DISTINCT** does not count **NULL**. The **COUNT(DISTINCT BY(FavoriteColors) %ID)** value is the same as the row count, because the **BY** clause does count a single **NULL** as a distinct value:

```
SELECT DISTINCT FavoriteColors,
       COUNT(DISTINCT FavoriteColors) AS DistColors,
       COUNT(DISTINCT BY(FavoriteColors) %ID) AS DistColorPeople
FROM Sample.Person
```

The following example use **GROUP BY** to return a row for each FavoriteColors value, including a row for NULL. Associated with each row are two counts. The first counts the number of records with that FavoriteColors option; records with NULL are not counted. The second counts the number of names associated with each FavoriteColor choice; since Name does not include NULL values, this enables a count of FavoriteColors with NULL:

```
SELECT FavoriteColors,
       COUNT(FavoriteColors) AS ColorPreference,
       COUNT(Name) AS People
FROM Sample.Person
GROUP BY FavoriteColors
```

The following example returns the count of person records for each Home_State value in Sample.Person:

```
SELECT Home_State, COUNT(*) AS AllPersons
FROM Sample.Person
GROUP BY Home_State
```

The following example uses **%AFTERHAVING** to return the count of person records and the count of persons over 65 for each state in which there is at least one person over 65:

```
SELECT Home_State, COUNT(Name) AS AllPersons,
       COUNT(Name %AFTERHAVING) AS Seniors
FROM Sample.Person
GROUP BY Home_State
HAVING Age > 65
ORDER BY Home_State
```

The following example uses both the **%FOREACH** and the **%AFTERHAVING** keywords. It returns a row for those states containing people whose names start with “A”, “M”, or “W” (HAVING clause and GROUP BY clause). Each state row contains the following values:

- **COUNT(Name)**: a count of all of the people in the database. (This number is the same for all rows.)
- **COUNT(Name %FOREACH(Home_State))**: a count of all of the people in the state.
- **COUNT(Name %AFTERHAVING)**: a count of all of the people in the database that meet the HAVING clause criteria. (This number is the same for all rows.)
- **COUNT(Name %FOREACH(Home_State) %AFTERHAVING)**: a count of all of the people in the state that meet the HAVING clause criteria.

```
SELECT Home_State,
       COUNT(Name) AS NameCount,
       COUNT(Name %FOREACH(Home_State)) AS StateNameCount,
       COUNT(Name %AFTERHAVING) AS NameCountHaving,
       COUNT(Name %FOREACH(Home_State) %AFTERHAVING) AS StateNameCountHaving
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'A%' OR Name LIKE 'M%' OR Name LIKE 'W%'
ORDER BY Home_State
```

The following example shows **COUNT** with a concatenation expression. It returns the total number of non-NULL values in the FavoriteColors field, and the total number of non-NULL values in FavoriteColors concatenated with two other fields, using the concatenate operator (**||**):

```
SELECT COUNT(FavoriteColors) AS Color,
       COUNT(FavoriteColors || Home_State) AS ColorState,
       COUNT(FavoriteColors || Spouse) AS ColorSpouse
FROM Sample.Person
```

When two fields are concatenated, **COUNT** counts only those rows in which neither field has a NULL value. Because every row in Sample.Person has a non-NULL Home_State value, the concatenation FavoriteColors || Home_State returns the same count as FavoriteColors. Because some rows in Sample.Person have a NULL value for Spouse, the

concatenation `FavoriteColors || Spouse` returns the count of rows which have non-NULL values for both `FavoriteColors` and `Spouse`.

See Also

- [Aggregate Functions](#) overview
- [AVG](#) aggregate function
- [SUM](#) aggregate function

%DLIST

An aggregate function that creates a Caché list of values.

```
%DLIST([ALL | DISTINCT [BY(col-list)]] string-expr [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Specifies that %DLIST returns a list of all values for <i>string-expr</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that %DLIST returns a %List structured list containing only the unique <i>string-expr</i> values. DISTINCT can specify a BY(<i>col-list</i>) subclause, where <i>col-list</i> can be a single field or a comma-separated list of fields.
<i>string-expr</i>	An SQL expression that evaluates to a string. Usually the name of a column from the selected table.
%FOREACH(<i>col-list</i>)	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.
%AFTERHAVING	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

The **%DLIST** [aggregate function](#) returns a Caché %List structure containing the values in the specified column as list elements.

A simple **%DLIST** (or **%DLIST ALL**) returns Caché list composed of all the non-NULL values for *string-expr* in the selected rows. Rows where *string-expr* is NULL are not included as elements in the list structure.

A **%DLIST DISTINCT** returns a Caché list composed of all the distinct (unique) non-NULL values for *string-expr* in the selected rows: %DLIST(DISTINCT col1). NULL is not included as an element in the %List structure. %DLIST(DISTINCT BY(col2) col1) returns a %List of elements including only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

For further information about Caché list structures, see [\\$LIST](#) and related functions.

%DLIST and %SelectMode

You can use the [%SelectMode](#) property to specify the data display mode returned by **%DLIST**: 0=Logical (the default), 1=ODBC, 2=Display.

Note that **%DLIST** in ODBC mode separates column value lists with commas, and **\$LISTTOSTRING** (by default) returns elements within a %List column value separated with commas.

%DLIST and ORDER BY

The **%DLIST** function combines the values of a table column from multiple rows into %List structured list of values. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within this list. Under certain circumstances, **%DLIST** results may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

Related Aggregate Functions

- **%DLIST** returns a Caché list of values.

- [LIST](#) returns a comma-separated list of values.
- [JSON_ARRAYAGG](#) returns a JSON array of values.
- [XMLAGG](#) returns a concatenated string of values.

Examples

The following Embedded SQL example returns a host variable containing a Caché list of all of the values listed in the Home_State column of the Sample.Person table that start with the letter “A”:

```
&sql(SELECT %DLIST(Home_State)
      INTO :statelist
      FROM Sample.Person
      WHERE Home_State %STARTSWITH 'A')
WRITE "The states (as list):",statelist,!
WRITE "The states (as string):", $LISTTOSTRING(statelist,"^")
```

Note that this Caché list contains elements with duplicate values.

The following Embedded SQL example returns a host variable containing a Caché list of all of the distinct (unique) values listed in the Home_State column of the Sample.Person table that start with the letter “A”:

```
&sql(SELECT %DLIST(DISTINCT Home_State)
      INTO :statelist
      FROM Sample.Person
      WHERE Home_State %STARTSWITH 'A')
WRITE "The states (as list):",statelist,!
WRITE "The states (as string):", $LISTTOSTRING(statelist,"^")
```

The following SQL example creates a Caché list of all of the values found in the Home_City column for each of the states, and a count of these city values by state. Every Home_State row contains a list of all of the Home_City values for that state. These lists may include duplicate city names:

```
SELECT Home_State,
       %DLIST(Home_City) AS AllCities,
       COUNT(Home_City) AS CityCount
FROM Sample.Person
GROUP BY Home_State
```

Perhaps more useful would be a list of all of the *distinct* values found in the Home_City column for each of the states, as shown in the following example:

```
SELECT Home_State,
       %DLIST(DISTINCT Home_City) AS CitiesList,
       COUNT(DISTINCT Home_City) AS DistinctCities,
       COUNT(Home_City) AS TotalCities
FROM Sample.Person
GROUP BY Home_State
```

Note that this example returns integer counts of both the distinct city names and the total city names for each state.

The following example returns %List structures of Home_State values that begin with “A”. It returns as %List elements the distinct Home_State values (DISTINCT Home_State); the Home_State values corresponding to distinct Home_City values (DISTINCT BY(Home_City) Home_State), which may possibly including one unique NULL for Home_City; and all Home_State values:

```
SELECT %DLIST(DISTINCT Home_State) AS DistStates,
       %DLIST(DISTINCT BY(Home_City) Home_State) AS DistCityStates,
       %DLIST(Home_State) AS AllStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

The following Dynamic SQL example uses the %SelectMode property to specify the ODBC display mode for the %List structure FavoriteColors date field. ODBC mode returns the value for each column as a comma-separated list, and the **\$LISTTOSTRING** function specifies a different delimiter (in this example, ||) to separate the values from the different columns:

```

ZNSPACE "SAMPLES"
SET myquery = "SELECT %DLIST(FavoriteColors) AS colors FROM Sample.Person WHERE Name %STARTSWITH 'A'"

SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
  WRITE $LISTTOSTRING(rset.colors,"|"),!
}
WRITE !,"End of data"

```

The following example uses the %AFTERHAVING keyword. It returns a row for each Home_State that contains at least one Name value that fulfills the HAVING clause condition (a name that begins with “M”). The first %DLIST function returns a list of all of the names for that state. The second %DLIST function returns a list containing only those names that fulfill the HAVING clause condition:

```

SELECT Home_State,
       %DLIST(Name) AS AllNames,
       %DLIST(Name %AFTERHAVING) AS HaveClauseNames
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'M%'
ORDER BY Home_state

```

See Also

- [Aggregate Functions](#) overview
- [SELECT](#)
- [\\$LIST](#) function
- [JSON_ARRAYAGG](#) aggregate function
- [LIST](#) aggregate function
- [XMLAGG](#) aggregate function

JSON_ARRAYAGG

An aggregate function that creates a JSON format array of values.

```
JSON_ARRAYAGG([ALL | DISTINCT [BY(col-list)]] string-expr [%FOREACH(col-list)]
[%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Specifies that JSON_ARRAYAGG returns a JSON array containing all values for <i>string-expr</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that JSON_ARRAYAGG returns a JSON array containing only the unique <i>string-expr</i> values. DISTINCT can specify a <code>BY(col-list)</code> subclause, where <i>col-list</i> can be a single field or a comma-separated list of fields.
<i>string-expr</i>	An SQL expression that evaluates to a string. Usually the name of a column from the selected table.
<code>%FOREACH(col-list)</code>	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on <code>%FOREACH</code> .
<code>%AFTERHAVING</code>	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

The **JSON_ARRAYAGG** [aggregate function](#) returns a JSON format array of the values in the specified column. For further details on JSON array format, refer to the **JSON_ARRAY** function.

A simple **JSON_ARRAYAGG** (or **JSON_ARRAYAGG ALL**) returns a JSON array containing all the values for *string-expr* in the selected rows. Rows where *string-expr* is the empty string (") are represented by ("\"u0000") in the array. Rows where *string-expr* is NULL are not included in the array. If there is only one *string-expr* value, and it is the empty string ("), **JSON_ARRAYAGG** returns the JSON array ["\"u0000 "]. If all *string-expr* values are NULL, **JSON_ARRAYAGG** returns an empty JSON array [].

A **JSON_ARRAYAGG DISTINCT** returns a JSON array composed of all the different (unique) values for *string-expr* in the selected rows: `JSON_ARRAYAGG(DISTINCT col1)`. The NULL *string-expr* is not included in the JSON array. `JSON_ARRAYAGG(DISTINCT BY(col2) col1)` returns a JSON array containing only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

The **JSON_ARRAYAGG** *string-expr* cannot be a stream field. Specifying a stream field results in an SQLCODE -37.

Data Values Containing Escaped Characters

- **Double Quote:** If a *string-expr* value contains a double quote character ("), **JSON_ARRAYAGG** represents this character using the literal escape sequence \".
- **Backslash:** If a *string-expr* value contains a backslash character (\), **JSON_ARRAYAGG** represents this character using the literal escape sequence \\.
- **Single Quote:** When a *string-expr* value contains a single quote as a literal character, Caché SQL requires that this character must be escaped by doubling it as two single quote characters (' '. **JSON_ARRAYAGG** represents this character as a single quote character ' .

Maximum JSON Array Size

The default `JSON_ARRAYAGG` return type is `VARCHAR(8192)`. This length includes the JSON array formatting characters as well as the field data characters. If you anticipate the value returned will need to be longer than 8192, you can use the `CAST` function to specify a larger return value. For example, `CAST(JSON_ARRAYAGG(value) AS VARCHAR(12000))`. If the actual JSON array returned is longer than the `JSON_ARRAYAGG` return type length, Caché truncates the JSON array at the return type length without issuing an error. Because truncating a JSON array removes its closing `]` character, this makes the return value invalid.

JSON_ARRAYAGG and %SelectMode

You can use the `%SelectMode` property to specify the data display values for the elements in the JSON array: 0=Logical (the default), 1=ODBC, 2=Display. If the *string-expr* contains a `%List` structure, the elements are represented in ODBC mode separated by a comma, and in Logical and Display mode with `%List` format characters represented by `\` escape sequences. Refer to [\\$ZCONVERT](#) “Encoding Translation” for an table listing these JSON `\` escape sequences.

JSON_ARRAYAGG and ORDER BY

The `JSON_ARRAYAGG` function combines the values of a table column from multiple rows into a JSON array of element values. Because an `ORDER BY` clause is applied to the query result set after all aggregate fields are evaluated, `ORDER BY` cannot directly affect the sequence of values within this list. Under certain circumstances, `JSON_ARRAYAGG` results may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

Related Aggregate Functions

- `LIST` returns a comma-separated list of values.
- `%DLIST` returns a Caché list containing an element for each value.
- `XMLAGG` returns a concatenated string of values.

Examples

The following Embedded SQL example returns a host variable containing a JSON array of all of the values in the `Home_State` column of the `Sample.Person` table that start with the letter “A”:

```
&sql(SELECT JSON_ARRAYAGG(Home_State)
      INTO :statearray
      FROM Sample.Person
      WHERE Home_State %STARTSWITH 'A')
WRITE "JSON array of states:",!,statearray
```

Note that this JSON array contains duplicate values.

The following Dynamic SQL example returns a host variable containing a JSON array of all of the distinct (unique) values in the `Home_State` column of the `Sample.Person` table that start with the letter “A”:

```
ZNSPACE "SAMPLES"
SET myquery = 2
SET myquery(1) = "SELECT JSON_ARRAYAGG(DISTINCT Home_State) AS DistinctStates "
SET myquery(2) = "FROM Sample.Person WHERE Home_State %STARTSWITH 'A' "
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following SQL example creates a JSON array of all of the values found in the `Home_City` column for each of the states, and a count of these city values by state. Every `Home_State` row contains a JSON array of all of the `Home_City` values for that state. These JSON arrays may include duplicate city names:

```

SELECT Home_State,
       COUNT(Home_City) AS CityCount,
       JSON_ARRAYAGG(Home_City) AS ArrayAllCities
FROM Sample.Person
GROUP BY Home_State

```

Perhaps more useful would be a JSON array of all of the *distinct* values found in the Home_City column for each of the states, as shown in the following Dynamic SQL example:

```

ZNSPACE "SAMPLES"
SET myquery = 4
SET myquery(1) = "SELECT Home_State,COUNT(DISTINCT Home_City) AS DistCityCount,"
SET myquery(2) = "COUNT(Home_City) AS TotCityCount,"
SET myquery(3) = "JSON_ARRAYAGG(DISTINCT Home_City) AS ArrayDistCities "
SET myquery(4) = "FROM Sample.Person GROUP BY Home_State"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"

```

Note that this example returns integer counts of both the distinct city names and the total city names for each state.

The following Dynamic SQL example uses the %SelectMode property to specify the ODBC display mode for the JSON array of values returned by the DOB date field:

```

ZNSPACE "SAMPLES"
SET myquery = 2
SET myquery(1) = "SELECT JSON_ARRAYAGG(DOB) AS DOBs "
SET myquery(2) = "FROM Sample.Person WHERE Name %STARTSWITH 'A' "
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"

```

The following Dynamic SQL example uses the %FOREACH keyword. It returns a row for each distinct Home_State containing a JSON array of age values for that Home_State.

```

ZNSPACE "SAMPLES"
SET myquery = 3
SET myquery(1) = "SELECT DISTINCT Home_State,"
SET myquery(2) = "JSON_ARRAYAGG(Age %FOREACH(Home_State)) AgesForState "
SET myquery(3) = "FROM Sample.Person WHERE Home_State %STARTSWITH 'M' "
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"

```

The following Dynamic SQL example uses the %AFTERHAVING keyword. It returns a row for each Home_State that contains at least one Name value that fulfills the HAVING clause condition (a name that begins with “M”). The first **JSON_ARRAYAGG** function returns a JSON array of all of the names for that state. The second **JSON_ARRAYAGG** function returns a JSON array containing only those names that fulfill the HAVING clause condition:

```

ZNSPACE "SAMPLES"
SET myquery = 4
SET myquery(1) = "SELECT Home_State,JSON_ARRAYAGG(Name) AS AllNames,"
SET myquery(2) = "JSON_ARRAYAGG(Name %AFTERHAVING) AS HavingClauseNames "
SET myquery(3) = "FROM Sample.Person GROUP BY Home_State "
SET myquery(4) = "HAVING Name LIKE 'M%' ORDER BY Home_State"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"

```

See Also

- [Aggregate Functions](#) overview
- [JSON_ARRAY](#) function
- [IS JSON](#) predicate condition
- [LIST](#) aggregate function
- [%DLIST](#) aggregate function
- [XMLAGG](#) aggregate function
- [SELECT](#) statement

LIST

An aggregate function that creates a comma-separated list of values.

```
LIST([ALL | DISTINCT [BY(col-list)]] string-expr [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Specifies that LIST returns a list of all values for <i>string-expr</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that LIST returns a list containing only the unique <i>string-expr</i> values. DISTINCT can specify a <code>BY(col-list)</code> subclause, where <i>col-list</i> can be a single field or a comma-separated list of fields.
<i>string-expr</i>	An SQL expression that evaluates to a string. Usually the name of a column from the selected table.
<code>%FOREACH(col-list)</code>	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on <code>%FOREACH</code> .
<code>%AFTERHAVING</code>	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

The **LIST aggregate function** returns a comma-separated list of the values in the specified column.

A simple **LIST** (or **LIST ALL**) returns a string containing a comma-separated list composed of all the values for *string-expr* in the selected rows. Rows where *string-expr* is the empty string (") are represented by a placeholder comma in the comma-separated list. Rows where *string-expr* is NULL are not included in the comma-separated list. If there is only one *string-expr* value, and it is the empty string ("), **LIST** returns the empty string.

A **LIST DISTINCT** returns a string containing a comma-separated list composed of all the distinct (unique) values for *string-expr* in the selected rows: `LIST(DISTINCT col1)`. The NULL *string-expr* is not included in the comma-separated list. `LIST(DISTINCT BY(col2) col1)` returns a comma-separated list containing only those `col1` field values in records where the `col2` values are distinct (unique). Note however that the distinct `col2` values may include a single NULL as a distinct value.

Data Values Containing Commas

Because **LIST** uses commas to separate *string-expr* values, **LIST** should not be used for data values that contain commas. Use `%DLIST` or `JSON_ARRAYAGG` instead.

LIST and %SelectMode

You can use the `%SelectMode` property to specify the data display mode returned by **LIST**: 0=Logical (the default), 1=ODBC, 2=Display.

Note that **LIST** separates column values with commas, and ODBC mode separates elements within a `%List` column value with commas. Therefore, using ODBC mode when using **LIST** on a `%List` structure produces ambiguous results.

LIST and ORDER BY

The **LIST** function combines the values of a table column from multiple rows into a single comma-separated list of values. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within this list. Under certain circumstances, **LIST** results may appear in

sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

Maximum LIST Size

The largest permitted **LIST** return value is the [maximum string length](#) configured for your system. If your system is configured for Long Strings (the default), the longest list is 3,641,144 characters.

Related Aggregate Functions

- **LIST** returns a comma-separated list of values.
- **%DLIST** returns a Caché list containing an element for each value.
- **JSON_ARRAYAGG** returns a JSON array of values.
- **XMLAGG** returns a concatenated string of values.

Examples

The following Embedded SQL example returns a host variable containing a comma-separated list of all of the values listed in the Home_State column of the Sample.Person table that start with the letter “A”:

```
&sql(SELECT LIST(Home_State)
      INTO :statelist
      FROM Sample.Person
      WHERE Home_State %STARTSWITH 'A')
WRITE "The states are:",!,statelist
```

Note that this list contains duplicate values.

The following Embedded SQL example returns a host variable containing a comma-separated list of all of the distinct (unique) values listed in the Home_State column of the Sample.Person table that start with the letter “A”:

```
&sql(SELECT LIST(DISTINCT Home_State)
      INTO :statelist
      FROM Sample.Person
      WHERE Home_State %STARTSWITH 'A')
WRITE "The distinct states are:",!,statelist
```

The following SQL example creates a comma-separated list of all of the values found in the Home_City column for each of the states, and a count of these city values by state. Every Home_State row contains a list of all of the Home_City values for that state. These lists may include duplicate city names:

```
SELECT Home_State,
       COUNT(Home_City) AS CityCount,
       LIST(Home_City) AS ListAllCities
FROM Sample.Person
GROUP BY Home_State
```

Perhaps more useful would be a comma-separated list of all of the *distinct* values found in the Home_City column for each of the states, as shown in the following example:

```
SELECT Home_State,
       COUNT(DISTINCT Home_City) AS DistCityCount,
       COUNT(Home_City) AS TotCityCount,
       LIST(DISTINCT Home_City) AS DistCitiesList
FROM Sample.Person
GROUP BY Home_State
```

Note that this example returns integer counts of both the distinct city names and the total city names for each state.

The following example returns lists of Home_State values that begin with “A”. It returns the distinct Home_State values (DISTINCT Home_State); the Home_State values corresponding to distinct Home_City values (DISTINCT BY(Home_City) Home_State), which may possibly including one unique NULL for Home_City; and all Home_State values:

```
SELECT LIST(DISTINCT Home_State) AS DistStates,  
       LIST(DISTINCT BY(Home_City) Home_State) AS DistCityStates,  
       LIST(Home_State) AS AllStates  
FROM Sample.Person  
WHERE Home_State %STARTSWITH 'A'
```

The following Dynamic SQL example uses the %SelectMode property to specify the ODBC display mode for the list of values returned by the DOB date field:

```
ZNSPACE "SAMPLES"  
SET myquery = "SELECT LIST(DOB) AS DOBs FROM Sample.Person WHERE Name %STARTSWITH 'A' "  
SET tStatement = ##class(%SQL.Statement).%New()  
SET tStatement.%SelectMode=1  
SET qStatus = tStatement.%Prepare(myquery)  
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}  
SET rset = tStatement.%Execute()  
DO rset.%Display()  
WRITE !,"End of data"
```

The following Dynamic SQL example uses the %FOREACH keyword. It returns a row for each distinct Home_State containing a list of age values for that Home_State:

```
ZNSPACE "SAMPLES"  
SET myquery = 3  
SET myquery(1) = "SELECT DISTINCT Home_State,"  
SET myquery(2) = "LIST(Age %FOREACH(Home_State)) AgesForState "  
SET myquery(3) = "FROM Sample.Person WHERE Home_State %STARTSWITH 'M' "  
SET tStatement = ##class(%SQL.Statement).%New()  
SET tStatement.%SelectMode=1  
SET qStatus = tStatement.%Prepare(.myquery)  
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}  
SET rset = tStatement.%Execute()  
DO rset.%Display()  
WRITE !,"End of data"
```

The following example uses the %AFTERHAVING keyword. It returns a row for each Home_State that contains at least one Name value that fulfills the HAVING clause condition (a name that begins with “M”). The first LIST function returns a list of all of the names for that state. The second LIST function returns a list containing only those names that fulfill the HAVING clause condition:

```
SELECT Home_State,  
       LIST(Name) AS AllNames,  
       LIST(Name %AFTERHAVING) AS HavingClauseNames  
FROM Sample.Person  
GROUP BY Home_State  
HAVING Name LIKE 'M%'  
ORDER BY Home_State
```

See Also

- [Aggregate Functions](#) overview
- [%DLIST](#) aggregate function
- [JSON_ARRAYAGG](#) aggregate function
- [XMLAGG](#) aggregate function
- [SELECT](#) statement

MAX

An aggregate function that returns the maximum data value in a specified column.

```
MAX([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Applies the aggregate function to all values. ALL has no effect on the value returned by MAX . It is provided for SQL-92 compatibility.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that each unique value is considered. DISTINCT has no effect on the value returned by MAX . It is provided for SQL-92 compatibility.
<i>expression</i>	Any valid expression. Usually the name of a column that contains the values from which the maximum value is to be returned.
%FOREACH(<i>col-list</i>)	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.
%AFTERHAVING	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

The **MAX** [aggregate function](#) returns the largest (maximum) of the values of *expression*. Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query.

MAX can be used in a **SELECT** query or subquery that references either a table or a view. **MAX** can appear in a SELECT list or HAVING clause alongside ordinary field values.

MAX cannot be used in a WHERE clause. **MAX** cannot be used in the ON clause of a JOIN, unless the SELECT is a subquery.

Like most other aggregate functions, **MAX** cannot be applied to a [stream field](#). Attempting to do so generates an SQLCODE -37 error.

Unlike most other aggregate functions, the ALL and DISTINCT keywords, including MAX(DISTINCT BY(col2) col1), perform no operation in **MAX**. They are provided for SQL-92 compatibility.

Data Values

The specified field used by **MAX** can be numeric or nonnumeric. For a numeric data type field, maximum is defined as highest in numeric value; thus -3 is higher than -7. For a non-numeric data type field, maximum is defined as highest in string [collation sequence](#); thus '-7' is higher than '-3'.

An empty string (") value is treated as CHAR(0).

A predicate uses the [collation type](#) defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

When the field's defined collation type is SQLUPPER, **MAX** returns strings in all uppercase letters. Thus SELECT MAX(Name) returns 'ZWIG', regardless of the original lettercase of the data. But because comparisons are performed using uppercase collation, the clause HAVING Name=MAX(Name) selects rows with the Name value 'Zwig', 'ZWIG', and 'zwig'.

For numeric values, the scale returned is the same as the *expression* scale.

NULL values in data fields are ignored when deriving a **MAX** aggregate function value. If no rows are returned by the query, or the data field value for all rows returned is NULL, **MAX** returns NULL.

Changes Made During the Current Transaction

Like all aggregate functions, **MAX** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to [SET TRANSACTION](#) and [START TRANSACTION](#).

Examples

The following query returns the highest (maximum) salary in the Sample.Employee database:

```
SELECT '$' || MAX(Salary) As TopSalary
FROM Sample.Employee
```

The following query returns one row for each state that contains at least one employee with a salary smaller than \$25,000. Using the **%AFTERHAVING** keyword, each row returns the maximum employee salary smaller than \$25,000. Each row also returns the minimum salary and the maximum salary for all employees in that state:

```
SELECT Home_State,
'$' || MAX(Salary %AFTERHAVING) AS MaxSalaryBelow25K,
'$' || MIN(Salary) AS MinSalary,
'$' || MAX(Salary) AS MaxSalary
FROM Sample.Employee
GROUP BY Home_State
HAVING Salary < 25000
ORDER BY Home_State
```

The following query returns the lowest (minimum) and highest (maximum) name in collation sequence found in the Sample.Employee database:

```
SELECT Name, MIN(Name), MAX(Name)
FROM Sample.Employee
```

Note that **MIN** and **MAX** convert Name values to uppercase before comparison.

The following query returns the highest (maximum) salary for an employee whose Home_State is 'VT' in the Sample.Employee database:

```
SELECT MAX(Salary)
FROM Sample.Employee
WHERE Home_State = 'VT'
```

The following query returns the number of employees and the highest (maximum) employee salary for each Home_State in the Sample.Employee database:

```
SELECT Home_State,
COUNT(Home_State) As NumEmployees,
MAX(Salary) As TopSalary
FROM Sample.Employee
GROUP BY Home_State
ORDER BY TopSalary
```

See Also

- [Aggregate Functions](#) overview
- [MIN](#) aggregate function

MIN

An aggregate function that returns the minimum data value in a specified column.

```
MIN([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Applies the aggregate function to all values. ALL has no effect on the value returned by MIN . It is provided for SQL-92 compatibility.
DISTINCT	<i>Optional</i> — Specifies that each unique value is considered. DISTINCT has no effect on the value returned by MIN . It is provided for SQL-92 compatibility.
<i>expression</i>	Any valid expression. Usually the name of a column that contains the values from which the minimum value is to be returned.
%FOREACH(<i>col-list</i>)	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.
%AFTERHAVING	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

The **MIN** [aggregate function](#) returns the smallest (minimum) of the values of *expression*. Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query.

MIN can be used in a [SELECT](#) query or subquery that references either a table or a view. **MIN** can appear in a SELECT list or HAVING clause alongside ordinary field values.

MIN cannot be used in a WHERE clause. **MIN** cannot be used in the ON clause of a JOIN, unless the SELECT is a subquery.

Like most other aggregate functions, **MIN** cannot be applied to a [stream field](#). Attempting to do so generates an SQLCODE -37 error.

Unlike most other aggregate functions, the ALL and DISTINCT keywords, including MIN(DISTINCT BY(col2) col1), perform no operation in **MIN**. They are provided for SQL-92 compatibility.

Data Values

The specified field used by **MIN** can be numeric or nonnumeric. For a numeric data type field, minimum is defined as lowest in numeric value; thus -7 is lower than -3. For a non-numeric data type field, minimum is defined as lowest in string [collation sequence](#); thus '-3' is lower than '-7'.

An empty string (") value is treated as CHAR(0).

A predicate uses the [collation type](#) defined for the field. By default, string data type fields are defined with SQLUPPER collation, which is not case-sensitive. The “Collation” chapter of *Using Caché SQL* provides details on defining the [string collation default for the current namespace](#) and specifying a [non-default field collation type when defining a field/property](#).

When the field's defined collation type is SQLUPPER, **MIN** returns strings in all uppercase letters. Thus SELECT MIN(Name) returns 'AARON', regardless of the original lettercase of the data. But because comparisons are performed using uppercase collation, the clause HAVING Name=MIN(Name) selects rows with the Name value 'Aaron', 'AARON', and 'aaron'.

For numeric values, the scale returned is the same as the *expression* scale.

NULL values in data fields are ignored when deriving a **MIN** aggregate function value. If no rows are returned by the query, or the data field value for all rows returned is NULL, **MIN** returns NULL.

Changes Made During the Current Transaction

Like all aggregate functions, **MIN** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to [SET TRANSACTION](#) and [START TRANSACTION](#).

Examples

In the following examples a dollar sign (\$) is concatenated to Salary amounts.

The following query returns the lowest (minimum) salary in the Sample.Employee database:

```
SELECT '$' || MIN(Salary) AS LowSalary
FROM Sample.Employee
```

The following query returns one row for each state that contains at least one employee with a salary larger than \$75,000. Using the **%AFTERHAVING** keyword, each row returns the minimum employee salary larger than \$75,000. Each row also returns the minimum salary and the maximum salary for all employees in that state:

```
SELECT Home_State,
'$' || MIN(Salary %AFTERHAVING) AS MinSalaryAbove75K,
'$' || MIN(Salary) AS MinSalary,
'$' || MAX(Salary) AS MaxSalary
FROM Sample.Employee
GROUP BY Home_State
HAVING Salary > 75000
ORDER BY MinSalaryAbove75K
```

The following query returns the lowest (minimum) and highest (maximum) name in collation sequence found in the Sample.Employee database:

```
SELECT Name, MIN(Name), MAX(Name)
FROM Sample.Employee
```

Note that **MIN** and **MAX** convert Name values to uppercase before comparison.

The following query returns the lowest (minimum) salary for an employee whose Home_State is 'VT' in the Sample.Employee database:

```
SELECT MIN(Salary)
FROM Sample.Employee
WHERE Home_State = 'VT'
```

The following query returns the number of employees and the lowest (minimum) employee salary for each Home_State in the Sample.Employee database:

```
SELECT Home_State,
COUNT(Home_State) As NumEmployees,
MIN(Salary) As LowSalary
FROM Sample.Employee
GROUP BY Home_State
ORDER BY LowSalary
```

See Also

- [Aggregate Functions](#) overview
- [MAX](#) aggregate function

STDDEV, STDDEV_SAMP, STDDEV_POP

Aggregate functions that return the statistical standard deviation of a data set.

```
STDDEV([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
STDDEV_SAMP([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
STDDEV_POP([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Specifies that standard deviation functions return the standard deviation of all values for <i>expression</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that standard deviation functions return the standard deviation of the distinct (unique) <i>expression</i> values. DISTINCT can specify a <code>BY(col-list)</code> subclause, where <i>col-list</i> can be a single field or a comma-separated list of fields.
<i>expression</i>	Any valid expression. Usually the name of a column that contains the data values to be analyzed for standard deviation.
<code>%FOREACH(col-list)</code>	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on <code>%FOREACH</code> .
<code>%AFTERHAVING</code>	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

These three standard deviation [aggregate functions](#) return the statistical standard deviation of the distribution of the values of *expression*, after discarding NULL values. That is, the amount of standard deviation from the mean value of the data set, expressed as a positive number. The larger the return value, the more variation there is within the data set of values.

The **STDDEV**, **STDDEV_SAMP** (sample), and **STDDEV_POP** (population) functions are derived from the corresponding variance aggregate functions:

STDDEV	VARIANCE
STDDEV_SAMP	VAR_SAMP
STDDEV_POP	VAR_POP

The standard deviation is the square root of the corresponding variance value. Refer to these [variance aggregate functions](#) for further details.

These standard deviation functions can be used in a [SELECT](#) query or subquery that references either a table or a view. They can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

These standard deviation functions cannot be used in a **WHERE** clause. They cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

These standard deviation functions return a value of data type NUMERIC with a precision of 36 and a scale of 17, unless *expression* is data type DOUBLE in which case it returns data type DOUBLE.

These functions are normally applied to a field or expression that has a numeric value. They evaluate nonnumeric values, including the empty string (''), as zero (0).

These standard deviation functions ignore NULL values in data fields. If no rows are returned by the query, or the data field value for all rows returned is NULL, they return NULL.

The standard deviation functions, like all aggregate functions, can take an optional [DISTINCT clause](#). `STDDEV (DISTINCT col1)` returns the standard deviation of those col1 field values that are distinct (unique). `STDDEV (DISTINCT BY (col2) col1)` returns the standard deviation of the col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Changes Made During the Current Transaction

Like all aggregate functions, standard deviation functions always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to [SET TRANSACTION](#) and [START TRANSACTION](#).

Examples

The following example uses **STDDEV** to return the standard deviation in the ages of the employees in `Sample.Employee`, and the standard deviation in the distinct ages represented by one or more employees:

```
SELECT STDDEV(Age) AS AgeSD,STDDEV(DISTINCT Age) AS PerAgeSD
FROM Sample.Employee
```

The following example uses **STDDEV_POP** to return the population standard deviation in the ages of the employees in `Sample.Employee`, and the standard deviation in the distinct ages represented by one or more employees:

```
SELECT STDDEV_POP(Age) AS AgePopSD,STDDEV_POP(DISTINCT Age) AS PerAgePopSD
FROM Sample.Employee
```

See Also

- [Aggregate Functions](#) overview
- [VARIANCE](#), [VAR_SAMP](#), [VAR_POP](#) aggregate functions
- [AVG](#) aggregate function
- [COUNT](#) aggregate function

SUM

An aggregate function that returns the sum of the values of a specified column.

```
SUM([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Specifies that SUM return the sum of all values for <i>expression</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that SUM return the sum of the distinct (unique) values for <i>expression</i> . DISTINCT can specify a <code>BY(col-list)</code> subclause, where <i>col-list</i> can be a single field or a comma-separated list of fields.
<i>expression</i>	Any valid expression. Usually the name of a column that contains the data values to be summed.
<code>%FOREACH(col-list)</code>	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on <code>%FOREACH</code> .
<code>%AFTERHAVING</code>	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

The **SUM aggregate function** returns the sum of the values of *expression*. Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query.

SUM can be used in a **SELECT** query or subquery that references either a table or a view. **SUM** can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

SUM cannot be used in a **WHERE** clause. **SUM** cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

SUM, like all aggregate functions, can take an optional **DISTINCT clause**. `SUM(DISTINCT col1)` totals only those `col1` field values that are distinct (unique). `SUM(DISTINCT BY(col2) col1)` totals only those `col1` field values in records where the `col2` values are distinct (unique). Note however that the distinct `col2` values may include a single `NULL` as a distinct value.

Data Values

SUM returns data type `INTEGER` for an *expression* with data type `INT`, `SMALLINT`, or `TINYINT`. **SUM** returns data type `BIGINT` for an *expression* with data type `BIGINT`. **SUM** returns data type `DOUBLE` for an *expression* with data type `DOUBLE`. For all other numeric data types, **SUM** returns data type `NUMERIC`.

SUM returns a value with a precision of 18. The scale of the returned value is the same as the *expression* scale, with the following exception. If *expression* is a numeric value with data type `VARCHAR` or `VARBINARY`, the scale of the returned value is 8.

By default, aggregate functions use Logical (internal) data values, rather than Display values.

SUM is normally applied to a field or expression that has a numeric value. Because only minimal type checking is performed, it is possible (though rarely meaningful) to invoke it for nonnumeric fields. **SUM** evaluates nonnumeric values, including the empty string ("), as zero (0). If *expression* is data type `VARCHAR`, the return value to ODBC or JDBC is of data type `DOUBLE`.

`NULL` values in data fields are ignored when deriving a **SUM** aggregate function value. If no rows are returned by the query, or the data field value for all rows returned is `NULL`, **SUM** returns `NULL`.

Optimization

SQL optimization of a **SUM** calculation can use a [bitslice index](#), if this index is defined for the field.

Changes Made During the Current Transaction

Like all aggregate functions, **SUM** always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to [SET TRANSACTION](#) and [START TRANSACTION](#).

Examples

In the following examples a dollar sign (\$) is concatenated to Salary amounts.

The following query returns the sum of the salaries of all employees in the Sample.Employee database:

```
SELECT '$' || SUM(Salary) AS Total_Payroll
FROM Sample.Employee
```

The following query uses `%AFTERHAVING` to return the sum of all salaries and the sum of salaries over \$80,000 for each state in which there is at least one person with a salary > \$80,000:

```
SELECT Home_State,
       '$' || SUM(Salary) AS Total_Payroll,
       '$' || SUM(Salary %AFTERHAVING) AS Exec_Payroll
FROM Sample.Employee
GROUP BY Home_State
HAVING Salary > 80000
ORDER BY Home_State
```

The following query returns the sum and the average of the salaries for each job title in the Sample.Employee database:

```
SELECT Title,
       '$' || SUM(Salary) AS Total,
       '$' || AVG(Salary) AS Average
FROM Sample.Employee
GROUP BY Title
ORDER BY Average
```

The following query shows **SUM** used with an arithmetic expression. For each job title in the Sample.Employee database it returns the sum of the current salaries and the sum of the salaries with a 10% increase in pay:

```
SELECT Title,
       '$' || SUM(Salary) AS BeforeRaises,
       '$' || SUM(Salary * 1.1) AS AfterRaises
FROM Sample.Employee
GROUP BY Title
ORDER BY Title
```

The following query shows **SUM** used with a logical expression using the **CASE** statement. It counts all of the salaried employees, and uses **SUM** to count all of the salaried employees earning \$90,000 or more.

```
SELECT COUNT(Salary) As AllPaid,
       SUM(CASE WHEN (Salary >= 90000)
             THEN 1 ELSE 0 END) As TopPaid
FROM Sample.Employee
```

See Also

- [Aggregate Functions](#) overview
- [AVG](#) aggregate function
- [COUNT](#) aggregate function

VARIANCE, VAR_SAMP, VAR_POP

Aggregate functions that return the statistical variance of a data set.

```
VARIANCE([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)]
[%AFTERHAVING])

VAR_SAMP([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)]
[%AFTERHAVING])

VAR_POP([ALL | DISTINCT [BY(col-list)]] expression [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Specifies that statistical variance functions return the variance of all values for <i>expression</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that statistical variance functions return the variance of the distinct (unique) <i>expression</i> values. DISTINCT can specify a <code>BY(col-list)</code> subclause, where <i>col-list</i> can be a single field or a comma-separated list of fields.
<i>expression</i>	Any valid expression. Usually the name of a column that contains the data values to be analyzed for variance.
<code>%FOREACH(col-list)</code>	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on <code>%FOREACH</code> .
<code>%AFTERHAVING</code>	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

These three variance [aggregate functions](#) return the statistical variance of the values of *expression*, after discarding NULL values. That is, the amount of variation from the mean value of the data set, expressed as a positive number. The larger the return value, the more variation there is within the data set of values. Caché SQL also supplies aggregate functions to return the [standard deviation](#) corresponding to each of these variance functions.

There are slight variations in how this statistical variation is derived:

- **VARIANCE:** Returns 0 if all of the values in the data set have the same value (no variability). Returns 0 if the data set consists of only one value (no possible variability). Returns NULL if the data set has no values.

The **VARIANCE** calculation is:

$$\frac{(\text{SUM}(\text{expression}^2) * \text{COUNT}(\text{expression})) - \text{SUM}(\text{expression})^2}{\text{COUNT}(\text{expression}) * (\text{COUNT}(\text{expression}) - 1)}$$

- **VAR_SAMP:** Sample variance. Returns 0 if all of the values in the data set have the same value (no variability). Returns NULL if the data set consists of only one value (no possible variability). Returns NULL if the data set has no values. Uses the same variant calculation as **VARIANCE**.
- **VAR_POP:** Population variance. Returns 0 if all of the values in the data set have the same value (no variability). Returns 0 if the data set consists of only one value (no possible variability). Returns NULL if the data set has no values.

The **VAR_POP** calculation is:

$$\frac{(\text{SUM}(\text{expression}^2) * \text{COUNT}(\text{expression})) - (\text{SUM}(\text{expression}))^2}{\text{COUNT}(\text{expression})^2}$$

These variance aggregate functions can be used in a [SELECT](#) query or subquery that references either a table or a view. They can appear in a **SELECT** list or **HAVING** clause alongside ordinary field values.

These variance aggregate functions cannot be used in a **WHERE** clause. They cannot be used in the **ON** clause of a **JOIN**, unless the **SELECT** is a subquery.

These variance aggregate functions return a value of data type NUMERIC with a precision of 36 and a scale of 17, unless *expression* is data type DOUBLE in which case the function returns data type DOUBLE.

These variance aggregate functions are normally applied to a field or expression that has a numeric value. They evaluate nonnumeric values, including the empty string ("), as zero (0).

These variance aggregate functions ignore NULL values in data fields. If no rows are returned by the query, or the data field value for all rows returned is NULL, they return NULL.

The statistical variance functions, like all aggregate functions, can take an optional [DISTINCT clause](#). `VARIANCE(DISTINCT col1)` returns the variance of those col1 field values that are distinct (unique). `VARIANCE(DISTINCT BY(col2) col1)` returns the variance of the col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Changes Made During the Current Transaction

Like all aggregate functions, the variance functions always returns the current state of the data, including uncommitted changes, regardless of the current transaction's isolation level. For further details, refer to [SET TRANSACTION](#) and [START TRANSACTION](#).

Examples

The following example uses **VARIANCE** to return the variance in the ages of the employees in Sample.Employee, and the variance in the distinct ages represented by one or more employees:

```
SELECT VARIANCE(Age) AS AgeVar, VARIANCE(DISTINCT Age) AS PerAgeVar
FROM Sample.Employee
```

The following example uses **VAR_POP** to return the population variance in the ages of the employees in Sample.Employee, and the variance in the distinct ages represented by one or more employees:

```
SELECT VAR_POP(Age) AS AgePopVar, VAR_POP(DISTINCT Age) AS PerAgePopVar
FROM Sample.Employee
```

See Also

- [Aggregate Functions](#) overview
- [AVG](#) aggregate function
- [COUNT](#) aggregate function
- [STDDEV](#), [STDDEV_SAMP](#), [STDDEV_POP](#) aggregate functions

XMLAGG

An aggregate function that creates a concatenated string of values.

```
XMLAGG([ALL | DISTINCT [BY(col-list)]] string-expr [%FOREACH(col-list)] [%AFTERHAVING])
```

Arguments

ALL	<i>Optional</i> — Specifies that XMLAGG returns a concatenated string of all values for <i>string-expr</i> . This is the default if no keyword is specified.
DISTINCT	<i>Optional</i> — A DISTINCT clause that specifies that XMLAGG returns a concatenated string containing only the unique <i>string-expr</i> values. DISTINCT can specify a BY(col-list) subclause, where <i>col-list</i> can be a single field or a comma-separated list of fields.
<i>string-expr</i>	An SQL expression that evaluates to a string. Commonly this is the name of a column from which to retrieve data.
%FOREACH(<i>col-list</i>)	<i>Optional</i> — A column name or a comma-separated list of column names. See SELECT for further information on %FOREACH.
%AFTERHAVING	<i>Optional</i> — Applies the condition found in the HAVING clause.

Description

The **XMLAGG** aggregate function returns a concatenated string of all values from *string-expr*. The return value is of data type VARCHAR, with a default length of 4096.

- A simple **XMLAGG** (or **XMLAGG ALL**) returns a string containing a concatenated string composed of all the values for *string-expr* in the selected rows. Rows where *string-expr* is NULL are ignored.

The following two examples both return the same single value, a concatenated string of all of the values listed in the Home_State column of the Sample.Person table.

```
SELECT XMLAGG(Home_State) AS All_State_Values
FROM Sample.Person
```

```
SELECT XMLAGG(ALL Home_State) AS ALL_State_Values
FROM Sample.Person
```

Note that this concatenated string contains duplicate values.

- An **XMLAGG DISTINCT** returns a concatenated string composed of all the distinct (unique) values for *string-expr* in the selected rows: **XMLAGG(DISTINCT col1)**. Rows where *string-expr* is NULL are ignored. **XMLAGG(DISTINCT BY(col2) col1)** returns a concatenated string containing only those col1 field values in records where the col2 values are distinct (unique). Note however that the distinct col2 values may include a single NULL as a distinct value.

Rows where *string-expr* is NULL are omitted from the return value. Rows where *string-expr* is the empty string (") are omitted from the return value if at least one non-empty string value is returned. If the only non-NULL *string-expr* values are the empty string ("), the return value is a single empty string.

XMLAGG does not support data stream fields. Specifying a stream field for *string-expr* results in an SQLCODE -37.

XML and XMLAGG

One common use of **XMLAGG** is to tag each data item from a column. This is done by combining **XMLAGG** and **XMLELEMENT** as shown in the following example:

```
SELECT XMLAGG(XMLELEMENT("para",Home_State))
FROM Sample.Person
```

This results in an output string such as the following:

```
<para>LA</para><para>MN</para><para>LA</para><para>NH</para><para>ME</para>...
```

XMLAGG and ORDER BY

The **XMLAGG** function concatenates values of a table column from multiple rows into a single string. Because an **ORDER BY** clause is applied to the query result set after all aggregate fields are evaluated, **ORDER BY** cannot directly affect the sequence of values within this string. Under certain circumstances, **XMLAGG** results may appear in sequential order, but this ordering should not be relied upon. The values listed within a given aggregate result value cannot be explicitly ordered.

Related Aggregate Functions

- **XMLAGG** returns a string of concatenated values.
- **LIST** returns a comma-separated list of values.
- **%DLIST** returns a Caché list containing an element for each value.
- **JSON_ARRAYAGG** returns a JSON array of values.

Examples

The following example creates a concatenated string of all of the distinct values found in the FavoriteColors column of the Sample.Person table. Thus every row has the same value for the All_Colors column. Note that while some rows have a NULL value for FavoriteColors, this value is not included in the concatenated string. Data values are returned in internal format.

```
SELECT Name,FavoriteColors,
       XMLAGG(DISTINCT FavoriteColors) AS All_Colors_In_Table
FROM Sample.Person
ORDER BY FavoriteColors
```

The following example returns concatenated strings of Home_State values that begin with “A”. It returns the distinct Home_State values (DISTINCT Home_State); the Home_State values corresponding to distinct Home_City values (DISTINCT BY(Home_City) Home_State), which may possibly including one unique NULL for Home_City; and all Home_State values:

```
SELECT XMLAGG(DISTINCT Home_State) AS DistStates,
       XMLAGG(DISTINCT BY(Home_City) Home_State) AS DistCityStates,
       XMLAGG(Home_State) AS AllStates
FROM Sample.Person
WHERE Home_State %STARTSWITH 'A'
```

The following example creates a concatenated string of all of the distinct values found in the Home_City column for each of the states. Every row from the same state contains a list of all of the distinct city values for that state:

```
SELECT Home_State, Home_City,
       XMLAGG(DISTINCT Home_City %FOREACH(Home_State)) AS All_Cities_In_State
FROM Sample.Person
ORDER BY Home_State
```

The following example uses the %AFTERHAVING keyword. It returns a row for each Home_State that contains at least one Name value that fulfills the HAVING clause condition (a name that begins with either “C” or “K”). The first **XMLAGG** function returns a concatenated string consisting of all of the names for that state. The second **XMLAGG** function returns a concatenated string consisting of only those names that fulfill the HAVING clause condition:

```

SELECT Home_State,
       XMLAGG(Name) AS AllNames,
       XMLAGG(Name %AFTERHAVING) AS HaveClauseNames
FROM Sample.Person
GROUP BY Home_State
HAVING Name LIKE 'C%' OR Name LIKE 'K%'
ORDER BY Home_state

```

For the following examples, suppose we have the following table, AutoClub:

Name	Make	Model	Year
Smith,Joe	Pontiac	Firebird	1971
Smith,Joe	Saturn	SW2	1997
Smith,Joe	Pontiac	Bonneville	1999
Jones,Scott	Ford	Mustang	1966
Jones,Scott	Mazda	Miata	2000

The query:

```

SELECT DISTINCT Name, XMLAGG(Make) AS String_Of_Makes
FROM AutoClub WHERE Name = 'Smith,Joe'

```

returns:

Name	String_Of_Makes
Smith,Joe	PontiacSaturnPontiac

The query:

```

SELECT DISTINCT Name, XMLAGG(DISTINCT Make) AS String_Of_Makes
FROM AutoClub WHERE Name = 'Smith,Joe'

```

returns:

Name	String_Of_Makes
Smith,Joe	PontiacSaturn

See Also

- [Aggregate Functions](#) overview
- [%DLIST](#) aggregate function
- [JSON_ARRAYAGG](#) aggregate function
- [LIST](#) aggregate function
- [XMLELEMENT](#) function
- [SELECT](#) statement

SQL Functions

ABS

A numeric function that returns the absolute value of a numeric expression.

```
ABS(numeric-expression)
{fn ABS(numeric-expression)}
```

Arguments

<i>numeric-expression</i>	A number whose absolute value is to be returned.
---------------------------	--

Description

ABS returns the absolute value, which is always zero or a positive number. **ABS** returns the same data type as *numeric-expression*. If *numeric-expression* is not a number (for example, the string 'abc', or the empty string '') **ABS** returns 0. **ABS** returns <null> when passed a NULL value.

Note that **ABS** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

This function can also be invoked from ObjectScript using the **ABS()** method call:

```
WRITE $SYSTEM.SQL.ABS(-0099)
```

Examples

The following example shows the two forms of **ABS**:

```
SELECT ABS(-99) AS AbsGen, {fn ABS(-99)} AS AbsODBC
```

both returns 99.

The following examples show how **ABS** handles some other numbers. Caché SQL converts *numeric-expression* to [canonical form](#), deleting leading and trailing zeros and evaluating exponents, before invoking **ABS**.

```
SELECT ABS(007) AS AbsoluteValue
```

returns 7.

```
SELECT ABS(-0.000) AS AbsoluteValue
```

returns 0.

```
SELECT ABS(-99E4) AS AbsoluteValue
```

returns 990000.

```
SELECT ABS(-99E-4) AS AbsoluteValue
```

returns .0099.

See Also

- SQL functions: [CONVERT TO_NUMBER](#)
- ObjectScript function: [\\$ZABS](#)

ACOS

A scalar numeric function that returns the arc-cosine, in radians, of a given cosine.

```
{fn ACOS(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type float or real, whose value is between -1 and 1. This is the cosine of the angle.
-------------------------	--

Description

ACOS takes a numeric value and returns the inverse (arc) of its cosine as a floating point number. The value of *float-expression* must be a signed decimal number ranging from 1 to -1 (inclusive). A number outside of this range causes a runtime error, generating an SQLCODE -400 (fatal error occurred). **ACOS** returns NULL if passed a NULL value. **ACOS** treats nonnumeric strings, including the empty string ("), as the numeric value 0.

ACOS returns a value of data type FLOAT with a precision of 19 and a scale of 18.

ACOS can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the [DEGREES](#) function to convert radians to degrees. You can use the [RADIANS](#) function to convert degrees to radians.

Examples

The following examples show the effect of **ACOS** on two cosines:

```
SELECT {fn ACOS(0.52)} AS ArcCosine
```

returns 1.023945...

```
SELECT {fn ACOS(-1)} AS ArcCosine
```

returns pi (3.14159...).

See Also

- SQL functions: [ASIN](#) [ATAN](#) [COS](#) [COT](#) [SIN](#) [TAN](#)
- ObjectScript function: [\\$ZARCCOS](#)

%ALPHAUP

Deprecated. A collation function that converts alphabetic characters to the ALPHAUP collation format.

```
%ALPHAUP(expression)
%ALPHAUP expression
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
-------------------	--

Description

This is a deprecated collation function. Please refer to [%SQLUPPER](#) for new development. **%SQLUPPER** provides superior handling for non-alphabetic characters.

%ALPHAUP converts *expression* to the ALPHAUP format:

- Converts all letters to uppercase.
- Removes all punctuation characters, except the comma and question mark. ([%STRING](#) removes all punctuation characters, except the comma.)
- Removes all blank spaces (leading, trailing, and embedded).

%ALPHAUP (unlike **%SQLUPPER** and **%STRING**) does not force numerics to be interpreted as a string. SQL converts a numeric to [canonical form](#) (removing leading and trailing zeros, expanding exponents, etc.) before passing the numeric to the function. (SQL does not convert numeric strings to canonical form.) **%ALPHAUP** then removes the period (used as the decimal separator character in many locales) and the minus sign, as well as other punctuation and blanks. For this reason, **%ALPHAUP** must be used with caution on any expression containing non-alphabetic information.

%ALPHAUP is a Caché SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",6)
```

This function can also be invoked from ObjectScript using the **ALPHAUP()** method call:

```
WRITE $SYSTEM.SQL.ALPHAUP("The quick, BROWN fox.")
```

Alphanumeric Collation Order

The case conversion functions collate data values that begin with a number using different algorithms, as follows:

%ALPHAUP and %STRING	%SQLUPPER, %SQLSTRING, and all other case conversion functions
5988 Clinton Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 641 First Place, 6572 First Avenue, 6643 First Street, 665 Ash Drive, 66 Main Street, 672 Main Court, 6754 Oak Court, 6986 Madison Blvd, 6 Oak Avenue, 7000 Ash Court, 709 Oak Avenue	5988 Clinton Avenue, 6 Oak Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 641 First Place, 6572 First Avenue, 66 Main Street, 6643 First Street, 665 Ash Drive, 672 Main Court, 6754 Oak Court, 6986 Madison Blvd, 7000 Ash Court, 709 Oak Avenue

Examples

In the following example, **%ALPHAUP** is used to convert the Name field to uppercase, so that the contains operator (I) can test for the letter “Y”. This query returns all names in Sample.Person that contain the letter “Y”, either uppercase or lowercase:

```
SELECT Name FROM Sample.Person
WHERE %ALPHAUP(Name) [ 'Y'
```

The following embedded SQL example shows how **%ALPHAUP** can be used with Unicode alphabetic characters, in this case Greek letters:

```
IF $SYSTEM.Version.IsUnicode() {
SET greek=$CHAR(952,945,955,945,963,963,945)
WRITE !,"lowercase Greek: ",greek
&sql(SELECT %ALPHAUP(:greek)
      INTO :capgreek
      FROM Sample.Person)
WRITE !,"uppercase Greek: ",capgreek
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

(Note that the above example requires a Unicode installation of Caché.)

The following cautionary examples show how **%ALPHAUP** may treat as identical strings that are in fact quite different:

```
SELECT %ALPHAUP('Max Wells'),%ALPHAUP('Maxwell S.'),
      %ALPHAUP('Release 3.2'),%ALPHAUP('Re: Lease 32'),
      %ALPHAUP('12/2/04'),%ALPHAUP('1/22/04'),
      %ALPHAUP('-36.5 degrees'),%ALPHAUP('365 Degrees')
```

See Also

[%SQLUPPER](#)

[Collation](#) chapter in *Using Caché SQL*

ASCII

A string function that returns the integer ASCII code value of the first (leftmost) character of a string expression.

```
ASCII(string-expression)
{fn ASCII(string-expression)}
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR). A string expression of type CHAR or VARCHAR.
--------------------------	---

Description

ASCII returns NULL if passed a NULL or an empty string value. The returning of NULL for empty string is consistent with SQL Server.

Note that **ASCII** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Examples

The following examples both returns 90, which is the ASCII value of the character Z:

```
SELECT ASCII('Z') AS AsciiCode
```

```
SELECT {fn ASCII('ZEBRA')} AS AsciiCode
```

Caché SQL converts numerics to [canonical form](#) before performing **ASCII** conversion. The following example returns 55, which is the ASCII value of the number 7:

```
SELECT ASCII(+007) AS AsciiCode
```

This number parsing is not done if the numeric is presented as a string. The following example returns 43, which is the ASCII value of the plus (+) character:

```
SELECT ASCII('+007') AS AsciiCode
```

See Also

- SQL functions: [CHAR](#)
- ObjectScript functions: [\\$ASCI](#) [\\$ZLASCII](#) [\\$ZWASCII](#)

ASIN

A scalar numeric function that returns the arc-sine, in radians, of the sine of an angle.

```
{fn ASIN(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type float, whose value is between -1 and 1. This is the sine of the angle.
-------------------------	--

Description

ASIN returns the inverse (arc) of the sine of an angle as a floating point number. The value of *float-expression* must be a signed decimal number ranging from 1 to -1 (inclusive). A number outside of this range causes a runtime error, generating an SQLCODE -400 (fatal error occurred). **ASIN** returns NULL if passed a NULL value. **ASIN** treats nonnumeric strings, including the empty string ("), as the numeric value 0.

ASIN returns a value of data type FLOAT with a precision of 19 and a scale of 18.

ASIN can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the [DEGREES](#) function to convert radians to degrees. You can use the [RADIANS](#) function to convert degrees to radians.

Examples

The following examples show the effect of **ASIN** on two sines.

```
SELECT {fn ASIN(0.52)} AS ArcSine
```

returns 0.5468509506...

```
SELECT {fn ASIN(-1.00)} AS ArcSine
```

returns -1.5707963267...

See Also

- SQL functions: [ACOS](#) [ATAN](#) [COS](#) [COT](#) [SIN](#) [TAN](#)
- ObjectScript function: [\\$ZARCSIN](#)

ATAN

A scalar numeric function that returns the arc-tangent, in radians, of the tangent of an angle.

```
{fn ATAN(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type float. This is the tangent of the angle.
-------------------------	--

Description

ATAN takes any numeric value and returns the inverse (arc) of the tangent of an angle as a floating point number. **ATAN** returns NULL if passed a NULL value. **ATAN** treats nonnumeric strings, including the empty string ("), as the numeric value 0.

ATAN returns a value of data type FLOAT with a precision of 36 and a scale of 18.

ATAN can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the [DEGREES](#) function to convert radians to degrees. You can use the [RADIANS](#) function to convert degrees to radians.

Example

The following example shows the effect of **ATAN**:

```
SELECT {fn ATAN(0.52)} AS ArcTangent
```

returns 0.47951929199...

See Also

- SQL functions: [ACOS](#) [ASIN](#) [COS](#) [COT](#) [SIN](#) [TAN](#)
- ObjectScript function: [\\$ZARCTAN](#)

CAST

A function that converts a given expression to a specified data type.

```
CAST(expr AS CHAR | CHARACTER | VARCHAR | NCHAR | NVARCHAR)
CAST(expr AS CHAR(n) | CHARACTER(n) | VARCHAR(n))
CAST(expr AS CHAR VARYING | CHARACTER VARYING)
CAST(expr AS INT | INTEGER | BIGINT | SMALLINT | TINYINT)
CAST(expr AS DEC | DECIMAL | NUMERIC)
CAST(expr AS DEC(p[,s]) | DECIMAL(p[,s]) | NUMERIC(p[,s])
CAST(expr AS FLOAT | REAL)
CAST(expr AS DOUBLE)
CAST(expr AS MONEY | SMALLMONEY)
CAST(expr AS DATE)
CAST(expr AS TIME)
CAST(expr AS TIMESTAMP | DATETIME | SMALLDATETIME)
CAST(expr AS BIT)
CAST(expr AS BINARY | BINARY VARYING | VARBINARY)
CAST(expr AS BINARY(n) | BINARY VARYING(n) | VARBINARY(n))
CAST(expr AS GUID)
```

Arguments

<i>expr</i>	An SQL expression.
<i>n</i>	An integer, indicating the maximum number of characters to return.
<i>p,s</i>	<i>Optional</i> — <i>p</i> =Precision (maximum number of total digits), expressed as an integer. <i>s</i> =Scale (maximum number of decimal digits), expressed as an integer. If scale is not specified, it defaults to 15.

Description

The SQL **CAST** function converts the data type of an expression to another data type.

You can cast an expression to any of the following types:

- **CHAR** or **CHARACTER**: represent a numeric or a string by its initial character. **VARCHAR** with no *n* defaults to a length of 30 characters when specified to **CAST** or **CONVERT**. Otherwise, the **VARCHAR** data type (with no specified size) is mapped to a **MAXLEN** of 1 character, as shown in the [Data Types](#) table. **NCHAR** is equivalent to **CHAR**; **NVARCHAR** is equivalent to **VARCHAR**.
- **CHAR(n)**, **CHARACTER(n)**, or **VARCHAR(n)**: represent a numeric or a string by the number of characters specified by *n*.
- **CHAR VARYING** or **CHARACTER VARYING**: represent a numeric or a string by the number of characters in the original value.
- **INT**, **INTEGER**, **BIGINT**, **SMALLINT**, and **TINYINT**: represent a numeric by its integer portion. Decimal digits are truncated.
- **DEC**, **DECIMAL**, and **NUMERIC**: represent a numeric by the number of digits in the original value. Converts using the Caché **\$DECIMAL** function, which converts **\$DOUBLE** values to **\$DECIMAL** values. The *p* (precision), if specified, is retained as part of the defined data type, but does not affect the value returned by **CAST**. If you specify a *s* (scale) value of a positive integer, the decimal value is rounded to the specified number of digits. (The appropriate number of trailing zeros are included for Display mode, but are truncated for Logical mode and ODBC mode.) If you specify *s*=0, the numeric value is rounded to an integer. If you specify *s*=-1, the numeric value is truncated to an integer.
- **FLOAT** and **REAL**: represent a numeric by the number of digits in the original value, with a precision of 18 and a scale of 9.

- **DOUBLE** represents the IEEE floating point standard. For further details, refer to the ObjectScript [\\$DOUBLE](#) function.
- **MONEY** and **SMALLMONEY** are currency numeric data types. The scale for currency data types is always 4.
- **DATE**: represents a date. Dates can be represented in any of the following formats, depending on context: the display date format for your locale (for example, MM/DD/YYYY); the ODBC date format (YYYY-MM-DD); or the \$HOROLOG integer date storage format (nnnnn).
- **TIME**: represents a time. Times can be represented in any of the following formats, depending on context: the display time format for your locale (for example, hh:mm:ss); the ODBC date format (hh:mm:ss); or the \$HOROLOG integer time storage format (nnnnn).
- **TIMESTAMP**, **DATETIME**, and **SMALLDATETIME**: represents a date and time stamp with the format YYYY-MM-DD hh:mm:ss. This corresponds to the ObjectScript [\\$ZTIMESTAMP](#) special variable.
- **BIT** represents a single binary value.
- **BINARY**, **BINARY VARYING**, and **VARBINARY** represent a value of data type %Library.Binary (xDBC data type **BINARY**). The optional *n* length defaults to 1 for **BINARY**, 30 for **BINARY VARYING** and **VARBINARY**. No conversion of the data is actually performed when casting to a binary value. Caché does truncate the length of the value at the specified *n* length.
- **GUID** represents a 36-character value of data type %Library.UniqueIdentifier. If you supply an *expr* longer than 36 characters, **CAST** returns the first 36 characters of *expr*.

For a list of the data types supported by Caché SQL, see [Data Types](#). For other data type conversions, refer to the [CONVERT](#) function. If you specify a **CAST** with an unsupported data type, Caché issues an SQLCODE -376.

Casting Numerics

A numeric value can be cast to a numeric data type or to a character data type.

When casting a numeric results in a shortened value, the numeric is truncated, not rounded. For example, casting 98.765 to **INT** returns 98, to **CHAR** returns 9, and to **CHAR(4)** returns 98.7. Note that casting a negative number to **CHAR** returns just the negative sign, and casting a fractional number to **CHAR** returns just the decimal point.

A numeric value can consist of the digits 0 through 9, a decimal point, one or more leading signs (+ or –), and the exponent sign (the letter E or e) followed by, at most, one + or – sign. A numeric cannot contain group separator characters (commas). For further details, see the [literals](#) section of “Language Elements” in *Using Caché SQL*.

Before a cast is performed, Caché SQL resolves a numeric to its canonical form: Exponentiation is performed. Caché strips leading and trailing zeros, a leading plus sign, and a trailing decimal point. Multiple signs are resolved before casting a numeric. However, SQL treats double negative signs as a [comment](#) indicator; encountering double negative signs in a number results in Caché processing the remainder of that line of code as a comment.

A Caché floating point number can take a **DEC**, **DECIMAL**, **NUMERIC**, or **FLOAT** data type. The **DOUBLE** data type represents floating point numbers according to the IEEE floating point standard. The Caché floating point data types have greater precision than the **DOUBLE** data type, and are preferable for most applications. You cannot use **CAST** to cast a floating point number to the **DOUBLE** data type; instead, use the ObjectScript [\\$DOUBLE](#) function.

When a numeric value is cast to a date or time data type, it displays in SQL as zero (0); however, when a numeric cast as a date or time is passed out of embedded SQL to ObjectScript, it displays as the corresponding \$HOROLOG value.

Casting Character Strings

You can cast a character string to another character data type, returning either a single character, the first *n* characters, or the entire character string.

Before a cast is performed, Caché SQL resolves embedded quote characters ('can't'=can't) and string concatenation ('can||'not'=cannot). Leading and trailing blanks are retained.

When a character string is cast to a numeric type, it always returns the single digit zero (0).

You can cast a character string to the DATE, TIME, or TIMESTAMP data type. The following operations result in a valid value:

- **DATE:** A string of the format 'yyyy-mm-dd' can be cast to DATE. This string format corresponds to ODBC date format. Value and range checking are performed; the input date value must be a valid date. Missing leading zeros in month and day fields are added. In SQL, this cast displays as the locale's date display format. For example, '2004-11-23' might display as '11/23/2004'. In embedded SQL, this cast is returned as the corresponding \$HOROLOG date integer. An invalid ODBC date or a non-numeric string is represented as 0 in logical mode when cast to DATE; date 0 is displayed as 1840-12-31.
- **TIME:** A string of the format 'hh:mm:ss' or 'hh:mm:ss.nn' can be cast to TIME. This string format corresponds to ODBC time format. Value and range checking are performed. Missing leading zeros are added. In embedded SQL, this cast is returned as the corresponding \$HOROLOG time integer. An invalid ODBC time or a non-numeric string is represented as 0 in logical mode when cast to TIME; time 0 is displayed as 00:00:00.
- **TIMESTAMP:** A string consisting of a valid date and time, a valid date, or a valid time can be cast to TIMESTAMP. The date portion can be in a variety of formats. A missing date portion defaults to 1841-01-01. A missing time portion defaults to 00:00:00. Two-digit years 00 through 49 are converted to 21st century years; two-digit years 50 through 99 are converted to 20th century years. Missing leading zeros are added. Fractional seconds (if specified) can be preceded by either a period (.) or a colon (:). The symbols have different meanings. A period indicates a standard fraction; thus 12:00:00.4 indicates four-tenths of a second, and 12:00:00.004 indicates four-thousandth of a second. A colon indicates that what follows is in thousandths of a second; thus 12:00:00:4 indicates four-thousandth of a second. The permitted number of digits following a colon is limited to three.

Casting NULL and the Empty String

NULL can be cast to any data type and returns NULL.

The empty string (' ') casts as follows:

- All character data types return NULL.
- All numeric data types return 0 (zero), with the appropriate number of trailing fractional zeros. The DOUBLE data type returns zero with no trailing fractional zeros.
- The DATE data type returns 12/31/1840.
- The TIME data type returns 00:00:00.
- The TIMESTAMP, DATETIME, and SMALLDATETIME data types return NULL.
- The BIT data type returns 0.
- All binary data types return NULL.

Casting Dates

You can cast a date to a date data type, to a numeric data type, or to a character data type.

Casting a date to the TIMESTAMP, DATETIME, or SMALLDATETIME data type returns a timestamp with the format YYYY-MM-DD hh:mm:ss. Since a date does not have a time portion, the time portion of the resulting timestamp is always 00:00:00. If the *expr* value is not a valid date in the locale's date display format, **CAST** returns NULL.

The following Dynamic SQL example casts a field of DATE data type to TIMESTAMP:

```
ZNSPACE "SAMPLES"
SET myquery=2
SET myquery(1)="SELECT TOP 5 DOB,CAST(DOB AS TIMESTAMP) AS TStamp,"
SET myquery(2)="FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example casts a field of **TIMESTAMP** data type to **DATE**:

```
ZNSPACE "SAMPLES"
SET myquery=2
SET myquery(1)="SELECT TOP 5 EventDate,CAST(EventDate AS DATE) AS Horolog,"
SET myquery(2)="FROM Aviation.Event"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Casting a date to a numeric data type returns the **\$HOROLOG** value for the date. This is an integer value representing the number of days since Dec. 31, 1840.

Casting a date to a character data type returns either the complete date, or as much of the date as the length of the data type permits. However, the display format is not the same for all character data types. The **CHAR VARYING** and **CHARACTER VARYING** data types return the complete date in display format. For example, if a date displays as **MM/DD/YYYY**, these data types return the date as a character string with the same format. The other character data types return the date (or a part thereof) as a character string in **ODBC** date format. For example, if a date displays as **mm/dd/yyyy**, these data types return the date as a character string with the format **YYYY-MM-DD**. Thus for the date **04/24/2004**, the **CHAR** data type returns '2' (the first character of the year), and a **CHAR(8)** returns '2004-04-'

Casting a Bit Value

You can cast an *expr* value **AS BIT** to return a 0 or 1. If *expr* is 1 or any other non-zero numeric value, it returns 1. If *expr* is "TRUE", "True", or "true", it returns 1. (The word "True" can be represented in any combination of uppercase and lowercase, but *cannot* be abbreviated as "T".) If *expr* is any other non-numeric value, it returns 0. If *expr* is 0, it returns 0.

In the following example, the first five **CAST** operations return 1, the second five **CAST** operations return 0:

```
SELECT CAST(1 AS BIT) AS One,
       CAST(7 AS BIT) AS Num,
       CAST(743.6 AS BIT) AS Frac,
       CAST(0.3 AS BIT) AS Zerofrac,
       CAST('tRuE' AS BIT) AS TrueWord,
       CAST(0 AS BIT) AS Zero,
       CAST('FALSE' AS BIT) AS FalseWord,
       CAST('T' AS BIT) AS T,
       CAST('F' AS BIT) AS F,
       CAST(0.0 AS BIT) AS Zerodot
```

Examples

The following example uses the **CAST** function to present an average as an integer, not a floating point. Note that the **CAST** truncates the number, rather than rounding it:

```
SELECT DISTINCT AVG(Age) AS AvgAge,
       CAST(AVG(Age) AS INTEGER) AS IntAvgAge
FROM Sample.Person
```

The following example shows how the **CAST** function converts pi (a floating point number) to different numeric data types:

```
SELECT
  CAST({fn PI()} As INTEGER) As IntegerPi,
  CAST({fn PI()} As SMALLINT) As SmallIntPi,
  CAST({fn PI()} As DECIMAL) As DecimalPi,
  CAST({fn PI()} As NUMERIC) As NumericPi,
  CAST({fn PI()} As FLOAT) As FloatPi,
  CAST({fn PI()} As DOUBLE) As DoublePi
```

Note in the following example that the precision and scale values are parsed, but do not change the value returned by **CAST**:

```
SELECT
  CAST({fn PI()} As DECIMAL) As DecimalPi,
  CAST({fn PI()} As DECIMAL(6,3)) As DecimalPSPi
```

The following example shows how the **CAST** function converts pi (a floating point number) to different character data types:

```
SELECT
  CAST({fn PI()} As CHAR) As CharPi,
  CAST({fn PI()} As CHAR(4)) As CharNPi,
  CAST({fn PI()} As CHAR VARYING) As CharVaryingPi,
  CAST({fn PI()} As VARCHAR(4)) As VarCharNPi
```

The following example shows how the **CAST** function converts Name (a character string) to different character data types:

```
SELECT DISTINCT
  CAST(Name As CHAR) As CharName,
  CAST(Name As CHAR(4)) As CharNName,
  CAST(Name As CHAR VARYING) As CharVaryingName,
  CAST(Name As VARCHAR(4)) As VarCharNName
FROM Sample.Person
```

The following example shows what happens when you use the **CAST** function to convert Name (a character string) to different numeric data types. In every case, the value returned is 0 (zero):

```
SELECT DISTINCT
  CAST(Name As INT) As IntName,
  CAST(Name As SMALLINT) As SmallIntName,
  CAST(Name As DEC) As DecName,
  CAST(Name As NUMERIC) As NumericName,
  CAST(Name As FLOAT) As FloatName
FROM Sample.Person
```

The following example casts a date field (DOB) to a numeric data type and several character data types. Casting a date to a numeric returns the **\$HOROLOG** integer equivalent. Casting a date to a character data type returns either a date string in input format (CHAR VARYING or CHARACTER VARYING) or the date (partial or full) in ODBC date string format:

```
SELECT DISTINCT DOB,
  CAST(DOB As INT) AS IntDate,
  CAST(DOB As CHAR) AS CharDate,
  CAST(DOB As CHAR(6)) AS CharNDate,
  CAST(DOB As CHAR VARYING) AS CharVaryDate,
  CAST(DOB As VARCHAR(10)) AS VarCharNDate
FROM Sample.Person
```

The following example casts character strings to the DATE and TIME data types:

```
SELECT CAST('1936-11-26' As DATE) AS StringToDate,
  CAST('14:33:45.78' AS TIME) AS StringToTime
```

Only a string with the format YYYY-MM-DD can be converted to a date. Strings with other formats return 0. Note that fractional seconds are truncated (not rounded) when converting a string to the TIME data type.

The following example casts a date to the TIMESTAMP data type:

```
SELECT DISTINCT DOB,
  CAST(DOB As TIMESTAMP) AS DateToTstamp
FROM Sample.Person
```

The resulting timestamp is in the format: YYYY-MM-DD hh:mm:ss.

The following example casts a character string to the `TIME` data type, then casts the resulting time to the `TIMESTAMP` data type:

```
SELECT CAST(CAST('14:33:45.78' AS TIME) AS TIMESTAMP) AS TimeToTstamp
```

The resulting timestamp is in the format: `YYYY-MM-DD hh:mm:ss`. The time portion is supplied by the nested `CAST`; the date portion is the current system date.

See Also

- [Data type, CONVERT](#)
- [TO_CHAR, TO_DATE, TO_NUMBER, TO_TIMESTAMP](#)

CEILING

A numeric function that returns the smallest integer greater than or equal to a given numeric expression.

```
CEILING(numeric-expression)
{fn CEILING(numeric-expression)}
```

Arguments

<i>numeric-expression</i>	A number whose ceiling is to be calculated.
---------------------------	---

Description

CEILING returns the nearest integer value greater than or equal to *numeric-expression*. The returned value has the same data type as *numeric-expression* and a scale of 0. When *numeric-expression* is a NULL value, an empty string ("), or any nonnumeric string, **CEILING** returns NULL.

Note that **CEILING** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

This function can also be invoked from ObjectScript using the **CEILING()** method call:

```
$SYSTEM.SQL.CEILING(numeric-expression)
```

Examples

The following examples show how **CEILING** converts a fraction to its ceiling integer:

```
SELECT CEILING(167.111) AS CeilingNum1,
       CEILING(167.456) AS CeilingNum2,
       CEILING(167.999) AS CeilingNum3
```

all return 168.

```
SELECT {fn CEILING(167.00)} AS CeilingNum1,
       {fn CEILING(167.00)} AS CeilingNum2
```

return 167.

```
SELECT CEILING(-167.111) AS CeilingNum1,
       CEILING(-167.456) AS CeilingNum2,
       CEILING(-167.999) AS CeilingNum3
```

all return -167.

```
SELECT CEILING(-167.00) AS CeilingNum
```

returns -167.

The following example uses a subquery to reduce a large table of US Zip Codes (postal codes) to one representative city for each ceiling Latitude integer:

```
SELECT City,State,CEILING(Latitude) AS CeilingLatitude
FROM (SELECT City,State,Latitude,CEILING(Latitude) AS CeilingNum
      FROM Sample.USZipCode)
GROUP BY CeilingNum
ORDER BY CeilingNum DESC
```

See Also

- [FLOOR](#)
- [ROUND](#)

CHAR

A string function that returns the character that has the ASCII code value specified in a string expression.

```
CHAR(code-value)  
{fn CHAR(code-value)}
```

Arguments

<i>code-value</i>	An integer code that corresponds to a character.
-------------------	--

Description

CHAR returns the character that corresponds to the specified integer code value. On Unicode systems, you can specify the integer code for any Unicode character, 0 through 65535. **CHAR** returns NULL if *code-value* is a integer that exceeds the permissible range of values.

CHAR returns an empty string (") if *code-value* is a nonnumeric string. **CHAR** returns NULL if passed a NULL value.

Note that **CHAR** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Examples

The following examples both return the character Z:

```
SELECT CHAR(90) AS CharCode  
  
SELECT {fn CHAR(90)} AS CharCode
```

The following example returns the Greek letter lambda:

```
IF $SYSTEM.Version.IsUnicode() {  
  &sql(SELECT {fn CHAR(955)}  
    INTO :greeklet)  
  WRITE !,"Greek letter: ",greeklet  
}  
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

Note that the above example requires a Unicode installation of Caché.

See Also

- SQL functions: [ASCII](#) [CHAR_LENGTH](#) [CHARACTER_LENGTH](#)
- ObjectScript functions: [\\$CHAR](#) [\\$ZLCHAR](#) [\\$ZWCHAR](#)

CHARACTER_LENGTH

A function that returns the number of characters in an expression.

```
CHARACTER_LENGTH(expression)
```

Arguments

<i>expression</i>	An expression, which can be the name of a column, a string literal, or the result of another scalar function. The underlying data type can be a character type (such as CHAR or VARCHAR), a numeric, or a data stream.
-------------------	--

Description

CHARACTER_LENGTH returns an integer value representing the number of characters, not the number of bytes, in the specified *expression*. The *expression* can be a string, or any other data type such as a numeric or a [data stream field](#). This integer count returned including leading and trailing blanks and the string-termination character. **CHARACTER_LENGTH** returns NULL if passed a NULL value, and 0 if passed an empty string (") value.

Numbers are parsed to [canonical form](#) before counting the characters; quoted number strings are not parsed. In the following example, the first **CHARACTER_LENGTH** returns 1 (because number parsing removes leading and trailing zeros), the second **CHARACTER_LENGTH** returns 8.

```
SELECT CHARACTER_LENGTH(007.0000) AS NumLen,
       CHARACTER_LENGTH('007.0000') AS NumStringLength
```

Note: The **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** functions are identical. All of them accept a [stream field](#) argument. The **LENGTH** and **\$LENGTH** functions do not accept a stream field argument.

LENGTH also differs from these functions by stripping trailing blanks and the string-termination character before counting characters. **\$LENGTH** also differs from these functions because it returns 0 if passed a NULL value, and 0 if passed an empty string.

Examples

The following example returns the number of characters in the state abbreviation field (Home_State) in the Sample.Employee table. (All U.S. states have a two-letter postal abbreviation):

```
SELECT DISTINCT CHARACTER_LENGTH(Home_State) AS StateLength
       FROM Sample.Employee
```

The following example returns the names of the employees and the number of characters in each employee name, ordered by ascending number of characters:

```
SELECT Name,
       CHARACTER_LENGTH(Name) AS NameLength
       FROM Sample.Employee
       ORDER BY NameLength
```

The following examples return the number of characters in a character stream field (Notes) and a binary stream field (Picture) in the Sample.Employee table:

```
SELECT DISTINCT CHARACTER_LENGTH(Notes) AS NoteLen
       FROM Sample.Employee WHERE Notes IS NOT NULL
```

```
SELECT DISTINCT CHARACTER_LENGTH(Picture) AS PicLen
       FROM Sample.Employee WHERE Picture IS NOT NULL
```

The following Embedded SQL example demonstrates how **CHARACTER_LENGTH** handles Unicode characters. **CHARACTER_LENGTH** counts the number of characters, regardless of their byte length.

```
IF $SYSTEM.Version.IsUnicode() {  
  SET a=$CHAR(960)_"FACE"  
  WRITE !,a  
  &sql(SELECT CHARACTER_LENGTH(:a) INTO :b)  
  IF SQLCODE'=0 {WRITE !,"Error code ",SQLCODE }  
  ELSE {WRITE !,"The CHARACTER length is ",b }  
}  
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

returns 5.

See Also

- SQL functions: [CHAR](#), [CHAR_LENGTH](#), [DATALENGTH](#), [LENGTH](#), [LEN](#), [\\$LENGTH](#)
- ObjectScript function: [\\$LENGTH](#)

CHARINDEX

A string function that returns the position of a substring within a string, with optional search start point.

```
CHARINDEX(substring, string[, start])
```

Arguments

<i>substring</i>	A substring to match within <i>string</i> .
<i>string</i>	A string expression that is the target for the substring search.
<i>start</i>	<i>Optional</i> — The starting point for substring search, specified as a positive integer. A character count from the beginning of <i>string</i> , counting from 1. To search from the beginning of <i>string</i> , omit this argument or specify a <i>start</i> of 0 or 1. A negative number, the empty string, NULL, or a nonnumeric value is treated as 0.

Description

CHARINDEX searches a string for a substring. If a match is found, it returns the starting position of the first matching substring, counting from 1. If the substring cannot be found, **CHARINDEX** returns 0.

The empty string is a string value. You can, therefore, use the empty string for either string argument value. The *start* argument treats an empty string value as 0. However, note that the ObjectScript empty string is passed to Caché SQL as NULL.

NULL is not a string value in Caché SQL. For this reason, specifying NULL for either **CHARINDEX** string argument returns NULL.

CHARINDEX is case-sensitive. Use one of the case-conversion functions to locate both uppercase and lowercase instances of a letter or character string.

This function provides compatibility with Transact-SQL implementations.

CHARINDEX, POSITION, \$FIND, and INSTR

CHARINDEX, **POSITION**, **\$FIND**, and **INSTR** all search a string for a specified substring and return an integer position corresponding to the first match. **CHARINDEX**, **POSITION**, and **INSTR** return the integer position of the first character of the matching substring. **\$FIND** returns the integer position of the first character after the end of the matching substring. **CHARINDEX**, **\$FIND**, and **INSTR** support specifying a starting point for substring search. **INSTR** also support specifying the substring occurrence from that starting point.

The following example demonstrates these four functions, specifying all optional arguments. Note that the positions of *string* and *substring* differ in these functions:

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

For a list of functions that search for a substring, refer to [String Manipulation](#).

Examples

The following example searches for the substring KONG. It returns 6, the character position of this substring within the string:

```
SELECT CHARINDEX('KONG','KING KONG')
```

The following example searches for all Name field values that contain the substring 'Fred':

```
SELECT Name
FROM Sample.Person
WHERE CHARINDEX('Fred',Name)>0
```

The following example matches a substring after the first 10 characters:

```
SELECT CHARINDEX('Re','Reduce, Reuse, Recycle',10)
```

it returns 16.

The following example specifies a *start* location beyond the length of the string:

```
SELECT CHARINDEX('Re','Reduce, Reuse, Recycle',99)
```

it returns 0.

The following example shows that **CHARINDEX** handles the empty string (") just like any other string value:

```
SELECT CHARINDEX('', 'King Kong'),
       CHARINDEX('K', ''),
       CHARINDEX('', '')
```

In the above example, the first and second **CHARINDEX** functions return 0 (no match). The third returns 1, because the empty string matches the empty string at position 1.

The following example shows that **CHARINDEX** does not treat NULL as a string value. Specifying NULL for either string always returns NULL:

```
SELECT CHARINDEX(NULL, 'King Kong'),
       CHARINDEX('K', NULL),
       CHARINDEX(NULL, NULL)
```

See Also

- [\\$FIND](#) function
- [INSTR](#) function
- [POSITION](#) function
- [String Manipulation](#)

CHAR_LENGTH

A function that returns the number of characters in an expression.

```
CHAR_LENGTH(expression)
```

Arguments

<i>expression</i>	An expression, which can be the name of a column, a string literal, or the result of another scalar function. The underlying data type can be a character type (such as CHAR or VARCHAR), a numeric, or a data stream.
-------------------	--

Description

CHAR_LENGTH returns an integer value representing the number of characters, not the number of bytes, in the specified *expression*. The *expression* can be a string, or any other data type such as a numeric or a [data stream field](#). This integer count returned including leading and trailing blanks and the string-termination character. **CHARACTER_LENGTH** returns NULL if passed a NULL value, and 0 if passed an empty string (") value.

Numbers are parsed to [canonical form](#) before counting the characters; quoted number strings are not parsed. In the following example, the first **CHAR_LENGTH** returns 1 (because number parsing removes leading and trailing zeros), the second **CHAR_LENGTH** returns 8.

```
SELECT CHAR_LENGTH(007.0000) AS NumLen,
       CHAR_LENGTH('007.0000') AS NumStringLength
```

Note: The **CHAR_LENGTH**, **CHARACTER_LENGTH**, and **DATALENGTH** functions are identical. All of them accept a [stream field](#) argument. The **LENGTH** and **\$LENGTH** functions do not accept a stream field argument.

LENGTH also differs from these functions by stripping trailing blanks and the string-termination character before counting characters. **\$LENGTH** also differs from these functions because it returns 0 if passed a NULL value, and 0 if passed an empty string.

Examples

The following example returns the number of characters in the state abbreviation field (Home_State) in the Sample.Employee table. (All U.S. states have a two-letter postal abbreviation):

```
SELECT DISTINCT CHAR_LENGTH(Home_State) AS StateLength
FROM Sample.Employee
```

The following example returns the names of the employees and the number of characters in each employee name, ordered by ascending number of characters:

```
SELECT Name,
       CHAR_LENGTH(Name) AS NameLength
FROM Sample.Employee
ORDER BY NameLength
```

The following examples return the number of characters in a character stream field (Notes) and a binary stream field (Picture) in the Sample.Employee table:

```
SELECT DISTINCT CHAR_LENGTH(Notes) AS NoteLen
FROM Sample.Employee WHERE Notes IS NOT NULL
```

```
SELECT DISTINCT CHAR_LENGTH(Picture) AS PicLen
FROM Sample.Employee WHERE Picture IS NOT NULL
```

The following Embedded SQL example shows how **CHAR_LENGTH** handles Unicode characters. **CHAR_LENGTH** counts the number of characters, regardless of their byte length:

```
IF $SYSTEM.Version.IsUnicode() {  
  SET a=$CHAR(960)_"FACE"  
  WRITE !,a  
  &sql(SELECT CHAR_LENGTH(:a) INTO :b)  
  IF SQLCODE'=0 {WRITE !,"Error code ",SQLCODE }  
  ELSE {WRITE !,"The CHAR length is ",b }  
}  
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

returns 5.

See Also

- SQL functions: [CHAR](#), [CHARACTER_LENGTH](#), [DATALENGTH](#), [LENGTH](#), [LEN](#), [\\$LENGTH](#)
- ObjectScript function: [\\$LENGTH](#)

COALESCE

A function that returns the value of the first expression that is not NULL.

```
COALESCE(expression, expression)
```

Arguments

<i>expression</i>	A series of expressions to be evaluated. Multiple expressions are specified as a comma-separated list. This expression list has a limit of 140 expressions.
-------------------	---

Description

The **COALESCE** function evaluates a list of expressions in left-to-right order and returns the value of the first non-NULL expression. If all expressions evaluate to NULL, NULL is returned.

Non-numeric expressions (such as strings or dates) must all be of the same data type, and return a value of that data type. Specifying expressions with incompatible data types results in an SQLCODE -378, and a %msg error message value. You can use the **CAST** function to convert an *expression* to a compatible data type.

Numeric expressions may be of different data types. If you specify numeric expressions with different data types, the data type returned is the *expression* data type most compatible with all of the possible result values, the data type with the highest [data type precedence](#). The following data types are compatible and are specified in order of precedence (highest to lowest): DOUBLE, NUMERIC, BIGINT, INTEGER, SMALLINT, TINYINT.

A string is returned unchanged; leading and trailing blanks are retained. A number is returned in canonical form, with leading and trailing zeros removed.

For further details on NULL handling, refer to the [NULL and the Empty String](#) section of “Language Elements” in *Using Caché SQL*.

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual [equality \(or non-equality\) condition comparison](#).

SQL Function	Comparison Test	Return Value
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1

Examples

The following Embedded SQL example takes a series of host variable values and returns the first (value *d*) that is not NULL. Note that the ObjectScript empty string ("") is translated as NULL in Caché SQL:

```
SET (a,b,c,e)=""
SET d="firstdata"
SET f="nextdata"
&sql(SELECT COALESCE(:a,:b,:c,:d,:e,:f) INTO :x)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The first non-null value is: ",x }
```

The following example compares the values of two columns in left-to-right order and returns the value of the first non-NULL column. The FavoriteColors column is NULL for some rows; the Home_State column is never NULL. For **COALESCE** to compare the two, FavoriteColors must be cast as a string:

```
SELECT TOP 25 Name,FavoriteColors,Home_State,
COALESCE(CAST(FavoriteColors AS VARCHAR),Home_State) AS CoalesceCol
FROM Sample.Person
```

The following Dynamic SQL example compares **COALESCE** to the other NULL-processing functions:

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT TOP 50 %ID,"_
              "IFNULL(FavoriteColors,'blank') AS Ifn2Col,"_
              "IFNULL(FavoriteColors,'blank','value') AS Ifn3Col,"_
              "COALESCE(CAST(FavoriteColors AS VARCHAR),Home_State) AS CoalesceCol,"_
              "ISNULL(FavoriteColors,'blank') AS IsnullCol,"_
              "NULLIF(FavoriteColors,$LISTBUILD('Orange')) AS NullifCol,"_
              "NVL(FavoriteColors,'blank') AS NvlCol"_
              " FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

See Also

- [CASE](#) command
- [IFNULL](#) function
- [ISNULL](#) function
- [NULLIF](#) function
- [NVL](#) function

CONCAT

A scalar string function that returns a character string as a result of concatenating two character expressions.

```
{fn CONCAT(string-expression1,string-expression2)}
```

Arguments

string-expression1,
string-expression2

The string expressions to be concatenated. The expressions can be the name of a column, a string literal, a numeric, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).

Description

CONCAT concatenates two strings to return a concatenated string. You can perform exactly the same operation using the concatenate operator (||).

You can concatenate any combination of numerics or numeric strings; the concatenation result is a numeric string. Caché SQL converts numerics to **canonical form** (exponents are expanded and leading and trailing zeros are removed) before concatenation. Numeric strings are not converted to canonical form before concatenation.

You can concatenate leading or trailing blanks to a string. Concatenating a NULL value to a string results in a NULL; this is the industry-wide SQL standard.

The **STRING** function can also be used to concatenate two or more expressions into a single string.

Examples

The following example concatenates the Home_State and Home_City columns to create a location value. The concatenation is shown twice, using the **CONCAT** function and the concatenate operator:

```
SELECT {fn CONCAT(Home_State,Home_City)} AS LocationFunc,  
Home_State||Home_City AS LocationOp  
FROM Sample.Person
```

The following example shows what happens when you attempt to concatenate a string and a NULL:

```
SELECT {fn CONCAT(Home_State,NULL)} AS StrNull  
FROM Sample.Person
```

The following example shows that numbers are converted to canonical form before concatenation. To avoid this, you can specify the number as a string, as shown:

```
SELECT {fn CONCAT(Home_State,0012.00E2)} AS StrNum,  
{fn CONCAT(Home_State,'0012.00E2')} AS StrStrNum  
FROM Sample.Person
```

The following example shows that trailing blank spaces are retained:

```
SELECT CHAR_LENGTH({fn CONCAT(Home_State,'          ')}) AS StrSpace  
FROM Sample.Person
```

See Also

[ASCII CHAR STRING SUBSTRING](#)

CONVERT

A function that converts a given expression to a specified data type.

```
CONVERT(datatype, expression[, format-code])
{fn CONVERT(expression, datatype)}
```

Arguments

<i>expression</i>	The expression to be converted.
<i>datatype</i>	The data type to which <i>expression</i> is to be converted.
<i>format-code</i>	<i>Optional</i> — An integer code that specifies date and time formats, used to convert between date/time/timestamp data types and character data types. This argument is only used with the general scalar syntax form.

Description

Two different implementations of the **CONVERT** function are described here. Both convert an expression in one data type to a corresponding value in another data type. Both perform date and time conversions.

Note: The arguments in these two implementations of **CONVERT** are presented in a different order. The first is a general Caché scalar function compatible with MS SQL Server, which takes three arguments. The second is a Caché ODBC scalar function with two arguments. These two forms of **CONVERT** are handled separately in the text that follows.

- **CONVERT(datatype,expression)** function supports conversion of [stream data](#). For example, you can convert the contents of a character stream field to a string of data type VARCHAR.
- **{fn CONVERT(expression,datatype)}** does not support conversion of stream data; specifying a stream field to *expression* results in an SQLCODE -37 error.

Specifying an invalid value to either version of **CONVERT** results in an SQLCODE -141.

If an *expression* does not have a defined data type (for example a host variable supplied by Caché ObjectScript) its data type defaults to the string data type.

For a list of the data types supported by Caché SQL, see [Data Types](#). For other data type conversions, refer to the [CAST](#) function.

CONVERT(datatype,expression,format-code)

This is the MS SQL Server compatible function. It takes as *datatype* any valid Caché SQL data type, including [character stream data](#). For a list of the data types supported by Caché SQL, see [Data Types](#).

You can truncate a string by performing a VARCHAR-to-VARCHAR conversion, specifying an output string length shorter than the *expression* string length.

When using **CONVERT** (or **CAST**), if a character data type (such as CHAR or VARCHAR) has no specified length, the default maximum length is 30 characters. If a binary data type (such as BINARY or VARBINARY) has no specified length, the default maximum length is 30 characters. Otherwise, these data types with no specified length are mapped to a MAXLEN of 1 character, as shown in the [Data Types](#) table.

You can perform a BIT data type conversion. The permitted values are 1, 0, or NULL. If you specify any other value, Caché issues an SQLCODE -141 error. In the following Embedded SQL example, both are BIT conversions of a NULL:

```

SET a=""
&sql(SELECT CONVERT(BIT,:a),
      CONVERT(BIT,NULL)
      INTO :x,:y)
WRITE !,"SQLCODE=",SQLCODE
WRITE !,"the host variable is:",x
WRITE !,"the NULL keyword is:",y

```

The optional *format-code* argument specifies a date, datetime, or time format. This format can either be used to define the output when converting from a date/time/timestamp data type to a character string, or to define the input when converting from a character string to a date/time/timestamp data type. The following format codes are supported; format codes that output a two-digit year are listed in the first column; formats that either output a four-digit year or do not output a year at all are listed in the second column:

Two-digit year codes	Four-digit year codes	Format
	0 or 100	Mon dd yyyy hh:mmAM (or PM)
1	101	mm/dd/yy
2	102	yy.dd.mm
3	103	dd/mm/yy
4	104	dd.mm.yy
5	105	dd-mm-yy
6	106	dd Mon yy
7	107	Mon dd, yy (no leading zero when dd < 10)
	8 or 108	hh:mm:ss
	9 or 109	Mon dd yyyy hh:mm:ss:nnnAM (or PM)
10	110	mm-dd-yy
11	111	yy.mm.dd
12	112	yymmdd
	13 or 113	dd Mon yyyy hh:mm:ss:nnn (24 hour)
	14 or 114	hh:mm:ss.nnn (24 hour)
	20 or 120	yyyy-mm-dd hh:mm:ss (24 hour)
	21 or 121	yyyy-mm-dd hh:mm:ss.nnnn (24 hour)
	126	yyyy-mm-ddThh:mm:ss.nnnn (24 hour)
	130	dd Mon yyyy hh:mm:ss:nnnAM (or PM)
	131	dd/mm/yyyy hh:mm:ss:nnnAM (or PM)

The following are features of date and time conversions:

- Range of Values: The range of permitted dates is 1840-12-31 through 9999-12-31.
- Default Values:
 - When converting a time value to `TIMESTAMP`, `DATETIME`, or `SMALLDATETIME`, the date defaults to 1900-01-01. Note that for `{fn CONVERT()}` the date defaults to the current date.
 - When converting a date value to `TIMESTAMP`, `DATETIME`, or `SMALLDATETIME`, the time defaults to 00:00:00.

- Default Format: If *format-code* is not specified, **CONVERT** attempts to determine the format from the specified value. If it cannot, it defaults to *format-code* 100.
- Two-digit Years: Two-digit years from 00 through 49 are converted to 21st century dates (2000 through 2049); two-digit years from 50 through 99 are converted to 20th century dates (1950 through 1999).
- Fractional seconds: fractional seconds can be preceded by either a period (.) or a colon (:). The symbols have different meanings:
 - A period is the default, and can be used with all format codes. A period indicates a standard fraction; thus 12:00:00.4 indicates four-tenths of a second, and 12:00:00.004 indicates four-thousandth of a second. There is no limit on the number of digits of fractional precision.
 - A colon can only be used with the following *format-code* values: 9/109, 13/113, 130, and 131. A colon indicates that the number that follows is in thousandths of a second; thus 12:00:00:4 indicates four-thousandth of a second (12:00:00.004). The permitted number of digits following a colon is limited to three.

Specifying an *expression* with an invalid format or a format that does not match the *format-code* generates an SQLCODE -141 error. Specifying a non-existent *format-code* returns 1900-01-01 00:00:00.

{fn CONVERT(expression,datatype)}

This is the ODBC scalar function. It supports the following ODBC explicit data type conversions. You must use the “SQL_” keywords for specifying data type conversions with this form of **CONVERT**. In the following table, where there are two groups of conversion data types, the first group converts both the data value and the data type, the second group converts the data type but does not convert the data value:

Source	Conversion
Any numeric data type	SQL_VARCHAR, SQL_DOUBLE, SQL_DATE, SQL_TIME
%String	SQL_DATE, SQL_TIME, SQL_TIMESTAMP
%Date	SQL_VARCHAR, SQL_TIMESTAMP SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT, SQL_DATE
%Time	SQL_VARCHAR, SQL_TIMESTAMP SQL_VARCHAR, SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT, SQL_TIME
%TimeStamp	SQL_DATE, SQL_TIME SQL_VARCHAR, SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT
Any non-stream data type	SQL_INTEGER, SQL_BIGINT, SQL_SMALLINT, SQL_TINYINT
Any non-stream data type	SQL_DOUBLE

SQL_VARCHAR is the standard ODBC representation. When converting to SQL_VARCHAR, dates and times are converted to their appropriate ODBC representations; numeric datatype values are converted to a string representation. When converting from SQL_VARCHAR, the value must be a valid ODBC Time, Timestamp, or Date representation.

- When converting a time value to `SQL_TIMESTAMP`, the date defaults to the current date. Note that for `CONVERT()` the date defaults to 1900-01-01.
- When converting a date value to `SQL_TIMESTAMP`, the time defaults to 00:00:00.

In this syntactic form, fractional seconds can be preceded by either a period (.) or a colon (:). The symbols have different meanings. A period indicates a standard fraction; thus `12:00:00.4` indicates four-tenths of a second, and `12:00:00.004` indicates four-thousandth of a second. A colon indicates that what follows is in thousandths of a second; thus `12:00:00:4` indicates four-thousandth of a second. The permitted number of digits following a colon is limited to three.

When converting to an integer data type or the `SQL_DOUBLE` data type, data values (including dates and times) are converted to a numeric representation. For `SQL_DATE`, this is the number of days since January 1, 1841. For `SQL_TIME`, this is the number of seconds since midnight. Input strings are truncated when a nonnumeric character is encountered. The integer data types also truncates decimal digits, returning the integer portion of the number.

{fn CONVERT(expression,datatype)} does not support conversion of stream data; specifying a stream field to *expression* results in an `SQLCODE -37` error.

A `NULL` converted to any data type remains `NULL`.

An empty string (""), or any nonnumeric string value converts as follows:

- `SQL_VARCHAR` and `SQL_TIMESTAMP` return the supplied value.
- Numeric data types convert to 0 (zero).
- `SQL_DATE` and `SQL_TIME` convert to `NULL`.

For other data type conversions, refer to the [CAST](#) function.

CONVERT Class Method

You can also perform data type conversions using the `CONVERT()` method call, using “SQL_” keywords for specifying data types:

```
$$SYSTEM.SQL.CONVERT(expression,convert-to-type,convert-from-type)
```

as shown in the following example:

```
WRITE $$SYSTEM.SQL.CONVERT(60945,"SQL_VARCHAR","SQL_DATE")
```

Examples

CONVERT() Examples

The following examples uses the Caché scalar syntactical form of `CONVERT`.

The following example compares the conversion of a fractional number using the `DECIMAL` and `DOUBLE` data types:

```
SELECT CONVERT(DECIMAL,-123456789.0000123456789) AS DecimalVal,
       CONVERT(DOUBLE,-123456789.0000123456789) AS DoubleVal
```

The following example converts a character stream field to a `VARCHAR` text string. It also displays the length of the character stream field using `CHAR_LENGTH`:

```
SELECT Notes,CONVERT(VARCHAR(80),Notes) AS NoteText,CHAR_LENGTH(Notes) AS TextLen
FROM Sample.Employee WHERE Notes IS NOT NULL
```

The following example shows several conversions of the date-of-birth field (`DOB`) to a formatted character string:

```

SELECT DOB,
       CONVERT(VARCHAR(20),DOB) AS DOBDefault,
       CONVERT(VARCHAR(20),DOB,100) AS DOB100,
       CONVERT(VARCHAR(20),DOB,107) AS DOB107,
       CONVERT(VARCHAR(20),DOB,114) AS DOB114,
       CONVERT(VARCHAR(20),DOB,126) AS DOB126
FROM Sample.Person

```

The default format and the code 100 format are the same. Because the DOB field does not contain a time value, formats that display time (here including the default, 100, 114, and 126) supply a zero value, which represents 12:00AM (midnight). The code 126 format provides a date and time string that contains no spaces.

{fn CONVERT()} Examples

The following examples use the ODBC syntactical form of **CONVERT**.

The following embedded SQL example converts a mixed string to an integer. Caché truncates the string at the first nonnumeric character and then converts the resulting numeric to [canonical form](#):

```

SET a="007 James Bond"
&sql(SELECT {fn CONVERT(:a,SQL_INTEGER)} INTO :x)
WRITE !,"SQLCODE=",SQLCODE
WRITE !,"the host variable is:",x

```

returns the integer 7.

The following example converts dates in the "DOB" (Date Of Birth) column to the SQL_TIMESTAMP data type.

```

SELECT DOB,{fn CONVERT(DOB,SQL_TIMESTAMP)} AS DOBtoTstamp
FROM Sample.Person

```

The resulting timestamp is in the format: *yyyy-mm-dd hh:mm:ss*.

The following example converts dates in the "DOB" (Date Of Birth) column to the SQL_INTEGER data type.

```

SELECT DOB,{fn CONVERT(DOB,SQL_INTEGER)} AS DOBtoInt
FROM Sample.Person

```

The resulting integer is the \$HOROLOG count of days since December 31, 1840.

The following example converts dates in the "DOB" (Date Of Birth) column to the SQL_VARCHAR data type.

```

SELECT DOB,{fn CONVERT(DOB,SQL_VARCHAR)} AS DOBtoVchar
FROM Sample.Person

```

The resulting string is in the format: *yyyy-mm-dd*.

See Also

- [CAST](#) function
- [Data Types](#)

COS

A scalar numeric function that returns the cosine, in radians, of an angle.

```
{fn COS(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type FLOAT. This is an angle expressed in radians.
-------------------------	---

Description

COS takes any numeric value and returns the cosine as a floating point number. The returned value is within the range -1 to 1, inclusive. **COS** returns NULL if passed a NULL value. **COS** treats nonnumeric strings as the numeric value 0.

COS returns a value of data type FLOAT with a precision of 19 and a scale of 18.

COS can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the [DEGREES](#) function to convert radians to degrees. You can use the [RADIANS](#) function to convert degrees to radians.

Examples

These examples show the effect of **COS** on two sines.

```
SELECT {fn COS(0.52)} AS Cosine
```

returns 0.86781.

```
SELECT {fn COS(-.31)} AS Cosine
```

returns 0.95233.

See Also

- SQL functions: [ACOS](#) [ASIN](#) [ATAN](#) [COT](#) [SIN](#) [TAN](#)
- ObjectScript function: [\\$ZCOS](#)

COT

A scalar numeric function that returns the cotangent, in radians, of an angle.

```
{fn COT(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type FLOAT. This is an angle expressed in radians.
-------------------------	---

Description

COT takes any nonzero number and returns its cotangent as a floating point number. **COT** returns NULL if passed a NULL value. A numeric value of 0 (zero) causes a runtime error, generating an SQLCODE -400 (fatal error occurred). **COT** treats nonnumeric strings as the numeric value 0.

COT returns a value of data type FLOAT with a precision of 36 and a scale of 18.

COT can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the [DEGREES](#) function to convert radians to degrees. You can use the [RADIANS](#) function to convert degrees to radians.

Examples

The following examples show the effect of **COT**:

```
SELECT {fn COT(0.52)} AS Cotangent
```

returns 1.74653.

```
SELECT {fn COT(124.1332)} AS Cotangent
```

returns -0.040312.

See Also

- SQL functions: [ACOS](#) [ASIN](#) [ATAN](#) [COS](#) [SIN](#) [TAN](#)
- ObjectScript function: [\\$ZCOT](#)

CURDATE

A scalar date/time function that returns the current local date.

```
{fn CURDATE()}
{fn CURDATE}
```

Description

CURDATE takes no arguments and returns the date as type DATE. Note that the argument parentheses are optional. **CURDATE** returns the current local date for this timezone; it adjusts for local time variants, such as [Daylight Saving Time](#).

CURDATE in Logical mode returns the current local date in [\\$HOROLOG](#) format; for example, 62970. **CURDATE** in Display mode returns the current local date in the default format for the locale. For example, in an American locale 05/28/2013, in a European locale 28/05/2013, in a Russian locale 28.05.2013.

To specify a different date format, use the [TO_DATE](#) function. To change the default date format, use the [SET OPTION](#) command with the DATE_FORMAT, YEAR_OPTION, or DATE_SEPARATOR options.

To return just the current date, use **CURDATE** or [CURRENT_DATE](#). These functions return their values in DATE data type. The [CURRENT_TIMESTAMP](#), [GETDATE](#) and [NOW](#) functions can also be used to return the current date and time as a [TIMESTAMP](#) data type.

Note that all Caché SQL time and date functions except [GETUTCDATE](#) are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can use [GETUTCDATE](#) or the ObjectScript [\\$ZTIMESTAMP](#) special variable.

These data types perform differently when using embedded SQL. The DATE data type stores values as integers in [\\$HOROLOG](#) format; when displayed in SQL they are converted to date display format; when returned from embedded SQL they are returned as integers. A [TIMESTAMP](#) data type stores and displays its value in the same format. You can use the [CONVERT](#) function to change the data type of dates and times.

Examples

The following examples both return the current date:

```
SELECT {fn CURDATE()} AS Today
SELECT {fn CURDATE} AS Today
```

The following Embedded SQL example returns the current date. Because this date is stored in [\\$HOROLOG](#) format, it is returned as an integer:

```
&sql(SELECT {fn CURDATE()} INTO :a)
WRITE !,"Current date is: ",a
```

The following example shows how **CURDATE** can be used in a **SELECT** statement to return all records that have a shipment date that is the same or later than today's date:

```
SELECT * FROM Orders
WHERE ShipDate >= {fn CURDATE()}
```

See Also

- SQL functions: [CURRENT_DATE](#) [CURRENT_TIME](#) [CURRENT_TIMESTAMP](#) [CURTIME](#) [GETDATE](#) [GETUTCDATE](#) [NOW](#)
- ObjectScript function: [\\$ZDATE](#)

CURRENT_DATE

A date/time function that returns the current local date.

CURRENT_DATE

Description

CURRENT_DATE takes no arguments and returns the date as type DATE. Argument parentheses are not permitted. **CURRENT_DATE** returns the current local date for this timezone; it adjusts for local time variants, such as [Daylight Saving Time](#).

CURRENT_DATE in Logical mode returns the current local date in [\\$HOROLOG](#) format; for example, 62970. **CURRENT_DATE** in Display mode returns the current local date in the default format for the locale. For example, in an American locale 05/28/2013, in a European locale 28/05/2013, in a Russian locale 28.05.2013.

To specify a different date format, use the [TO_DATE](#) function. To change the default date format, use the [SET OPTION](#) command with the DATE_FORMAT, YEAR_OPTION, or DATE_SEPARATOR options.

To return just the current date, use **CURRENT_DATE** or [CURDATE](#). These functions return their values in DATE data type. The [CURRENT_TIMESTAMP](#), [GETDATE](#) and [NOW](#) functions can also be used to return the current date and time as a [TIMESTAMP](#) data type.

Note that all Caché SQL time and date functions except [GETUTCDATE](#) are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can use [GETUTCDATE](#) or the ObjectScript [\\$ZTIMESTAMP](#) special variable.

These data types perform differently when using embedded SQL. The DATE data type stores values as integers in [\\$HOROLOG](#) format; when displayed in SQL they are converted to date display format; when returned from embedded SQL they are returned as integers. A [TIMESTAMP](#) data type stores and displays its value in the same format. You can use the [CONVERT](#) function to change the datatype of dates and times.

CURRENT_DATE can be used as a default specification keyword in [CREATE TABLE](#) or [ALTER TABLE](#).

Examples

The following example returns the current date, converted to Display mode:

```
SELECT CURRENT_DATE AS Today
```

The following Embedded SQL example returns the current date as stored. Because this date is stored in [\\$HOROLOG](#) format, it is returned as an integer:

```
&sql(SELECT CURRENT_DATE INTO :a)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
  ELSE {
    WRITE !,"Current date is: ",a }
```

The following example shows how **CURRENT_DATE** can be used in a **WHERE** clause to return records of people born in the last 1000 days:

```
SELECT Name,DOB,Age
FROM Sample.Person
WHERE DOB > CURRENT_DATE - 1000
```

See Also

[CURDATE](#), [CURRENT_TIME](#), [CURRENT_TIMESTAMP](#), [CURTIME](#), [GETDATE](#), [GETUTCDATE](#), [NOW](#)

CURRENT_TIME

A date/time function that returns the current local time.

```
CURRENT_TIME
CURRENT_TIME (precision)
```

Arguments

<i>precision</i>	A positive integer that specifies the time precision as the number of digits of fractional seconds. The default is 0 (no fractional seconds); this default is configurable.
------------------	---

CURRENT_TIME returns the **TIME** data type.

Description

CURRENT_TIME takes either no arguments or a precision argument. Empty argument parentheses are not permitted.

CURRENT_TIME returns the current local time for this timezone. It adjusts for local time variants, such as [Daylight Saving Time](#).

CURRENT_TIME in Logical mode returns the current local time in **\$HOROLOG** format; for example, 37065.

CURRENT_TIME in Display mode returns the current local time in the default format for the locale; for example, 10:18:27.

To change the default time format, use the **SET OPTION** command with the **TIME_FORMAT** and **TIME_PRECISION** options. You can configure fractional seconds of precision, as described below.

To return just the current time, use **CURRENT_TIME** or **CURTIME**. These functions return their values in **TIME** data type. The **CURRENT_TIMESTAMP**, **GETDATE** and **NOW** functions can also be used to return the current date and time as a **TIMESTAMP** data type.

Note that all Caché SQL time and date functions except **GETUTCDATE** are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can use **GETUTCDATE** or the ObjectScript **\$ZTIMESTAMP** special variable.

These data types perform differently when using embedded SQL. The **TIME** data type stores values as integers in **\$HOROLOG** format (as the number of seconds since midnight); when displayed in SQL they are converted to time display format; when returned from embedded SQL they are returned as integers. A **TIMESTAMP** data type stores and displays its value in the same format. You can use the **CAST** or **CONVERT** function to change the datatype of times and dates.

CURRENT_TIME can be used as a default specification keyword in **CREATE TABLE** or **ALTER TABLE**.

CURRENT_TIME cannot specify a *precision* argument when used as a default specification keyword.

Fractional Seconds Precision

CURRENT_TIME can return up to nine digits of fractional seconds of precision. The default for the number of digits of precision can be configured using the following:

- **SET OPTION** with the **TIME_PRECISION** option.
- The **\$SYSTEM.SQL.SetDefaultTimePrecision()** method call.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; digits of precision in excess of the precision available on your system are returned as zeroes.

Examples

The following example returns the current system time:

```
SELECT CURRENT_TIME
```

The following example returns the current system time with three digits of fractional seconds precision:

```
SELECT CURRENT_TIME(3)
```

The following Embedded SQL example returns the current time. Because this time is stored in \$HOROLOG format, it is returned as an integer:

```
&sql(SELECT CURRENT_TIME INTO :a)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"Current time is: ",a }
```

The following example sets the LastCall field in the selected row of the Contacts table to the current system time:

```
UPDATE Contacts SET LastCall = CURRENT_TIME
WHERE Contacts.ItemNumber=:item
```

See Also

- SQL concepts: [Data Type](#), [Date and Time Constructs](#)
- SQL time functions: [CAST](#), [CONVERT](#), [CURTIME](#), [HOUR](#), [MINUTE](#), [SECOND](#)
- SQL timestamp functions: [CURRENT_TIMESTAMP](#), [GETDATE](#), [GETUTCDATE](#), [NOW](#), [SYSDATE](#), [TIMESTAMPADD](#), [TIMESTAMPDIFF](#)
- Caché ObjectScript: [\\$ZTIME](#) function, [\\$HOROLOG](#) special variable, [\\$ZTIMESTAMP](#) special variable

CURRENT_TIMESTAMP

A date/time function that returns the current local date and time.

```
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP(precision)
```

Arguments

<i>precision</i>	A positive integer that specifies the time precision as the number of digits of fractional seconds. The default is 0 (no fractional seconds); this default is configurable.
------------------	---

Description

CURRENT_TIMESTAMP takes either no arguments or a precision argument. Empty argument parentheses are not permitted.

CURRENT_TIMESTAMP returns the current local date and time for this [timezone](#); it adjusts for local time variants, such as [Daylight Saving Time](#).

CURRENT_TIMESTAMP can return a timestamp in %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff).

You can use \$HOROLOG to store or return the current local date and time in internal format.

To change the default datetime string format, use the [SET OPTION](#) command with the various date and time options.

You can specify **CURRENT_TIMESTAMP**, with or without *precision*, as the field default value when defining a datetime field using **CREATE TABLE** or **ALTER TABLE**.

Fractional Seconds Precision

CURRENT_TIMESTAMP has two syntax forms:

- Without argument parentheses, **CURRENT_TIMESTAMP** is functionally identical to [NOW](#). It uses the system-wide default time precision.
- With argument parentheses, **CURRENT_TIMESTAMP(*precision*)**, is functionally identical to **GETDATE**, except that the **CURRENT_TIMESTAMP()** *precision* argument is mandatory. **CURRENT_TIMESTAMP()** always returns its specified *precision* and ignores the configured system-wide default time precision.

Fractional seconds are always truncated, not rounded, to the specified precision. In **TIMESTAMP** data type format, the maximum possible digits of precision is nine. The actual number of digits supported is determined by the *precision* argument, the configured default time precision, and the system capabilities. If you specify a *precision* larger than the configured default time precision, the additional digits of precision are returned as trailing zeros.

Configuring Precision

The default precision can be configured using the following:

- [SET OPTION](#) with the **TIME_PRECISION** option.
- The **\$SYSTEM.SQL.SetDefaultTimePrecision()** method call.
- Go to the Management Portal, select **[System] > [Configuration] > [General SQL Settings]**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; *precision* digits in excess of the precision available on your system are returned as zeroes.

Date and Time Functions Compared

GETDATE and **NOW** can also be used to return the current local date and time as a **TIMESTAMP** data type. **GETDATE** supports precision, **NOW** does not support precision.

SYSDATE is identical to **CURRENT_TIMESTAMP**, with the exception that **SYSDATE** does not support *precision*. **CURRENT_TIMESTAMP** is the preferred Caché SQL function; **SYSDATE** is provided for compatibility with other vendors.

GETUTCDATE can be used to return the universal (independent of time zone) date and time as a **TIMESTAMP** data type value. Note that all Caché SQL time and date functions except **GETUTCDATE** are specific to the local time zone setting. To get a universal (time zone independent) timestamp, you can use **GETUTCDATE** or the ObjectScript **\$ZTIMESTAMP** special variable.

To return just the current local date, use **CURDATE** or **CURRENT_DATE**. To return just the current local time, use **CURRENT_TIME** or **CURTIME**. These functions return their values in **DATE** or **TIME** data type. None of these functions support precision.

The **TIMESTAMP** data type storage format and display format are the same. The **TIME** and **DATE** data types store their values as integers in **\$HOROLOG** format; when displayed in SQL they are converted to date or time display format. Embedded SQL returns them in logical (storage) format by default. You can change the Embedded SQL returned value format using the **#SQLCompile Select** macro preprocessor directive, as described in the “ObjectScript Macros and the Macro Preprocessor” chapter of *Using Caché ObjectScript*.

You can use the **CAST** or **CONVERT** function to change the data type of dates and times.

Examples

The following example returns the current local date and time three different ways: in **TIMESTAMP** data type format with system default time precision, with a *precision* of two digits of fractional seconds, and in **\$HOROLOG** internal storage format with full seconds:

```
SELECT
    CURRENT_TIMESTAMP AS FullSecStamp,
    CURRENT_TIMESTAMP(2) AS FracSecStamp,
    $HOROLOG AS InternalFullSec
```

The following Embedded SQL example sets a locale default time precision. The first **CURRENT_TIMESTAMP** specifies no *precision*; it returns the current time with the default time precision. The second **CURRENT_TIMESTAMP** specifies *precision*; this overrides the configured default time precision. The *precision* argument can be larger or smaller than the default time precision setting:

```
InitialVal
    SET pre=##class(%SYS.NLS.Format).GetFormatItem("TimePrecision")
ChangeVal
    SET x=##class(%SYS.NLS.Format).SetFormatItem("TimePrecision",4)
    &sql(SELECT CURRENT_TIMESTAMP,CURRENT_TIMESTAMP(2) INTO :a,:b)
    IF SQLCODE'=0 {
        WRITE !,"Error code ",SQLCODE }
    ELSE {
        WRITE !,"Timestamp is: ",a
        WRITE !,"Timestamp is: ",b }
RestoreVal
    SET x=##class(%SYS.NLS.Format).SetFormatItem("$TimePrecision",pre)
```

The following Embedded SQL example compares local (time zone specific) and universal (time zone independent) time stamps:

```
&sql(SELECT CURRENT_TIMESTAMP,GETUTCDATE() INTO :a,:b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"Local timestamp is: ",a
    WRITE !,"UTC timestamp is: ",b
    WRITE !,"$ZTIMESTAMP is: ",$ZDATETIME($ZTIMESTAMP,3,,3)
}
```

The following example sets the LastUpdate field in the selected row of the Orders table to the current system date and time. If LastUpdate is data type %TimeStamp, **CURRENT_TIMESTAMP** returns the current date and time as an ODBC timestamp:

```
UPDATE Orders SET LastUpdate = CURRENT_TIMESTAMP
WHERE Orders.OrderNumber=:ord
```

The following example creates a table named Orders, which records product orders received:

```
CREATE TABLE Orders (
  OrderId      INT NOT NULL,
  ClientId     INT,
  ItemName     CHAR(40) NOT NULL,
  OrderDate    TIMESTAMP DEFAULT CURRENT_TIMESTAMP(3),
  PRIMARY KEY (OrderId))
```

The OrderDate column contains the date and time that the order was received. It uses the **TIMESTAMP** data type and inserts the current system date and time as the default value using the **CURRENT_TIMESTAMP** function with a precision of 3.

See Also

- SQL concepts: [Data Type](#), [Date and Time Constructs](#)
- SQL timestamp functions: [CAST](#), [CONVERT](#), [GETDATE](#), [GETUTCDATE](#), [NOW](#), [SYSDATE](#), [TIMESTAMPADD](#), [TIMESTAMPDIFF](#), [TO_TIMESTAMP](#)
- SQL current date and time functions: [CURDATE](#), [CURRENT_DATE](#), [CURRENT_TIME](#), [CURTIME](#)
- ObjectScript: [\\$ZDATETIME](#) function, [\\$SHOROLOG](#) special variable, [\\$ZTIMESTAMP](#) special variable

CURTIME

A scalar date/time function that returns the current local time.

```
{fn CURTIME()}
{fn CURTIME}
```

Description

CURTIME takes no arguments and returns the time as a **TIME** data type. Note that the argument parentheses are optional. **CURTIME** returns the current local time for this timezone; it adjusts for local time variants, such as [Daylight Saving Time](#).

CURTIME in Logical mode returns the current local time in **\$HOROLOG** format; for example, 37065. **CURTIME** in Display mode returns the current local time in the default format for the locale; for example, 10:18:27.

Hours are represented in 24-hour format.

To change the default time format, use the **SET OPTION** command with the **TIME_FORMAT** and **TIME_PRECISION** options.

To return just the current time, use **CURTIME** or **CURRENT_TIME**. These functions return their values in **TIME** data type. The **CURRENT_TIMESTAMP**, **GETDATE** and **NOW** functions can also be used to return the current date and time as a **TIMESTAMP** data type.

Note that all Caché SQL time and date functions except **GETUTCDATE** are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can use **GETUTCDATE** or the ObjectScript **\$ZTIMESTAMP** special variable.

These data types perform differently when using embedded SQL. The **TIME** data type stores values as integers in **\$HOROLOG** format (as the number of seconds since midnight); when displayed in SQL they are converted to time display format; when returned from embedded SQL they are returned as integers. A **TIMESTAMP** data type stores and displays its value in the same format. You can use the **CAST** or **CONVERT** function to change the datatype of times and dates.

Examples

The following examples both return the current system time:

```
SELECT {fn CURTIME()} AS TimeNow
```

```
SELECT {fn CURTIME} AS TimeNow
```

The following Embedded SQL example returns the current time. Because this time is stored in **\$HOROLOG** format, it is returned as an integer:

```
&sql(SELECT {fn CURTIME} INTO :a)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"Current time is: ",a }
```

The following example sets the **LastCall** field in the selected row of the **Contacts** table to the current system time:

```
UPDATE Contacts Set LastCall = {fn CURTIME()}
WHERE Contacts.ItemNumber=:item
```

See Also

- SQL concepts: [Data Type Date and Time Constructs](#)
- SQL time functions: [CAST CONVERT CURRENT_TIME HOUR MINUTE SECOND](#)

- SQL timestamp functions: [CURRENT_TIMESTAMP](#) [GETDATE](#) [GETUTCDATE](#) [NOW](#) [TIMESTAMPADD](#) [TIMESTAMPDIFF](#)
- ObjectScript: [\\$ZTIME](#) function [\\$HOROLOG](#) special variable [\\$ZTIMESTAMP](#) special variable

DATABASE

A scalar string function that returns the database name qualifier.

```
{fn DATABASE( )}
```

Description

DATABASE returns the current qualifier for the name of the database corresponding to the connection handle. In Caché, **DATABASE** always returns the empty string ("").

DATALENGTH

A function that returns the number of characters in an expression.

```
DATALENGTH(expression)
```

Arguments

<i>expression</i>	An expression, which can be the name of a column, a string literal, or the result of another scalar function. The underlying data type can be a character type (such as CHAR or VARCHAR), a numeric, or a data stream.
-------------------	--

Description

Note: The **DATALENGTH**, **CHAR_LENGTH**, and **CHARACTER_LENGTH** functions are identical. Use of the **CHAR_LENGTH** function is recommended for new code. **DATALENGTH** is provided for TSQL compatibility. Refer to [CHAR_LENGTH](#) for further details.

See Also

- [CHAR_LENGTH](#) function
- [CHARACTER_LENGTH](#) function

DATE

A function that takes a timestamp and returns a date.

```
DATE(timestamp)
```

Arguments

<i>timestamp</i>	An expression that specifies a timestamp or other date or date and time representation.
------------------	---

Description

DATE takes a timestamp expression and returns a date. The return value is of data type DATE. This is functionally the same as `CAST(timestamp AS DATE)`. It accepts *timestamp* values with any of the following data types: %Library.TimeStamp, %Library.Date, and %Library.Integer or %Library.Numeric (for implicit logical dates, such as +\$HOROLOG). It can also accept %Library.String values that are in a format compatible with %Library.TimeStamp (a valid ODBC date).

An invalid ODBC date string is evaluated as zero, which corresponds to the date December 31, 1840. A *timestamp* may contain just an ODBC format date or an ODBC format date and time. Although only the date portion of the ODBC *timestamp* is converted, the entire string is validated. An ODBC *timestamp* fails validation if the date portion is incomplete, if either the date or time portion contain an out-of-range value (including leap year calculations), or if *timestamp* contains any invalid format characters or trailing characters.

An empty string (") argument returns 0 (December 31, 1840). A NULL argument returns NULL.

This function can also be invoked from ObjectScript using the **DATE()** method call:

```
WRITE $SYSTEM.SQL.DATE("2016-02-23 12:37:45")
```

\$HOROLOG and \$ZTIMESTAMP

\$HOROLOG and **\$ZTIMESTAMP** return character string values. When a character string is cast to a numeric type, it always returns a numeric value of zero (0). The Caché DATE data type value for 0 is December 31, 1840.

Therefore, in order to interpret **\$HOROLOG** or **\$ZTIMESTAMP** as the current date, you must prefix it with a plus (+) sign, which forces numeric interpretation. This is shown in the following examples:

```
SELECT DATE($HOROLOG),DATE($ZTIMESTAMP) // both return 12/31/1840
```

```
SELECT DATE(+$HOROLOG),DATE(+$ZTIMESTAMP) // both return the current date
```

ODBC Date Strings

The **DATE** function and the **\$SYSTEM.SQL.DATE()** method can both take an ODBC date format string. They validate the input string. If it passes validation, they return the corresponding date. If it fails validation, they return 0. Validation is performed as following:

- The string must correspond to ODBC format: `yyyy-mm-dd hh:mm:ss.xx`. The entire string is parsed for correct format, not just the date portion of the string.
- The string must contain (at least) a full date: `yyyy-mm-dd`. Leading zeros may be omitted or included. The time portion is optional, and any part of the time portion may be included: `yyyy-mm-dd hh:`.
- Each numeric element of the string (both date portion and time portion) must contain a valid value. For example, month values must be in the range of 1 through 12 (inclusive). Day values cannot exceed the number of days for the specified month. Leap year days are calculated.

- Dates must be within the Caché date range. The minimum date is 1840-12-31, the maximum date is 9999-12-31.

Examples

The following examples take a value of data type `%Library.TimeStamp`:

```
SET myquery = "SELECT {fn NOW} AS NowCol,DATE({fn NOW}) AS DateCol"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

```
SET myquery = "SELECT CURRENT_TIMESTAMP AS TSCol,DATE(CURRENT_TIMESTAMP) AS DateCol"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

```
SET myquery = "SELECT GETDATE() AS GetDateCol,DATE(GETDATE()) AS DateCol"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following example takes a string value in `%Library.TimeStamp` format:

```
SET myquery = "SELECT '1995-09-10 13:14:23' AS DateStrCol,DATE('1995-09-10 13:14:23') AS DateCol"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following examples take string values that represent dates in Caché logical format. In order to properly convert these values to `%Library.Date` data type, the value must be prefixed with a plus sign (+) to force numeric evaluation:

```
SET myquery = "SELECT $H AS HoroCol,DATE(+$H) AS DateCol"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

```
SET myquery = "SELECT $ZTIMESTAMP AS TSCol,DATE(+$ZTIMESTAMP) AS DateCol"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

See Also

- [CAST](#) function
- [CURDATE](#) and [CURRENT_DATE](#) functions
- [CURRENT_TIMESTAMP](#) function
- [GETUTCDATE](#) function
- [NOW](#) function
- [TO_TIMESTAMP](#) function
- [\\$HOROLOG](#) special variable
- [\\$ZTIMESTAMP](#) special variable

DATEADD

A date/time function that returns a timestamp calculated by adding or subtracting a number of date part units (such as hours or days) to a date or timestamp.

```
DATEADD(datepart, integer-exp, date-exp)
```

Arguments

<i>datepart</i>	The name (or abbreviation) of a date or time part. This name can be specified in uppercase or lowercase, with or without enclosing quotes. The <i>datepart</i> can be specified as a literal or a host variable.
<i>integer-exp</i>	A numeric expression of any number type. The value is truncated to an integer (positive or negative). The value indicates the number of <i>datepart</i> units that will be added to (or subtracted from) <i>date-exp</i> .
<i>date-exp</i>	The date/time expression to be modified. This can be a date string, or a timestamp string, or a function such as CURRENT_DATE. The value returned is always a timestamp, in %TimeStamp data type format.

Description

The **DATEADD** function modifies a date/time expression by incrementing the specified date part by the specified number of units. For example, if *datepart* is 'month' and *integer-exp* is 5, **DATEADD** increments *date-exp* by five months. You can also decrement a date part by specifying a negative integer for *integer-exp*.

The calculated date is returned as a complete date/time expression (a timestamp). **DATEADD** returns data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

DATEADD always returns a valid date, taking into account the number of days in a month, and calculating for leap year. For example, incrementing January 31 by one month returns February 28 (the highest valid date in the month), unless the specified year is a leap year, in which case it returns February 29. Incrementing a leap year date of February 29 by one year returns February 28. Incrementing a leap year date of February 29 by four years returns February 29.

If you specify a *date-exp* that includes fractional seconds, the returned value also includes fractional seconds. If you omit the time portion of *date-exp*, **DATEADD** returns a default time of 00:00:00. If you omit the date portion of *date-exp*, **DATEADD** returns a default date of 1900-01-01.

DATEADD and **TIMESTAMPADD** handle quarters (3-month intervals); **DATEDIFF** and **TIMESTAMPDIFF** do not handle quarters.

Similar time/date modification operations can be performed using the **TIMESTAMPADD** ODBC scalar function.

This function can also be invoked from ObjectScript using the **DATEADD()** method call:

```
$SYSTEM.SQL.DATEADD(datepart, integer-exp, date-exp)
```

Datepart Argument

The *datepart* argument can be one of the following date/time components, either the full name (the Date Part column) or its abbreviation (the Abbreviation column). These *datepart* component names and abbreviations are not case-sensitive.

Date Part	Abbreviations	<i>integer-exp = 1</i>
year	yyyy, yy	Increments year by 1.
quarter	qq, q	Increments month by 3.
month	mm, m	Increments month by 1.
week	wk, ww	Increments day by 7.
weekday	dw	Increments day by 1.
day	dd, d	Increments day by 1.
dayofyear	dy, y	Increments day by 1.
hour	hh	Increments hour by 1.
minute	mi, n	Increments minute by 1.
second	ss, s	Increments second by 1.
millisecond	ms	Increments by .001 of a second.

Incrementing or decrementing a date part causes other date parts to be modified appropriately. For example, incrementing the hour past midnight automatically increments the day, which may in turn increment the month, and so forth.

A *datepart* can be specified as a quoted string or without quotes. These syntax variants perform slightly different operations:

- Quotes: `DATEADD('month', 12, $HOROLOG)`: the *datepart* is treated as a literal when creating cached queries. Caché SQL performs [literal substitution](#). This produces a more generally reusable cached query.
- No quotes: `DATEADD(month, 12, $HOROLOG)`: the *datepart* is treated as a keyword when creating cached queries. No literal substitution. This produces a more specific cached query.

If you specify an invalid *datepart* value as a literal, an `SQLCODE -8` error code is issued. However, if you supply an invalid *datepart* value as a host variable, no `SQLCODE` error is issued and the **DATEPART** function returns a value of `NULL`.

Date Expression Formats

The *date-exp* argument can be in any of the following formats, and may include or omit fractional seconds:

- A Caché %Date logical value (+\$H)
- A Caché %TimeStamp (%Library.TimeStamp) logical value (YYYY-MM-DD HH:MM:SS)
- A Caché %String (or compatible) value

The Caché %String (or compatible) value can be in any of the following formats:

- 99999,99999 (\$H format)
- *Sybase/SQL-Server-date Sybase/SQL-Server-time*
- *Sybase/SQL-Server-time Sybase/SQL-Server-date*
- *Sybase/SQL-Server-date* (default time is 00:00:00)
- *Sybase/SQL-Server-time* (default date is 01/01/1900)

Sybase/SQL-Server-date is one of these five formats:

```
mmdelimiterdddelimiter[yy]yy
dd Mmm[mm][,][yy]yy
dd [yy]yy Mmm[mm]
YYYY Mmm[mm] dd
YYYY [dd] Mmm[mm]
```

where *delimiter* is a slash (/), hyphen (-), or period (.).

Sybase/SQL-Server-time represents one of these three formats:

```
HH:MM[:SS:SSS][{AM|PM}]
HH:MM[:SS.S]
HH[' ']{AM|PM}
```

If the year is given as two digits, Caché checks the sliding window to interpret the date. The system default for the sliding window can be set via the %DATE utility, which is documented only in the “[Legacy Documentation](#)” chapter in *Using InterSystems Documentation*. For information on setting the sliding window for the current process, see the documentation for the ObjectScript [\\$ZDATE](#), [\\$ZDATEH](#), [\\$ZDATETIME](#) and [\\$ZDATETIMEH](#) functions.

Note that **DATEADD** is provided for Sybase and Microsoft SQL Server compatibility.

Range and Value Checking

DATEADD performs the following checks on input values. If a value fails a check, the null string is returned.

- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits.
- Date values must be within a valid range. Years: 1841 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 0 through 23. Minutes: 0 through 59. Seconds: 0 through 59.
- The incremented year value returned must be within the range 1841 through 9999. Incrementing beyond this range returns <null>.
- The number of days in a month must match the month and year. For example, the date '02-29' is only valid if the specified year is a leap year.
- Date values less than 10 may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.

Examples

The following example adds 1 week to the specified date:

```
SELECT DATEADD('week',1,'1999-12-20') AS NewDate
```

it returns 1999-12-27 00:00:00, because adding 1 week adds 7 days. Note that **DATEADD** supplies the omitted time portion.

The following example adds 5 months to the original timestamp:

```
SELECT DATEADD(MM,5,'1999-12-20 12:00:00') AS NewDate
```

it returns 2000-05-20 12:00:00 because adding 5 months also increments the year.

The following example also adds 5 months to the original timestamp:

```
SELECT DATEADD('mm',5,'1999-01-31 12:00:00') AS NewDate
```

it returns 1999-06-30 12:00:00. Here **DATEADD** modified the day value as well as the month, because simply incrementing the month would result in June 31, which is an invalid date.

The following example adds 45 minutes to the original timestamp:

```
SELECT DATEADD(MI,45,'1999-12-20 12:00:00') AS NewTime
```

it returns 1999-12-20 12:45:00.

The following example also adds 45 minutes to the original timestamp, but in this case the addition increments the date:

```
SELECT DATEADD('mi',45,'1999-12-20 23:30:00') AS NewTime
```

it returns 1999-12-21 00:15:00.

The following example decrements the original timestamp by 45 minutes:

```
SELECT DATEADD(N,-45,'1999-12-20 12:00:00') AS NewTime
```

it returns 1999-12-20 11:15:00.

The following example adds 60 days to the current date, adjusting for the varying lengths of months:

```
SELECT DATEADD(D,60,CURRENT_DATE) AS NewDate
```

In the following example, the first **DATEADD** adds 92 days to the specified date, the second **DATEADD** adds 1 quarter to the specified date:

```
SELECT DATEADD('dd',92,'1999-12-20') AS NewDateD,  
       DATEADD('qq',1,'1999-12-20') AS NewDateQ
```

The first returns 2000-03-21 00:00:00; the second returns 2000-03-20 00:00:00. Incrementing by a quarter increments the month field by 3, and, when needed, increments the year field. It also corrects for the maximum number of days for a given month.

The above examples all use date part abbreviations. However, you can also specify the date part by its full name, as in this example:

```
SELECT DATEADD('day',92,'1999-12-20') AS NewDate
```

it returns 2000-03-21 00:00:00.

The following Embedded SQL example uses host variables to perform the same **DATEADD** operation as the previous example:

```
SET x="day"  
SET datein="1999-12-20"  
&sql(SELECT DATEADD(:x,92,:datein)  
      INTO :dateout)  
WRITE "in: ",datein,!, "out: ",dateout
```

See Also

- [DATEDIFF](#) function
- [DATENAME](#) function
- [DATEPART](#) function
- [TIMESTAMPADD](#) function
- [TIMESTAMPDIFF](#) function

DATEDIFF

A date/time function that returns an integer difference for a specified datepart between two dates.

```
DATEDIFF (datepart , startdate , enddate )
```

Arguments

<i>datepart</i>	The name (or abbreviation) of a date or time part. This name can be specified in uppercase or lowercase, with or without enclosing quotes. The <i>datepart</i> can be specified as a literal or a host variable.
<i>startdate</i>	The starting date/time for the interval. May be a date, a time, or a datetime in a variety of standard formats.
<i>enddate</i>	The ending date/time for the interval. May be a date, a time, or a datetime in a variety of standard formats. <i>startdate</i> is subtracted from <i>enddate</i> to determine how many <i>datepart</i> intervals are between the two dates.

Description

The **DATEDIFF** function returns the INTEGER number of the specified *datepart* difference between the two specified dates. The date range begins at *startdate* and ends at *enddate*. (If *enddate* is earlier than *startdate*, **DATEDIFF** returns a negative INTEGER value.)

DATEDIFF returns the total number of the specified unit between *startdate* and *enddate*. For example, the number of minutes between two datetime values evaluates the date component as well as the time component, and adds 1440 minutes for each day difference. **DATEDIFF** returns the count of the specified date part boundaries crossed between *startdate* and *enddate*. For example, that any two dates that specify sequential years (for example 2010-09-23 and 2011-01-01) return a year **DATEDIFF** of 1, regardless of whether the actual duration between the two dates is more than or less than 365 days. Similarly, the number of minutes between 12:23:59 and 12:24:05 is 1, although only 6 seconds actually separate the two values.

Note that **DATEDIFF** is provided for Sybase and Microsoft SQL Server compatibility. Similar time/date comparison operations can be performed using the [TIMESTAMPDIFF](#) ODBC scalar function.

This function can also be invoked from ObjectScript using the **DATEDIFF()** method call:

```
$SYSTEM.SQL.DATEDIFF (datepart , startdate , enddate )
```

Specifying an invalid *datepart*, *startdate*, or *enddate* to the **DATEDIFF()** method generates a <ZDDIF> error.

Datepart Argument

The *datepart* argument can be one of the following date/time components, either the full name (the Date Part column) or its abbreviation (the Abbreviation column). These *datepart* component names and abbreviations are not case-sensitive.

Date Part	Abbreviations
year	yyyy, yy
month	mm, m
week	wk, ww
weekday	dw
day	dd, d
dayofyear	dy
hour	hh
minute	mi, n
second	ss, s
millisecond	ms

The `weekday` and `dayofyear` *datepart* values are functionally identical to the `day` *datepart* value.

DATEDIFF and **TIMESTAMPDIFF** do not handle quarters (3-month intervals).

If you specify a *startdate* and *enddate* that include fractional seconds, you can return the difference as a number of fractional seconds, expressed as thousands of a second (.001), as shown in the following example:

```
SELECT DATEDIFF('ms', '62871,56670.10', '62871,56670.27'), /* returns 170 */
       DATEDIFF('ms', '62871,56670.1111', '62871,56670.27222') /* returns 161.12 */
```

A *datepart* can be specified as a quoted string or without quotes. These syntax variants perform slightly different operations:

- Quotes: `DATEDIFF('month', '2004-02-25', $HOROLOG)`: the *datepart* is treated as a literal when creating cached queries. Caché SQL performs [literal substitution](#). This produces a more generally reusable cached query.
- No quotes: `DATEDIFF(month, '2004-02-25', $HOROLOG)`: the *datepart* is treated as a keyword when creating cached queries. No literal substitution. This produces a more specific cached query.

Date Expression Formats

The *startdate* and *enddate* arguments can be in different data type formats.

The *startdate* and *enddate* arguments can be in any of the following formats:

- A Caché %Date logical value (+\$H), also known as \$HOROLOG format.
- A Caché %TimeStamp (%Library.TimeStamp) logical value (YYYY-MM-DD HH:MM:SS.FFF), also known as ODBC format.
- A Caché %String (or compatible) value.

The Caché %String (or compatible) value can be in any of the following formats, and may include or omit fractional seconds:

- 99999,99999 (\$HOROLOG format). The \$HOROLOG special variable does not return fractional seconds. However, you can specify a value in \$HOROLOG format that includes fractional seconds: 99999,99999,999
- *Sybase/SQL-Server-date* *Sybase/SQL-Server-time*
- *Sybase/SQL-Server-time* *Sybase/SQL-Server-date*
- *Sybase/SQL-Server-date* (default time is 00:00:00)
- *Sybase/SQL-Server-time* (default date is 01/01/1900)

Sybase/SQL-Server-date is one of these five formats:

```
mm/dd/[yy]yy
dd Mmm[mm][,][yy]yy
dd [yy]yy Mmm[mm]
YYYY Mmm[mm] dd
YYYY [dd] Mmm[mm]
```

In the first syntactic format the delimiter can be a slash (/), a hyphen (-), or a period (.).

Sybase/SQL-Server-time represents one of these three formats:

```
HH:MM[:SS[:FFF]] [ {AM|PM} ]
HH:MM[:SS[.FFF]]
HH[ ' ' ] {AM|PM}
```

Years

If the year is given as two digits or the date is omitted entirely, Caché checks the sliding window to interpret the date. The system-wide default for the sliding window is 1900; thus by default a two-digit year is assumed to be in the 20th century. This is shown in the following example:

```
SELECT DATEDIFF('year', '10/11/14', '09/14/2015'),
       DATEDIFF('year', '12:00:00', '2004-07-09 12:00:00')
```

The sliding window default can be set system-wide or for the current process via the %DATE utility, which is documented only in the “[Legacy Documentation](#)” chapter in *Using InterSystems Documentation*. For information on establishing a sliding window for interpreting a specified date with a two-digit year, see the documentation for the ObjectScript [\\$ZDATE](#), [\\$ZDATEH](#), [\\$ZDATETIME](#) and [\\$ZDATETIMEH](#) functions.

Fractional Seconds

DATEDIFF returns fractional seconds as milliseconds (a three-digit integer) regardless of the number of fractional digits precision in *startdate* and *enddate*. Fractional digits beyond three are represented as fractional milliseconds. This is shown in the following example:

```
SELECT DATEDIFF('ms', '12:00:00.1', '12:00:00.2'),
       DATEDIFF('ms', '12:00:00.10009', '12:00:00.20007')
```

Some NLS locales specify the fractional separator as a comma (European usage) rather than as a period. If the current locale is one of these locales, **DATEDIFF** accepts either a period or a comma as the fractional seconds separator character for local date formats. You cannot use a comma as the fractional seconds separator for a date in \$HOROLOG format, or a date in ODBC format. Attempting to do so generates an SQLCODE -8. Both of these formats require a period regardless of the current NLS locale.

Time Differences Independent of TimeFormat

DATEDIFF returns a time difference in seconds and milliseconds, even when the TimeFormat for the current process is set to not return seconds. This is shown in the following example:

```
SET tfmt=##class(%SYS.NLS.Format).GetFormatItem("TimeFormat")
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",1)
WRITE "datetime values (with seconds) are: ",!,
      $ZDATETIME("62871,56670.10",1,-1)," ",$ZDATETIME("62871,56673.27",1,-1),!
&sql(SELECT DATEDIFF('ss','62871,56670.10','62871,56673.27') INTO :x)
WRITE "DATEDIFF number of seconds is: ",x,!
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",2)
WRITE "datetime values (without seconds) are: ",!,
      $ZDATETIME("62871,56670.10",1,-1)," ",$ZDATETIME("62871,56673.27",1,-1),!
&sql(SELECT DATEDIFF('ss','62871,56670.10','62871,56673.27') INTO :x)
WRITE "DATEDIFF number of seconds is: ",x,!
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeFormat",tfmt)
```

Range and Value Checking

DATEDIFF performs the following checks on input values:

- All specified parts of the *startdate* and *enddate* must be valid before any **DATEDIFF** operation can be performed.
- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits. If you omit the date portion of an input value, **DATEDIFF** defaults to '1900-01-01'. An invalid date value results in an SQLCODE -8 error.
- Date and time values must be within a valid range. Years: 0001 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 00 through 23. Minutes: 0 through 59. Seconds: 0 through 59. The number of days in a month must match the month and year. For example, the date '02-29' is only valid if the specified year is a leap year. An invalid date value results in an SQLCODE -8 error.
- Date values less than 10 (month and day) may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.
- Time values may be wholly or partially omitted. If *startdate* or *enddate* specifies an incomplete time, zeros are supplied for the unspecified parts.
- An hour value less than 10 must include a leading zero. Omitting this leading zero results in an SQLCODE -8 error.

Error Handling

- In Embedded SQL, if you specify an invalid *datepart* as an input variable, an SQLCODE -8 error code is issued. If you specify an invalid *datepart* as a literal, a <SYNTAX> error occurs. If you specify an invalid *startdate* or *enddate* as either an input variable or a literal, an SQLCODE -8 error code is issued.
- In Dynamic SQL, if you supply an invalid *datepart*, *startdate*, or *enddate*, the **DATEDIFF** function returns a value of NULL. No SQLCODE error is issued.

Examples

The following example returns 353 because there are 353 days (D) between the two timestamps:

```
SELECT DATEDIFF(D, '1999-01-01 00:00:00', '1999-12-20 12:00:00')
```

In the following example, each **DATEDIFF** returns 1, because the year portion of the dates differs by 1. The actual duration between the dates is not considered:

```
SELECT DATEDIFF('yyyy', '1910-08-21', '1911-08-21') AS ExactYear,
       DATEDIFF('yyyy', '1910-06-30', '1911-01-01') AS HalfYear,
       DATEDIFF('yyyy', '1910-01-01', '1911-12-31') AS Nearly2Years,
       DATEDIFF('yyyy', '1910-12-31 11:59:59', '1911-01-01 00:00:00') AS NewYearSecond
```

Note that the above examples use an abbreviation for the date part. However, you can specify the full name, as in this example:

```
SELECT DATEDIFF('year', '1968-09-10 13:19:00', '1999-12-20 00:00:00')
```

The following Embedded SQL example uses host variables to perform the same **DATEDIFF** operation as the previous example:

```
SET x="year"
SET date1="1968-09-10 13:19:00"
SET date2="1999-12-20 00:00:00"
&sql(SELECT DATEDIFF(:x, :date1, :date2)
      INTO :diff)
WRITE diff
```

The following example uses a subquery to return those records where the person date of birth is 1500 days or less from the current date:

```
SELECT Name, Age, DOB
FROM (SELECT Name, Age, DOB, DATEDIFF('dy', DOB, $HOROLOG) AS DaysTo FROM Sample.Person)
WHERE DaysTo <= 1500
ORDER BY Age
```

See Also

- [DATEADD](#) function
- [DATENAME](#) function
- [DATEPART](#) function
- [TIMESTAMPADD](#) function
- [TIMESTAMPDIFF](#) function

DATENAME

A date/time function that returns a string representing the value of the specified part of a date/time expression.

```
DATENAME(datepart, date-expression)
```

Arguments

<i>datepart</i>	The type of date/time information to return. The name (or abbreviation) of a date or time part. This name can be specified in uppercase or lowercase, with or without enclosing quotes. The <i>datepart</i> can be specified as a literal or a host variable.
<i>date-expression</i>	A date, time, or timestamp expression from which the <i>datepart</i> value is to be returned. <i>date-expression</i> must contain a value of type <i>datepart</i> .

Description

The **DATENAME** function returns the name of the specified part (such as the month "June") of a date/time value. The result is returned as data type VARCHAR(20). If the result is numeric (such as "23" for the day), it is still returned as a VARCHAR(20) string. To return this information as an integer, use **DATEPART**. To return a string containing multiple date parts, use **TO_DATE**.

Note that **DATENAME** is provided for Sybase and Microsoft SQL Server compatibility.

This function can also be invoked from ObjectScript using the **DATENAME()** method call:

```
$$SYSTEM.SQL.DATENAME(datepart, date-expression)
```

Datepart Argument

The *datepart* argument can be a string containing one (and only one) of the following date/time components, either the full name (the Date Part column) or its abbreviation (the Abbreviation column). These *datepart* component names and abbreviations are not case-sensitive.

Date Part	Abbreviations	Return Values
year	yyyy, yy	1840-9999
quarter	qq, q	1-4
month	mm	January,...December
week	wk, ww	1-53
weekday	dw	Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
dayofyear	dy, y	1-366
day	dd, d	1-31
hour	hh	0-23
minute	mi, n	0-59
second	ss, s	0-59 (with fractional seconds, if provided)
millisecond	ms	0-999 (with precision of 3)

If you specify an invalid *datepart* value as a literal, an SQLCODE -8 error code is issued. However, if you supply an invalid *datepart* value as a host variable, no SQLCODE error is issued and the **DATENAME** function returns a value of NULL.

The preceding table shows the default return values for the various date parts. You can modify the returned values for several of these date parts by using the [SET OPTION](#) command with various time and date options.

week: Caché can be configured to determine the week of the year for a given date using either the Caché default algorithm or the ISO 8601 standard algorithm. For further details, refer to the [WEEK](#) function.

weekday: The Caché default for weekday is to designate Sunday as first day of the week (*weekday*=1). However, you can configure the first day of the week to another value, or you can apply the ISO 8601 standard which designates Monday as first day of the week. For further details, refer to the [DAYOFWEEK](#) function.

second: Caché returns a string containing the *date-expression* value for seconds and (if provided) fractional seconds with the precision of *date-expression*.

millisecond: Caché returns a string containing the number of milliseconds (thousandths of a second). If the *date-expression* has more than three fractional digits of precision, Caché truncates it to three digits and returns this number as a string.

A *datepart* can be specified as a quoted string or without quotes. These syntax variants perform slightly different operations:

- Quotes: `DATENAME('month', '2004-02-25')`: the *datepart* is treated as a literal when creating cached queries. Caché SQL performs [literal substitution](#). This produces a more generally reusable cached query.
- No quotes: `DATENAME(month, '2004-02-25')`: the *datepart* is treated as a keyword when creating cached queries. No literal substitution. This produces a more specific cached query.

Date Expression Formats

The *date-expression* argument can be in any of the following formats:

- A Caché %Date logical value (+\$H)
- A Caché %TimeStamp (%Library.TimeStamp) logical value (YYYY-MM-DD HH:MM:SS.FFF), also known as ODBC format.
- A Caché %String (or compatible) value

The Caché %String (or compatible) value can be in any of the following formats:

- 99999,99999 (\$H format)
- *Sybase/SQL-Server-date* *Sybase/SQL-Server-time*
- *Sybase/SQL-Server-time* *Sybase/SQL-Server-date*
- *Sybase/SQL-Server-date* (default time is 00:00:00)
- *Sybase/SQL-Server-time* (default date is 01/01/1900)

Sybase/SQL-Server-date is one of these five formats:

```
mmdelimiterddddelimiter[yy]yy
dd Mmm[mm][,][yy]yy
dd [yy]yy Mmm[mm]
yyyy Mmm[mm] dd
yyyy [dd] Mmm[mm]
```

where *delimiter* is a slash (/), hyphen (-), or period (.)

If the year is given as two digits, Caché checks the sliding window to interpret the date. The system default for the sliding window can be set via the %DATE utility, which is documented only in the [“Legacy Documentation”](#) chapter in *Using*

InterSystems Documentation. For information on setting the sliding window for the current process, see the documentation for the ObjectScript [\\$ZDATE](#), [\\$ZDATEH](#), [\\$ZDATETIME](#) and [\\$ZDATETIMEH](#) functions.

Sybase/SQL-Server-time represents one of these three formats:

```
HH:MM[:SS:SSS] [ {AM|PM} ]
HH:MM[:SS.S]
HH[ ' ' ] {AM|PM}
```

If the *date-expression* specifies a time format but does not specify a date format, **DATENAME** defaults to the date 1900-01-01, which has a weekday value of Monday.

Range and Value Checking

DATENAME performs the following checks on input values. If a value fails a check, the null string is returned.

- A valid *date-expression* may consist of a date string (yyyy-mm-dd), a time string (hh:mm:ss), or a date and time string (yyyy-mm-dd hh:mm:ss). If both date and time are specified, both must be valid. For example, you can return a Year value if no time string is specified, but you cannot return a Year value if an invalid time string is specified.
- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. For example, you cannot return a Year value if the Day value is omitted. Years must be specified as four digits.
- A time string must be properly formatted with the appropriate separator character. Because a time value can be zero, you can omit one or more time elements (either retaining or omitting the separator characters) and these elements will be returned with a value of zero. Thus, 'hh:mm:ss', 'hh:mm:', 'hh:mm', 'hh::ss', 'hh:.', 'hh', and ':::' are all valid. To omit the Hour element, *date-expression* must not have a date portion of the string, and you must retain at least one separator character (:).
- Date and time values must be within a valid range. Years: 1841 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 0 through 23. Minutes: 0 through 59. Seconds: 0 through 59.
- The number of days in a month must match the month and year. For example, the date '02-29' is only valid if the specified year is a leap year.
- Most date and time values less than 10 may include or omit a leading zero. However, an Hour value of less than 10 must include the leading zero if it is part of a datetime string. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.
- If *date-expression* specifies a time format but does not specify a date format, **DATENAME** does not perform range validation for the time component values.

Examples

In the following example, each **DATENAME** returns 'Wednesday' because that is the day of week ('dw') of the specified date:

```
SELECT DATENAME('dw', '2004-02-25') AS DayName,
       DATENAME(dw, '02/25/2004') AS DayName,
       DATENAME('DW', 59590) AS DayName
```

The following example returns 'December' because that is the month name ('mm') of the specified date:

```
SELECT DATENAME('mm', '1999-12-20 12:00:00') AS MonthName
```

The following example returns '1999' (as a string) because that is the year ('yy') of the specified date:

```
SELECT DATENAME('yy', '1999-12-20 12:00:00') AS Year
```

Note that the above examples use the abbreviations of the date parts. However, you can specify the full name, as in this example:

```
SELECT DATENAME('year', '1999-12-20 12:00:00') AS Year
```

The following example returns the current quarter, week-of-year, and day-of-year. Each value is returned as a string:

```
SELECT DATENAME('Q', $HOROLOG) AS Q,  
       DATENAME('WK', $HOROLOG) AS WkCnt,  
       DATENAME('DY', $HOROLOG) AS DayCnt
```

The following Embedded SQL example passes in the *datepart* and *date-expression* as a host variables:

```
SET a="year"  
SET b=$HOROLOG  
&sql(SELECT DATENAME(:a, :b) INTO :c)  
WRITE "this year is: ",c
```

The following example uses a subquery to returns records from Sample.Person whose day of birth was a Wednesday:

```
SELECT Name AS WednesdaysChild, DOB  
FROM (SELECT Name, DOB, DATENAME('dw', DOB) AS Wkday FROM Sample.Person)  
WHERE Wkday='Wednesday'  
ORDER BY DOB
```

See Also

- SQL functions: [DATEADD](#), [DATEDIFF](#), [DATEPART](#), [TO_DATE](#), [TIMESTAMPADD](#), [TIMESTAMPDIFF](#)
- ObjectScript function: [\\$ZDATETIME](#)

DATEPART

A date/time function that returns an integer representing the value of the specified part of a date/time expression.

```
DATEPART(datepart, date-expression)
```

Arguments

<i>datepart</i>	The type of date/time information to return. The name (or abbreviation) of a date or time part. This name can be specified in uppercase or lowercase, with or without enclosing quotes. The <i>datepart</i> can be specified as a literal or a host variable.
<i>date-expression</i>	A date, time, or timestamp expression from which the <i>datepart</i> value is to be returned. <i>date-expression</i> must contain a value of type <i>datepart</i> .

Description

The **DATEPART** function returns the *datepart* information about a specified date/time expression as an integer. To return this information as a character string, use **DATENAME**. If the *datepart* value is `sqltimestamp` (or `sts`), the **DATEPART** can return either data type `TIMESTAMP` or `INTEGER`, as described below.

DATEPART returns the value of only one element of *date-expression*; to return a string containing multiple date parts, use **TO_DATE**.

This function can also be invoked from ObjectScript using the **DATEPART()** method call:

```
$$SYSTEM.SQL.DATEPART(datepart, date-expression)
```

DATEPART is provided for Sybase and Microsoft SQL Server compatibility.

Datepart Argument

The *datepart* argument can be one of the following date/time components, either the full name (the Date Part column) or its abbreviation (the Abbreviations column). These *datepart* component names and abbreviations are not case-sensitive.

Date Part	Abbreviations	Return Values
year	yyyy, yy	1840-9999
quarter	qq, q	1-4
month	mm, m	1-12
week	wk, ww	1-53
weekday	dw	1-7 (Sunday,...,Saturday)
dayofyear	dy, y	1-366
day	dd, d	1-31
hour	hh	0-23
minute	mi, n	0-59
second	ss, s	0-59
millisecond	ms	0-999 (with precision of 3).
sqltimestamp	sts	SQL_TIMESTAMP: yyyy-mm-dd hh:mm:ss

The preceding table shows the default return values for the various date parts. You can modify the returned values for several of these date parts by using the [SET OPTION](#) command with various time and date options.

week: Caché can be configured to determine the week of the year for a given date using either the Caché default algorithm or the ISO 8601 standard algorithm. For further details, refer to the [WEEK](#) function.

weekday: The Caché default for weekday is to designate Sunday as first day of the week (`weekday=1`). However, you can configure the first day of the week to another value, or you can apply the ISO 8601 standard which designates Monday as first day of the week. For further details, refer to the [DAYOFWEEK](#) function. Note that the ObjectScript **\$ZDATE** and **\$ZDATETIME** functions count week days from 0 through 6 (not 1 through 7).

second: If the *date-expression* contains fractional seconds, Caché returns `second` as a decimal number with whole seconds as the integer component, and fractional seconds as the decimal component. Precision is not truncated.

millisecond: Caché returns three fractional digits of precision, with trailing zeroes removed. If the *date-expression* has more than three fractional digits of precision, Caché truncates it to three digits.

sqltimestamp: Caché converts the input data to timestamp format and supplies zero values for the time elements, if necessary. Can return either data type `TIMESTAMP` or data type `INTEGER` (see below). The `sqltimestamp` (abbreviated `sts`) *datepart* value is for use *only* with **DATEPART**. Do not attempt to use this value in other contexts.

A *datepart* can be specified as a quoted string, without quotes, or with parentheses around a quoted string. No [literal substitution](#) is performed on *datepart*, regardless of how specified; literal substitution is performed on *date-expression*. All *datepart* values return a data type `INTEGER` value, except `sqltimestamp` (or `sts`), which returns its value as a character string of data type `TIMESTAMP`.

Date Input Formats

The *date-expression* argument can be in any of the following formats:

- A Caché %Date logical value (+\$H)
- A Caché %TimeStamp (%Library.TimeStamp) logical value (YYYY-MM-DD HH:MM:SS.FFF), also known as ODBC format.
- A Caché %String (or compatible) value

The Caché %String (or compatible) value can be in any of the following formats:

- 99999,99999 (\$H format)
- *Sybase/SQL-Server-date Sybase/SQL-Server-time*
- *Sybase/SQL-Server-time Sybase/SQL-Server-date*
- *Sybase/SQL-Server-date* (default time is 00:00:00)
- *Sybase/SQL-Server-time* (default date is 01/01/1900)

Sybase/SQL-Server-date is one of these five formats:

```
mmdelimiterdddelimiter[yy]yy
dd Mmm[mm][,][yy]yy
dd [yy]yy Mmm[mm]
YYYY Mmm[mm] dd
YYYYY [dd] Mmm[mm]
```

where *delimiter* is a slash (/), hyphen (-), or period (.

If the year is given as two digits, Caché checks the sliding window to interpret the date. The system default for the sliding window can be set via the %DATE utility, which is documented only in the “[Legacy Documentation](#)” chapter in *Using InterSystems Documentation*. For information on setting the sliding window for the current process, see the documentation for the ObjectScript [\\$ZDATE](#), [\\$ZDATEH](#), [\\$ZDATETIME](#) and [\\$ZDATETIMEH](#) functions.

Sybase/SQL-Server-time represents one of these three formats:

```
HH:MM[:SS:SSS][ {AM|PM} ]
HH:MM[:SS.S]
HH[ ' ' ] {AM|PM}
```

If the *date-expression* specifies a time format but does not specify a date format, **DATENAME** defaults to the date 1900-01-01, which has a weekday value of 2 (Monday).

For `sqltimestamp`, time is returned as a 24-hour clock. Fractional seconds are truncated.

Invalid Argument Error Codes

If you specify an invalid *datepart* option, **DATEPART** generates an SQLCODE -8 error code, and the following %msg: 'badopt' is not a recognized DATEPART option.

If you specify an invalid *date-expression* value (for example, an alphabetic text string), **DATEPART** generates an SQLCODE -400 error code, and the following %msg: Invalid input to DATEPART() function:

`DATEPART('year', 'badval')`. If you specify a *date-expression* that fails validation (as described below), **DATEPART** generates an SQLCODE -400 error code, and the following %msg: Unexpected error occurred: <ILLEGAL VALUE>datepart.

Range and Value Checking

DATEPART performs the following checks on *date-expression* values. If a value fails a check, the null string is returned.

- A valid *date-expression* may consist of a date string (yyyy-mm-dd), a time string (hh:mm:ss), or a date and time string (yyyy-mm-dd hh:mm:ss). If both date and time are specified, both must be valid. For example, you can return a Year value if no time string is specified, but you cannot return a Year value if an invalid time string is specified.
- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. For example, you cannot return a Year value if the Day value is omitted. Years must be specified as four digits.
- A time string must be properly formatted with the appropriate separator character. Because a time value can be zero, you can omit one or more time elements (either retaining or omitting the separator characters) and these elements will be returned with a value of zero. Thus, 'hh:mm:ss', 'hh:mm:', 'hh:mm', 'hh:ss', 'hh:', 'hh', and ':::' are all valid. To omit the Hour element, *date-expression* must not have a date portion of the string, and you must retain at least one separator character (:).
- Date and time values must be within a valid range. Years: 1841 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 0 through 23. Minutes: 0 through 59. Seconds: 0 through 59.
- The number of days in a month must match the month and year. For example, the date '02-29' is only valid if the specified year is a leap year.
- Most date and time values less than 10 may include or omit a leading zero. However, an Hour value of less than 10 must include the leading zero if it is part of a datetime string. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.
- If *date-expression* specifies a time format but does not specify a date format, **DATEPART** does not perform range validation for the time component values.

Examples

In the following example, each **DATEPART** returns the year portion of the datetime string (in this case, 2004) as an integer. Note that *date-expression* can be in various formats, and *datepart* can be specified as either the datepart name or datepart abbreviation, quoted or unquoted:

```
SELECT DATEPART('yy', '2004-12-20 12:00:00') AS YearDTS,
       DATEPART('year', '2004-02-25') AS YearDS,
       DATEPART('YYYY', '02/25/2004') AS YearD,
       DATEPART('YEAR', 59590) AS YearHD,
       DATEPART('Year', '59590,23456') AS YearHDT
```

The following example returns the current year and quarter, based on the \$HOROLOG value:

```
SELECT DATEPART('yyyy', $HOROLOG) AS Year, DATEPART('q', $HOROLOG) AS Quarter
```

The following Embedded SQL example uses host variables to supply the **DATEPART** argument values:

```
SET x="year"
SET datein="2004-02-25"
&sql(SELECT DATEPART(:x, :datein)
      INTO :partout)
WRITE "the ", x, " is ", partout
```

The following example returns the birth day-of-week for the Sample.Person table, ordered by day of week:

```
SELECT Name, DOB, DATEPART('weekday', DOB) AS bday
FROM Sample.Person
ORDER BY bday, DOB
```

In the following example, each **DATEPART** returns 20 as the minutes portion of the *date-expression* string:

```
SELECT DATEPART('mi', '1999-1-20 12:20:07') AS Minutes,
       DATEPART('n', '1999-02-20 10:20:') AS Minutes,
       DATEPART('MINUTE', '1999-02-20 10:20') AS Minutes
```

In the following example, each **DATEPART** returns 0 as the seconds portion of the *date-expression* string:

```
SELECT DATEPART('ss', '1999-02-20 03:20:') AS Seconds,
       DATEPART('S', '1999-02-20 03:20') AS Seconds,
       DATEPART('Second', '1999-02-20') AS Seconds
```

The following example returns the full SQL timestamp as a **TIMESTAMP** data type. **DATEPART** fills in the missing time information to return a timestamp of '2004-02-25 00:00:00':

```
SELECT DATEPART('sqltimestamp', '2/25/2004') AS DTStamp
```

The following example returns the full SQL timestamp as an **INTEGER** data type. **DATEPART** fills in the missing time information to return a timestamp of '2004-02-25 00:00:00':

```
SELECT DATEPART('sqltimestamp', '2/25/2004') AS DTStampAsInt
```

The following example supplies a date and time in \$HOROLOG format, and returns a timestamp of '2004-02-25 06:30:56':

```
SELECT DATEPART('sqltimestamp', '59590,23456') AS DTStamp
```

The following example uses a subquery with **DATEPART** to return those people whose birthday is leap year day (February 29th):

```
SELECT Name, DOB
FROM (SELECT Name, DOB, DATEPART('dd', DOB) AS DayNum, DATEPART('mm', DOB) AS Month FROM Sample.Person)
WHERE Month=2 AND DayNum=29
```

See Also

- [DATEDIFF](#) function
- [DATENAME](#) function
- [TIMESTAMPADD](#) function
- [TIMESTAMPDIFF](#) function
- [TO_DATE](#) function

DAY

A date function that returns the day of the month for a date expression.

```
DAY(date-expression)
{fn DAY(date-expression)}
```

Arguments

<i>date-expression</i>	An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

Note: The **DAY** function is an alias for the **DAYOFMONTH** function. **DAY** is provided for TSQL compatibility. Refer to [DAYOFMONTH](#) for further details.

See Also

- [DAYOFMONTH](#)

DAYNAME

A date function that returns the name of the day of the week for a date expression.

```
{fn DAYNAME(date-expression)}
```

Arguments

<i>date-expression</i>	An expression that evaluates to either a Caché date integer, an ODBC date, or a timestamp. This expression can be the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

DAYNAME returns the name of the day that corresponds to a specified date. The returned value is a character string with a maximum length of 15. The default day names returned are: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.

To change these default day name values, use the [SET OPTION](#) command with the WEEKDAY_NAME option.

The day name is calculated for a Caché date integer, a [\\$HOROLOG](#) or [\\$ZTIMESTAMP](#) value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted. The *date-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

DAYNAME checks that the date supplied is a valid date. The year must be between 1841 and 9999 (inclusive), the month 01 through 12, and the day appropriate for that month (for example, 02/29 is only valid on leap years). If the date is not valid, **DAYNAME** issues an SQLCODE -400 error (Fatal error occurred).

The same day of week information can be returned by using the [DATENAME](#) function. You can use [TO_DATE](#) to retrieve a day name or day name abbreviation with other date elements. To return an integer corresponding to the day of the week, use [DAYOFWEEK DATEPART](#) or [TO_DATE](#).

This function can also be invoked from ObjectScript using the **DAYNAME()** method call:

```
$SYSTEM.SQL.DAYNAME(date-expression)
```

Examples

The following examples both return the character string Wednesday because the day of the date (February 25, 2004) is a Wednesday. The first example takes a timestamp string:

```
SELECT {fn DAYNAME('2004-02-25 12:35:46')} AS Weekday
```

The second example takes a Caché date integer:

```
SELECT {fn DAYNAME(59590)} AS Weekday
```

The following examples all return the name of the current day of the week:

```
SELECT {fn DAYNAME({fn NOW()})} AS Wd_Now,
       {fn DAYNAME(CURRENT_DATE)} AS Wd_CurrDate,
       {fn DAYNAME(CURRENT_TIMESTAMP)} AS Wd_CurrTstamp,
       {fn DAYNAME($ZTIMESTAMP)} AS Wd_ZTstamp,
       {fn DAYNAME($HOROLOG)} AS Wd_Horolog
```

Note that **\$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time. This may affect the **DAYNAME** value.

The following embedded SQL example shows how **DAYNAME** responds to an invalid date (the year 2001 was not a leap year):

```
SET testdate="2001-02-29"
&sql(SELECT {fn DAYNAME(:testdate)}
INTO :a)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"returns: ",a }
QUIT
```

The following embedded SQL example shows how **DAYNAME** responds to an invalid date (the year 1835 is too early for Caché SQL):

```
SET testdate="1835-02-19"
&sql(SELECT {fn DAYNAME(:testdate)}
INTO :a)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"returns: ",a }
QUIT
```

See Also

- SQL functions: [DATENAME](#), [DATEPART](#), [DAYOFMONTH](#), [DAYOFWEEK](#), [DAYOFYEAR](#), [TO_DATE](#)
- ObjectScript function: [\\$ZDATE](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

DAYOFMONTH

A date function that returns the day of the month for a date expression.

```
{fn DAYOFMONTH(date-expression)}
```

Arguments

<i>date-expression</i>	A date or timestamp expression from which the day of the month value is to be returned. An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

DAYOFMONTH returns the day of the month as an integer from 1 to 31. The *date-expression* can be a Caché date integer, a **\$HOROLOG** or **\$ZTIMESTAMP** value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp or **\$HOROLOG** string is not evaluated and can be omitted. The *date-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

The **DAYOFMONTH** and **DAY** functions are functionally identical.

This function can also be invoked from ObjectScript using the **DAYOFMONTH()** method call:

```
WRITE $SYSTEM.SQL.DAYOFMONTH("2004-02-25")
```

Timestamp date-expression

The day (dd) portion of a timestamp string should be an integer in the range from 1 through 31. There is, however, no range checking for user-supplied values. Numbers greater than 31 and fractions are returned as specified. Because (–) is used as a separator character, negative numbers are not supported. Leading zeros are optional on input; leading zeros are suppressed on output.

DAYOFMONTH returns NULL when the day portion is '0', '00', or a nonnumeric value. NULL is also returned if the day portion of the date string is omitted entirely ('yyyy-mm hh:mm:ss'), or if no date expression is supplied.

The elements of a datetime string can be returned using the following SQL scalar functions: **YEAR**, **MONTH**, **DAYOFMONTH** (or **DAY**), **HOURL**, **MINUTE**, **SECOND**. The same elements can be returned by using the **DATEPART** or **DATENAME** function. **DATEPART** and **DATENAME** performs value and range checking on day values.

\$HOROLOG date-expression

When calculating day of the month for a **\$HOROLOG** value, **DAYOFMONTH** calculates leap years differences, including century day adjustments: 2000 is a leap year, 1900 and 2100 are not leap years.

DAYOFMONTH can process *date-expression* values prior to December 31, 1840 as negative integers. This is shown in the following example:

```
SELECT {fn DAYOFMONTH(-306)} AS DayOfMonthFeb, /* February 29, 1840 */
       {fn DAYOFMONTH(-305)} AS DayOfMonthMar /* March 1, 1840 */
```

The **LAST_DAY** function returns the date (in **\$HOROLOG** format) of the last day of the month for a specified date.

Examples

The following examples return the number 25 because the date specified is the twenty-fifth day of the month:

```
SELECT {fn DAYOFMONTH('2004-02-25')} AS DayNumTS,  
       {fn DAYOFMONTH(59590)} AS DayNumH
```

The following example also returns the number 25 for the day of the month. The year is omitted, but the separator character (–) serves as a placeholder:

```
SELECT {fn DAYOFMONTH('-02-25 11:45:32')} AS DayNum
```

The following examples return <null>:

```
SELECT {fn DAYOFMONTH('2000-02-00 11:45:32')} AS DayNum
```

```
SELECT {fn DAYOFMONTH('2000-02 11:45:32')} AS DayNum
```

```
SELECT {fn DAYOFMONTH('11:45:32')} AS DayNum
```

The following **DAYOFMONTH** examples all return the current day of the month:

```
SELECT {fn DAYOFMONTH({fn NOW()})} AS DoM_Now,  
       {fn DAYOFMONTH(CURRENT_DATE)} AS DoM_CurrD,  
       {fn DAYOFMONTH(CURRENT_TIMESTAMP)} AS DoM_CurrTS,  
       {fn DAYOFMONTH($HOROLOG)} AS DoM_Horolog,  
       {fn DAYOFMONTH($ZTIMESTAMP)} AS DoM_ZTS
```

Note that **\$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time. This may affect the **DAYOFMONTH** value.

The following example shows that leading zeros are suppressed. It returns a length of either 1 or 2, depending on the day of the month value:

```
SELECT LENGTH({fn DAYOFMONTH('2004-02-05')}),  
       LENGTH({fn DAYOFMONTH('2004-02-15')})
```

See Also

- SQL functions: [DATENAME](#), [DATEPART](#), [DAY](#), [DAYNAME](#), [DAYOFWEEK](#), [DAYOFYEAR](#), [LAST_DAY](#), [MONTH](#), [TO_DATE](#)
- ObjectScript function: [\\$ZDATE](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

DAYOFWEEK

A date function that returns the day of the week as an integer for a date expression.

```
{fn DAYOFWEEK(date-expression)}
```

Arguments

<i>date-expression</i>	A valid ODBC-format date or \$HOROLOG format date, with or without the time component. An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	---

Description

DAYOFWEEK takes a *date-expression* and returns an integer corresponding to the day of the week for that date. Days of the week are counted from the first day of the week; the Caché default is that Sunday is the first day of the week. Therefore, by default, the returned values represent these days:

- 1 — Sunday
- 2 — Monday
- 3 — Tuesday
- 4 — Wednesday
- 5 — Thursday
- 6 — Friday
- 7 — Saturday

The first day of the week default can be overridden system-wide or for specific namespaces, as described in “[Setting First Day of Week](#)”.

Note that the ObjectScript **\$ZDATE** and **\$ZDATETIME** functions count days of the week from 0 through 6 (not 1 through 7).

The *date-expression* can be a Caché date integer, a **\$HOROLOG** or **\$ZTIMESTAMP** value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted. The *date-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

The same day of week information can be returned by using the **DATEPART** or **TO_DATE** function. To return the name of the day of the week, use **DAYNAME**, **DATENAME**, or **TO_DATE**.

This function can also be invoked from ObjectScript using the **DAYOFWEEK()** method call:

```
$$SYSTEM.SQL.DAYOFWEEK(date-expression)
```

Date Validation

DAYOFWEEK performs the following checks on input values. If a value fails a check, the null string is returned.

- A valid *date-expression* may consist of a date string (yyyy-mm-dd), a date and time string (yyyy-mm-dd hh:mm:ss), a Caché date integer, or a \$HOROLOG value. **DAYOFWEEK** evaluates only the date portion of the *date-expression*.

- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits.
- Date values must be within a valid range. Years: 1841 through 9999. Months: 1 through 12. Days: 1 through 31.
- The number of days in a month must match the month and year. For example, the date '02-29' is only valid if the specified year is a leap year.
- Date values less than 10 may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.

Setting First Day of Week

By default, the first day of the week is Sunday. You can override this default system-wide by specifying `SET ^%SYS("sql", "sys", "day of week")=n`, where *n* values are 1=Monday through 7=Sunday. To set Monday as the first day of the week specify `SET ^%SYS("sql", "sys", "day of week")=1`. If Monday is the first day of the week, a Wednesday *date-expression* returns 3, rather than the 4 that would be returned if Sunday was the first day of the week. To reset the Caché default (Sunday as first day of week), specify `SET ^%SYS("sql", "sys", "day of week")=7`.

You can set the first day of the week for a specific namespace by specifying `SET ^%SYS("sql", "sys", "day of week", namespace)=n`, where *n* values are 1=Monday through 7=Sunday. To set Monday as the first day of the week for the USER namespace, specify `SET ^%SYS("sql", "sys", "day of week", "USER")=1`. Once the first day of the week is set at the namespace level, changing the system-wide setting by specifying `SET ^%SYS("sql", "sys", "day of week")=n` has no effect on that namespace. To restore the ability to change that namespace's first day of week default, you must kill `^%SYS("sql","sys","day of week",namespace)`. See example below.

Caché also supports the ISO 8601 standard for determining the day of the week, week of the year, and other date settings. This standard is principally used in European countries. The ISO 8601 standard begins counting the days of the week with Monday. To activate ISO 8601, `SET ^%SYS("sql", "sys", "week ISO8601")=1`; to deactivate, set it to 0. If week ISO8601 is activated and Caché `day of week` is undefined or set to the default (7=Sunday), the ISO 8601 standard overrides the Caché default. If Caché `day of week` is set to any other value, it overrides `week ISO8601` for **DAYOFWEEK**. See example below.

Examples

In the following example, both select-items return the number 6 (if Sunday is set as the first day of the week) because the specified *date-expression* (63876 = November 20, 2015) is a Friday:

```
SELECT {fn DAYOFWEEK('2015-11-20')}||'|'||DATENAME('dw','2015-11-20') AS ODBCDoW,
       {fn DAYOFWEEK(63876)}||'|'||DATENAME('dw','63876') AS HorologDoW
```

In the following example, all select-items return the integer corresponding to the current day of the week:

```
SELECT {fn DAYOFWEEK({fn NOW()})} AS DoW_Now,
       {fn DAYOFWEEK(CURRENT_DATE)} AS DoW_CurrDate,
       {fn DAYOFWEEK(CURRENT_TIMESTAMP)} AS DoW_CurrTstamp,
       {fn DAYOFWEEK($ZTIMESTAMP)} AS DoW_ZTstamp,
       {fn DAYOFWEEK($HOROLOG)} AS DoW_Horolog
```

Note that **\$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time. This may affect the **DAYOFWEEK** value.

The following Embedded SQL example demonstrates changing the first day of week for a namespace. It initially sets the system-wide first day of week (to 7), then sets the first day of week for a namespace (to 3). A subsequent system-wide first day of week change (to 2) has no effect on namespace first day of week until the program kills the namespace-specific setting. Killing the namespace-specific setting immediately resets that namespace's first day of week to the current system-wide value. Finally, the program restores the initial system-wide setting.

Note: The following program tests if you have namespace-specific first day of week settings for SAMPLES or USER. If you do, this program aborts to prevent changing these settings.

```

SetUp
  SET TestNsp="SAMPLES"
  SET ControlNsp="USER"
InitialDoWValues
  WRITE "Systemwide default DoW initial values",!
  DO TestDayofWeek()
  IF a=b {WRITE "No namespace-specific DoW defaults",!!}
  ELSE {WRITE "DoW initial settings are namespace-specific",!
        WRITE "Stopping this program"
        QUIT }
  SET initialDoW=^%SYS("sql","sys","day of week")
SetSystemwideDoW
  KILLL ^%SYS("sql","sys","day of week",TestNsp)
  KILLL ^%SYS("sql","sys","day of week",ControlNsp)
  SET ^%SYS("sql","sys","day of week")=7
  WRITE "Systemwide DoW set",!
  DO TestDayofWeek()
SetNamespaceDoW
  SET ^%SYS("sql","sys","day of week",TestNsp)=3
  WRITE TestNsp," namespace DoW set",!
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
  DO TestDayofWeek()
ResetSystemwideDoW
  SET ^%SYS("sql","sys","day of week")=2
  WRITE "Systemwide DoW set with ",TestNsp," DoW set",!
  DO TestDayofWeek()
KillNamespaceDoW
  KILLL ^%SYS("sql","sys","day of week",TestNsp)
  WRITE "Namespace ",TestNsp," DoW killed",!
  DO TestDayofWeek()
ResetSystemwideDoWDefault
  SET ^%SYS("sql","sys","day of week")=initialDoW
  WRITE "Systemwide DoW reset after ",TestNsp," DoW killed",!
  DO TestDayofWeek()
TestDayofWeek()
  ZNSPACE TestNsp
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
  WRITE "Today is the ",a," day of week in ",$NAMESPACE,!
  ZNSPACE ControlNsp
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :b)
  WRITE "Today is the ",b," day of week in ",$NAMESPACE,!!
  RETURN

```

The following Embedded SQL example shows the default day of the week and the day of the week with the ISO 8601 standard applied. It assumes that the Caché day of week is undefined or set to the default:

```

TestISO
  SET def=$DATA(^%SYS("sql","sys","week ISO8601"))
  IF def=0 {SET ^%SYS("sql","sys","week ISO8601")=0}
  ELSE {SET isoval=^%SYS("sql","sys","week ISO8601")}
  IF isoval=1 {GOTO UnsetISO }
  ELSE {SET isoval=0 GOTO DayofWeek }
UnsetISO
  SET ^%SYS("sql","sys","week ISO8601")=0
DayofWeek
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :a)
  WRITE "Today:",!
  WRITE "default day of week is ",a,!
  SET ^%SYS("sql","sys","week ISO8601")=1
  &sql(SELECT {fn DAYOFWEEK($HOROLOG)} INTO :b)
  WRITE "ISO8601 day of week is ",b,!
ResetISO
  SET ^%SYS("sql","sys","week ISO8601")=isoval

```

See Also

- SQL functions: [DATENAME](#), [DATEPART](#), [DAYNAME](#), [DAYOFMONTH](#), [DAYOFYEAR](#), [TO_DATE](#)
- ObjectScript function: [\\$ZDATE](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

DAYOFYEAR

A date function that returns the day of the year as an integer for a date expression.

```
{fn DAYOFYEAR(date-expression)}
```

Arguments

<i>date-expression</i>	A date expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

DAYOFYEAR returns an integer from 1 to 366 that corresponds to the day of the year for a given date expression.

DAYOFYEAR calculates leap year dates.

The day of year is calculated for a Caché date integer, a **\$HOROLOG** or **\$ZTIMESTAMP** value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted. The *date-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

When calculating day of the month for a **\$HOROLOG** value, **DAYOFYEAR** calculates leap years differences, including century day adjustments: 2000 is a leap year, 1900 and 2100 are not leap years.

DAYOFYEAR can process *date-expression* values prior to December 31, 1840 as negative integers. This is shown in the following example:

```
SELECT {fn DAYOFYEAR(-306)} AS LastDayFeb, /* February 29, 1840 */
       {fn DAYOFYEAR(-305)} AS FirstDayMar /* March 1, 1840 */
```

The same day count can be returned by using the **DATEPART** or **DATENAME** function. **DATEPART** and **DATENAME** performs value and range checking on date expressions.

This function can also be invoked from ObjectScript using the **DAYOFYEAR()** method call:

```
$$$SYSTEM.SQL.DAYOFYEAR(date-expression)
```

Examples

The following examples both return the number 64 because the day in the date expression (March 4, 2004) is the 64th day of the year (the leap year day is automatically counted):

```
SELECT {fn DAYOFYEAR('2004-03-04 12:45:37')} AS DayCount
SELECT {fn DAYOFYEAR(59598)} AS DayCount
```

The following examples all return the count for the current day:

```
SELECT {fn DAYOFYEAR({fn NOW()})} AS DNumNow,
       {fn DAYOFYEAR(CURRENT_DATE)} AS DNumCurrD,
       {fn DAYOFYEAR(CURRENT_TIMESTAMP)} AS DNumCurrTS,
       {fn DAYOFYEAR($HOROLOG)} AS DNumHorolog,
       {fn DAYOFYEAR($ZTIMESTAMP)} AS DNumZTS
```

Note that **\$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time. This may affect the **DAYOFYEAR** value.

The following example uses a subquery to return Employee records ordered by the day of year of each person's birthday:

```
SELECT Name,DOB
FROM (SELECT Name,DOB,{fn DAYOFYEAR(DOB)} AS BDay FROM Sample.Employee)
ORDER BY BDay
```

See Also

- SQL functions: [DATENAME](#), [DATEPART](#), [DAYNAME](#), [DAYOFMONTH](#), [DAYOFWEEK](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

DECODE

A function that evaluates a given expression and returns a specified value.

```
DECODE(expr { ,search,result}[ ,default ])
```

Arguments

<i>expr</i>	The expression to be decoded.
<i>search</i>	The value to which <i>expr</i> is compared.
<i>result</i>	The value which is returned if <i>expr</i> matches <i>search</i> .
<i>default</i>	<i>Optional</i> — The default value which is returned if <i>expr</i> does not match any <i>search</i> .

Description

You can specify multiple *search,result* pairs, separated by commas. You can specify one *default*. The maximum number of parameters in the **DECODE** expression (including *expr*, *search*, *result*, and *default*) is about 100. The *search*, *result*, and *default* values can be derived from expressions.

To evaluate a **DECODE** expression, Caché compares *expr* to each *search* value, one by one:

- If *expr* is equal to a *search* value, the corresponding *result* is returned.
- If *expr* is not equal to any *search* value, the *default* value is returned, or, if *default* is omitted, null is returned.

Caché evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them to *expr*. Therefore, Caché never evaluates a *search* if a previous *search* is equal to *expr*.

In a **DECODE** expression, Caché considers two nulls to be equivalent. If *expr* is null, Caché returns the *result* of the first *search* that is also null.

Note that **DECODE** is supported for Oracle compatibility.

Data Type of Returned Value

DECODE returns the data type of the first *result* argument. If the data type of the first *result* argument cannot be determined, **DECODE** returns VARCHAR. For numeric values, **DECODE** returns the largest length, precision, and scale from all of the possible *result* argument values.

If the data types of *result* and *default* are different, the data type returned is the type most compatible with all of the possible return values, the data type with the highest [data type precedence](#). For example, if *result* is an integer and *default* is a fractional number, **DECODE** returns a value with data type NUMERIC. This is because NUMERIC is the data type with the highest precedence that is compatible with both.

Examples

The following example “decodes” ages from 13 through 19 as 'Teen'; the *default* is 'Adult':

```
SELECT Name, Age, DECODE(Age,
    13, 'Teen', 14, 'Teen', 15, 'Teen', 16, 'Teen',
    17, 'Teen', 18, 'Teen', 19, 'Teen',
    'Adult') AS AgeBracket
FROM Sample.Person
WHERE Age > 12
```

The following example decodes NULLs. If there is no value for FavoriteColors, **DECODE** replaces it with the string 'No Preference'; otherwise, it returns the FavoriteColors value:

```
SELECT Name,DECODE(FavoriteColors,
                  NULL,'No Preference',
                  $LISTTOSTRING(FavoriteColors,'^')) AS ColorPreference
FROM Sample.Person
ORDER BY Name
```

The following example decodes color preferences. If the person has a single favorite color, that color name is replaced by a letter abbreviation. If the person has more than one favorite color, **DECODE** returns the FavoriteColors value:

```
SELECT Name,DECODE(FavoriteColors,
                  $LISTBUILD('Red'),'R',
                  $LISTBUILD('Orange'),'O',
                  $LISTBUILD('Yellow'),'Y',
                  $LISTBUILD('Green'),'G',
                  $LISTBUILD('Blue'),'B',
                  $LISTBUILD('Purple'),'V',
                  $LISTBUILD('White'),'W',
                  $LISTBUILD('Black'),'K',
                  $LISTTOSTRING(FavoriteColors,'^'))
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY FavoriteColors
```

Note that the **ORDER BY** clause sorts by the original field values. The following example sorts by the **DECODE** values:

```
SELECT Name,DECODE(FavoriteColors,
                  $LISTBUILD('Red'),'R',
                  $LISTBUILD('Orange'),'O',
                  $LISTBUILD('Yellow'),'Y',
                  $LISTBUILD('Green'),'G',
                  $LISTBUILD('Blue'),'B',
                  $LISTBUILD('Purple'),'V',
                  $LISTBUILD('White'),'W',
                  $LISTBUILD('Black'),'K',
                  $LISTTOSTRING(FavoriteColors,'^')) AS ColorCode
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
ORDER BY ColorCode
```

The following example decodes the numeric code in the Company code field in Employee records and returns the corresponding department name. If an employee's Company code is not 1 through 10, DECODE returns the default of "Admin (non-tech)":

```
SELECT Name,
DECODE (Company,
       1, 'TECH MARKETING', 2, 'TECH SALES', 3, 'DOCUMENTATION',
       4, 'BASIC RESEARCH', 5, 'SOFTWARE DEVELOPMENT', 6, 'HARDWARE DEVELOPMENT',
       7, 'QUALITY TESTING', 8, 'FIELD SUPPORT', 9, 'PHONE SUPPORT',
       10, 'TECH TRAINING',
       'Admin (non-tech)') AS TechJobs
FROM Sample.Employee
```

The expression has Company as the *expr* parameter and uses ten pairs of *search* and *result* parameters. "Admin (non-tech)" is the *default* parameter.

See Also

- [CASE](#)

DEGREES

A numeric function that converts radians to degrees.

```
DEGREES(numeric-expression)
```

Arguments

<i>numeric-expression</i>	The measure of an angle in radians. An expression that resolves to a numeric value.
---------------------------	---

Description

DEGREES takes an angle measurement in radians and returns the corresponding angle measurement in degrees. **DEGREES** returns NULL if passed a NULL value.

The returned data type is NUMERIC unless *numeric-expression* is data type DOUBLE, in which case the returned data type is DOUBLE. The default precision is 36. The default scale is 18.

You can use the [RADIAN](#)s function to convert degrees to radians.

Examples

The following Embedded SQL example returns the degree equivalents corresponding to the radian values 0 through 6:

```
SET a=0
WHILE a<7 {
&sql(SELECT DEGREES(:a) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"radians ",a," = degrees ",b
  SET a=a+1 }
}
```

See Also

- SQL functions: [CONVERT RADIAN TO_NUMBER](#)

%EXACT

A collation function that converts characters to the EXACT collation format.

```
%EXACT(expression)
```

```
%EXACT expression
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
-------------------	--

Description

%EXACT returns *expression* in the EXACT collation sequence. This collation sequence orders pure numeric values (values for which $x=+x$) in numeric order first, followed by all other characters in string order.

The EXACT collation sequence for strings is the same as the ANSI-standard ASCII collation sequence: digits are collated before uppercase alphabetic characters and uppercase alphabetic characters are collated before lowercase alphabetic characters. Punctuation characters occur at several places in the sequence.

%EXACT passes through NULLs unchanged.

%EXACT is a Caché SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class.

%EXACT collates an input string as either wholly numeric or as a mixed-character string in which numbers are treated the same as any other character. Compare this to **%MVR** collation, which sorts a string based on the numeric substrings within the string.

Examples

The following examples uses **%EXACT** to return all Name values that are higher in the collating sequence than 'Smith'. The first example uses parentheses syntax, the second omits the parentheses.

```
SELECT Name
FROM Sample.Person
WHERE Name > 'Smith'
```

```
SELECT Name
FROM Sample.Person
WHERE %EXACT(Name) > 'Smith'
```

```
SELECT Name
FROM Sample.Person
WHERE %EXACT Name > 'Smith'
```

See Also

- [ASCII](#) function
- [%MVR](#) collation function
- [%SQLSTRING](#) collation function
- [%SQLUPPER](#) collation function
- [%TRUNCATE](#) collation function

- [Collation](#) chapter in *Using Caché SQL*

EXP

A scalar numeric function that returns the exponential (inverse of natural logarithm) of a number.

```
{fn EXP(float-expression)}
```

Arguments

<i>float-expression</i>	The logarithmic exponent, which is an expression of type FLOAT.
-------------------------	---

Description

EXP is the exponential function e^n , where e is the constant 2.718281828. Therefore, to return the value of e , you can specify `{fn EXP(1)}`. **EXP** is the inverse of the natural logarithm function **LOG**.

EXP returns a value of data type FLOAT with a precision of 36 and a scale of 18. **EXP** returns NULL if passed a NULL value.

EXP can only be used as an ODBC scalar function (with the curly brace syntax).

Examples

The following example returns the constant e :

```
SELECT {fn EXP(1)} AS e_constant
```

returns 2.718281828...

The following Embedded SQL example returns the exponential values for the integers 0 through 10:

```
SET a=0
WHILE a<11 {
  &sql(SELECT {fn EXP(:a)} INTO :b)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
  ELSE {
    WRITE !,"Exponential of ",a," = ",b
    SET a=a+1 }
}
```

The following Embedded SQL example demonstrates that **EXP** is the inverse of **LOG**:

```
SET x=7
&sql(SELECT {fn EXP(:x)} AS Exp,
           {fn LOG(:x)} AS Log,
           {fn EXP({fn LOG(:x)})} AS ExpOfLog
        INTO :a,:b,:c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE "Exponential of ",x," = ",a,!
  WRITE "Natural log of ",x," = ",b,!
  WRITE "Exp of Log of ",x," = ",c
}
```

Note in the third function call the small discrepancy between the number input and the calculated return value. The next example shows how to handle this computational discrepancy.

The following Embedded SQL example shows the relationship between the **LOG** and **EXP** functions for the integers 1 through 10:

```
SET a=1
WHILE a<11 {
&sql(SELECT {fn LOG(:a)} INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"Logarithm of ",a," = ",b }
&sql(SELECT ROUND({fn EXP(:b)},12) INTO :c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Exponential of log ",b," = ",c
  SET a=a+1 }
}
```

Note that the **ROUND** function is needed here to correct for very small discrepancies caused by system calculation limitations. In the above example, **ROUND** is set arbitrarily to 12 decimal digits for this purpose.

See Also

- SQL functions: [LOG LOG10 POWER ROUND](#)
- ObjectScript function: [\\$ZEXP](#)

%EXTERNAL

A format-transformation function that returns an expression in DISPLAY format.

```
%EXTERNAL(expression)
```

```
%EXTERNAL expression
```

Arguments

<i>expression</i>	The expression to be converted. A field name, an expression containing a field name, or a function that returns a value in a convertible data type, such as DATE or %List. Cannot be a stream field.
-------------------	--

Description

%EXTERNAL converts *expression* to DISPLAY format, regardless of the current select mode (display mode). The DISPLAY format represents data in the VARCHAR data type with whatever data conversion the field or data type Logical-ToDisplay method performs.

%EXTERNAL is commonly used on a **SELECT** list *select-item*. It can be used in a **WHERE** clause, but this use is discouraged because using **%EXTERNAL** prevents the use of indexes on the specified field.

Applying **%EXTERNAL** changes the column header name to a value such as “Expression_1”; it is therefore usually desirable to specify a column name alias, as shown in the examples below.

Whether **%EXTERNAL** converts a date depends on the data type returned by the date field or function. **%EXTERNAL** converts **CURDATE**, **CURRENT_DATE**, **CURTIME**, and **CURRENT_TIME** values. It does not convert **CURRENT_TIMESTAMP**, **GETDATE**, **GETUTCDATE**, **NOW**, and **\$HOROLOG** values.

When **%EXTERNAL** converts a %List structure to DISPLAY format, the displayed list elements appear to be separated by a blank space. This “space” is actually the two non-display characters CHAR(13) and CHAR(10).

%EXTERNAL is a Caché SQL extension.

To convert an *expression* to LOGICAL format, regardless of the current select mode, use the **%INTERNAL** function. To convert an *expression* to ODBC format, regardless of the current select mode, use the **%ODBCOUT** function.

For further details on display format options, refer to “[Data Display Options](#)” in the “Caché SQL Basics” chapter of *Using Caché SQL*.

Examples

The following Dynamic SQL example returns Date of Birth (DOB) data values in the current select mode format, and the same data using the **%EXTERNAL** function. For the purpose of demonstration, in this program the **%SelectMode** value is determined randomly for each invocation:

```
ZNSPACE "SAMPLES"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=$RANDOM(3)
IF tStatement.%SelectMode=0 {WRITE "Select mode LOGICAL",! }
ELSEIF tStatement.%SelectMode=1 {WRITE "Select mode ODBC",! }
ELSEIF tStatement.%SelectMode=2 {WRITE "Select mode DISPLAY",! }
SET myquery = 2
SET myquery(1) = "SELECT TOP 5 DOB,%EXTERNAL(DOB) AS ExtDOB "
SET myquery(2) = "FROM Sample.Person"
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following examples show the two syntax forms for this function; they are otherwise identical. They specify the **%EXTERNAL** (DISPLAY format), **%INTERNAL** (LOGICAL format), and **%ODBCOUT** (ODBC format) of a %List field:

```
SELECT TOP 10 %EXTERNAL(FavoriteColors) AS ExtColors,  
             %INTERNAL(FavoriteColors) AS IntColors,  
             %ODBCOUT(FavoriteColors) AS ODBCColors  
FROM Sample.Person
```

```
SELECT TOP 10 %EXTERNAL FavoriteColors AS ExtColors,  
             %INTERNAL FavoriteColors AS IntColors,  
             %ODBCOUT FavoriteColors AS ODBCColors  
FROM Sample.Person
```

The following example converts date of birth (DOB) and rounded date of birth (DOB) values to **%EXTERNAL** (DISPLAY format):

```
SELECT %EXTERNAL(DOB) AS DOB,  
       %INTERNAL(ROUND(DOB,-3)) AS DOBGroup,  
       %EXTERNAL(ROUND(DOB,-3)) AS RoundedDOB  
FROM Sample.Person  
GROUP BY (ROUND(DOB,-3))  
ORDER BY DOBGroup
```

See Also

- [%INTERNAL](#), [%ODBCIN](#), [%ODBCOUT](#)
- SQL concepts: [Data Types](#), [Date and Time Constructs](#)

\$EXTRACT

A string function that extracts characters from a string by position.

```
$EXTRACT(string[,from[,to]])
```

Arguments

<i>string</i>	The target string from which the substring is to be extracted.
<i>from</i>	<i>Optional</i> — The position within the target string for a single character, or the beginning of a range of characters (inclusive), to be extracted. Specified as a positive integer counting from 1.
<i>to</i>	<i>Optional</i> — The end position (inclusive) for a range of characters to be extracted. Specified as a positive integer counting from 1.

Description

\$EXTRACT returns a substring from a specified position in *string*. The nature of the substring returned depends on the arguments used.

- **\$EXTRACT**(*string*) extracts the first character in the string.
- **\$EXTRACT**(*string,from*) extracts the character in the position specified by *from*. For example, if variable *var1* contains the string “ABCD”, the following command extracts “B” (the second character):

```
SELECT $EXTRACT('ABCD',2) AS Extracted
```

- **\$EXTRACT**(*string,from,to*) extracts the range of characters starting with the *from* position and ending with the *to* position. For example, the following command extracts the string “Alabama” (that is, all characters from position 5 to position 11, inclusive) from the string “1234Alabama567”:

```
SELECT $EXTRACT('1234Alabama567',5,11) AS Extracted
```

This function returns data of type VARCHAR.

Arguments

string

The *string* value can be a variable name, a numeric value, a string literal, or any valid expression.

from

The *from* value must be a positive integer (however, see Note). If a fractional number, the fraction is truncated and only the integer portion is used.

If the *from* value is greater than the number of characters in the string, **\$EXTRACT** returns a null string.

If *from* is specified without the *to* argument, it extracts the single specified character.

If used with the *to* argument, it identifies the start of the range to be extracted and must be less than the value of *to*. If *from* equals *to*, **\$EXTRACT** returns the single character at the specified position. If *from* is greater than *to*, **\$EXTRACT** returns a null string.

to

The *to* argument must be used with the *from* argument. It must be a positive integer. If a fractional number, the fraction is truncated and only the integer portion is used.

If the *to* value is greater than or equal to the *from* value, **\$EXTRACT** returns the specified substring. If *to* is greater than the length of the string, **\$EXTRACT** returns the substring from the *from* position to the end of the string. If *to* is less than *from*, **\$EXTRACT** returns a null string.

Examples

The following example returns “S”, the fourth character in the string:

```
SELECT $EXTRACT('THIS IS A TEST',4) AS Extracted
```

The following example returns a substring “THIS IS” which is composed of the first through seventh characters.

```
SELECT $EXTRACT('THIS IS A TEST',1,7) AS Extracted
```

The following Embedded SQL example extracts the second character (“B”) from *a* and assigns this value to variable *y*.

```
SET a="ABCD"
&sql(SELECT $EXTRACT(:a,2) INTO :y)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The extract returns ",y }
```

The following Embedded SQL example shows that the one-argument format is equivalent to the two-argument format when the *from* value is “1”. Both **\$EXTRACT** functions return “H”.

```
SET a="HELLO"
&sql(SELECT $EXTRACT(:a),$EXTRACT(:a,1) INTO :b,:c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The one-arg form returns ",b
  WRITE !,"The two-arg form returns ",c }
```

Notes

\$EXTRACT Compared with \$PIECE and \$LIST

\$EXTRACT returns a substring of characters by integer position from a string. **\$PIECE** and **\$LIST** both work on specially formatted strings.

\$PIECE returns a substring from a standard character string using delimiter characters within the string.

\$LIST returns a sublist of elements from an encoded list by the integer position of elements (not characters). **\$LIST** cannot be used on ordinary strings, and **\$EXTRACT** cannot be used on encoded lists.

The **\$EXTRACT**, **\$FIND**, **\$LENGTH**, and **\$PIECE** functions operate on standard character strings. The various **\$LIST** functions operate on encoded character strings, which are incompatible with standard character strings. The only exceptions are the **\$LISTGET** function and the one-argument and two-argument forms of **\$LIST**, which take an encoded character string as input, but output a single element value as a standard character string.

\$EXTRACT and Unicode

The **\$EXTRACT** function operates on characters, not bytes. Therefore, Unicode strings are handled the same as ASCII strings, as shown in the following embedded SQL example using the Unicode character for “pi” (**\$CHAR(960)**):

```

IF $SYSTEM.Version.IsUnicode() {
  SET a="QT PIE"
  SET b=("QT "$CHAR(960))
  &sql(SELECT
  $EXTRACT(:a,-33,4),
  $EXTRACT(:a,4,4),
  $EXTRACT(:a,4,99),
  $EXTRACT(:b,-33,4),
  $EXTRACT(:b,4,4),
  $EXTRACT(:b,4,99)
  INTO :a1,:a2,:a3,:b1,:b2,:b3)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
  ELSE {
    WRITE !,"ASCII form returns ",!,a1!,a2!,a3
    WRITE !,"Unicode form returns ",!,b1!,b2!,b3 }
  }
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}

```

Null and Invalid Arguments

- When *string* is a null string, a null string is returned.
- When *from* is a number larger than the string length, a null string is returned.
- When *from* is zero or a negative number, and no *to* is specified, a null string is returned.
- When *to* is zero, a negative number, or a number smaller than *from*, a null string is returned.
- When *to* is a valid value, *from* can be zero or a negative number. **\$EXTRACT** treats such *from* values as 1.

No SQLCODE error is generated for invalid argument values.

In following example, the negative *from* value is evaluated as 1; **\$EXTRACT** returns the substring “THIS IS” composed of the first through seventh characters.

```
SELECT $EXTRACT('THIS IS A TEST',-7,7)
```

In following embedded SQL example, all **\$EXTRACT** function calls return the null string:

```

SET a="THIS IS A TEST"
SET b=""
&sql(SELECT
$EXTRACT(:a,33),
$EXTRACT(:a,-7),
$EXTRACT(:a,3,2),
$EXTRACT(:a,-7,0),
$EXTRACT(:a,-7,-10),
$EXTRACT(:b,-33,4),
$EXTRACT(:b,4,4),
$EXTRACT(:b,4,99),
$EXTRACT(NULL,-33,4),
$EXTRACT(NULL,4,4),
$EXTRACT(NULL,4,99)
INTO :a1,:a2,:a3,:a4,:a5,:b1,:b2,:b3,:c1,:c2,:c3)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"FROM too big: ",a1
  WRITE !,"FROM negative, no TO: ",a2
  WRITE !,"TO smaller than FROM: ",a3
  WRITE !,"TO not a positive integer: ",a4,a5
  WRITE !,"LIST is null string: ",b1,b2,b3,c1,c2,c3 }
}

```

See Also

- SQL functions: [\\$FIND](#) [\\$LENGTH](#) [\\$LIST](#) [\\$LISTGET](#) [\\$PIECE](#)
- ObjectScript functions: [\\$EXTRACT](#) [\\$FIND](#) [\\$LENGTH](#) [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTGET](#) [\\$PIECE](#)

\$FIND

A string function that returns the end position of a substring within a string, with optional search start point.

```
$FIND(string,substring[,start])
```

Arguments

<i>string</i>	The target string that is to be searched. It can be a variable name, a numeric value, a string literal, or any valid expression.
<i>substring</i>	The substring that is to be searched for. It can be a variable name, a numeric value, a string literal, or any valid expression.
<i>start</i>	<i>Optional</i> — The starting point for substring search, specified as a positive integer. A character count from the beginning of <i>string</i> , counting from 1. To search from the beginning of <i>string</i> , omit this argument or specify a <i>start</i> of 0 or 1. A negative number, the empty string, or a nonnumeric value is treated as 0. Specifying <i>start</i> as NULL causes \$FIND to return <null>.

Description

\$FIND returns an integer specifying the end position of a substring within a string. **\$FIND** searches *string* for *substring*. If *substring* is found, **\$FIND** returns the integer position of the first character following *substring*. If *substring* is not found, **\$FIND** returns a value of 0.

You can include the *start* option to specify a starting position for the search. If *start* is greater than the number of characters in *string*, **\$FIND** returns a value of 0. If *start* is omitted, string position 1 is the default. If *start* is zero, a negative number, or a nonnumeric string, position 1 is the default.

\$FIND is case-sensitive. Use one of the case-conversion functions to locate both uppercase and lowercase instances of a letter or character string.

This function returns data of type SMALLINT.

\$FIND, POSITION, CHARINDEX, and INSTR

\$FIND, POSITION, CHARINDEX, and INSTR all search a string for a specified substring and return an integer position corresponding to the first match. **\$FIND** returns the integer position of the first character after the end of the matching substring. **CHARINDEX, POSITION, and INSTR** return the integer position of the first character of the matching substring. **CHARINDEX, \$FIND, and INSTR** support specifying a starting point for substring search. **INSTR** also support specifying the substring occurrence from that starting point.

The following example demonstrates these four functions, specifying all optional arguments. Note that the positions of *string* and *substring* differ in these functions:

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

For a list of functions that search for a substring, refer to [String Manipulation](#).

Examples

In the following example, *string* contains the string “ABCDEFGH” and *substring* contains the string “BCD”. The **\$FIND** function returns the value 5, indicating the position of the character (“E”) that follows “BCD”:

```
SELECT $FIND('ABCDEFGH','BCD') AS SubPoint
```

The following example searches the numeric 987654321 for the number 7. It returns 4, the position following the *substring*:

```
SELECT $FIND(987654321,7) AS SubPoint
```

The following example returns 3, the position of the character that follow the first instance of the *substring* "AA":

```
SELECT $FIND('AAAAAA','AA') AS SubPoint
```

In the following example, **\$FIND** searches for a substring that is not in the string. It returns zero (0):

```
SELECT $FIND('AABBCCDD','AC') AS SubPoint
```

In the following example, **\$FIND** begins its search with the seventh character. This example returns 14, the position of the character that follows the next occurrence of "R":

```
SELECT $FIND('EVERGREEN FOREST','R',7) AS SubPoint
```

In the following example, **\$FIND** begins its search after the last character in string. It returns zero (0):

```
SELECT $FIND('ABCDEFGF','G',10) AS SubPoint
```

The following Embedded SQL example shows that a *start* less than 1 is treated as 1:

```
SET a="ABCDEFGF"
SET b="F"
&sql(SELECT
  $FIND(:a,:b),
  $FIND(:a,:b,1),
  $FIND(:a,:b,0),
  $FIND(:a,:b,-35)
INTO :a1,:a2,:a3,:a4)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The input string: ",a
  WRITE !,"Two-arg: ",a1
  WRITE !,"3rd arg 1: ",a2
  WRITE !,"3rd arg 0: ",a3
  WRITE !,"3rd arg negative: ",a4 }
```

The following Embedded SQL example uses **\$FIND** to search a string containing the Unicode character for pi, \$CHAR(960). The first **\$FIND** returns 5, the character following pi. The second **\$FIND** also returns 5; it begins its search at character 4, which happens to be pi, the character sought. The third **\$FIND** begins its search at character 5; it returns 13, the position following the next occurrence of pi. Note that position 13 is returned, even though position 12 is the last character in the string:

```
IF $SYSTEM.Version.IsUnicode() {
SET a="QT "_$CHAR(960)_" HONEY "_$CHAR(960)
SET b=$CHAR(960)
&sql(SELECT
  $FIND(:a,:b),
  $FIND(:a,:b,4),
  $FIND(:a,:b,5)
INTO :a1,:a2,:a3)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The input string: ",a
  WRITE !,"From beginning: ",a1
  WRITE !,"From position 4: ",a2
  WRITE !,"From position 5: ",a3 }
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- [CHARINDEX](#) function
- [INSTR](#) function
- [POSITION](#) function

- [String Manipulation](#)

FLOOR

A numeric function that returns the largest integer less than or equal to a given numeric expression.

```
FLOOR(numeric-expression)
{fn FLOOR(numeric-expression)}
```

Arguments

<i>numeric-expression</i>	A number whose floor is to be calculated.
---------------------------	---

Description

FLOOR returns the nearest integer value less than or equal to *numeric-expression*. The returned value has the same data type as *numeric-expression* and a scale of 0. When *numeric-expression* is a NULL value, an empty string ("), or a nonnumeric string, **FLOOR** returns NULL.

Note that **FLOOR** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

This function can also be invoked from ObjectScript using the **FLOOR()** method call:

```
$SYSTEM.SQL.FLOOR(numeric-expression)
```

Examples

The following examples show how **FLOOR** converts a fraction to its floor integer:

```
SELECT FLOOR(167.111) AS FloorNum1,
       FLOOR(167.456) AS FloorNum2,
       FLOOR(167.999) AS FloorNum3
```

all return 167.

```
SELECT {fn FLOOR(167.00)} AS FloorNum1,
       {fn FLOOR(167)} AS FloorNum2
```

return 167.

```
SELECT FLOOR(-167.111) AS FloorNum1,
       FLOOR(-167.456) AS FloorNum2,
       FLOOR(-167.999) AS FloorNum3
```

all return -168.

```
SELECT FLOOR(-167.00) AS FloorNum
```

returns -167.

The following example uses a subquery to reduce a large table of US Zip Codes (postal codes) to one representative city for each floor Latitude integer:

```
SELECT City,State,FLOOR(Latitude) AS FloorLatitude
FROM (SELECT City,State,Latitude,FLOOR(Latitude) AS FloorNum
      FROM Sample.USZipCode)
GROUP BY FloorNum
ORDER BY FloorNum DESC
```

See Also

- [CEILING](#)
- [ROUND](#)

GETDATE

A date/time function that returns the current local date and time.

```
GETDATE([precision])
```

Arguments

<i>precision</i>	<i>Optional</i> — A positive integer that specifies the time precision as the number of digits of fractional seconds. The default is 0 (no fractional seconds); this default is configurable. A <i>precision</i> value is optional, the parentheses are mandatory.
------------------	--

Description

GETDATE returns the current local date and time for this [timezone](#) as a timestamp; it adjusts for local time variants, such as [Daylight Saving Time](#).

GETDATE returns a [timestamp](#) in %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff).

To change the default datetime string format, use the [SET OPTION](#) command with the various date and time options.

GETDATE can be used in a **SELECT** statement select list or in the **WHERE** clause of a query. In designing a report, **GETDATE** can be used to print the current date and time each time the report is produced. **GETDATE** is also useful for tracking activity, such as logging the time that a transaction occurred.

GETDATE can be used in **CREATE TABLE** to specify a field's default value. **GETDATE** is a synonym for [CURRENT_TIMESTAMP](#) and is provided for compatibility with Sybase and Microsoft SQL Server.

The [CURRENT_TIMESTAMP](#) and [NOW](#) functions can also be used to return the current local date and time as a timestamp. [CURRENT_TIMESTAMP](#) supports precision, [NOW](#) does not support precision.

To return just the current date, use [CURDATE](#) or [CURRENT_DATE](#). To return just the current time, use [CURRENT_TIME](#) or [CURTIME](#). These functions use the DATE or TIME data type. None of these functions support precision.

A [TIMESTAMP](#) data type stores and displays its value in the same format. The [TIME](#) and [DATE](#) data types store their values as integers in [\\$HOROLOG](#) format. They can be displayed in either Display format or Logical (storage) format. You can use the [CAST](#) or [CONVERT](#) function to change the data type of dates and times.

Universal Time (UTC)

GETDATE returns the current local date and time. All Caché SQL timestamp, date, and time functions except [GETUTCDATE](#) are specific to the local time zone setting. [GETUTCDATE](#) returns the current UTC (universal) date and time as a [TIMESTAMP](#) value. You can also use the ObjectScript [\\$ZTIMESTAMP](#) special variable to get a current timestamp that is universal (independent of time zone).

Fractional Seconds Precision

GETDATE can return up to nine digits of precision. The number of digits of precision returned is set using the *precision* argument. The default for the *precision* argument can be configured using the following:

- [SET OPTION](#) with the [TIME_PRECISION](#) option.
- The `$SYSTEM.SQL.SetDefaultTimePrecision()` method call.
- Go to the Management Portal, select **Configuration, SQL and Object Settings, General SQL Settings ([System] > [Configuration] > [General SQL Settings])**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; *precision* digits in excess of the precision available on your system are returned as zeroes.

Fractional seconds are always truncated, not rounded, to the specified precision.

Examples

The following example returns the current date and time in `TIMESTAMP` format:

```
SELECT GETDATE() AS DateTime
```

The following example returns the current date and time with two digits of precision:

```
SELECT GETDATE(2) AS DateTime
```

The following Embedded SQL example compares local (time zone specific) and universal (time zone independent) timestamps:

```
&sql(SELECT GETDATE(),GETUTCDATE() INTO :a,:b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Local timestamp is:  ",a
  WRITE !,"UTC timestamp is:   ",b
  WRITE !,"$ZTIMESTAMP is:     ",$ZDATETIME($ZTIMESTAMP,3,,3)
}
```

The following example sets the `LastUpdate` field in the selected row of the `Orders` table to the current system date and time. **GETDATE** returns the current date and time as an ODBC timestamp:

```
UPDATE Orders SET LastUpdate = GETDATE()
WHERE Orders.OrderNumber=:ord
```

In the following example, the **CREATE TABLE** statement uses **GETDATE** to set a default value for the `StartDate` field:

```
CREATE TABLE Employees(
  EmpId      INT NOT NULL,
  LastName   CHAR(40) NOT NULL,
  FirstName  CHAR(20) NOT NULL,
  StartDate  TIMESTAMP DEFAULT GETDATE())
```

See Also

- SQL concepts: [Data Type](#), [Date and Time Constructs](#)
- SQL timestamp functions: [CAST](#), [CONVERT](#), [CURRENT_TIMESTAMP](#), [GETUTCDATE](#), [NOW](#), [SYSDATE](#), [TIMESTAMPADD](#), [TIMESTAMPDIFF](#), [TO_TIMESTAMP](#)
- SQL current date and time functions: [CURDATE](#), [CURRENT_DATE](#), [CURRENT_TIME](#), [CURTIME](#)
- ObjectScript: [\\$ZDATETIME](#) function, [\\$SHOROLOG](#) special variable, [\\$ZTIMESTAMP](#) special variable

GETUTCDATE

A date/time function that returns the current UTC date and time.

```
GETUTCDATE([precision])
```

Arguments

<i>precision</i>	<i>Optional</i> — A positive integer that specifies the time precision as the number of digits of fractional seconds. The default is 0 (no fractional seconds); this default is configurable.
------------------	---

Description

GETUTCDATE returns Universal Time Constant (UTC) date and time as a timestamp. Because UTC time is the same everywhere on the planet, does not depend on the local timezone and is not subject to local time variants (such as [Daylight Saving Time](#)), this function is useful for applying consistent timestamps when users in different time zones access the same database.

GETUTCDATE returns a [timestamp](#) in %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff).

To change the default datetime string format, use the [SET OPTION](#) command with the various date and time options.

Typical uses for **GETUTCDATE** are in the **SELECT** statement select list or in the **WHERE** clause of a query. In designing a report, **GETUTCDATE** can be used to print the current date and time each time the report is produced.

GETUTCDATE is also useful for tracking activity, such as logging the time that a transaction occurred.

GETUTCDATE can be used in **CREATE TABLE** to specify a field's default value.

Other SQL Functions

GETUTCDATE returns the current UTC date and time as a timestamp.

All other timestamp functions return the local date and time: [GETDATE](#), [CURRENT_TIMESTAMP](#), [NOW](#), and [SYSDATE](#) return the current local date and time as a timestamp.

GETDATE and **CURRENT_TIMESTAMP** provide a *precision* argument.

NOW, argumentless **CURRENT_TIMESTAMP**, and **SYSDATE** do not provide a *precision* argument; they take the system-wide default time precision.

[CURDATE](#) and [CURRENT_DATE](#) return the current local date. [CURTIME](#) and [CURRENT_TIME](#) return the current local time. These functions use the DATE or TIME data type. None of these functions support precision.

A **TIMESTAMP** data type stores and displays its value in the same format. The **TIME** and **DATE** data types store their values as integers in [\\$HOROLOG](#) format and can be displayed in a variety of formats.

Note that all Caché SQL timestamp functions except **GETUTCDATE** are specific to the local time zone setting. To get a current timestamp that is universal (independent of time zone) you can also use the ObjectScript [\\$ZTIMESTAMP](#) special variable. Note that you can set the *precision* for **GETUTCDATE**; [\\$ZTIMESTAMP](#) always returns a precision of 3.

Fractional Seconds Precision

GETUTCDATE can return up to nine digits of precision. The number of digits of precision returned is set using the *precision* argument. The default for the *precision* argument can be configured using the following:

- [SET OPTION](#) with the `TIME_PRECISION` option.
- The `$SYSTEM.SQL.SetDefaultTimePrecision()` method call.

- Go to the Management Portal, select **Configuration, SQL and Object Settings, General SQL Settings ([System] > [Configuration] > [General SQL Settings])**. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; *precision* digits in excess of the precision available on your system are returned as zeroes.

Fractional seconds are always truncated, not rounded, to the specified precision.

Examples

The following example returns the current date and time as a UTC timestamp and as a local timestamp, both in **TIMESTAMP** format:

```
SELECT GETUTCDATE() AS UTCDateTime,
       GETDATE() AS LocalDateTime
```

The following example returns the current UTC date and time with fractional seconds having two digits of precision:

```
SELECT GETUTCDATE(2) AS DateTime
```

The following Embedded SQL example compares local (time zone specific) and universal (time zone independent) timestamps:

```
&sql(SELECT GETDATE(),GETUTCDATE() INTO :a,:b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Local timestamp is: ",a
  WRITE !,"UTC timestamp is:   ",b
  WRITE !,"$ZDATETIME is:     ",$ZDATETIME($ZTIMESTAMP,3,,3)
}
```

The following example sets the **LastUpdate** field in the selected row of the **Orders** table to the current UTC date and time. **GETUTCDATE** returns the current UTC date and time as an ODBC timestamp:

```
UPDATE Orders SET LastUpdate = GETUTCDATE()
WHERE Orders.OrderNumber=:ord
```

In the following example, the **CREATE TABLE** statement uses **GETUTCDATE** to set a default value for the **OrderRcvd** field:

```
CREATE TABLE Orders(
  OrderId      INT NOT NULL,
  ItemName     CHAR(40) NOT NULL,
  Quantity     INT NOT NULL,
  OrderRcvd    TIMESTAMP DEFAULT GETUTCDATE())
```

See Also

- SQL concepts: [Data Type](#), [Date and Time Constructs](#)
- SQL timestamp functions: [CAST](#), [CONVERT](#), [CURRENT_TIMESTAMP](#), [GETDATE](#), [NOW](#), [SYSDATE](#), [TIMESTAMPADD](#), [TIMESTAMPDIFF](#), [TO_TIMESTAMP](#)
- SQL current date and time functions: [CURDATE](#), [CURRENT_DATE](#), [CURRENT_TIME](#), [CURTIME](#)
- ObjectScript: [\\$ZDATETIME](#) function, [\\$SHOROLOG](#) special variable, [\\$ZTIMESTAMP](#) special variable

GREATEST

A function that returns the greatest value from a list of values.

```
GREATEST(expression,expression[,...])
```

Arguments

<i>expression</i>	An expression that resolves to a number or a string. The values of these expressions are compared to each other. An <i>expression</i> can be a field name, a literal, an arithmetic expression, a host variable, or an object reference. You can list up to 140 comma-separated expressions.
-------------------	--

Description

GREATEST returns the greatest value from a comma-separated list of *expression* values. Expressions are evaluated in left-to-right order. If only one *expression* is provided, **GREATEST** returns that value. If any *expression* is NULL, **GREATEST** returns NULL.

If all of the *expression* values resolve to canonical numbers, they are compared in numeric order. If a quoted string contains a number in canonical format, it is compared in numeric order. However, if a quoted string contains a number not in canonical format (for example, '00', '0.4', or '+4'), it is compared as a string. String comparisons are performed character-by-character in collation order. Any string value is greater than any numeric value.

The empty string is greater than any numeric value, but less than any other string value.

If the returned value is a number, **GREATEST** returns it in canonical format (leading and trailing zeros removed, etc.). If the returned value is a string, **GREATEST** returns it unchanged, including any leading or trailing blanks.

The inverse function of **GREATEST** is [LEAST](#).

Data Type of Returned Value

If the data types of the *expression* values are different, the data type returned is the type most compatible with all of the possible return values, the data type with the highest [data type precedence](#). For example, if one *expression* is an integer and another *expression* is a fractional number, **GREATEST** returns a value with data type NUMERIC. This is because NUMERIC is the data type with the highest precedence that is compatible with both.

Examples

In the following example, each **GREATEST** compares three canonical numbers:

```
SELECT GREATEST(22,2.2,-21) AS HighNum,
       GREATEST('2.2','22','-21') AS HighNumStr
```

In the following example, each **GREATEST** compare three numeric strings. However, each **GREATEST** contains one string that is non-canonical; these non-canonical values are compared as character strings. A character string is always greater than a number:

```
SELECT GREATEST('22','+2.2','-21'),
       GREATEST('0.2','22','-21')
```

In the following example, each **GREATEST** compare three strings and returns the value with the highest collation sequence:

```
SELECT GREATEST('A','a',''),
       GREATEST('a','ab','abc'),
       GREATEST('#','0','7'),
       GREATEST('##','00','77')
```

The following example compares two dates, treated as canonical numbers: the date of birth as a `SHOROLOG` integer, and the integer 58073 converted to a date. It returns the date of birth for each person born in the 21st century. Anyone born before January 1, 2000 is displayed with the default birth date of December 31, 1999:

```
SELECT Name,GREATEST(DOB,TO_DATE(58073)) AS NewMillenium
FROM Sample.Person
```

See Also

- SQL functions: [LEAST CONVERT TO_NUMBER](#)

HOURL

A time function that returns the hour for a datetime expression.

```
{fn HOUR(time-expression)}
```

Arguments

<i>time-expression</i>	An expression that is the name of a column, the result of another scalar function, or a string or numeric literal. It must resolve either to a datetime string or a time integer, where the underlying data type can be represented as %Time or %TimeStamp.
------------------------	---

Description

HOURL returns an integer specifying the hour for a given time or datetime value. The hour is calculated for a [\\$HOROLOG](#) or [\\$ZTIMESTAMP](#) value, an ODBC format date string, or a timestamp.

A *time-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

To change this default time format, use the [SET OPTION](#) command.

Note that you can supply a time integer (number of elapsed seconds), but not a time string (hh:mm:ss). You must supply a datetime string (yyyy-mm-dd hh:mm:ss). You can omit the seconds (:ss) or minutes and seconds (mm:ss) portion of a datetime string and still return the hour portion. The *time-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

Hours are expressed in 24-hour time. The hours (hh) portion should be an integer in the range from 0 through 23. There is, however, no range checking for user-supplied values. Numbers greater than 23, negative numbers, and fractions are returned as specified. Leading zeros are optional on input; leading zeros are suppressed on output.

HOURL returns a value of 0 hours when the hours portion is '0', '00', or a nonnumeric value. Zero hours is also returned if no time expression is supplied, or if the hours portion of the time expression is omitted (':mm:ss' or '::~ss').

The same time information can be returned using [DATEPART](#) or [DATENAME](#).

This function can also be invoked from ObjectScript using the **HOURL()** method call:

```
$SYSTEM.SQL.HOURL(time-expression)
```

Examples

The following examples both return the number 18 because the *time-expression* value is 18:45:38:

```
SELECT {fn HOUR('2000-02-16 18:45:38')} AS Hour_Given
```

```
SELECT {fn HOUR(67538)} AS Hour_Given
```

The following example also returns 18. The seconds (or minutes and seconds) portion of the time value can be omitted.

```
SELECT {fn HOUR('2000-02-16 18:45')} AS Hour_Given
```

The following example returns 0 hours, because the time portion of the datetime string has been omitted:

```
SELECT {fn HOUR('2000-02-16')} AS Hour_Given
```

The following examples all return the hours portion of the current time:

```
SELECT {fn HOUR(CURRENT_TIME)} AS H_CurrentT,
       {fn HOUR({fn CURTIME()})} AS H_CurT,
       {fn HOUR({fn NOW()})} AS H_Now,
       {fn HOUR($HOROLOG)} AS H_Horolog,
       {fn HOUR($ZTIMESTAMP)} AS H_ZTS
```

Note that **\$ZTIMESTAMP** returns Coordinated Universal Time (UTC). The other *time-expression* values return the local time.

The following example shows that leading zeros are suppressed. The first HOUR function returns a length 2, the others return a length of 1. An omitted time is considered to be 0 hours, which has a length of 1:

```
SELECT LENGTH({fn HOUR('2004-02-05 11:45')}),
       LENGTH({fn HOUR('2004-02-15 03:45')}),
       LENGTH({fn HOUR('2004-02-15 3:45')}),
       LENGTH({fn HOUR('2004-02-15')})
```

The following Embedded SQL example shows that the **HOUR** function recognizes the TimeSeparator character specified for the locale:

```
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeSeparator",".")
&sql(SELECT {fn HOUR('2000-02-16 18.45.38')} INTO :a)
WRITE "hour=",a
```

See Also

- SQL concepts: [Data Type Date and Time Constructs](#)
- SQL functions: [MINUTE SECOND CURRENT_TIME CURTIME NOW DATEPART DATENAME](#)
- ObjectScript function: [\\$ZTIME](#)
- ObjectScript special variables: [\\$HOROLOG \\$ZTIMESTAMP](#)

IFNULL

A function that tests for NULL and returns the appropriate expression.

```
IFNULL(expression-1,expression-2 [,expression-3])
{fn IFNULL(expression-1,expression-2)}
```

Arguments

<i>expression-1</i>	The expression to be evaluated to determine if it is NULL or not.
<i>expression-2</i>	An expression that is returned if <i>expression-1</i> is NULL.
<i>expression-3</i>	<i>Optional</i> — An expression that is returned if <i>expression-1</i> is not NULL. If <i>expression-3</i> is not specified, a NULL value is returned when <i>expression-1</i> is not NULL.

Description

Caché supports **IFNULL** as both an SQL general function and an ODBC scalar function. Note that while these two perform very similar operations, they are functionally different. The SQL general function supports three arguments. The ODBC scalar function supports two arguments. The two-argument forms of the SQL general function and the ODBC scalar function are not the same; they return different values when *expression-1* is not null.

The SQL general function evaluates whether *expression-1* is NULL. It never returns *expression-1*:

- If *expression-1* is NULL, *expression-2* is returned.
- If *expression-1* is not NULL, *expression-3* is returned.
- If *expression-1* is not NULL, and there is no *expression-3*, NULL is returned.

The ODBC scalar function evaluates whether *expression-1* is NULL. It either returns *expression-1* or *expression-2*:

- If *expression-1* is NULL, *expression-2* is returned.
- If *expression-1* is not NULL, *expression-1* is returned.

The possible data type(s) of *expression-2* and *expression-3* must be compatible with the data type of *expression-1*. If *expression-2* and *expression-3* have different data types, the data type **IFNULL** returns is the data type with the higher (more inclusive) [data type precedence](#). If *expression-2* and *expression-3* have different length, precision, or scale, **IFNULL** returns the greater length, precision, or scale of the two expressions.

Refer to [NULL](#) section of the “Language Elements” chapter of *Using Caché SQL* for further details on NULL handling.

Display-to-Logical Conversion

When executing in SELECTMODE=DISPLAY, SQL does not convert input arguments from Display to Logical. Therefore, if *expression-1* is a %List field, the *expression-2* field must be specified as a %List. In the following example FavoriteColors is a %List field, so the 'No Preference' value must be specified as a %List:

```
SELECT Name,
       IFNULL(FavoriteColors,$LISTBUILD('No Preference')) AS ColorPref
FROM Sample.Person
```

If the SELECTMODE is LOGICAL or ODBC, or SELECTMODE is RUNTIME, and RUNTIMEMODE is DISPLAY, this Display-to-Logical conversion is performed, so the *expression-2* field can be specified as a string. Both Dynamic SQL and Embedded SQL perform this Display-to-Logical conversion, as shown in the following two examples:

```

ZNSPACE "SAMPLES"
SET myquery=3
  SET myquery(1)="SELECT Name,"
  SET myquery(2)="IFNULL(FavoriteColors,'No Preference') AS ColorChoice "
  SET myquery(3)="FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"

&sql(DECLARE C1 CURSOR FOR
  SELECT Name,IFNULL(FavoriteColors,'No Preference')
  INTO :name,:colorpref
  FROM Sample.Person
  ORDER BY Name )
&sql(OPEN C1)
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
  WRITE name, ": ", colorpref,!
  &sql(FETCH C1)
}
&sql(CLOSE C1)

```

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual [equality \(or non-equality\) condition comparison](#).

SQL Function	Comparison Test	Return Value
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2,ex3) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex

Examples

In the following example, the general function and the ODBC scalar function both returns the second expression (99) because the first expression is NULL:

```
SELECT IFNULL(NULL,99) AS NullGen,{fn IFNULL(NULL,99)} AS NullODBC
```

In the following example, the general function and the ODBC scalar function examples return different values. The general function returns <null> because the first expression is not NULL. The ODBC example returns the first expression (33) because the first expression is not NULL:

```
SELECT IFNULL(33,99) AS NullGen,{fn IFNULL(33,99)} AS NullODBC
```

The following Dynamic SQL example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns NULL:

```
ZNSPACE "SAMPLES"
SET myquery=3
  SET myquery(1)="SELECT Name,"
  SET myquery(2)="IFNULL(FavoriteColors,'No Preference') AS ColorChoice "
  SET myquery(3)="FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns the value of FavoriteColors:

```
ZNSPACE "SAMPLES"
SET myquery=3
  SET myquery(1)="SELECT Name,"
  SET myquery(2)="IFNULL(FavoriteColors,'No Preference',FavoriteColors) AS ColorChoice "
  SET myquery(3)="FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=2
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns the string 'Preference':

```
SELECT Name,
IFNULL(FavoriteColors,'No Preference','Preference') AS ColorPref
FROM Sample.Person
```

The following ODBC syntax examples return the string 'No Preference' if FavoriteColors is NULL, otherwise they return the FavoriteColors data value:

```
SELECT Name,
  {fn IFNULL(FavoriteColors,$LISTBUILD('No Preference'))} AS ColorPref
FROM Sample.Person
```

```
ZNSPACE "SAMPLES"
SET myquery=3
  SET myquery(1)="SELECT Name,"
  SET myquery(2)="{fn IFNULL(FavoriteColors,'No Preference')}" AS ColorChoice "
  SET myquery(3)="FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following example uses **IFNULL** in the **WHERE** clause. It selects people under the age of 21 who do not have favorite color preferences. If FavoriteColors is NULL, **IFNULL** returns the Age field value, which is used for the condition test:

```
SELECT Name, FavoriteColors, Age
FROM Sample.Person
WHERE 21 > IFNULL(FavoriteColors, Age)
ORDER BY Age
```

Refer to the [NULL](#) predicate (**IS NULL**, **IS NOT NULL**) for similar functionality.

See Also

- [CASE](#) command
- [COALESCE](#) function
- [ISNULL](#) function
- [NULLIF](#) function
- [NVL](#) function
- [NULL](#) predicate

INSTR

A string function that returns the position of a substring within a string, with optional search start point and occurrence count.

```
INSTR(string, substring[, start[, occurrence]])
```

Arguments

<i>string</i>	The string expression within which to search for <i>substring</i> . It can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
<i>substring</i>	A substring that is believed to occur within <i>string</i> .
<i>start</i>	<i>Optional</i> — The starting point for substring search, specified as a positive integer. A character count from the beginning of <i>string</i> , counting from 1. To search from the beginning of <i>string</i> , omit this argument or specify a <i>start</i> of 1. A <i>start</i> value of 0, the empty string, NULL, or a nonnumeric value cause INSTR to return 0. Specifying <i>start</i> as a negative number causes INSTR to return <null>.
<i>occurrence</i>	<i>Optional</i> — A non-zero integer that specifies which occurrence of <i>substring</i> to return when searching from the <i>start</i> position. The default is to return the position of the first occurrence.

Description

INSTR searches *string* for *substring*, and returns the position of the first character of *substring*. The position is returned as an integer, counting from the beginning of *string*. If *substring* is not found, 0 (zero) is returned. **INSTR** returns NULL if passed a NULL value for either argument.

INSTR supports specifying *start* as the starting point for substring search. **INSTR** also support specifying the substring *occurrence* from that starting point.

INSTR is case-sensitive. Use one of the case-conversion functions to locate both uppercase and lowercase instances of a letter or character string.

This function can also be invoked from ObjectScript using the **INSTR()** method call:

```
WRITE $SYSTEM.SQL.INSTR("The broken brown briefcase", "br", 6, 2)
```

INSTR, CHARINDEX, POSITION, and \$FIND

INSTR, **CHARINDEX**, **POSITION**, and **\$FIND** all search a string for a specified substring and return an integer position corresponding to the first match. **CHARINDEX**, **POSITION**, and **INSTR** return the integer position of the first character of the matching substring. **\$FIND** returns the integer position of the first character after the end of the matching substring. **CHARINDEX**, **\$FIND**, and **INSTR** support specifying a starting point for substring search. **INSTR** also support specifying the substring occurrence from that starting point.

The following example demonstrates these four functions, specifying all optional arguments. Note that the positions of *string* and *substring* differ in these functions:

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br', 'The broken brown briefcase', 6) AS Charindex,
       $FIND('The broken brown briefcase', 'br', 6) AS Find,
       INSTR('The broken brown briefcase', 'br', 6, 2) AS Inst
```

For a list of functions that search for a substring, refer to [String Manipulation](#).

Examples

The following example returns 11, because “b” is the 11th character in the string:

```
SELECT INSTR('The quick brown fox','b',1) AS PosInt
```

The following example returns the length of the last name (surname) for each name in the `Sample.Person` table. It locates the comma used to separate the last name from the rest of the name field, then subtracts 1 from that position:

```
SELECT Name,  
INSTR(Name,',',1)-1 AS LNameLen  
FROM Sample.Person
```

The following example returns the position of the first instance of the letter “B” in each name in the `Sample.Person` table. Because **INSTR** is case-sensitive, the **%SQLUPPER** function is used to convert all name values to uppercase before performing the search. Because **%SQLUPPER** adds a blank space at the beginning of a string, this example subtracts 1 to get the actual letter position. Searches that do not locate the specified string return zero (0); in this example, because of the subtraction of 1, the value displayed for these searches is -1:

```
SELECT Name,  
INSTR(%SQLUPPER(Name),'B',1)-1 AS BPos  
FROM Sample.Person
```

See Also

- [CHARINDEX](#) function
- [\\$FIND](#) function
- [POSITION](#) function
- [String Manipulation](#)

%INTERNAL

A format-transformation function that returns an expression in LOGICAL format.

```
%INTERNAL(expression)
```

```
%INTERNAL expression
```

Arguments

<i>expression</i>	The expression to be converted. A field name, an expression containing a field name, or a function that returns a value in a convertible data type, such as DATE or %List.
-------------------	--

Description

%INTERNAL converts *expression* to LOGICAL format, regardless of the current select mode (display mode). The LOGICAL format is the in-memory format of data (the format upon which operations are performed). **%INTERNAL** is commonly used on a **SELECT** list *select-item*.

%INTERNAL can be used in a **WHERE** clause, but this use is strongly discouraged because using **%INTERNAL** prevents the use of indexes on the specified field, and **%INTERNAL** forces all comparisons to be case-sensitive, even if the field has default collation.

Applying **%INTERNAL** changes the column header name to a value such as “Expression_1”; it is therefore usually desirable to specify a column name alias, as shown in the examples below.

%INTERNAL converts a value of data type %Date to an INTEGER data type value. **%INTERNAL** converts a value of data type %Time to a NUMERIC (15,9) data type value. This conversion is provided because an ODBC or JDBC client does not recognize Caché logical %Date and %Time values.

Whether **%INTERNAL** converts a date depends on the data type returned by the date field or function. **%INTERNAL** converts [CURDATE](#), [CURRENT_DATE](#), [CURTIME](#), and [CURRENT_TIME](#) values. It does not convert [CURRENT_TIMES-TAMP](#), [GETDATE](#), [GETUTCDATE](#), [NOW](#), and [\\$HOROLOG](#) values.

A stream field cannot be specified as an argument to Caché unary functions, including all format-transformation functions, with the exception of **%INTERNAL**. The **%INTERNAL** function permits a [stream field](#) as an *expression* value, but performs no operation on that stream field.

%INTERNAL is a Caché SQL extension.

To convert an *expression* to DISPLAY format, regardless of the current select mode, use the [%EXTERNAL](#) function. To convert an *expression* to ODBC format, regardless of the current select mode, use the [%ODBCOUT](#) function.

For further details on display format options, refer to “[Data Display Options](#)” in the “Caché SQL Basics” chapter of *Using Caché SQL*.

Examples

The following Dynamic SQL example returns Date of Birth (DOB) data values in the current select mode format, and the same data using the **%INTERNAL** function. For the purpose of demonstration, in this program the [%SelectMode](#) value is determined randomly for each invocation:

```

ZNSPACE "SAMPLES"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=$RANDOM(3)
  IF tStatement.%SelectMode=0 {WRITE "Select mode LOGICAL",! }
  ELSEIF tStatement.%SelectMode=1 {WRITE "Select mode ODBC",! }
  ELSEIF tStatement.%SelectMode=2 {WRITE "Select mode DISPLAY",! }
SET myquery = 2
SET myquery(1) = "SELECT TOP 5 DOB,%INTERNAL(DOB) AS IntDOB "
SET myquery(2) = "FROM Sample.Person"
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"

```

The following examples show the two syntax forms for this function; they are otherwise identical. They specify the **%EXTERNAL** (DISPLAY format), **%INTERNAL** (LOGICAL format), and **%ODBCOUT** (ODBC format) of a %List field:

```

SELECT TOP 10 %EXTERNAL(FavoriteColors) AS ExtColors,
              %INTERNAL(FavoriteColors) AS IntColors,
              %ODBCOUT(FavoriteColors) AS ODBCColors
FROM Sample.Person

```

```

SELECT TOP 10 %EXTERNAL FavoriteColors AS ExtColors,
              %INTERNAL FavoriteColors AS IntColors,
              %ODBCOUT FavoriteColors AS ODBCColors
FROM Sample.Person

```

See Also

- [%EXTERNAL, %ODBCIN, %ODBCOUT](#)
- [SQL concepts: Data Types, Date and Time Constructs](#)

ISNULL

A function that tests for NULL and returns the appropriate expression.

```
ISNULL(check-expression, replace-expression)
```

Arguments

<i>check-expression</i>	The expression to be evaluated.
<i>replace-expression</i>	An expression that is returned if <i>check-expression</i> is NULL.

Description

ISNULL evaluates *check-expression* and returns one of two values:

- If *check-expression* is NULL, *replace-expression* is returned.
- If *check-expression* is not NULL, *check-expression* is returned.

The possible data type(s) of *replace-expression* must be compatible with the data type of *check-expression*. The data type returned in DISPLAY mode or ODBC mode is determined by the data type of *check-expression*.

Note that the **ISNULL** function is the same as the **NVL** function, which is provided for Oracle compatibility.

Refer to [NULL](#) section of the “Language Elements” chapter of *Using Caché SQL* for further details on NULL handling.

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual [equality \(or non-equality\) condition comparison](#).

SQL Function	Comparison Test	Return Value
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2,ex3) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex

Examples

In the following example, the first **ISNULL** returns the second expression (99) because the first expression is NULL. The second **ISNULL** returns the first expression (33) because the first expression is not NULL:

```
SELECT ISNULL(NULL,99) AS IsNullT,ISNULL(33,99) AS IsNullF
```

The following Dynamic SQL example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns the value of FavoriteColors:

```
ZNSPACE "SAMPLES"
SET myquery=3
  SET myquery(1)="SELECT Name,"
  SET myquery(2)="ISNULL(FavoriteColors,'No Preference') AS ColorChoice "
  SET myquery(3)="FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

Compare the behavior of **ISNULL** with **IFNULL**:

```
ZNSPACE "SAMPLES"
SET myquery=3
  SET myquery(1)="SELECT Name,"
  SET myquery(2)="IFNULL(FavoriteColors,'No Preference') AS ColorChoice "
  SET myquery(3)="FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

See Also

- [CASE](#) command
- [COALESCE](#) function
- [IFNULL](#) function
- [NULLIF](#) function
- [NVL](#) function

ISNUMERIC

A numeric function that tests for a valid number.

```
ISNUMERIC(check-expression)
```

Arguments

<i>check-expression</i>	The expression to be evaluated.
-------------------------	---------------------------------

Description

ISNUMERIC evaluates *check-expression* and returns one of the following values:

- Returns 1 if *check-expression* is a valid number. A valid number can either be a numeric expression or a string that represents a valid number.
 - A numeric expression is first converted to [canonical form](#), resolving multiple leading signs; therefore, a numeric expression such as `+-+-+34` is a valid number.
 - A numeric string is not converted before evaluation. A numeric string must have at most one leading sign to evaluate as a valid number. A numeric string with a trailing decimal point evaluates as a valid number.
- Returns 0 if *check-expression* is not a valid number. Any string that contains a non-numeric character is not a valid number. A numeric string with more than one leading sign, such as `'+-+-+34'`, is not evaluated as a valid number. A Caché encoded list always returns 0, even if its element(s) are valid numbers. An empty string `ISNUMERIC('')` returns 0.
- Returns NULL if *check-expression* is NULL. `ISNUMERIC(NULL)` returns null.

ISNUMERIC generates an SQLCODE -7, exponent out of range error if a scientific notation exponent is greater than 308 (308 – (number of integers - 1)). For example, `ISNUMERIC(1E309)` and `ISNUMERIC(111E307)` both generate this error code. If an exponent numeric string less than or equal to `'1E145'` returns 1; an exponent numeric string greater than `'1E145'` returns 0.

The **ISNUMERIC** function is very similar to the ObjectScript **\$ISVALIDNUM** function. However, these two functions return different values when the input value is NULL.

Examples

In the following example, all of the **ISNUMERIC** functions return 1:

```
SELECT ISNUMERIC(99) AS MyInt,
       ISNUMERIC('-99') AS MyNegInt,
       ISNUMERIC('-0.99') AS MyNegFrac,
       ISNUMERIC('-0.00') AS MyNegZero,
       ISNUMERIC('-0.09'+7) AS MyAdd,
       ISNUMERIC('5E2') AS MyExponent
```

The following example returns NULL if FavoriteColors is NULL; otherwise, it returns 0, because FavoriteColors is not a numeric field:

```
SELECT Name,
       ISNUMERIC(FavoriteColors) AS ColorPref
FROM Sample.Person
```

See Also

- [IFNULL](#) function

- [ISNULL](#) function
- [NULLIF](#) function
- ObjectScript function: [\\$ISVALIDNUM](#)

JSON_ARRAY

A conversion function that returns data as a JSON array.

```
JSON_ARRAY(select-items [NULL ON NULL | ABSENT ON NULL])
```

Arguments

<i>select-items</i>	An expression or a comma-separated list of expressions. These expressions can include column names, aggregate functions, arithmetic expressions, literals, and the literal NULL.
ABSENT ON NULL NULL ON NULL	<i>Optional</i> — A keyword phrase specifying how to represent NULL values in the returned JSON array. NULL ON NULL (the default) represents NULL (absent) data with the word null (not quoted). ABSENT ON NULL omits NULL data from the JSON array; it does not leave a placeholder comma. This keyword phrase has no effect on empty string values.

Description

JSON_ARRAY takes an expression or (more commonly) a comma-separated list of expressions and returns a JSON array containing those values. **JSON_ARRAY** can be combined in a **SELECT** statement with other types of select-items. **JSON_ARRAY** can be specified in other locations where an SQL function can be used, such as in a **WHERE** clause.

The returned JSON array has the following format:

```
[ element1 , element2 , element3 ]
```

JSON_ARRAY returns each array element value as either a string (enclosed in double quotes), or a number. Numbers are returned in canonical format. A numeric string is returned as a literal, enclosed in double quotes. All other data types (for example, Date or \$List) are returned as a string.

JSON_ARRAY does not support asterisk (*) syntax as a way to specify all fields in a table. It does support the **COUNT(*)** aggregate function.

The returned JSON array column is labeled as an Expression (by default); you can specify a column alias for a **JSON_ARRAY**.

Select Mode and Collation

The current [%SelectMode](#) property determines the format of the returned JSON array values. By changing the Select Mode, all Date and %List elements are included in the JSON array as strings with that Select Mode format.

You can override the current Select Mode by applying a format-transformation function ([%EXTERNAL](#), [%INTERNAL](#), [%ODBCIN](#), [%ODBCOUT](#)) to individual field names within **JSON_ARRAY**. Applying a format-transformation function to a **JSON_ARRAY** has no effect, because the elements of a JSON array are strings.

You can apply a [collation function](#) to individual field names within **JSON_ARRAY** or to an entire **JSON_ARRAY**:

- A collation function applied to a **JSON_ARRAY** applies the collation after JSON array formatting. Therefore, `%SQLUPPER(JSON_ARRAY(f1, f2))` converts all the JSON array element values to uppercase. `%SQLUPPER(JSON_ARRAY(f1, f2))` inserts a space before the JSON array, not before the elements of the array; therefore it does not force numbers to be parsed as strings. Collation functions that strip punctuation (such as `%ALPHAUP`) remove the enclosing brackets from the returned value, making it no longer a JSON array.
- A collation function applied to an element within a **JSON_ARRAY** applies that collation. Therefore `JSON_ARRAY('Abc', %SQLUPPER('Abc'))` returns `["Abc", " ABC"]` (note leading space); and

`JSON_ARRAY(007,%SQLSTRING(007))` returns `[7," 7"]`. Because `%SQLUPPER` inserts a space before the value, it is generally preferable to specify a case transformation function such as `LCASE` or `UCASE`. You can apply collation to both an element and to the whole array: `%SQLUPPER(JSON_ARRAY('Abc',%SQLSTRING('Abc')))` returns `["ABC"," ABC"]`

ABSENT ON NULL

If you specify the optional `ABSENT ON NULL` keyword phrase, a column value which is `NULL` (or the `NULL` literal) is not included in the JSON array. No placeholder is included in the JSON array. This can result in JSON arrays with different numbers of elements. For example, the following program returns JSON arrays where for some records the 3rd array element is `Age`, and for other records the 3rd element is `FavoriteColors`:

```
SELECT JSON_ARRAY(%ID,Name,FavoriteColors,Age ABSENT ON NULL) FROM Sample.Person
```

If you specify no keyword phrase, the default is `NULL ON NULL`: `NULL` is represented by the word `null` (not delimited by quotes) as a comma-separated array element. Thus all JSON arrays returned by a `JSON_ARRAY` function will have the same number of array elements.

Examples

The following example applies `JSON_ARRAY` to format a JSON array containing a comma-separated list of field values:

```
SELECT TOP 3 JSON_ARRAY(%ID,Name,Age,Home_State) FROM Sample.Person
```

The following example applies `JSON_ARRAY` to format a JSON array with a single element containing the `Name` field values:

```
SELECT TOP 3 JSON_ARRAY(Name) FROM Sample.Person
```

The following example applies `JSON_ARRAY` to format a JSON array containing literals and field values:

```
SELECT TOP 3 JSON_ARRAY('Employee from',%TABLENAME,Name,SSN) FROM Sample.Employee
```

The following example applies `JSON_ARRAY` to format a JSON array containing nulls and field values:

```
SELECT JSON_ARRAY(Name,FavoriteColors) FROM Sample.Person
WHERE Name %STARTSWITH 'S'
```

The following example applies `JSON_ARRAY` to format a JSON array containing field values from joined tables:

```
SELECT TOP 3 JSON_ARRAY(E.%TABLENAME,E.Name,C.%TABLENAME,C.Name)
FROM Sample.Employee AS E,Sample.Company AS C
```

The following Dynamic SQL example sets the ODBC `%SelectMode`, which determines how all fields, including JSON array values are represented. The query overrides this Select Mode for specific JSON array elements by applying the `%EXTERNAL` format-transformation function:

```
ZNSPACE "SAMPLES"
SET myquery = 3
SET myquery(1) = "SELECT TOP 8 DOB,JSON_ARRAY(Name,DOB,FavoriteColors) AS ODBCMode, "
SET myquery(2) = "JSON_ARRAY(Name,DOB,%EXTERNAL(DOB),%EXTERNAL(FavoriteColors)) AS ExternalTrans
"
SET myquery(3) = "FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
WRITE "SelectMode is ODBC",!
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)}
DO rset.%Display()
WRITE !,"End of data"
```

The following example uses **JSON_ARRAY** in a **WHERE** clause to perform a Contains test on multiple columns without using OR syntax:

```
SELECT Name,Home_City,Home_State FROM Sample.Person
WHERE JSON_ARRAY(Name,Home_City,Home_State) [ 'X'
```

See Also

- [SELECT](#) statement
- [WHERE](#) clause
- [JSON_OBJECT](#) function
- [IS JSON](#) predicate condition
- [Overview of Predicates](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*

JSON_OBJECT

A conversion function that returns data as a JSON object.

```
JSON_OBJECT(select-items [NULL ON NULL | ABSENT ON NULL])
```

Arguments

<i>select-items</i>	A key:value pair or a comma-separated list of key:value pairs. A key is a user-specified literal string delimited with single quotes. A value can be a column name, an aggregate function, an arithmetic expression, a numeric or string literal, or the literal NULL.
ABSENT ON NULL NULL ON NULL	<i>Optional</i> — A keyword phrase specifying how to represent NULL values in the returned JSON object. NULL ON NULL (the default) represents NULL (absent) data with the word null (not quoted). ABSENT ON NULL omits NULL data from the JSON object; it removes the key:value pair when value is NULL and does not leave a placeholder comma. This keyword phrase has no effect on empty string values.

Description

JSON_OBJECT takes a comma-separated list of key:value pairs (for example, 'mykey':colname) and returns a JSON object containing those values. You can specify any single-quoted string as a key name; **JSON_OBJECT** does not enforce any naming conventions or uniqueness check for key names. You can specify for value a column name or other expression.

JSON_OBJECT can be combined in a **SELECT** statement with other types of select-items. **JSON_OBJECT** can be specified in other locations where an SQL function can be used, such as in a **WHERE** clause.

A returned JSON object has the following format:

```
{ "key1" : "value1" , "key2" : "value2" , "key3" : "value3" }
```

JSON_OBJECT returns object values as either a string (enclosed in double quotes), or a number. Numbers are returned in canonical format. A numeric string is returned as a literal, enclosed in double quotes. All other data types (for example, Date or \$List) are returned as a string, with the current [%SelectMode](#) determining the format of the returned value.

JSON_OBJECT returns both key and value values in DISPLAY or ODBC mode if that is the select mode for the query.

JSON_OBJECT does not support asterisk (*) syntax as a way to specify all fields in a table.

The returned JSON object column is labeled as an Expression (by default); you can specify a column alias for a **JSON_OBJECT**.

Select Mode and Collation

The current [%SelectMode](#) property determines the format of the returned JSON object values. By changing the Select Mode, all Date and %List values are included in the JSON object as strings with that Select Mode format. You can override the current Select Mode by applying a format-transformation function ([%EXTERNAL](#), [%INTERNAL](#), [%ODBCIN](#), [%ODBCOUT](#)) to individual field names within **JSON_OBJECT**. Applying a format-transformation function to a **JSON_OBJECT** has no effect, because the key:value pairs of a JSON object are strings.

The default collation determines the collation of the returned JSON object values. You can apply a [collation function](#) to a **JSON_OBJECT**, converting both keys and values. Generally, you should not apply a collation function to **JSON_OBJECT** because keys are case-sensitive. Caché applies the collation after JSON object formatting. Therefore,

`%SQLUPPER(JSON_OBJECT('k1':f1, 'k2':f2))` converts all the JSON object key and value strings to uppercase. `%SQLUPPER` inserts a space before the JSON object, not before the values within the object. Collation functions that strip

punctuation (such as %ALPHAUP) remove the enclosing curly braces from the returned value, making it no longer a JSON object.

Within **JSON_OBJECT**, you can apply a collation function to the value portion of a key:value pair. Because %SQLUPPER inserts a space before the value, it is generally preferable to specify a case transformation function such as **LCASE**, **UCASE**, or **LOWER**.

ABSENT ON NULL

If you specify the optional ABSENT ON NULL keyword phrase, a column value which is NULL (or the NULL literal) is not included in the JSON object. No placeholder is included in the JSON object. This can result in JSON objects with different numbers of key:value pairs. For example, the following program returns JSON objects where for some records the 3rd key:value pair is Age, and for other records the 3rd key:value pair is FavoriteColors:

```
SELECT JSON_OBJECT('id':%ID,'name':Name,'colors':FavoriteColors,'years':Age ABSENT ON NULL) FROM
Sample.Person
```

If you specify no keyword phrase, the default is NULL ON NULL: NULL is represented by the word null (not delimited by quotes) as the value of the key:value pair. Thus all JSON objects returned by a **JSON_OBJECT** function will have the same number of key:value pairs.

Examples

The following Dynamic SQL example applies **JSON_OBJECT** to format a JSON object containing field values:

```
ZNSPACE "SAMPLES"
SET myquery = 2
SET myquery(1) = "SELECT TOP 3 JSON_OBJECT('id':%ID,'name':Name,'birth':DOB,"
SET myquery(2) = "'age':Age,'state':Home_State) FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {DO rset.%Print(" ^ ")}
WRITE !,"Total row count=",rset.%ROWCOUNT
```

The following Dynamic SQL example applies **JSON_OBJECT** to format a JSON object containing literals and field values:

```
ZNSPACE "SAMPLES"
SET myquery = 2
SET myquery(1) = "SELECT TOP 3 JSON_OBJECT('lit':'Employee from','t':%TABLENAME,"
SET myquery(2) = "'name':Name,'num':SSN) FROM Sample.Employee"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {DO rset.%Print(" ^ ")}
WRITE !,"Total row count=",rset.%ROWCOUNT
```

The following Dynamic SQL example applies **JSON_OBJECT** to format a JSON object containing nulls and field values:

```
ZNSPACE "SAMPLES"
SET myquery = 2
SET myquery(1) = "SELECT JSON_OBJECT('name':Name,'colors':FavoriteColors) FROM Sample.Person"
SET myquery(2) = " WHERE Name %STARTSWITH 'S'"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {DO rset.%Print(" ^ ")}
WRITE !,"Total row count=",rset.%ROWCOUNT
```

The following Dynamic SQL example sets the ODBC %SelectMode, which determines how all fields, including JSON object values are represented. The query overrides this Select Mode for specific **JSON_OBJECT** values by applying the **%EXTERNAL** format-transformation function:

```

ZNSPACE "SAMPLES"
SET myquery = 3
  SET myquery(1) = "SELECT TOP 8 JSON_OBJECT('ODBCBday':DOB,'DispBday':%EXTERNAL(DOB)), "
  SET myquery(2) = "JSON_OBJECT('ODBCcolors':FavoriteColors,'DispColors':%EXTERNAL(FavoriteColors))"
"
  SET myquery(3) = "FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=1
WRITE "SelectMode is ODBC",!
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)}
DO rset.%Display()
WRITE !,"End of data"

```

The following Dynamic SQL example applies **JSON_OBJECT** to format a JSON object containing field values from joined tables:

```

ZNSPACE "SAMPLES"
SET myquery = 2
SET myquery(1) = "SELECT TOP 3 JSON_OBJECT('e.t':E.%TABLENAME,'e.name':E.Name,'c.t':C.%TABLENAME,"
SET myquery(2) = "'c.name':C.Name) FROM Sample.Employee AS E,Sample.Company AS C"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {DO rset.%Print(" ^ ")}
WRITE !,"Total row count=",rset.%ROWCOUNT

```

The following Dynamic SQL example uses **JSON_OBJECT** in a **WHERE** clause to perform a Contains test on multiple columns without using OR syntax:

```

ZNSPACE "SAMPLES"
SET myquery = 2
SET myquery(1) = "SELECT Name,Home_City,Home_State FROM Sample.Person"
SET myquery(2) = " WHERE JSON_OBJECT('name':Name,'city':Home_City,'state':Home_State) [ 'X'"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {DO rset.%Print(" ^ ")}
WRITE !,"Total row count=",rset.%ROWCOUNT

```

See Also

- [SELECT](#) statement
- [WHERE](#) clause
- [JSON_ARRAY](#) function
- [IS JSON](#) predicate condition
- [Overview of Predicates](#)
- “[Querying the Database](#)” chapter in *Using Caché SQL*

\$JUSTIFY

A function that right-aligns a value within a specified width, optionally rounding to a specified number of fractional digits.

```
$JUSTIFY(expression,width[,decimal])
```

Arguments

<i>expression</i>	The value that is to be right-aligned. It can be a numeric value, a string literal, or an expression that resolves to a numeric or string.
<i>width</i>	The number of characters within which <i>expression</i> is to be right-aligned. A positive integer or an expression that evaluates to a positive integer.
<i>decimal</i>	<i>Optional</i> — The number of fractional digits. A positive integer or an expression that evaluates to a positive integer. Caché rounds or pads the number of fractional digits in <i>expression</i> to this value. If you specify <i>decimal</i> , Caché treats <i>expression</i> as a numeric.

Description

\$JUSTIFY returns the value specified by *expression* right-aligned within the specified *width*. You can include the *decimal* argument to decimal-align numbers within *width*.

- `$JUSTIFY(expression,width)`: the 2-argument syntax right-justifies *expression* within *width*. It does not perform any conversion of *expression*. The *expression* can be a numeric or a nonnumeric string.
- `$JUSTIFY(expression,width,decimal)`: the 3-argument syntax converts *expression* to a [canonical number](#), rounds or zero pads fractional digits to *decimal*, then right-justifies the resulting numeric value within *width*. If *expression* is a nonnumeric string or NULL, Caché converts it to 0, pads it, then right-justifies it.

SQLCODE -380 is issued if you specify too few arguments. SQLCODE -381 is issued if you specify too many arguments.

\$JUSTIFY recognizes the DecimalSeparator character for the current locale. It adds or deletes a DecimalSeparator character as needed. The DecimalSeparator character depends upon the locale; commonly it is either a period (.) for American-format locales, or a comma (,) for European-format locales. To determine the DecimalSeparator character for your locale, invoke the following method:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator")
```

\$JUSTIFY, ROUND, and TRUNCATE

When rounding to a fixed number of fractional digits is important — for example, when representing monetary amounts — use **\$JUSTIFY**, which returns the specified number of trailing zeros following the rounding operation. When *decimal* is larger than the number of fractional digits in *expression*, **\$JUSTIFY** zero-pads. **\$JUSTIFY** also right-aligns the numbers, so that the DecimalSeparator characters align in a column of numbers.

ROUND also rounds to a specified number of fractional digits, but its return value is always normalized, removing trailing zeros. For example, `ROUND(10.004,2)` returns 10, not 10.00. Unlike **\$JUSTIFY**, **ROUND** allows you to specify either rounding (the default), or truncation.

TRUNCATE truncates to a specified number of fractional digits. Unlike **ROUND**, if the truncation results in trailing zeros, these trailing zeros are preserved. However, unlike **\$JUSTIFY**, **TRUNCATE** does not zero-pad.

ROUND and **TRUNCATE** allow you to round (or truncate) to the left of the decimal separator. For example, `ROUND(128.5,-1)` returns 130.

\$JUSTIFY and LPAD

The two-argument form of **LPAD** and the two-argument form of **\$JUSTIFY** both right-align a string by padding it with leading spaces. These two-argument forms differ in how they handle an output *width* that is shorter than the length of the input *expression*: **LPAD** truncates the input string to fit the specified output length. **\$JUSTIFY** expands the output length to fit the input string. This is shown in the following example:

```
SELECT '>' || LPAD(12345,10) || '<' AS lpadplus,
       '>' || $JUSTIFY(12345,10) || '<' AS justifyplus,
       '>' || LPAD(12345,3) || '<' AS lpaddingus,
       '>' || $JUSTIFY(12345,3) || '<' AS justifyminus
```

The three-argument form of **LPAD** allows you to left pad with characters other than spaces.

Arguments

expression

The value to be right-justified, and optionally expressed as a numeric with a specified number of fractional digits.

- If string justification is desired, do not specify *decimal*. The *expression* can contain any characters. **\$JUSTIFY** right-justifies *expression*, as described in *width*.
- If numeric justification is desired, specify *decimal*. If *decimal* is specified, Caché supplies *expression* to **\$JUSTIFY** as a **canonical number**. It resolves leading plus and minus signs and removes leading and trailing zeros. It truncates *expression* at the first nonnumeric character. If *expression* begins with a nonnumeric character (such as a currency symbol), Caché converts the *expression* value to 0. Canonical conversion does not recognize NumericGroupSeparator characters, currency symbols, multiple DecimalSeparator characters, or trailing plus or minus signs. For further details on how Caché converts a numeric to a canonical number, and Caché handling of a numeric string containing nonnumeric characters, refer to the [Numbers](#) section of the “Data Types and Values” chapter of *Using Caché ObjectScript*.

After **\$JUSTIFY** receives *expression* as a canonical number, **\$JUSTIFY** performs its operation and either rounds or zero-pads this canonical number to *decimal* number of fractional digits, then right-justifies the result, as described in *width*.

width

The *width* in which to right-justify the converted *expression*. If *width* is greater than the length of *expression* (after numeric and fractional digit conversion), Caché right-justifies to *width*, left-padding as needed with blank spaces. If *width* is less than the length of *expression* (after numeric and fractional digit conversion), Caché sets *width* to the length of the *expression* value.

Specify *width* as a positive integer. A *width* value of 0, the empty string ("), NULL, or a nonnumeric string is treated as a *width* of 0, which means that Caché sets *width* to the length of the *expression* value.

decimal

The number of fractional digits. If *expression* contains more fractional digits, **\$JUSTIFY** rounds the fractional portion to this number of fractional digits. If *expression* contains fewer fractional digits, **\$JUSTIFY** pads the fractional portion with zeros to this number of fractional digits, adding a Decimal Separator character, if needed. If *decimal*=0, **\$JUSTIFY** rounds *expression* to an integer value and deletes the Decimal Separator character.

If the *expression* value is less than 1, **\$JUSTIFY** inserts a leading zero before the DecimalSeparator character.

The **\$DOUBLE** values INF, -INF, and NAN are returned unchanged by **\$JUSTIFY**, regardless of the *decimal* value.

Examples

The following Dynamic SQL example performs right-justification on strings. No numeric conversion is performed:

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT TOP 20 Age,$JUSTIFY(Name,18),DOB FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example performs numeric right-justification with a specified number of fractional digits:

```
ZNSPACE "SAMPLES"
SET myquery = 2
SET myquery(1) = "SELECT TOP 20 $JUSTIFY(Salary,10,2) AS FullSalary,"
SET myquery(2) = "$JUSTIFY(Salary/7,10,2) AS SeventhSalary FROM Sample.Employee"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example performs numeric right-justification with a specified number of fractional digits, and string right-justification of the same numeric value:

```
SET myquery = 2
SET myquery(1) = "SELECT $JUSTIFY({fn ACOS(-1)},8,3) AS ArcCos3,"
SET myquery(2) = "$JUSTIFY({fn ACOS(-1)},8) AS ArcCosAll"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following Dynamic SQL example performs numeric right-justification with the **\$DOUBLE** values INF and NAN:

```
DO ##class(%SYSTEM.Process).%IEEEEError(0)
SET x=$DOUBLE(1.2e500)
SET y=x-x
SET myquery = 2
SET myquery(1) = "SELECT $JUSTIFY(?,12,2) AS INFtest,"
SET myquery(2) = "$JUSTIFY(?,12,2) AS NANtest"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(x,y)
DO rset.%Display()
```

See Also

- [LPAD](#) function
- [ROUND](#) function
- [TRUNCATE](#) function

LAST_DAY

A date function that returns the date of the last day of the month for a date expression.

```
LAST_DAY(date-expression)
```

Arguments

<i>date-expression</i>	An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

LAST_DAY returns the date of the last day of the specified month as an integer in \$HOROLOG format. Leap years differences are calculated, including century day adjustments: 2000 is a leap year, 1900 and 2100 are not leap years.

The *date-expression* can be a Caché date integer, a **\$HOROLOG** or **\$ZTIMESTAMP** value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of a %TimeStamp string is optional. The *date-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

LAST_DAY returns 0 (in Display mode 12/31/1840) when an invalid date is specified: the day or month as zero; the month greater than 12; or the day larger than the number of days in that month on that year.

This function can also be invoked from ObjectScript using the **LASTDAY()** method call:

```
WRITE $SYSTEM.SQL.LASTDAY("2004-02-25"),!  
WRITE $SYSTEM.SQL.LASTDAY(59590)
```

Examples

The following examples return the last day of the month as a Caché date integer. Whether this value is displayed as an integer or as a date string depends on the current SQL Display Mode setting.

The following two examples both return the number 59594 (which corresponds to '2004-02-29') because the last day of the month on the specified date is February 29 (2004 is a leap year):

```
SELECT LAST_DAY('2004-02-25')  
  
SELECT LAST_DAY(59590)
```

The following examples all return the Caché date integer corresponding to the last day of the current month:

```
SELECT LAST_DAY({fn NOW()}) AS LD_Now,  
       LAST_DAY(CURRENT_DATE) AS LD_CurrDate,  
       LAST_DAY(CURRENT_TIMESTAMP) AS LD_CurrTstamp,  
       LAST_DAY($ZTIMESTAMP) AS LD_ZTstamp,  
       LAST_DAY($HOROLOG) AS LD_Horolog
```

See Also

- SQL functions: [DATENAME](#), [DATEPART](#), [DAY](#), [DAYOFYEAR](#), [MONTH](#), [YEAR](#), [TO_DATE](#)
- ObjectScript function: [\\$ZDATE](#)
- ObjectScript special variables: [\\$HOROLOG](#) [\\$ZTIMESTAMP](#)

LAST_IDENTITY

A scalar function that returns the identity of the last row inserted, updated, deleted, or fetched.

```
LAST_IDENTITY ( )
```

Description

The **LAST_IDENTITY** function returns the **%ROWID** local variable value. The **%ROWID** local variable is set to a value in **Embedded SQL** or ODBC. The **%ROWID** local variable is not set to a value by Dynamic SQL, the SQL Shell, or the Management Portal SQL interface. Dynamic SQL instead sets a **%ROWID** object property.

The **LAST_IDENTITY** function takes no arguments. Note that the argument parentheses are required.

LAST_IDENTITY returns the IDENTITY field value of the last row affected by the current process. If the table has no IDENTITY field, it returns the row ID (**%ROWID**) of the last row affected by the current process. The returned value is data type INTEGER.

- For an Embedded SQL **INSERT**, **UPDATE**, **DELETE** or **TRUNCATE TABLE** statement, **LAST_IDENTITY** returns the IDENTITY or **%ROWID** value of the last row modified.
- For an Embedded SQL **cursor-based SELECT** statement, **LAST_IDENTITY** returns the IDENTITY or **%ROWID** value of the last row retrieved. However, if the cursor-based **SELECT** statement includes a **DISTINCT** keyword or a **GROUP BY** clause, **LAST_IDENTITY** is not changed; it returns its prior value (if any).
- For an Embedded SQL single-row (non-cursor) **SELECT** statement, **LAST_IDENTITY** is not changed. The prior value (if any) is returned.

At process initiation, **LAST_IDENTITY** returns NULL. Following a **NEW %RowID**, **LAST_IDENTITY** returns NULL.

If no rows were affected by an operation, **LAST_IDENTITY** is not changed; **LAST_IDENTITY** returns its prior value (if any). Following a **NEW %RowID**, invoking **LAST_IDENTITY** returns NULL, but invoking **%ROWID** generates an <UNDEFINED> error.

For further details on IDENTITY fields, see **CREATE TABLE**. For further details on **%ROWID**, see the “Embedded SQL” chapter of *Using Caché SQL*.

Examples

The following example uses two Embedded SQL programs to return **LAST_IDENTITY**. The first example creates a new table **Sample.Students**. The second example populates this table with data, then performs a cursor-based **SELECT** on the data, returning **LAST_IDENTITY** for each operation.

Please run the two Embedded SQL programs in the order shown. (It is necessary to use two embedded SQL programs here because embedded SQL cannot compile an **INSERT** statement unless the referenced table already exists.)

```
WRITE !,"Creating table"
&sql(CREATE TABLE Sample.Students (
  StudentName VARCHAR(30),
  StudentAge INTEGER,
  StudentID IDENTITY))
IF SQLCODE=0 {
  WRITE !,"Created table, SQLCODE=",SQLCODE }
ELSEIF SQLCODE=-201 {
  WRITE !,"Table already exists, SQLCODE=",SQLCODE }

WRITE !,"Populating table"
NEW %ROWCOUNT,%ROWID
&sql(INSERT INTO Sample.Students (StudentName,StudentAge)
  SELECT Name,Age FROM Sample.Person WHERE Age <= '21')
IF SQLCODE=0 {
  WRITE !,%ROWCOUNT," records added, last RowID is ",%ROWID,! }
ELSE {
  WRITE !,"Insert failed, SQLCODE=",SQLCODE }
```

```
&sql(SELECT LAST_IDENTITY()  
      INTO :insertID  
      FROM Sample.Students)  
      WRITE !,"INSERT Last Identity is: ",insertID,!  
/* Cursor-based SELECT Query */  
      &sql(DECLARE C1 CURSOR FOR  
          SELECT StudentName INTO :name FROM Sample.Students  
          WHERE StudentAge = '17')  
&sql(OPEN C1)  
      QUIT:(SQLCODE'=0)  
&sql(FETCH C1)  
      WHILE (SQLCODE = 0) {  
          WRITE name," is seventeen",!  
          &sql(FETCH C1) }  
&sql(CLOSE C1)  
      WRITE !,%ROWCOUNT," records queried, last RowID is ",%ROWID,!  
&sql(SELECT LAST_IDENTITY()  
      INTO :qId)  
      WRITE !,"SELECT Last Identity is: ",qId,!  
&sql(DROP TABLE Sample.Students)
```

See Also

- [INSERT, UPDATE, DELETE, TRUNCATE TABLE](#)
- [DECLARE, OPEN, FETCH, CLOSE](#)
- [Embedded SQL in *Using Caché SQL*](#)

LCASE

A case-transformation function that converts all uppercase letters in a string to lowercase letters.

```
LCASE(string-expression)
{fn LCASE(string-expression)}
```

Arguments

<i>string-expression</i>	The string expression whose characters are to be converted to lowercase. The expression can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
--------------------------	---

Description

LCASE converts uppercase letters to lowercase for display purposes. It has no effects on non-alphabetic characters. It leaves unchanged punctuation and leading and trailing blank spaces.

LCASE does not force numerics to be interpreted as a string. Caché SQL converts numerics to canonical form, removing leading and trailing zeros. Caché SQL does not convert numeric strings to canonical form.

The **LOWER** function can also be used convert uppercase letters to lowercase.

LCASE does not affect **collation**. The **%SQLUPPER** function is the preferred way in SQL to convert a data value for not case-sensitive collation. Refer to **%SQLUPPER** for further information on case transformation for collation.

Examples

The following example returns each person's name in lowercase letters:

```
SELECT TOP 10 Name, {fn LCASE(Name)} AS LowName
FROM Sample.Person
```

LCASE also works on Unicode (non-ASCII) alphabetic characters, as shown in the following Embedded SQL example, which converts Greek letters from uppercase to lowercase:

```
IF $SYSTEM.Version.IsUnicode() {
  SET a=$CHAR(920,913,923,913,931,931,913)
  &sql(SELECT LCASE(:a) INTO :b )
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
  ELSE {WRITE !,a,!,b }
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- SQL functions: [LOWER UCASE](#)
- ObjectScript function: [\\$ZCONVERT](#)

LEAST

A function that returns the least value from a list of values.

```
LEAST(expression,expression[,...])
```

Arguments

<i>expression</i>	An expression that resolves to a number or a string. The values of these expressions are compared to each other and the least value returned. An <i>expression</i> can be a field name, a literal, an arithmetic expression, a host variable, or an object reference. You can list up to 140 comma-separated expressions.
-------------------	---

Description

LEAST returns the smallest (least) value from a comma-separated list of values. Expressions are evaluated in left-to-right order. If only one *expression* is provided, **LEAST** returns that value. If any *expression* is NULL, **LEAST** returns NULL.

If all of the *expression* values resolve to canonical numbers, they are compared in numeric order. If a quoted string contains a number in canonical format, it is compared in numeric order. However, if a quoted string contains a number not in canonical format (for example, '00', '0.4', or '+4'), it is compared as a string. String comparisons are performed character-by-character in collation order. Any string value is greater than any numeric value.

The empty string is greater than any numeric value, but less than any other string value.

If the returned value is a number, **LEAST** returns it in canonical format (leading and trailing zeros removed, etc.). If the returned value is a string, **LEAST** returns it unchanged, including any leading or trailing blanks.

The inverse function of **LEAST** is **GREATEST**.

Data Type of Returned Value

If the data types of the *expression* values are different, the data type returned is the type most compatible with all of the possible return values, the data type with the highest [data type precedence](#). For example, if one *expression* is an integer and another *expression* is a fractional number, **LEAST** returns a value with data type NUMERIC. This is because NUMERIC is the data type with the highest precedence that is compatible with both.

Examples

In the following example, each **LEAST** compares three canonical numbers:

```
SELECT LEAST(22,2.2,-21) AS HighNum,
       LEAST('2.2','22','-21') AS HighNumStr
```

In the following example, each **LEAST** compare three numeric strings. However, each **LEAST** contains one string that is non-canonical; these non-canonical values are compared as character strings. A character string is always greater than a number:

```
SELECT LEAST('22','+2.2','21'),
       LEAST('0.2','22','21')
```

In the following example, each **LEAST** compare three strings and returns the value with the lowest collation sequence:

```
SELECT LEAST('A','a',''),
       LEAST('a','aa','abc'),
       LEAST('#','0','7'),
       LEAST('##','00','77')
```

The following example compares two dates, treated as canonical numbers: the date of birth as a `SHOROLOG` integer, and the integer 58074 converted to a date. It returns the date of birth for each person born in the 20th century. Anyone born after December 31, 1999 is displayed with the default birth date of January 1, 2000:

```
SELECT Name,LEAST(DOB,TO_DATE(58074)) AS NewMillenium
FROM Sample.Person
```

See Also

- SQL functions: [GREATEST CONVERT TO_NUMBER](#)

LEFT

A scalar string function that returns a specified number of characters from the beginning (leftmost position) of a string expression.

```
{fn LEFT(string-expression,count)}
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
<i>count</i>	An integer that specifies the number of characters to return from the starting position of <i>string-expression</i> .

Description

LEFT returns the specified number of characters from the beginning of a string. **LEFT** does not pad strings; if you specify a larger number of characters than are in the string, **LEFT** returns the string. **LEFT** returns NULL if passed a NULL value for either argument.

LEFT can only be used as an ODBC scalar function (with the curly brace syntax).

Examples

The following example returns the seven leftmost characters from each name in the Sample.Person table:

```
SELECT Name, {fn LEFT(Name,7)} AS ShortName
FROM Sample.Person
```

The following embedded SQL example shows how **LEFT** handles a *count* that is longer than the string itself:

```
&sql(SELECT Name, {fn LEFT(Name,40)}
INTO :a,:b
FROM Sample.Person)
IF SQLCODE'=0 {
WRITE !,"Error code ",SQLCODE }
ELSE {
WRITE !,a,"=original",!,b,"=LEFT 40" }
```

No padding is performed.

See Also

[LTRIM](#) [RIGHT](#) [RTRIM](#)

LEN

A string function that returns the number of characters in a string expression.

```
LEN(string-expression)
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
--------------------------	--

Description

Note: The **LEN** function is an alias for the **LENGTH** function. **LEN** is provided for TSQL compatibility. Refer to [LENGTH](#) for further details.

See Also

- [LENGTH](#)

LENGTH

A string function that returns the number of characters in a string expression.

```
LENGTH(string-expression)
{fn LENGTH(string-expression)}
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
--------------------------	--

Description

LENGTH returns an integer that denotes the number of characters, not the number of bytes, of the given string expression. The *string-expression* can be a string (from which trailing blanks are removed), or a number (which Caché converts to canonical form).

Note that **LENGTH** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

LENGTH and the other length functions (**\$LENGTH**, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH**) all perform the following operations:

- **LENGTH** returns the length of the Logical (internal data storage) value of a field, not the display value, regardless of the SelectMode setting. All SQL functions always use the internal storage value of a field.
- **LENGTH** returns the length of the [canonical form](#) of a number. A number in canonical form excludes leading and trailing zeros, leading signs (except a single minus sign), and a trailing decimal separator character. **LENGTH** returns the string length of a numeric string. A numeric string is not converted to canonical form.
- **LENGTH** does not exclude leading blanks from strings. You can remove leading blanks from a string using the [LTRIM](#) function.
- **LENGTH** returns a value of [data type](#) INTEGER (%Library.Integer).

LENGTH differs from the other length functions (**\$LENGTH**, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH**) when performing the following operations:

- **LENGTH** excludes trailing blanks and the string-termination character.
\$LENGTH, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** do not exclude trailing blanks and terminators.
- **LENGTH** returns NULL if passed a NULL value, and 0 if passed an empty string.
CHARACTER_LENGTH, **CHAR_LENGTH**, and **DATALENGTH** also return NULL if passed a NULL value, and 0 if passed an empty string. **\$LENGTH** returns 0 if passed a NULL value, and 0 if passed an empty string.
- **LENGTH** does not support data stream fields. Specifying a stream field for *string-expression* results in an SQLCODE -37.
\$LENGTH also does not support stream fields. **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** functions do support data stream fields.

Examples

In the following example, Caché first converts each number to canonical form (removing leading and trailing zeros, resolving leading signs, and removing a trailing decimal separator character). Each **LENGTH** returns a length of 1:

```
SELECT {fn LENGTH(7.00)} AS CharCount,
       {fn LENGTH(+007)} AS CharCount,
       {fn LENGTH(007.)} AS CharCount,
       {fn LENGTH(00000.00)} AS CharCount,
       {fn LENGTH(-0)} AS CharCount
```

In the following example, the first **LENGTH** removes the leading zero, returning a length value of 2; the second **LENGTH** treats the numeric value as a string, and does not remove the leading zero, returning a length value of 3:

```
SELECT LENGTH(0.7) AS CharCount,
       LENGTH('0.7') AS CharCount
```

The following example returns the value 12:

```
SELECT LENGTH('INTERSYSTEMS') AS CharCount
```

The following example shows how **LENGTH** handles leading and trailing blanks. The first **LENGTH** returns 15, because **LENGTH** excludes trailing blanks, but not leading blanks. The second **LENGTH** returns 12, because **LTRIM** excludes the leading blanks:

```
SELECT LENGTH(' INTERSYSTEMS ') AS CharCount,
       LENGTH(LTRIM(' INTERSYSTEMS ')) AS CharCount
```

The following example returns the number of characters in each Name value in the Sample.Person table:

```
SELECT Name, {fn LENGTH(Name)} AS CharCount
FROM Sample.Person
ORDER BY CharCount
```

The following example returns the number of characters in the DOB (date of birth) field. Note that the length returned (by **LENGTH**, **CHAR_LENGTH**, and **CHARACTER_LENGTH**) is the internal (\$HOROLOG) format of the date, not the display format. The display length of DOB is ten characters; all three length functions return the internal length of 5:

```
SELECT DOB, {fn LENGTH(DOB)} AS LenCount,
       CHAR_LENGTH(DOB) AS CCount,
       CHARACTER_LENGTH(DOB) AS CtrCount
FROM Sample.Person
```

The following Embedded SQL example gives the length of a string of Unicode characters. The length returned is the number of characters (7), not the number of bytes.

```
IF $SYSTEM.Version.IsUnicode() {
  SET a=$CHAR(920,913,923,913,931,931,913)
  &sql(SELECT LENGTH(:a) INTO :b )
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
  ELSE {
    WRITE !,"The Greek Sea: ",a,!, $LENGTH(a),!,b }
  }
} ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

(Note that the above example requires a Unicode installation of Caché.)

See Also

- SQL functions: [CHAR_LENGTH](#), [CHARACTER_LENGTH](#), [DATALENGTH](#), [LEN](#), [\\$LENGTH](#)
- ObjectScript function: [\\$LENGTH](#)

\$LENGTH

A string function that returns the number of characters or the number of delimited substrings in a string.

```
$LENGTH(expression[,delimiter])
```

Arguments

<i>expression</i>	The target string. It can be a numeric value, a string literal, the name of any variable, or any valid expression.
<i>delimiter</i>	<i>Optional</i> — A string that demarcates separate substrings in the target string. It must be a string literal, but can be of any length. The enclosing quotation marks are required.

Description

\$LENGTH returns the number of characters in a specified string or the number of substrings in a specified string, depending on the arguments used.

- **\$LENGTH**(*expression*) returns the number of characters in the string. If the expression is an empty string ("), **\$LENGTH** returns 0. If the expression is NULL, **\$LENGTH** returns 0.
- **\$LENGTH**(*expression,delimiter*) returns the number of substrings within the string. **\$LENGTH** returns the number of substrings separated from one another by the indicated *delimiter*. This number is always equal to the number of delimiter instances found in the *expression* string, plus one.

This function returns data of type SMALLINT.

\$LENGTH(*expression*) and other Length Functions

\$LENGTH(*expression*) and the other length functions (**LENGTH**, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH**) all perform the following operations:

- **\$LENGTH** returns the length of the Logical (internal data storage) value of a field, not the display value, regardless of the SelectMode setting. All SQL functions always use the internal storage value of a field.
- **\$LENGTH** returns the length of the [canonical form](#) of a number. A number in canonical form excludes leading and trailing zeros, leading signs (except a single minus sign), and a trailing decimal separator character. **\$LENGTH** returns the string length of a numeric string. A numeric string is not converted to canonical form.
- **\$LENGTH** does not exclude leading blanks from strings. You can remove leading blanks from a string using the [LTRIM](#) function.

\$LENGTH differs from the other length functions (**LENGTH**, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH**) when performing the following operations:

- **\$LENGTH** does not exclude trailing blanks and terminators.
CHARACTER_LENGTH, **CHAR_LENGTH**, and **DATALENGTH** also do not exclude trailing blanks and terminators. **LENGTH** excludes trailing blanks and the string-termination character.
- **\$LENGTH** returns 0 if passed a NULL value, and 0 if passed an empty string.
LENGTH, **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALENGTH** return NULL if passed a NULL value, and 0 if passed an empty string.
- **\$LENGTH** does not support data stream fields. Specifying a stream field for *string-expression* results in an SQLCODE -37.

LENGTH also does not support stream fields. **CHARACTER_LENGTH**, **CHAR_LENGTH**, and **DATALength** functions do support data stream fields.

NULL and Empty String Arguments

\$LENGTH(*expression*) does not distinguish between the empty string (") and NULL (the absence of a value). It returns a length of 0 for both an empty string (") value and for NULL.

\$LENGTH(*expression,delimiter*) with a non-null delimiter returns a delimited substring count of 1 if no match occurred. The full string is a single substring containing no delimiters. This is true even when *expression* is the empty string ("), or *expression* is NULL. However, an empty string does match itself, returning a value of 2.

The following table shows the possible combinations of a string ('abc'), empty string ("), or NULL *expression* value paired with a non-matching string ('^'), empty string ("), or NULL *delimiter* value:

\$LENGTH (NULL) = 0	\$LENGTH (") = 0	\$LENGTH ('abc') = 3
\$LENGTH (NULL,NULL) = 0	\$LENGTH (" ,NULL) = 0	\$LENGTH ('abc',NULL) = 0
\$LENGTH (NULL,") = 1	\$LENGTH (" ,") = 2	\$LENGTH ('abc',") = 1
\$LENGTH (NULL,'^') = 1	\$LENGTH (" ,'^') = 1	\$LENGTH ('abc','^') = 1

Examples

The following example returns 6, the length of the string:

```
SELECT $LENGTH('ABCDEG') AS StringLength
```

The following example returns 3, the number of substrings within the string, as delimited by the dollar sign (\$) character.

```
SELECT $LENGTH('ABC$DEF$EFG', '$') AS SubStrings
```

If the specified delimiter is not found in the string **\$LENGTH** returns 1, because the only substring is the string itself:

```
SELECT $LENGTH('ABCDEG', '$') AS SubStrings
```

In the following embedded SQL example, the first **\$LENGTH** function returns 11, the number of characters in *a* (including, of course, the space character). The second **\$LENGTH** function returns 2, the number of substrings in *a* using *b*, the space character, as the substring delimiter.

```
SET a="HELLO WORLD"
SET b=" "
&sql(SELECT
$LENGTH(:a),
$LENGTH(:a,:b)
INTO :a1,:a2 )
IF SQLCODE'=0 {
WRITE !,"Error code ",SQLCODE }
ELSE {
WRITE !,"The input string: ",a
WRITE !,"Number of characters: ",a1
WRITE !,"Number of substrings: ",a2 }
```

The following example returns 0 because the string tested is the null string:

```
SELECT $LENGTH(NULL) AS StringLength
```

The following example returns 1 because a delimiter is specified and not found. There is one substring, which is the null string:

```
SELECT $LENGTH(NULL, '$') AS SubStrings
```

The following example returns 0 because the delimiter is the null string:

```
SELECT $LENGTH('ABCDEFG',NULL) AS SubStrings
```

Notes

\$LENGTH, \$PIECE, and \$LIST

- **\$LENGTH** with one argument returns the number of characters in a string. This function can be used with the **\$EXTRACT** function, which locates a substring by position and returns the substring value.
- **\$LENGTH** with two arguments returns the number of substrings in a string, based on a delimiter. This function can be used with the **\$PIECE** function, which locates a substring by a delimiter and returns the substring value.
- **\$LENGTH** should not be used on encoded lists created using **\$LISTBUILD** or **\$LIST**. Use **\$LISTLENGTH** to determine the number of substrings (list elements) in an encoded list string.

The **\$LENGTH**, **\$FIND**, **\$EXTRACT**, and **\$PIECE** functions operate on standard character strings. The various **\$LIST** functions operate on encoded character strings, which are incompatible with standard character strings. The only exceptions are the **\$LISTGET** function and the one-argument and two-argument forms of **\$LIST**, which take an encoded character string as input, but output a single element value as a standard character string.

See Also

- SQL functions: [CHAR_LENGTH](#), [CHARACTER_LENGTH](#), [DATALENGTH](#), [\\$EXTRACT](#), [\\$FIND](#), [LENGTH](#), [\\$LIST](#), [\\$LISTGET](#), [\\$PIECE](#)
- ObjectScript functions: [\\$EXTRACT](#), [\\$FIND](#), [\\$LENGTH](#), [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTGET](#), [\\$PIECE](#)

\$LIST

A list function that returns elements in a list.

```
$LIST(list[,position[,end]])
```

Arguments

<i>list</i>	An expression that evaluates to a valid list. A <i>list</i> is an encoded character string containing one or more elements. You can create a <i>list</i> using the SQL or ObjectScript \$LISTBUILD or \$LISTFROMSTRING functions. You can extract a <i>list</i> from an existing list using the SQL or ObjectScript \$LIST function.
<i>position</i>	<i>Optional</i> — The starting position in the specified list. An expression that evaluates to an integer.
<i>end</i>	<i>Optional</i> — The ending position in the specified list. An expression that evaluates to an integer.

Description

\$LIST returns elements from a list. The elements returned depend on the arguments used.

- **\$LIST(list)** returns the first element in the list as a text string.
- **\$LIST(list,position)** returns the element indicated by the specified position as a text string. The *position* argument must evaluate to an integer.
- **\$LIST(list,position,end)** returns a “sublist” (an encoded list string) containing the elements of the list from the specified start *position* through the specified *end* position.

This function returns data of type VARCHAR.

Arguments

list

An encoded character string containing one or more elements. You can create a list using the SQL **\$LISTBUILD** function or the ObjectScript **\$LISTBUILD** function. You can convert a delimited string into a list using the SQL **\$LISTFROMSTRING** function or the ObjectScript **\$LISTFROMSTRING** function. You can extract a list from an existing list using the SQL **\$LIST** function or the ObjectScript **\$LIST** function.

A list can be supplied to the SQL **\$LIST** function by using a host variable, or by specifying a **\$LISTBUILD** within SQL. Both are shown in the following Embedded SQL example:

```
SET mylist=$LISTBUILD("Red","Blue","Green")
&sql (SELECT $LIST(:mylist,2),$LIST($LISTBUILD('Red','Blue','Green'),3)
INTO :a,:b )
IF SQLCODE'=0 {
  WRITE "Error code ",SQLCODE,! }
ELSE {
  WRITE !,"The host variable list element is ",a,!
  WRITE !,"The SQL $LISTBUILD list element is ",b,! }
```

A list can be extracted from another list by using the **\$LIST** function:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,2,3)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
&sql(SELECT $LIST(:b,1)
INTO :c )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The element returned is ",c }
}

```

In the following Embedded SQL example, *subList* is not a valid *list* argument, because it is a single element returned as an ordinary string, not an encoded list string. Only the three-argument form of **\$LIST** returns an encoded list string. In this case, an SQLCODE -400 fatal error is generated:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,2)
INTO :sublist )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
&sql(SELECT $LIST(:sublist,1)
INTO :c )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The sublist is"
  ZZDUMP c ; Variable not set
}
}

```

position

The position of a list element to return. List elements are counted from 1. If *position* is omitted, the first element is returned. If the value of *position* is 0 or greater than the number of elements in the list, Caché SQL does not return a value. If the value of *position* is negative one (-1), **\$LIST** returns the final element in the list.

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,-1)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The last element is ",b }

```

If the *end* argument is specified, *position* specifies the first element in a range of elements. Even when only one element is returned (when *position* and *end* are the same number) this element is returned as an encoded list string. Thus, **\$LIST(x, 2)** (which returns the element as an ordinary string) is not identical to **\$LIST(x, 2, 2)** (which returns the element as an encoded list string).

end

The position of the last element in a range of elements. You must specify *position* to specify *end*. When *end* is specified, the value returned is an encoded list string. Because of this encoding, such strings should only be processed by other **\$LIST** functions.

If the value of *end* is:

- greater than *position*, an encoded string containing a list of elements is returned.
- equal to *position*, an encoded string containing the one element is returned.
- less than *position*, no value is returned.
- greater than the number of elements in *list*, it is equivalent to specifying the final element in the list.
- negative one (-1), it is equivalent to specifying the final element in the list.

When specifying *end*, you can specify a *position* value of zero (0). In this case, 0 is equivalent to 1.

Examples

In the following Embedded SQL example, the two **WRITE** statements both return “Red”, the first element in the list. The first writes the first element by default, the second writes the first element because the *position* argument is set to 1:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a),$LIST(:a,1)
INTO :b,:c )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The one-arg sublist is ",b
  WRITE !,"The two-arg sublist is ",c }
```

The following Embedded SQL example returns “Blue”, the second element in the list:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,2)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The second element is ",b }
```

The following Embedded SQL example returns “Red Blue”, a two-element list string beginning with the first element and ending with the second element in the list. **ZZDUMP** is used rather than **WRITE**, because a list string contains special (non-printing) encoding characters:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,1,2)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The encoded sublist is"
  ZZDUMP b ; Prints "Red Blue "
}
```

The following Embedded SQL example returns the last element in a list of unknown length. Here, the last element is returned first as an ordinary string, then as an encoded list string:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTLENGTH(:a),$LIST(:a,-1)
INTO :b,:plain )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  &sql(SELECT $LIST(:a,:b,-1)
INTO :encoded )
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
  ELSE {
    WRITE !,"The final element as a string: ",plain
    WRITE !,"The final element as an encoded string: "
    ZZDUMP encoded }
}
```

Notes

Invalid Argument Values

If the expression in the *list* argument does not evaluate to a valid list, an SQLCODE -400 fatal error is generated:

```

SET a="the quick brown fox"
&sql(SELECT $LIST(:a,1)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The sublist is"
  ZZDUMP b ; Variable not set
}

```

If the value of the *position* argument or the *end* argument is less than -1, an SQLCODE -400 fatal error is generated:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,-2,3)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The sublist is"
  ZZDUMP b ; Variable not set
}

```

If the value of the *position* argument refers to a nonexistent list member and no *end* argument is used, an SQLCODE -400 fatal error is generated:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,7)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The sublist is"
  ZZDUMP b ; Variable not set
}

```

However, if an *end* argument is used, no error occurs, and the null string is returned.

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LIST(:a,7,-1)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Error code ",SQLCODE
  WRITE !,"The sublist is"
  ZZDUMP b ; Prints a null string
}

```

If the value of the *position* argument identifies an element with an undefined value, an SQLCODE -400 fatal error is generated:

```

SET a=$LISTBUILD("Red",,"Green")
&sql(SELECT $LIST(:a,2)
INTO :b )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The sublist is"
  ZZDUMP b ; Variable not set
}

```

Two-Argument and Three-Argument \$LIST

\$LIST(list,1) is not equivalent to **\$LIST(list,1,1)** because the former returns a string, while the latter returns a single-element list string. If there are no elements to return, the two-argument form does not return a value; the three-argument form returns a null string.

Unicode

If one Unicode character appears in a list element, that entire list element is represented as Unicode (wide) characters. Other elements in the list are not affected.

The following Embedded SQL example shows two lists. The *a* list consists of two elements which contain only ASCII characters. The *b* list consists of two elements: the first element contains a Unicode character (**\$CHAR(960)** = the pi symbol); the second element contains only ASCII characters.

```

IF $SYSTEM.Version.IsUnicode() {
  SET a=$LISTBUILD("ABC"_$CHAR(68),"XYZ")
  SET b=$LISTBUILD("ABC"_$CHAR(960),"XYZ")
  &sql(SELECT $LIST(:a,1),$LIST(:a,2),$LIST(:b,1),$LIST(:b,2)
  INTO :a1,:a2,:b1,:b2 )
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
  ELSE {
    WRITE !,"The ASCII list a elements: "
    ZZDUMP a1
    ZZDUMP a2
    WRITE !,"The Unicode list b elements: "
    ZZDUMP b1
    ZZDUMP b2 }
  }
ELSE {WRITE "This example requires a Unicode installation of Caché"}

```

Note that Caché encodes the first element of *b* entirely in wide Unicode characters. The second element of *b* contains no Unicode characters, and thus Caché encodes it using narrow ASCII characters.

See Also

- SQL functions: [\\$LISTBUILD](#), [\\$LISTDATA](#), [\\$LISTFIND](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTLENGTH](#), [\\$LISTSAME](#), [\\$LISTTOSTRING](#), [\\$PIECE](#)
- ObjectScript functions: [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTDATA](#), [\\$LISTFIND](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTLENGTH](#), [\\$LISTNEXT](#), [\\$LISTSAME](#), [\\$LISTTOSTRING](#), [\\$LISTVALID](#)

\$LISTBUILD

A list function that builds a list from strings.

```
$LISTBUILD(element, ...)
```

Argument

<i>element</i>	Any expression, or comma-separated list of expressions.
----------------	---

Description

\$LISTBUILD takes one or more expressions and returns a list with one element for each expression.

The following functions can be used to create a list:

- **\$LISTBUILD**, which creates a list from multiple strings, one string per element.
- **\$LISTFROMSTRING**, which creates a list from a single string containing multiple delimited elements.
- **\$LIST**, which extracts a sublist from an existing list.

\$LISTBUILD is used with the other Caché SQL list functions: **\$LIST**, **\$LISTDATA**, **\$LISTFIND**, **\$LISTFROMSTRING**, **\$LISTGET**, **\$LISTLENGTH**, and **\$LISTTOSTRING**.

Note: **\$LISTBUILD** and the other **\$LIST** functions use an optimized binary representation to store data elements. For this reason, equivalency tests may not work as expected with some **\$LIST** data. Data that might, in other contexts, be considered equivalent, may have a different internal representation. For example, `$LISTBUILD(1)` is not equal to `$LISTBUILD('1')`.

For the same reason, a list string value returned by **\$LISTBUILD** should not be used in character search and parse functions that use a delimiter character, such as **\$PIECE** and the two-argument form of **\$LENGTH**. Elements in a list created by **\$LISTBUILD** are not marked by a character delimiter, and thus can contain any character.

Examples

The following Embedded SQL example takes three strings and produces a three-element list:

```
SET x="Red"
SET y="White"
SET z="Blue"
&sql(SELECT $LISTBUILD(:x,:y,:z)
      INTO :listout)
IF SQLCODE=0 {WRITE listout," length ",$LISTLENGTH(listout)}
ELSE {WRITE "Error code:",SQLCODE}
```

Notes

Omitting Arguments

Omitting an element expression yields an element whose value is NULL. For example, the following Embedded SQL contains two **\$LISTBUILD** statements that both produce a three-element list whose second element has an undefined (NULL) value:

```

SET x="Red"
SET y="White"
SET z="Blue"
&sql(SELECT $LISTBUILD(:x,,:z),
      $LISTBUILD(:x,',':z)
      INTO :list1,list2)
IF SQLCODE=0 {WRITE list1," length ",$LISTLENGTH(list1),!
              WRITE list2," length ",$LISTLENGTH(list2)}
ELSE {WRITE "Error code:",SQLCODE}

```

Additionally, if a **\$LISTBUILD** expression is undefined, the corresponding list element has an undefined value. The following Embedded SQL example produces a two-element list whose first element is "Red" and whose second element has an undefined value:

```

&sql(SELECT $LISTBUILD('Red',:z)
      INTO :list1)
IF SQLCODE=0 {WRITE list1," length ",$LISTLENGTH(list1)}
ELSE {WRITE "Error code:",SQLCODE}

```

The following Embedded SQL example produces a two-element list. The trailing comma indicates the second element has an undefined value:

```

&sql(SELECT $LISTBUILD('Red',)
      INTO :list1)
IF SQLCODE=0 {WRITE list1," length ",$LISTLENGTH(list1)}
ELSE {WRITE "Error code:",SQLCODE}

```

Providing No Arguments

Invoking the **\$LISTBUILD** function with no arguments returns a list with one element whose data value is undefined. This is not the same as NULL. The following are valid **\$LISTBUILD** statements that create “empty” lists:

```

&sql(SELECT $LISTBUILD(),
      $LISTBUILD(NULL)
      INTO :list1,:list2)
IF SQLCODE=0 {
  ZSDUMP list1
  WRITE !,"length ",$LISTLENGTH(list1),!
  ZSDUMP list2
  WRITE !,"length ",$LISTLENGTH(list2),!
}
ELSE {WRITE "Error code:",SQLCODE}

```

The following are valid **\$LISTBUILD** statements that create a list element that contains an empty string:

```

&sql(SELECT $LISTBUILD(''),
      $LISTBUILD(CHAR(0))
      INTO :list1,:list2)
IF SQLCODE=0 {
  ZSDUMP list1
  WRITE !,"length ",$LISTLENGTH(list1),!
  ZSDUMP list2
  WRITE !,"length ",$LISTLENGTH(list2),!
}
ELSE {WRITE "Error code:",SQLCODE}

```

Nesting Lists

An element of a list may itself be a list. For example, the following statement produces a three-element list whose third element is the two-element list, "Walnut,Pecan":

```

SELECT $LISTBUILD('Apple','Pear',$LISTBUILD('Walnut','Pecan'))

```

Concatenating Lists

The result of concatenating two lists with the SQL Concatenate operator (||) is another list. For example, the following **SELECT** items produce the same list, "A,B,C":

```

SELECT $LISTBUILD('A','B','C') AS List,
       $LISTBUILD('A','B')||$LISTBUILD('C') AS CatList

```

In the following example, the first two select items result in the same two-element list; the third select item results in NULL (because concatenating NULL to anything results in NULL); the fourth and fifth select items result in the same three-element list:

```
SELECT
  $LISTBUILD('A','B') AS List,
  $LISTBUILD('A','B') || '' AS CatEStr,
  $LISTBUILD('A','B') || NULL AS CatNull,
  $LISTBUILD('A','B') || $LISTBUILD('') AS CatEList,
  $LISTBUILD('A','B') || $LISTBUILD(NULL) AS CatNList
```

Unicode

If one or more characters in a list element is a wide (Unicode) character, all characters in that element are represented as wide characters. To ensure compatibility across systems, **\$LISTBUILD** always stores these bytes in the same order, regardless of the hardware platform. Wide characters are represented as byte strings. For further details, refer to the ObjectScript **\$LISTBUILD** function in the *Caché ObjectScript Reference*.

See Also

- SQL functions: **\$LIST**, **\$LISTDATA**, **\$LISTFIND**, **\$LISTFROMSTRING**, **\$LISTGET**, **\$LISTLENGTH**, **\$LISTSAME**, **\$LISTTOSTRING**, **\$PIECE**
- ObjectScript functions: **\$LIST**, **\$LISTBUILD**, **\$LISTDATA**, **\$LISTFIND**, **\$LISTFROMSTRING**, **\$LISTGET**, **\$LISTLENGTH**, **\$LISTNEXT**, **\$LISTSAME**, **\$LISTTOSTRING**, **\$LISTVALID**

\$LISTDATA

A list function that indicates whether the specified element exists and has a data value.

```
$LISTDATA(list[,position])
```

Arguments

<i>list</i>	An expression that evaluates to a valid list. A <i>list</i> is an encoded character string containing one or more elements. You can create a <i>list</i> using the SQL or ObjectScript \$LISTBUILD or \$LISTFROMSTRING functions. You can extract a <i>list</i> from an existing list using the SQL or ObjectScript \$LIST function.
<i>position</i>	<i>Optional</i> — An integer expression specifying an element in <i>list</i> .

Description

\$LISTDATA checks for data in the requested element in a list. **\$LISTDATA** returns a value of 1 if the element indicated by the *position* argument is in the *list* and has a data value. **\$LISTDATA** returns a value of a 0 if the element is not in the *list* or does not have a data value.

This function returns data of type SMALLINT.

Arguments

list

An encoded character string containing one or more elements. You can create a list using the SQL [\\$LISTBUILD](#) function or the ObjectScript [\\$LISTBUILD](#) function. You can convert a delimited string into a list using the SQL [\\$LISTFROMSTRING](#) function or the ObjectScript [\\$LISTFROMSTRING](#) function. You can extract a list from an existing list using the SQL [\\$LIST](#) function or the ObjectScript [\\$LIST](#) function.

position

If you omit the *position* argument, **\$LISTDATA** evaluates the first element. If the value of the *position* argument is -1, it is equivalent to specifying the final element of the list. If the value of the *position* argument refers to a nonexistent list member, **\$LISTDATA** returns 0.

Examples

The following Embedded SQL examples show the results of the various values of the *position* argument.

All of the following **\$LISTDATA** statements return a value of 1:

```
KILL Y
SET a=$LISTBUILD("Red",,Y,"","Green")
&sql(SELECT $LISTDATA(:a), $LISTDATA(:a,1),
      $LISTDATA(:a,4), $LISTDATA(:a,5), $LISTDATA(:a,-1)
      INTO :b,:c, :d, :e, :f)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"1st element status ",b ; 1st element default
  WRITE !,"1st element status ",c ; 1st element specified
  WRITE !,"4th element status ",d ; 4th element null string
  WRITE !,"5th element status ",e ; 5th element in 5-element list
  WRITE !,"last element status ",f ; last element in 5-element list
}
```

The following **\$LISTDATA** statements return a value of 0 for the same five-element list:

```

KILL Y
SET a=$LISTBUILD("Red",,Y,"","Green")
&sql(SELECT $LISTDATA(:a,2), $LISTDATA(:a,3),
      $LISTDATA(:a,0), $LISTDATA(:a,6)
      INTO :b,:c, :d, :e)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"2nd element status ",b ; 2nd element is undefined
  WRITE !,"3rd element status ",c ; 3rd element is killed variable
  WRITE !,"0th element status ",d ; zero position nonexistent
  WRITE !,"6th element status ",e ; 6th element in 5-element list
}

```

Notes

Invalid Argument Values

If the expression in the *list* argument does not evaluate to a valid list, an SQLCODE -400 fatal error occurs:

```

&sql(SELECT $LISTDATA('fred') INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The the element is ",b }

```

If the value of the *position* argument is less than -1, an SQLCODE -400 fatal error occurs:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTDATA(:a,-3) INTO :c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"A neg-num position status ",c }

```

This does not occur when *position* is a nonnumeric value:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTDATA(:a,'g') INTO :c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Error code ",SQLCODE
  WRITE !,"A nonnumeric position status ",c }

```

See Also

- SQL functions: [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTFIND](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTLENGTH](#), [\\$LISTSAME](#), [\\$LISTTOSTRING](#), [\\$PIECE](#)
- ObjectScript functions: [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTDATA](#), [\\$LISTFIND](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTLENGTH](#), [\\$LISTNEXT](#), [\\$LISTSAME](#), [\\$LISTTOSTRING](#), [\\$LISTVALID](#)

\$LISTFIND

A list function that searches a specified list for the requested value.

```
$LISTFIND(list,value[,startafter])
```

Arguments

<i>list</i>	An expression that evaluates to a valid list. A <i>list</i> is an encoded character string containing one or more elements. You can create a <i>list</i> using the SQL or ObjectScript \$LISTBUILD or \$LISTFROMSTRING functions. You can extract a <i>list</i> from an existing list using the SQL or ObjectScript \$LIST function.
<i>value</i>	An expression containing the search element. A character string.
<i>startafter</i>	<i>Optional</i> — An integer expression interpreted as a list position. The search starts with the element after this position. Zero and -1 are valid values; -1 never returns an element. Zero is the default.

Description

\$LISTFIND searches the specified *list* for the first instance of the requested *value*. The search begins with the element after the position indicated by the *startafter* argument. If you omit the *startafter* argument, **\$LISTFIND** assumes a *startafter* value of 0 and starts the search with the first element (element 1). If the value is found, **\$LISTFIND** returns the position of the matching element. If the value is not found, **\$LISTFIND** returns a 0. The **\$LISTFIND** function will also return a 0 if the value of the *startafter* argument refers to a nonexistent list member.

This function returns data of type SMALLINT.

Examples

The following Embedded SQL example returns 2, the position of the first occurrence of the requested string:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTFIND(:a,'Blue') INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The position is ",b }
```

The following Embedded SQL example returns 0, indicating the requested string was not found:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTFIND(:a,'Orange') INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The position is ",b }
```

The following three Embedded SQL examples show the effect of using the *startafter* argument. The first example does not find the requested string and returns 0 because the requested string occurs at the *startafter* position:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTFIND(:a,'Blue',2) INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The position is ",b }
```

The second example finds the requested string at the first position by setting *startafter* to zero (the default value):

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTFIND(:a,'Red',0) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The position is ",b }

```

The third example finds the second occurrence of the requested string and returns 5, because the first occurs before the *startafter* position:

```

SET a=$LISTBUILD("Red","Blue","Green","Yellow","Blue")
&sql(SELECT $LISTFIND(:a,'Blue',3) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The position is ",b }

```

The **\$LISTFIND** function only matches complete elements. Thus, the following example returns 0 because no element of the list is equal to the string “B”, though all of the elements contain “B”:

```

SET a=$LISTBUILD("ABC","BCD","BBB")
&sql(SELECT $LISTFIND(:a,'B') INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The position is ",b }

```

Notes

Invalid Argument Values

If the expression in the *list* argument does not evaluate to a valid list, the **\$LISTFIND** function generates an SQLCODE -400 fatal error.

```

SET a="Blue"
&sql(SELECT $LISTFIND(:a,'Blue') INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The position is ",b }

```

If the value of the *startafter* argument is -1, **\$LISTFIND** always returns zero (0).

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTFIND(:a,'Blue',-1) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The position is ",b }

```

If the value of the *startafter* argument is less than -1, invoking the **\$LISTFIND** function generates an SQLCODE -400 fatal error.

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTFIND(:a,'Blue',-3) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The position is ",b }

```

See Also

- SQL functions: [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTDATA](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTLENGTH](#), [\\$LISTSAME](#), [\\$LISTTOSTRING](#), [\\$PIECE](#)
- ObjectScript functions: [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTDATA](#), [\\$LISTFIND](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTLENGTH](#), [\\$LISTNEXT](#), [\\$LISTSAME](#), [\\$LISTTOSTRING](#), [\\$LISTVALID](#)

\$LISTFROMSTRING

A list function that creates a list from a string.

```
$LISTFROMSTRING(string[,delimiter])
```

Arguments

<i>string</i>	A string to be converted into a Caché list. This string contains one or more elements, separated by a <i>delimiter</i> . The <i>delimiter</i> does not become part of the resulting Caché list.
<i>delimiter</i>	<i>Optional</i> — The delimiter used to separate substrings (elements) in <i>string</i> . Specify <i>delimiter</i> as a quoted string. If no <i>delimiter</i> is specified, the default is the comma (,) character.

Description

\$LISTFROMSTRING takes a quoted string containing delimited elements and returns a list. A list represents data in an encoded format which does not use delimiter characters. Thus a list can contain all possible characters, and is ideally suited for bitstring data. Lists are handled using the ObjectScript and Caché SQL **\$LIST** functions.

Arguments

string

A string literal (enclosed in single quotation marks), a numeric, or a variable or expression that evaluates to a string. This string can contain one or more substrings (elements), separated by a *delimiter*. The string data elements must not contain the *delimiter* character (or string), because the *delimiter* character is not included in the output list.

delimiter

A character (or string of characters) used to delimit substrings within the input string. It can be a numeric or string literal (enclosed in single quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character string, the individual characters of which can be used within string data. If you specify no delimiter, the default delimiter is the comma (,) character.

Examples

The following Embedded SQL example takes a string of names which are separated by a blank space, and creates a list:

```
SET names="Deborah Noah Martha Bowie"
&sql(SELECT $LISTFROMSTRING(:names,' ')
      INTO :namelist)
IF SQLCODE=0 {
  FOR n=1:1:$LISTLENGTH(namelist) {WRITE !,"element ",n," : ",$LIST(namelist,n)}
}
ELSE {WRITE !,"Error code;",SQLCODE }
```

The following Embedded SQL example uses the default delimiter (the comma character), and creates a list:

```
SET names="Deborah,Noah,Martha,Bowie"
&sql(SELECT $LISTFROMSTRING(:names)
      INTO :namelist)
IF SQLCODE=0 {
  FOR n=1:1:$LISTLENGTH(namelist) {WRITE !,"element ",n," : ",$LIST(namelist,n)}
}
ELSE {WRITE !,"Error code;",SQLCODE }
```

See Also

- SQL functions: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$PIECE](#)
- ObjectScript functions: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTNEXT](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$LISTVALID](#)

\$LISTGET

A list function that returns an element in a list or a specified default value.

```
$LISTGET(list[,position[,default]])
```

Arguments

<i>list</i>	An expression that evaluates to a valid list. A <i>list</i> is an encoded character string containing one or more elements. You can create a <i>list</i> using the SQL or ObjectScript \$LISTBUILD or \$LISTFROMSTRING functions. You can extract a <i>list</i> from an existing list using the SQL or ObjectScript \$LIST function.
<i>position</i>	<i>Optional</i> — An expression interpreted as a position in the specified list.
<i>default</i>	<i>Optional</i> — An expression that provides the value to return if the list element has an undefined value.

Description

\$LISTGET returns the requested element in the specified list as a standard character string. If the value of the *position* argument refers to a nonexistent member or identifies an element with an undefined value, the specified default value is returned.

The **\$LISTGET** function is identical to the one- and two-argument forms of the **\$LIST** function except that, under conditions that would cause **\$LIST** to return a null string, **\$LISTGET** returns a default value.

This function returns data of type VARCHAR.

You can use **\$LISTGET** to retrieve a field value from a serial container field. In the following example, Home is a serial container field, the third element of which is Home_State:

```
SELECT Name,$LISTGET(Home,3) AS HomeState
FROM Sample.Person
```

Arguments

list

An encoded character string containing one or more elements. You can create a list using the SQL **\$LISTBUILD** function or the ObjectScript **\$LISTBUILD** function. You can convert a delimited string into a list using the SQL **\$LISTFROMSTRING** function or the ObjectScript **\$LISTFROMSTRING** function. You can extract a list from an existing list using the SQL **\$LIST** function or the ObjectScript **\$LIST** function.

position

The *position* argument must evaluate to an integer. If it is omitted, by default, the function examines the first element of the list. If the value of the *position* argument is -1, it is equivalent to specifying the last element of the list.

default

A character string. If you omit the *default* argument, a zero-length string is assumed for the default value.

Examples

The **\$LISTGET** functions in the following Embedded SQL example both return “Red”, the first element in the list:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTGET(:a),$LISTGET(:a,1)
INTO :b,:c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The one-arg element returned is ",b
  WRITE !,"The two-arg element returned is ",c }

```

The **\$LISTGET** functions in the following Embedded SQL example both return “Green”, the third and last element in the list:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTGET(:a,3),$LISTGET(:a,-1)
INTO :b,:c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The third element is ",b
  WRITE !,"The last element is ",c }

```

The **\$LISTGET** functions in the following Embedded SQL example both return a value upon encountering the undefined 2nd element in the list. The first returns a question mark (?), which the user defined as the default value. The second returns a null string because a default value is not specified:

```

SET a=$LISTBUILD("Red",,"Green")
&sql(SELECT $LISTGET(:a,2,'?'),$LISTGET(:a,2)
INTO :b,:c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The default value is ",b
  WRITE !,"The no-default value is ",c }

```

The **\$LISTGET** functions in the following Embedded SQL example both specify a position greater than the last element in the three-element list. The first returns a null string because the default value is not specified. The second returns the user-specified default value, “ERR”:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTGET(:a,4),$LISTGET(:a,4,'ERR')
INTO :b,:c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The no-default 4th element is ",b
  WRITE !,"The default for 4th element is ",c }

```

The **\$LISTGET** functions in the following Embedded SQL example both return a null string:

```

SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTGET(:a,0),$LISTGET(NULL)
INTO :b,:c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The zero element is ",b
  WRITE !,"The NULL element is ",c }

```

Notes

Invalid Argument Values

If the expression in the *list* argument does not evaluate to a valid list, an SQLCODE -400 fatal error occurs because the **\$LISTGET** return variable remains undefined. This occurs even when a *default* value is supplied, as in the following Embedded SQL example:

```

&sql(SELECT $LISTGET('fred',1,'failsafe') INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"The non-list element is ",b ; Variable not set
}

```

If the value of the *position* argument is less than -1, an SQLCODE -400 fatal error occurs because the **\$LISTGET** return variable remains undefined. This occurs even when a *default* value is supplied, as in the following Embedded SQL example:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTGET(:a,-3,'failsafe') INTO :c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"A neg-num position returns ",c ; Variable not set
}
```

This does not occur when *position* is a nonnumeric value:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTGET(:a,'g','failsafe') INTO :c)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"A nonnumeric position returns ",c }
```

See Also

- SQL functions: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTLENGTH](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$PIECE](#)
- ObjectScript functions: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTNEXT](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$LISTVALID](#)

\$LISTLENGTH

A list function that returns the number of elements in a specified list.

```
$LISTLENGTH(list)
```

Argument

<i>list</i>	An expression that evaluates to a valid list. A <i>list</i> is an encoded character string containing one or more elements. You can create a <i>list</i> using the SQL or ObjectScript \$LISTBUILD or \$LISTFROMSTRING functions. You can extract a <i>list</i> from an existing list using the SQL or ObjectScript \$LIST function.
-------------	---

Description

\$LISTLENGTH returns the number of elements in *list*.

This function returns data of type SMALLINT.

Examples

The following Embedded SQL example returns 3, because there are 3 elements in the list:

```
SET a=$LISTBUILD("Red","Blue","Green")
&sql(SELECT $LISTLENGTH(:a) INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The number of elements is ",b }
```

The following SQL example also returns 3, because there are 3 elements in the list:

```
SELECT $LISTLENGTH($LISTBUILD('Red','Blue','Green'))
```

The following Embedded SQL example also returns 3. There are 3 elements in the list, though the second element contains no data:

```
SET a=$LISTBUILD("Red",,"Green")
&sql(SELECT $LISTLENGTH(:a) INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The number of elements is ",b }
```

In the following SQL example, each **\$LISTLENGTH** returns 3, because there are 3 elements in the list, though the second element contains no data:

```
SELECT $LISTLENGTH($LISTBUILD('Red','','Green')),
       $LISTLENGTH($LISTBUILD('Red',NULL,'Green')),
       $LISTLENGTH($LISTBUILD('Red','','Green'))
```

Notes

Invalid Lists

If *list* is not a valid list, an SQLCODE -400 fatal error is generated:

```
SET a="fred"
&sql(SELECT $LISTLENGTH(:a) INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The number of elements is ",b ; Variable not set
}
```

If the ObjectScript **\$LISTBUILD** function is used to build a list that contains only the null string, this is a valid *list*, containing one element:

```
SET a=$LISTBUILD("")
&sql(SELECT $LISTLENGTH(:a) INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The number of elements is ",b }
```

Null Lists

The SQL **\$LISTLENGTH** function and the ObjectScript **\$LISTLENGTH** function differ in how they handle a null list (a list containing no elements).

The following three Embedded SQL examples show how the **\$LISTLENGTH** SQL function handles a null list. In the first two examples, *list* is the null string, and a null string is returned:

```
SET a=""
&sql(SELECT $LISTLENGTH(:a)
INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The number of elements is ",b }

&sql(SELECT $LISTLENGTH(NULL)
INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The number of elements is ",b }
```

In the third example, *list* is the value **\$CHAR(0)**, which is an invalid list; an SQLCODE -400 fatal error is generated:

```
&sql(SELECT $LISTLENGTH(' ')
INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The number of elements is ",b }
```

Note that this differs from how the ObjectScript **\$LISTLENGTH** function handles a null list. In ObjectScript, the null string (") is used to represent a null list, a list containing no elements. Because it contains no list elements, it has a **\$LISTLENGTH** count of 0, as shown in the following example:

```
WRITE $LISTLENGTH("")
```

\$LISTLENGTH and Nested Lists

The following Embedded SQL example returns 3, because **\$LISTLENGTH** does not recognize the individual elements in nested lists:

```
SET a=$LISTBUILD("Apple", "Pear", $LISTBUILD("Walnut", "Pecan"))
&sql(SELECT $LISTLENGTH(:a)
INTO :b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,"The number of elements is ",b }
```

See Also

- SQL list functions: [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTDATA](#), [\\$LISTFIND](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTSAME](#), [\\$LISTTOSTRING](#)
- Other SQL functions: [\\$PIECE](#)

- ObjectScript list functions: [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTDATA](#), [\\$LISTFIND](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTLENGTH](#), [\\$LISTNEXT](#), [\\$LISTSAME](#), [\\$LISTTOSTRING](#), [\\$LISTVALID](#)

\$LISTSAME

A list function that compares two lists and returns a boolean value.

```
$LISTSAME(list1, list2)
```

Arguments

<i>list1</i>	An expression that evaluates to a valid list.
<i>list2</i>	An expression that evaluates to a valid list.

Description

\$LISTSAME compares the contents of two lists and returns 1 if the lists are the same. If the lists are not the same, **\$LISTSAME** returns 0. **\$LISTSAME** compares the two lists element-by-element. For two lists to be the same, they must contain the same number of elements and each element in *list1* must match the corresponding element in *list2*.

\$LISTSAME compares list elements using their string representations. **\$LISTSAME** comparisons are case-sensitive. **\$LISTSAME** compares the two lists element-by-element in left-to-right order. Therefore **\$LISTSAME** returns a value of 0 when it encounters the first non-matching pair of list elements; it does not check subsequent items to determine if they are valid list elements.

This function returns data of type SMALLINT.

Arguments

list (list1 and list2)

A *list* is an encoded character string containing one or more elements. You can create a list using the SQL **\$LISTBUILD** function or the ObjectScript **\$LISTBUILD** function. You can convert a delimited string into a list using the SQL **\$LISTFROMSTRING** function or the ObjectScript **\$LISTFROMSTRING** function. You can extract a list from an existing list using the SQL **\$LIST** function or the ObjectScript **\$LIST** function.

The following are examples of valid lists:

- **\$LISTBUILD('a', 'b', 'c')**: a three-element list.
- **\$LISTBUILD('a', '', 'c')**: a three-element list, the second element of which has a null string value.
- **\$LISTBUILD('a', , 'c')** or **\$LISTBUILD('a', NULL, 'c')**: a three-element list, the second element of which has no value.
- **\$LISTBUILD(NULL, NULL)** or **\$LISTBUILD(, NULL)**: a two-element list, the elements of which have no values.
- **\$LISTBUILD(NULL)** or **\$LISTBUILD()**: a one-element list, the element has no value.

If a *list* argument is NULL, **\$LISTSAME** returns NULL. If a *list* argument is not a valid list (and is not NULL), Caché SQL generates an SQLCODE -400 fatal error.

Examples

The following embedded SQL example uses **\$LISTSAME** to compare two list arguments:

```

SET a=$LISTBUILD("Red",,"Yellow","Green","", "Violet")
SET b=$LISTBUILD("Red",,"Yellow","Green","", "Violet")
&sql(SELECT $LISTSAME(:a,:b)
      INTO :c )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSEIF c=1 { WRITE "lists a and b are the same",! }
ELSE { WRITE "lists a and b are not the same",! }

```

The following SQL example compares lists with NULL, absent, or empty string elements:

```

SELECT $LISTSAME($LISTBUILD('Red',NULL,'Blue'),$LISTBUILD('Red',,'Blue')) AS NullAbsent,
       $LISTSAME($LISTBUILD('Red',NULL,'Blue'),$LISTBUILD('Red','','Blue')) AS NullEmpty,
       $LISTSAME($LISTBUILD('Red',,'Blue'),$LISTBUILD('Red','','Blue')) AS AbsentEmpty

```

\$LISTSAME comparison is not the same equivalence test as the one used by the Caché equal sign. An equal sign compares the two lists as encoded strings (character-by-character); **\$LISTSAME** compares the two lists element-by-element. This distinction is easily seen when comparing a number and a numeric string, as in the following example:

```

SET a = $LISTBUILD("365")
SET b = $LISTBUILD(365)
IF a=b
  { WRITE "Equal sign: lists a and b are the same",! }
ELSE { WRITE "Equal sign: lists a and b are not the same",! }
&sql(SELECT $LISTSAME(:a,:b)
      INTO :c )
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSEIF c=1 { WRITE "$LISTSAME: lists a and b are the same",! }
ELSE { WRITE "$LISTSAME: lists a and b are not the same",! }

```

The following SQL example compares lists containing numbers and numeric strings in canonical and non-canonical forms. When comparing a numeric list element and a string list element, the string list element must represent the numeric in canonical form; this is because Caché always reduces numbers to canonical form before performing a comparison. In the following example, **\$LISTSAME** compares a string and a number. The first three **\$LISTSAME** functions return 1 (identical); the fourth **\$LISTSAME** function returns 0 (not identical) because the string representation is not in canonical form:

```

SELECT $LISTSAME($LISTBUILD('365'),$LISTBUILD(365)),
       $LISTSAME($LISTBUILD('365'),$LISTBUILD(365.0)),
       $LISTSAME($LISTBUILD('365.5'),$LISTBUILD(365.5)),
       $LISTSAME($LISTBUILD('365.0'),$LISTBUILD(365.0))

```

See Also

- SQL functions: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTTOSTRING](#) [\\$PIECE](#)
- ObjectScript functions: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTNEXT](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$LISTVALID](#)

\$LISTTOSTRING

A list function that creates a string from a list.

```
$LISTTOSTRING(list[,delimiter])
```

Arguments

<i>list</i>	An expression that evaluates to a valid list. A <i>list</i> is an encoded character string containing one or more elements. You can create a <i>list</i> using the SQL or ObjectScript \$LISTBUILD or \$LISTFROMSTRING functions. You can extract a <i>list</i> from an existing list using the SQL or ObjectScript \$LIST function.
<i>delimiter</i>	<i>Optional</i> — A delimiter inserted to separate substrings. A delimiter can be one or more characters, specified as a quoted string. To concatenate the substrings with no delimiter, specify the empty string ("). If you specify no <i>delimiter</i> , the default is the comma (,) character.

Description

\$LISTTOSTRING takes a Caché list and converts it to a string. In the resulting string, the elements of the list are separated by the *delimiter*.

A list represents data in an encoded format which does not use delimiter characters. Thus a list can contain all possible characters, and is ideally suited for bitstring data. **\$LISTTOSTRING** converts this list to a string with delimited elements. It sets aside a specified character (or character string) to serve as a delimiter. These delimited elements can be handled using the **\$PIECE** function.

Note: The *delimiter* specified here must not occur in the source data. Caché makes no distinction between a character serving as a delimiter and the same character as a data character.

You can use **\$LISTTOSTRING** to retrieve field values from a serial container field as a delimited string. In the following example, Home is a serial container field. It contains the list elements Home_Street, Home_City, Home_State, and Home_Zip:

```
SELECT Name, $LISTTOSTRING(Home, '^') AS HomeAddress
FROM Sample.Person
```

Arguments

list

An encoded character string containing one or more elements. You can create a list using the SQL **\$LISTBUILD** function or the ObjectScript **\$LISTBUILD** function. You can convert a delimited string into a list using the SQL **\$LISTFROMSTRING** function or the ObjectScript **\$LISTFROMSTRING** function. You can extract a list from an existing list using the SQL **\$LIST** function or the ObjectScript **\$LIST** function.

If the expression in the *list* argument does not evaluate to a valid list, an SQLCODE -400 error occurs.

delimiter

A character (or string of characters) used to delimit substrings within the output string. It can be a numeric or string literal (enclosed in single quotation marks), a host variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character string, the individual characters of which can be used within string data.

If you specify no delimiter, the default delimiter is the comma (,) character. You can specify a null string (") as a delimiter; in this case, substrings are concatenated with no delimiter. To specify the single quote character as the delimiter, duplicate the quote character thus: '''' — four single quote characters.

Example

The following example converts the values of a list field to a string with the elements delimited by the colon (:) character:

```
SELECT
Name,
FavoriteColors AS ColorList,
$LISTTOSTRING(FavoriteColors,':') AS ColorStrings
FROM Sample.Person
WHERE FavoriteColors IS NOT NULL
```

See Also

- SQL functions: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTSAME](#) [\\$PIECE](#)
- ObjectScript functions: [\\$LIST](#) [\\$LISTBUILD](#) [\\$LISTDATA](#) [\\$LISTFIND](#) [\\$LISTFROMSTRING](#) [\\$LISTGET](#) [\\$LISTLENGTH](#) [\\$LISTNEXT](#) [\\$LISTSAME](#) [\\$LISTTOSTRING](#) [\\$LISTVALID](#)

LOG

A scalar numeric function that returns the natural logarithm of a given numeric expression.

```
{fn LOG(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type FLOAT.
-------------------------	------------------------------

Description

LOG returns the natural logarithm (base e) of *float-expression*. **LOG** returns a value of data type FLOAT with a precision of 21 and a scale of 18.

LOG can only be used as an ODBC scalar function (with the curly brace syntax).

Examples

The following example returns the natural logarithm of an integer:

```
SELECT {fn LOG(5)} AS Logarithm
```

returns 1.60943791...

The following Embedded SQL example shows the relationship between the **LOG** and **EXP** functions for the integers 1 through 10:

```
SET a=1
WHILE a<11 {
  &sql(SELECT {fn LOG(:a)} INTO :b)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
  ELSE {
    WRITE !,"Logarithm of ",a," = ",b }
  &sql(SELECT ROUND({fn EXP(:b)},12) INTO :c)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
  ELSE {
    WRITE !,"Exponential of log ",b," = ",c
    SET a=a+1 }
}
```

Note that the **ROUND** function is needed here to correct for very small discrepancies caused by system calculation limitations. In the above example, **ROUND** is set arbitrarily to 12 decimal digits for this purpose.

See Also

- SQL functions: [EXP](#), [LOG10](#), [ROUND](#)
- ObjectScript function: [\\$ZLN](#)

LOG10

A scalar numeric function that returns the base-10 logarithm of a given numeric expression.

```
{fn LOG10(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type FLOAT.
-------------------------	------------------------------

Description

LOG10 returns the base-10 logarithm value of *float-expression*. **LOG10** returns a value of data type FLOAT with a precision of 21 and a scale of 18.

LOG10 can only be used as an ODBC scalar function (with the curly brace syntax).

Examples

The following example returns the base-10 logarithm of an integer:

```
SELECT {fn LOG10(5)} AS Log10
```

returns .69897000433...

The following Embedded SQL example returns the base-10 logarithm values for the integers 1 through 10:

```
SET a=1
WHILE a<11 {
  &sql(SELECT {fn LOG10(:a)} INTO :b)
  IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE
    QUIT }
  ELSE {
    WRITE !,"Log-10 of ",a," = ",b
    SET a=a+1 }
}
```

See Also

- SQL functions: [EXP](#), [LOG](#), [ROUND](#)
- ObjectScript function: [\\$ZLOG](#)

LOWER

A case-transformation function that converts all uppercase letters in a string expression to lowercase letters.

```
LOWER(string-expression)
```

Arguments

<i>string-expression</i>	The string expression whose characters are to be converted to lowercase. The expression can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
--------------------------	---

Description

The **LOWER** function converts uppercase letters to lowercase for display purposes. This is the inverse of the **UPPER** function. **LOWER** has no effects on non-alphabetic characters. It leave unchanged punctuation, numbers, and leading and trailing blank spaces.

LOWER does not force a numeric to be interpreted as a string. Caché SQL converts numerics to canonical form, removing leading and trailing zeros. A numeric specified as a string is not converted to canonical form, and retains leading and trailing zeros.

The **LCASE** function can also be used convert uppercase letters to lowercase.

LOWER has no effect on [collation](#). The **%SQLUPPER** function is the preferred way in SQL to convert a data value for not case-sensitive collation. Refer to **%SQLUPPER** for further information on case transformation for collation.

Examples

The following example returns each person's name in lowercase letters:

```
SELECT Name,LOWER(Name) AS LowName
FROM Sample.Person
```

LOWER also works on Unicode (non-ASCII) alphabetic characters, as shown in the following Embedded SQL example, which converts Greek letters from uppercase to lowercase:

```
IF $SYSTEM.Version.IsUnicode() {
  SET a=$CHAR(920,913,923,913,931,931,913)
  &sql(SELECT LOWER(:a)
  INTO :b
  FROM Sample.Person)
  IF SQLCODE'=0 {WRITE !,"Error code ",SQLCODE }
  ELSE {WRITE !,a!,b }
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- SQL functions: [LCASE UCASE](#)
- ObjectScript function: [\\$ZCONVERT](#)

LPAD

A string function that returns a string left-padded to a specified length.

```
LPAD(string-expression,length[,padstring])
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, a host variable, or the result of another scalar function. Can be of any data type convertible to a VARCHAR data type. <i>string-expression</i> cannot be a stream.
<i>length</i>	An integer specifying the number of characters in the returned string.
<i>padstring</i>	<i>Optional</i> — A string consisting of a character or a string of characters used to pad the input <i>string-expression</i> . The <i>padstring</i> character or characters are appended to the left of <i>string-expression</i> to supply as many characters as need to create an output string of <i>length</i> characters. <i>padstring</i> may be a string literal, a column, a host variable, or the result of another scalar function. If omitted, the default is a blank space character.

Description

LPAD pads a string expression with leading pad characters. It returns a copy of the string padded to *length* number of characters. If the string expression is longer than *length* number of characters, the return string is truncated to *length* number of characters.

If *string-expression* is NULL, **LPAD** returns NULL. If *string-expression* is the empty string (") **LPAD** returns a string consisting entirely of pad characters. The returned string is type VARCHAR.

LPAD does not remove leading or trailing blanks; it pads the string including any leading or trailing blanks. To remove leading or trailing blanks before padding a string, use **LTRIM**, **RTRIM**, or **TRIM**.

LPAD and \$JUSTIFY

The two-argument form of **LPAD** and the two-argument form of **\$JUSTIFY** both right-align a string by padding it with leading spaces. These two-argument forms differ in how they handle an output *length* that is shorter than the length of the input *string-expression*: **LPAD** truncates the input string to fit the specified output length. **\$JUSTIFY** expands the output length to fit the input string. This is shown in the following example:

```
SELECT '>' || LPAD(12345,10) || '<' AS lpadplus,
       '>' || $JUSTIFY(12345,10) || '<' AS justifyplus,
       '>' || LPAD(12345,3) || '<' AS lpadminus,
       '>' || $JUSTIFY(12345,3) || '<' AS justifyminus
```

Examples

The following example left pads column values with ^ characters (when needed) to return strings of length 16. Note that some Name strings are left padded, some Name strings are right truncated to return strings of length 16.

```
SELECT TOP 15 Name,LPAD(Name,16,'^') AS Name16
FROM Sample.Person
```

The following example left pads column values with the ^=^ pad string (when needed) to return strings of length 20. Note that the pad name string is repeated as many times as needed, and that some return strings contain partial pad strings:

```
SELECT TOP 15 Name,LPAD(Name,20,'^=^') AS Name20
FROM Sample.Person
```

See Also

- [\\$JUSTIFY](#) function
- [RPAD](#) function
- [LTRIM](#) function
- [RTRIM](#) function
- [TRIM](#) function

LTRIM

A string function that returns a string with the leading blanks removed.

```
LTRIM(string-expression)
{fn LTRIM(string-expression)}
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
--------------------------	--

Description

LTRIM removes the leading blanks from a string expression, and returns the string as type VARCHAR. If *string-expression* is NULL, **LTRIM** returns NULL. If *string-expression* is a string consisting entirely of blank spaces, **LTRIM** returns the empty string ("").

LTRIM leave trailing blanks; to remove trailing blanks, use **RTRIM**. To remove leading and/or trailing characters of any type, use **TRIM**. To pad a string with leading blanks or other characters, use **LPAD**. To create a string of blanks, use **SPACE**.

Note that **LTRIM** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Examples

The following Embedded SQL example removes the five leading blanks from the string. It leaves the five trailing blanks:

```
SET a="      Test string with 5 leading and 5 trailing spaces.      "
&sql(SELECT {fn LTRIM(:a)} INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Before LTRIM",!,"start:",a,"end"
  WRITE !,"After LTRIM",!,"start:",b,"end" }
```

Returns:

```
Before LTRIM
start:      Test string with 5 leading and 5 trailing spaces.      :end
After LTRIM
start:Test string with 5 leading and 5 trailing spaces.      :end
```

See Also

[RTRIM](#) [TRIM](#) [LPAD](#) [SPACE](#)

%MINUS

A collation function that converts numbers to canonical collation format, then inverts the sign.

```
%MINUS(expression)
```

```
%MINUS expression
```

Arguments

<i>expression</i>	An expression, which can be the name of a column, a number or a string literal, an arithmetic expression, or the result of another function, where the underlying data type can be represented as any character type.
-------------------	---

Description

%MINUS converts numbers or numeric strings to canonical form, inverts the sign, then returns these *expression* values in numeric collation sequence.

%MINUS and **%PLUS** are functionally identical, except that **%MINUS** inverts the sign. It prefixes a minus sign to any number that resolves to a positive number, and removes the minus sign from any number that resolves to a negative number. Zero is never signed.

A number can contain leading and trailing zeros, multiple leading plus and minus signs, a single decimal point indicator (.), and the E exponent indicator. In canonical form, all arithmetic operations are performed, exponents are expanded, signs are resolved to either a single leading minus sign or no sign, and leading and trailing zeros are stripped.

A numeric literal can be specified with or without enclosing string delimiters. If a string contains non-numeric characters, **%MINUS** truncates the number at the first non-numeric character, and returns the numeric part in canonical form. A non-numeric string (any string that begins with a non-numeric character) is returned as 0. **%MINUS** also returns NULLs as 0.

%MINUS is a Caché SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

```
WRITE $SYSTEM.Util.Collation("++007.500",4)
```

Compare **%MINUS** to **%MVR** collation, which sorts a string based on the numeric substrings within the string.

Example

The following example uses **%MINUS** to return records in descending numeric order of the home street number:

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %MINUS(Home_Street)
```

Note that the above example orders the integer part of the street address in numerical order. Compare this with the following **ORDER BY DESC** example, which orders records by street addresses in collation sequence:

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY Home_Street DESC
```

See Also

- **%EXACT** collation function
- **%PLUS** collation function

- [%MVR](#) collation function
- [Collation](#) chapter in *Using Caché SQL*

MINUTE

A time function that returns the minute for a datetime expression.

```
{fn MINUTE(time-expression)}
```

Arguments

<i>time-expression</i>	An expression that is the name of a column, the result of another scalar function, or a string or numeric literal. It must resolve either to a datetime string or a time integer, where the underlying data type can be represented as %Time or %TimeStamp.
------------------------	---

Description

MINUTE returns an integer specifying the minutes for a given time or datetime value. Minutes are calculated for a [\\$HOROLOGY](#) or [\\$ZTIMESTAMP](#) value, an ODBC format date string, or a timestamp.

A *time-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

To change the default time format, use the [SET OPTION](#) command.

Note that you can supply a time integer (number of elapsed seconds), but not a time string (hh:mm:ss). You must supply a datetime string (yyyy-mm-dd hh:mm:ss). You can omit the seconds (:ss) portion of a datetime string and still return the minutes portion. The *time-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

The minutes (mm) portion should be an integer in the range from 0 through 59. There is, however, no range checking for user-supplied values. Numbers greater than 59, negative numbers and fractions are returned as specified. Leading zeros are optional on input; leading zeros are suppressed on output.

MINUTE returns zero minutes when the minutes portion is '0', '00', or a nonnumeric value. Zero minutes is also returned if no time expression is supplied, or the minutes portion of the time expression is omitted entirely ('hh', 'hh:', 'hh:', or 'hh:ss'), or if the time expression format is invalid.

The same time information can be returned using [DATEPART](#) or [DATENAME](#).

This function can also be invoked from ObjectScript using the **MINUTE()** method call:

```
$SYSTEM.SQL.MINUTE(time-expression)
```

Examples

The following examples both return the number 45 because it is the forty-fifth minute of the time expression in the datetime string:

```
SELECT {fn MINUTE('2000-02-16 18:45:38')} AS Minutes_Given
```

```
SELECT {fn MINUTE(67538)} AS Minutes_Given
```

The following example also returns 45. As shown here, the seconds portion of the time value can be omitted:

```
SELECT {fn MINUTE('2000-02-16 18:45')} AS Minutes_Given
```

The following example returns 0 minutes because the time expression has been omitted from the datetime string:

```
SELECT {fn MINUTE('2000-02-16')} AS Minutes_Given
```

The following examples all return the minutes portion of the current time:

```
SELECT {fn MINUTE(CURRENT_TIME)} AS Min_CurrentT,  
       {fn MINUTE({fn CURTIME()})} AS Min_CurT,  
       {fn MINUTE({fn NOW()})} AS Min_Now,  
       {fn MINUTE($HOROLOG)} AS Min_Horolog,  
       {fn MINUTE($ZTIMESTAMP)} AS Min_ZTS
```

The following example shows that leading zeros are suppressed. The first **MINUTE** function returns a length 2, the others return a length of 1. An omitted time is considered to be 0 minutes, which has a length of 1:

```
SELECT LENGTH({fn MINUTE('2004-02-05 11:45:00')}),  
       LENGTH({fn MINUTE('2004-02-15 03:05:00')}),  
       LENGTH({fn MINUTE('2004-02-15 3:5:0')}),  
       LENGTH({fn MINUTE('2004-02-15')})
```

The following Embedded SQL example shows that the **MINUTE** function recognizes the TimeSeparator character specified for the locale:

```
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeSeparator",".")  
&sql(SELECT {fn MINUTE('2000-02-16 18.45.38')}  
INTO :a)  
WRITE "minutes=",a
```

See Also

- SQL concepts: [Data Type](#), [Date and Time Constructs](#)
- SQL functions: [HOUR](#), [SECOND](#), [CURRENT_TIME](#), [CURTIME](#), [NOW](#), [DATEPART](#), [DATENAME](#)
- ObjectScript function: [\\$ZTIME](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

MOD

A scalar numeric function that returns the modulus (remainder) of a number divided by another.

```
{fn MOD(dividend,divisor)}
```

Arguments

<i>dividend</i>	A number that is the numerator (dividend) of the division.
<i>divisor</i>	A number that is the denominator (divisor) of the division.

Description

MOD returns the mathematical remainder (modulus) from the dividend by the divisor. It returns a negative or zero result for a division involving a negative divisor. **MOD** returns NULL if passed a NULL value for either argument. The value returned is of data type NUMERIC.

MOD can only be used as an ODBC scalar function (with the curly brace syntax).

Examples

The following example shows the remainder returned by **MOD**.

```
SELECT {fn MOD(5,3)} AS Remainder
```

returns 2.

The following example shows the remainder returned by **MOD** with a negative dividend.

```
SELECT {fn MOD(-5,3)} AS Remainder
```

returns 1.

The following example shows the remainder returned by **MOD** with a negative divisor.

```
SELECT {fn MOD(5,-3)} AS Remainder
```

returns -1.

See Also

[CEILING](#) [FLOOR](#) [ROUND](#) [TRUNCATE](#)

MONTH

A date function that returns the month as an integer for a date expression.

```
MONTH(date-expression)
{fn MONTH(date-expression)}
```

Arguments

<i>date-expression</i>	An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

MONTH returns an integer specifying the month. The month integer is calculated for a Caché date integer, a [\\$HOROLOG](#) or [\\$ZTIMESTAMP](#) value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type `%Library.TimeStamp` (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted. The *date-expression* can also be specified as data type `%Library.FilemanDate`, `%Library.FilemanTimestamp`, or `%MV.Date`.

The month (mm) portion of a date string should be an integer in the range from 1 through 12. There is, however, no range checking for user-supplied values. Numbers greater than 12, zero, and fractions are returned as specified. Because (–) is used as a separator character, negative numbers are not supported. Leading zeros are optional on input. Leading and trailing zeros are suppressed on output.

MONTH returns zero when the month portion is '0', '00', or a nonnumeric value. Zero is also returned if the month portion of the date string is omitted entirely ('yyyy—dd'), or if no date expression is supplied.

MONTH interprets the second numeric string encountered in a date string as the month value, so omitting the year portion of the date string ('mm-dd hh:mm:ss'), results in the second number encountered ('dd') being treated as the month value. Thus, a leading hyphen or some placeholder should be supplied for an unknown year value; for compatibility with Caché, 9999 is generally the preferred default year value.

Note that **MONTH** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

This function can also be invoked from ObjectScript using the **MONTH()** method call:

```
SYSTEM.SQL.MONTH(date-expression)
```

The elements of a datetime string can be returned using the following SQL functions: **YEAR**, **MONTH**, **DAY** (or **DAYOFMONTH**), **HOURL**, **MINUTE**, and **SECOND**. The same elements can be returned by using the [DATEPART](#) or [DATENAME](#) function. Date elements can be returned using [TO_DATE](#). **DATEPART** and **DATENAME** performs value and range checking on month values.

The [LAST_DAY](#) function returns the date of the last day of the specified month.

Examples

The following examples both return the number 2 because February is the second month of the year:

```
SELECT MONTH('2000-02-16') AS Month_Given
```

```
SELECT {fn MONTH(59589)} AS Month_Given
```

The following example sorts records in birthday order by month and day, ignoring the year component of the DOB:

```
SELECT Name,DOB AS Birthdays
FROM Sample.Person
ORDER BY MONTH(DOB),DAY(DOB),Name
```

The following examples returns zero because the month is omitted:

```
SELECT MONTH('2000--16') AS Month_Given  
  
SELECT {fn MONTH('12:34:55')} AS Month_Given  
  
SELECT MONTH('2000 12:34:55') AS Month_Given
```

The following example returns the number 2 because a placeholder character (-) has been supplied for the omitted year:

```
SELECT {fn MONTH('-02-16')} AS Month_Given
```

The following examples all return the current month:

```
SELECT {fn MONTH({fn NOW()})} AS MNow,  
       MONTH(CURRENT_DATE) AS MCurrD,  
       {fn MONTH(CURRENT_TIMESTAMP)} AS MCurrTS,  
       MONTH($HOROLOG) AS MHorolog,  
       {fn MONTH($ZTIMESTAMP)} AS MZTS
```

See Also

- SQL functions: [DATEPART](#), [DATENAME](#), [DAYOFMONTH](#), [LAST_DAY](#), [MONTHNAME](#), [TO_DATE](#)
- ObjectScript function: [\\$ZDATE](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

MONTHNAME

A date function that returns the name of the month for a date expression.

```
{fn MONTHNAME(date-expression)}
```

Arguments

<i>date-expression</i>	An expression that evaluates to either a Caché date integer, an ODBC date, or a timestamp. This expression can be the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

MONTHNAME takes as input a Caché date integer, a [\\$HOROLOG](#) or [\\$ZTIMESTAMP](#) value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type `%Library.TimeStamp` (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted. The *date-expression* can also be specified as data type `%Library.FilemanDate`, `%Library.FilemanTimestamp`, or `%MV.Date`.

MONTHNAME returns the name of the corresponding calendar month, January through December. The returned value is a character string with a maximum length of 15.

MONTHNAME checks that the date supplied is a valid date. The year must be between 1841 and 9999 (inclusive), the month 01 through 12, and the day appropriate for that month (for example, 02/29 is only valid on leap years). If the date is not valid, **MONTHNAME** issues an `SQLCODE -400` error (Fatal error occurred).

The names of months default to the full-length American English month names. To change these month name values, use the [SET OPTION](#) command with the `MONTH_NAME` option.

The same month name information can be returned by using the [DATENAME](#) function. You can use [TO_DATE](#) to retrieve a month name or a month name abbreviation with other date elements. To return an integer corresponding to the month, use [MONTH DATEPART](#) or [TO_DATE](#).

This function can also be invoked from ObjectScript using the **MONTHNAME()** method call:

```
$SYSTEM.SQL.MONTHNAME(date-expression)
```

Examples

The following examples both return the character string "February" because it is the month of the date expression (February 16, 2000):

```
SELECT {fn MONTHNAME('2000-02-16')} AS NameOfMonth
```

```
SELECT {fn MONTHNAME(59589)} AS NameOfMonth
```

The following examples all return the current month:

```
SELECT {fn MONTHNAME({fn NOW()})} AS MnameNow,
       {fn MONTHNAME(CURRENT_DATE)} AS MNameCurrDate,
       {fn MONTHNAME(CURRENT_TIMESTAMP)} AS MNameCurrTS,
       {fn MONTHNAME($HOROLOG)} AS MNameHorolog,
       {fn MONTHNAME($ZTIMESTAMP)} AS MNameZTS
```

The following Embedded SQL example shows how **MONTHNAME** responds to an invalid date (the year 2001 was not a leap year):

```
SET testdate="2001-02-29"
&sql(SELECT {fn MONTHNAME(:testdate)} INTO :a)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE,!
  WRITE %msg,! }
ELSE {
  WRITE !,"returns: ",a }
QUIT
```

The SQLCODE -400 error code is issued with the %msg indicating <ILLEGAL VALUE>. This validity test is performed before testing if the date is in the allowed range of years (after 1840).

The following embedded SQL example shows how **MONTHNAME** responds to a valid, but out-of-range date (the year 1835 is too early for Caché SQL):

```
SET testdate="1835-02-19"
&sql(SELECT {fn MONTHNAME(:testdate)} INTO :a)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE,!
  WRITE %msg,! }
ELSE {
  WRITE !,"returns: ",a }
QUIT
```

The SQLCODE -400 error code is issued with the %msg indicating <VALUE OUT OF RANGE>.

See Also

- SQL functions: [DATEPART](#) [DATENAME](#) [DAYOFMONTH](#) [MONTH](#) [TO_DATE](#)
- ObjectScript function: [\\$ZDATE](#)
- ObjectScript special variables: [\\$HOROLOG](#) [\\$ZTIMESTAMP](#)

NOW

A date/time function that returns the current local date and time.

```
NOW( )
{ fn NOW }
{ fn NOW( ) }
```

Description

NOW takes no arguments. The argument parentheses are optional for the ODBC scalar syntax; they are mandatory for the SQL standard function syntax.

NOW returns the current local date and time for this [timezone](#) as a timestamp; it adjusts for local time variants, such as [Daylight Saving Time](#).

NOW returns a [timestamp](#) in %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff).

To change the default datetime string format, use the [SET OPTION](#) command with the various date and time options.

You can use the [CAST](#) or [CONVERT](#) function to change the data type of timestamps, dates, and times.

Fractional Seconds of Precision

By default, **NOW** does not return fractional seconds of precision. It does not support a precision argument. However, by changing the [system-wide default time precision](#), you can cause all **NOW** functions system-wide to return this configured number of digits of fractional second precision. The initial configuration setting of the system-wide default time precision is 0 (no fractional seconds); the highest setting is 9.

[GETDATE](#) is functionally identical to **NOW**, except that [GETDATE](#) provides a *precision* argument that allows you to override the system-wide default time precision; if you omit the *precision* argument, [GETDATE](#) takes the configured system-wide default time precision.

[CURRENT_TIMESTAMP](#) has two syntax forms: Without argument parentheses, [CURRENT_TIMESTAMP](#) is functionally identical to **NOW**. With argument parentheses, [CURRENT_TIMESTAMP\(precision\)](#), is functionally identical to [GETDATE](#), except that the [CURRENT_TIMESTAMP\(\)](#) *precision* argument is mandatory. [CURRENT_TIMESTAMP\(\)](#) always returns its specified *precision* and ignores the configured system-wide default time precision.

Fractional seconds are always truncated, not rounded, to the specified precision.

[SYSDATE](#) is functionally identical to the argumentless [CURRENT_TIMESTAMP](#) function.

Other Current Time and Date Functions

NOW, [GETDATE](#), [CURRENT_TIMESTAMP](#), and [SYSDATE](#) all return the current local date and time, based on the local time zone setting.

[GETUTCDATE](#) returns the current Universal Time Constant (UTC) date and time as a timestamp. Because UTC time does not depend on the local timezone and is not subject to local time variants (such as [Daylight Saving Time](#)), this function is useful for applying consistent timestamps when users in different time zones access the same database. [GETUTCDATE](#) supports fractional seconds of precision. The current UTC timestamp is also provided by the ObjectScript [\\$ZTIMESTAMP](#) special variable.

To return just the current date, use [CURDATE](#) or [CURRENT_DATE](#). To return just the current time, use [CURRENT_TIME](#) or [CURTIME](#). The functions use the DATE or TIME data type. The TIME and DATE data types store their values as integers in [\\$HOROLOG](#) format. None of these functions support precision.

Examples

The following example shows the three syntax forms are equivalent; all return the current local date and time as a timestamp:

```
SELECT NOW(), {fn NOW}, {fn NOW() }
```

The following Embedded SQL example compares local (time zone specific) and universal (time zone independent) timestamps:

```
&sql(SELECT NOW(),GETUTCDATE() INTO :a,:b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Local timestamp is: ",a
  WRITE !,"UTC timestamp is: ",b
  WRITE !,"$ZTIMESTAMP is: ",$ZDATETIME($ZTIMESTAMP,3,,3)
}
```

The following example sets the LastUpdate field in the selected row of the Orders table to the current system date and time:

```
UPDATE Orders SET LastUpdate = {fn NOW()}
WHERE Orders.OrderNumber=:ord
```

See Also

- SQL concepts: [Data Type](#), [Date and Time Constructs](#)
- SQL timestamp functions: [CAST](#), [CONVERT](#), [CURRENT_TIMESTAMP](#), [GETDATE](#), [GETUTCDATE](#), [SYSDATE](#), [TIMESTAMPADD](#), [TIMESTAMPDIFF](#), [TO_TIMESTAMP](#)
- SQL current date and time functions: [CURDATE](#), [CURRENT_DATE](#), [CURRENT_TIME](#), [CURTIME](#)
- ObjectScript: [\\$ZDATETIME](#) function, [\\$HOROLOG](#) special variable, [\\$ZTIMESTAMP](#) special variable

NULLIF

A function that returns NULL if an expression is true.

```
NULLIF(expression1, expression2)
```

Arguments

<i>expression1</i>	An SQL expression.
<i>expression2</i>	An SQL expression.

Description

The **NULLIF** function returns NULL if *expression1* is equal to *expression2*, otherwise it returns *expression1*. The data type returned in DISPLAY mode or ODBC mode is determined by the data type of *expression1*.

NULLIF is equivalent to:

```
SELECT CASE  
WHEN value1 = value2 THEN NULL  
ELSE value1  
END  
FROM MyTable
```

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual [equality \(or non-equality\) condition comparison](#).

SQL Function	Comparison Test	Return Value
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2,ex3) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2) }	ex1 = NULL	True returns ex2 False returns ex1
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex

Examples

The following example uses the **NULLIF** function to set to null the display field of all records with Age=20:

```
SELECT Name, Age, NULLIF(Age, 20) AS Nulled20
FROM Sample.Person
```

See Also

- [CASE](#) command
- [COALESCE](#) function
- [IFNULL](#) function
- [ISNULL](#) function
- [NVL](#) function

NVL

A function that tests for NULL and returns the appropriate expression.

```
NVL(check-expression, replace-expression)
```

Arguments

<i>check-expression</i>	The expression to be evaluated.
<i>replace-expression</i>	The expression that is returned if <i>check-expression</i> is NULL.

Description

NVL evaluates *check-expression* and returns one of two values:

- If *check-expression* is NULL, *replace-expression* is returned.
- If *check-expression* is not NULL, *check-expression* is returned.

The arguments *check-expression* and *replace-expression* can have any data type. If their data types are different, Caché converts *replace-expression* to the data type of *check-expression* before comparing them. The data type of the return value is always the same as the data type of *check-expression*, unless *check-expression* is character data, in which case the return value's data type is VARCHAR2.

Note that **NVL** is supported for Oracle compatibility, and is the same as the **ISNULL** function.

Refer to [NULL](#) section of the “Language Elements” chapter of *Using Caché SQL* for further details on NULL handling.

NULL Handling Functions Compared

The following table shows the various SQL comparison functions. Each function returns one value if the logical comparison tests True (A same as B) and another value if the logical comparison tests False (A not same as B). These functions allow you to perform NULL logical comparisons. You cannot specify NULL in an actual [equality \(or non-equality\) condition comparison](#).

SQL Function	Comparison Test	Return Value
NVL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
IFNULL(ex1,ex2) [two-argument form]	ex1 = NULL	True returns ex2 False returns NULL
IFNULL(ex1,ex2,ex3) [three-argument form]	ex1 = NULL	True returns ex2 False returns ex3
{fn IFNULL(ex1,ex2)}	ex1 = NULL	True returns ex2 False returns ex1
ISNULL(ex1,ex2)	ex1 = NULL	True returns ex2 False returns ex1
NULLIF(ex1,ex2)	ex1 = ex2	True returns NULL False returns ex1
COALESCE(ex1,ex2,...)	ex = NULL for each argument	True tests next ex argument. If all ex arguments are True (NULL), returns NULL. False returns ex

Examples

This following example returns the *replace-expression* (99) because the *check-expression* is NULL:

```
SELECT NVL(NULL,99) AS NullTest
```

This following example returns the *check-expression* (33) because *check-expression* is not NULL:

```
SELECT NVL(33,99) AS NullTest
```

The following Dynamic SQL example returns the string 'No Preference' if FavoriteColors is NULL; otherwise, it returns the value of FavoriteColors:

```
ZNSPACE "SAMPLES"
SET myquery=3
SET myquery(1)="SELECT Name,"
SET myquery(2)="NVL(FavoriteColors,'No Preference') AS ColorChoice "
SET myquery(3)="FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

See Also

- [CASE](#) command
- [COALESCE](#) function

- [IFNULL](#) function
- [ISNULL](#) function
- [NULLIF](#) function

%OBJECT

A scalar function that opens a stream object and returns the corresponding `oref`.

```
%OBJECT(stream)
```

Arguments

<code>stream</code>	An expression that is the name of a stream field.
---------------------	---

Description

%OBJECT is used to open a stream object and return the `oref` (object reference) of the [stream field](#).

A **SELECT** on a stream field returns the fully formed `oid` (object ID) value of the stream field. A **SELECT %OBJECT** on a stream field returns the `oref` (object reference) of the stream field. This is shown in the following example, in which `Notes` and `Picture` are both stream fields:

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT TOP 3 Title,Notes,%OBJECT(Picture) AS Photo FROM Sample.Employee"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
    WRITE "String field: ",rset.Title,!
    WRITE "Stream field oid: ",rset.Notes,!
    WRITE "Stream field oref: ",rset.Photo,!!
}
WRITE !,"End of data"
```

If `stream` is not a stream field, **%OBJECT** generates an `SQLCODE -128` error.

%OBJECT can be used as an argument to the following functions:

- `CHARACTER_LENGTH(%OBJECT(streamfield))`, `CHAR_LENGTH(%OBJECT(streamfield))`, or `DATALength(%OBJECT(streamfield))`.
- `SUBSTRING(%OBJECT(streamfield),start,length)`.

You can perform the same operation by issuing a **SELECT** on a stream field, then opening the stream `oid` by calling the `$Stream.Object.%Open()` class method, which generates an `oref` from the `oid`:

```
SET oref = ##class(%Stream.Object).%Open(oid)
```

For information on `orefs`, see “[OREF Basics](#)” in *Using Caché Objects*. For information on `oids`, see “[Identifiers for Saved Objects: ID and OID](#)” in the same book.

See Also

- [SELECT](#)
- [Introduction to the Default SQL Projection](#) in the “Introduction to Persistent Objects” chapter of *Using Caché Objects*
- [Using Streams with SQL](#) in the “Streams” chapter of *Using Caché Objects*
- [Storing and Using BLOBs and CLOBs](#) chapter of *Using Caché SQL*

%ODBCIN

A format-transformation function that returns an expression in Logical format.

```
%ODBCIN(expression)
%ODBCIN expression
```

Arguments

<i>expression</i>	The expression to be converted.
-------------------	---------------------------------

Description

%ODBCIN returns *expression* in the Logical format after passing the value through the field or data type's `OdbcToLogical` method. The Logical format is the in-memory format of data (the format upon which operations are performed).

%ODBCIN is a Caché SQL extension.

For further details on display format options, refer to “[Data Display Options](#)” in the “Caché SQL Basics” chapter of *Using Caché SQL*.

Examples

The following example shows the default display format, the **%ODBCIN**, and the **%ODBCOUT** formats for the same field.

```
SELECT FavoriteColors,%ODBCIN(FavoriteColors) AS InVal,
%ODBCOUT(FavoriteColors) AS OutVal
FROM Sample.Person
```

The following example uses **%ODBCIN** in the WHERE clause:

```
SELECT Name,DOB,%ODBCOUT(DOB) AS Birthdate
FROM Sample.Person
WHERE DOB BETWEEN %ODBCIN('2000-01-01') AND %ODBCIN('2010-01-01')
```

See Also

[%EXTERNAL](#) [%INTERNAL](#) [%ODBCOUT](#)

%ODBCOUT

A format-transformation function that returns an expression in ODBC format.

```
%ODBCOUT(expression)
```

```
%ODBCOUT expression
```

Arguments

<i>expression</i>	The expression to be converted. A field name, an expression containing a field name, or a function that returns a value in a convertible data type, such as DATE or %List. Cannot be a stream field.
-------------------	--

Description

%ODBCOUT returns *expression* in the ODBC format after passing the value through the field or data type's LogicalToOdbc method. The ODBC format is the format in which data can be presented via ODBC. This format is used when data is exposed to ODBC/SQL. The available formats correspond to those defined by ODBC.

%ODBCOUT is commonly used on a **SELECT** list *select-item*. It can be used in a **WHERE** clause, but this use is discouraged because using **%ODBCOUT** prevents the use of indexes on the specified field.

Applying **%ODBCOUT** changes the column header name to a value such as "Expression_1"; it is therefore usually desirable to specify a column name alias, as shown in the examples below.

Whether **%ODBCOUT** converts a date depends on the data type returned by the date field or function. **%ODBCOUT** converts [CURDATE](#), [CURRENT_DATE](#), [CURTIME](#), and [CURRENT_TIME](#) values. It does not convert [CURRENT_TIMESTAMP](#), [GETDATE](#), [GETUTCDATE](#), [NOW](#), and [\\$HOROLOG](#) values.

%ODBCOUT is a Caché SQL extension.

For further details on display format options, refer to "[Data Display Options](#)" in the "Caché SQL Basics" chapter of *Using Caché SQL*.

Examples

The following example shows the default display format, the **%ODBCIN**, and the **%ODBCOUT** formats for the same field.

```
SELECT FavoriteColors,%ODBCIN(FavoriteColors) AS InVal,
%ODBCOUT(FavoriteColors) AS OutVal
FROM Sample.Person
```

See Also

- [%EXTERNAL](#), [%INTERNAL](#), [%ODBCIN](#)
- SQL concepts: [Data Types](#), [Date and Time Constructs](#)

%OID

A scalar function that returns OID of an ID field.

```
%OID(id_field)
```

Arguments

<i>id_field</i>	The field name of an ID field, or a reference field.
-----------------	--

Description

%OID takes a field name and returns the fully formed OID (object ID) for the object. The field must be either an ID field or a reference field (a foreign key field). Specifying any other type of field in *id_field* generates an SQLCODE -1 error.

Examples

The following example shows %OID used with a reference field:

```
SELECT Name, Spouse, %OID(Spouse)
FROM Sample.Person
WHERE Spouse IS NOT NULL
```

The following Embedded SQL example shows %OID used with a reference field:

```
&sql(SELECT Name, Spouse, %OID(Spouse)
      INTO :n,:s,:soid
      FROM Sample.Person)
WRITE !,"Name is:",n
WRITE !,"Spouse name is:",s
WRITE !,"Spouse OID is:",soid
```

See Also

- [SELECT](#)
- [%OBJECT](#)

PI

A scalar numeric function that returns the constant value of pi.

```
{fn PI()}  
{fn PI}
```

Description

PI takes no arguments. It returns the mathematical constant pi as data type **DECIMAL** with a precision of 19 and a scale of 18.

PI can only be invoked using ODBC scalar function (curly brace) syntax. Note that the argument parentheses are optional.

Examples

The following examples both return the value of pi:

```
SELECT {fn PI()} AS ExactPi
```

```
SELECT {fn PI} AS ExactPi
```

returns 3.141592653589793238.

See Also

- SQL functions: [ROUND](#)
- ObjectScript special variable: [\\$ZPI](#)

\$PIECE

A string function that returns a substring identified by a delimiter.

```
$PIECE(plist,delimiter[,from[,to]])
```

Arguments

<i>plist</i>	The target string from which a substring is to be returned.
<i>delimiter</i>	A delimiter used to identify substrings.
<i>from</i>	<i>Optional</i> — An integer that specifies the substring, or beginning of a range of substrings, to return from the target string. Substrings are separated by a <i>delimiter</i> , and counted from 1. If omitted, the first substring is returned.
<i>to</i>	<i>Optional</i> — An integer that specifies the ending substring for a range of substrings to return from the target string. Must be used with <i>from</i> .

Description

\$PIECE returns the specified substring (piece) from *plist*. The substring returned depends on the arguments used:

- **\$PIECE**(*plist,delimiter*) returns the first substring in *plist*. If *delimiter* occurs in *plist*, this is the substring that precedes the first occurrence of *delimiter*. If *delimiter* does not occur in *plist*, the returned substring is *plist*.
- **\$PIECE**(*plist,delimiter,from*) returns the substring which is the *n*th piece of *plist*, where the integer *n* is specified by the *from* argument, and pieces are separated by a *delimiter*. The delimiter is not returned.
- **\$PIECE**(*plist,delimiter,from,to*) returns a range of substrings including the substring specified in *from* through the substring specified in *to*. This four-argument form of **\$PIECE** returns a string that includes any intermediate occurrences of *delimiter* that occur between the *from* and *to* substrings. If *to* is greater than the number of substrings, the returned substring includes all substrings to the end of the *plist* string.

Arguments

plist

The target string from which the substring is to be returned. It can be a string literal, a variable name, or any valid expression that evaluates to a string.

A target string usually contains instances of a character (or character string) which are used as delimiters. This character or string cannot also be used as a data value within *plist*.

If you specify the null string (NULL) as the target string, **\$PIECE** returns <null>, the null string.

delimiter

The search string to be used to delimit substrings within *plist*. It can be a numeric or string literal (enclosed in quotation marks), the name of a variable, or an expression that evaluates to a string.

Commonly, a delimiter is a designated character which is never used within string data, but is set aside solely for use as a delimiter separating substrings. A delimiter can also be a multi-character search string, the individual characters of which can be used within string data.

If you specify the null string (NULL) as the delimiter, **\$PIECE** returns <null>, the null string.

from

The number of a substring within *plist*, counting from 1. It must be a positive integer, the name of an integer variable, or an expression that evaluates to a positive integer. Substrings are separated by delimiters.

- If the *from* argument is omitted or set to 1, **\$PIECE** returns the first substring of *plist*. If *plist* does not contain the specified delimiter, a *from* value of 1 returns *plist*.
- If the *from* argument identifies by count the last substring in *plist*, this substring is returned, regardless of whether it is followed by a delimiter.
- If the value of *from* is NULL, the empty string, zero, or a negative number, and no *to* argument is specified, **\$PIECE** returns a null string. However, if a *to* argument is specified, **\$PIECE** treats these *from* values the same as *from*=1.
- If the value of *from* is greater than the number of substrings in *plist*, **\$PIECE** returns a null string.

If the *from* argument is used with the *to* argument, it identifies the start of a range of substrings to be returned as a string, and should be less than the value of *to*.

to

The number of the substring within *plist* that ends the range initiated by the *from* argument. The returned string includes both the *from* and *to* substrings, as well as any intermediate substrings and the delimiters separating them. The *to* argument must be a positive integer, the name of an integer variable, or an expression that evaluates to a positive integer. The *to* argument must be used with *from* and should be greater than the value of *from*.

- If *from* is less than *to*, **\$PIECE** returns a string consisting of all of the delimited substrings within this range, including the *from* and *to* substrings. This returned string contains the substrings and the delimiters within this range.
- If *to* is greater than the number of delimited substrings, the returned string contains all the string data (substrings and delimiters) beginning with the *from* substring and continuing to the end of the *plist* string.
- If *from* is equal to *to*, the *from* substring is returned.
- If *from* is greater than *to*, **\$PIECE** returns a null string.
- If *to* is the null string (NULL), **\$PIECE** returns a null string.

Examples

The following example returns 'Red', the first substring as identified by the "," delimiter:

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',',',1)
```

The following example returns 'Blue', the third substring as identified by the "," delimiters:

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',',',3)
```

The following example returns 'Blue,Yellow,Orange', the third through fifth elements in *colorlist*, as delimited by ",":

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',',',3,5)
```

The following **\$PIECE** functions both return '123', showing that the two-argument form is equivalent to the three-argument form when *from* is 1:

```
SELECT $PIECE('123#456#789','#') AS TwoArg
```

```
SELECT $PIECE('123#456#789','#',1) AS ThreeArg
```

The following example uses the multi-character delimiter string '#-' to return the third substring '789'. Here, the component characters of the delimiter string, '#' and '-', can be used as data values; only the specified sequence of characters (#-) is set aside:

```
SELECT $PIECE('1#2-3#-#45##6#-#789', '#-', 3)
```

The following example returns 'MAR;APR;MAY'. These comprise the third through the fifth substrings, as identified by the ';' *delimiter*:

```
SELECT $PIECE('JAN;FEB;MAR;APR;MAY;JUN', ';', 3, 5)
```

The following example uses **\$PIECE** to extract the surname from employee names and vendor contact names, and then perform a JOIN which return instances where an employee has the same surname as a vendor contact:

```
SELECT E.Name, V.Contact
FROM Sample.Employee AS E INNER JOIN Sample.Vendor AS V
ON $PIECE(E.Name, ',')=$PIECE(V.Contact, ',')
```

Notes

Using \$PIECE to Unpack Data Values

\$PIECE is typically used to "unpack" data values that contain multiple fields delimited by a separator character. Typical delimiter characters include the slash (/), the comma (,), the space (), and the semicolon (;). The following sample values are good candidates for use with **\$PIECE**:

```
'John Jones/29 River St./Boston MA, 02095'
'Mumps;Measles;Chicken Pox;Diphtheria'
'45.23,52.76,89.05,48.27'
```

\$PIECE and \$LENGTH

The two-argument form of **\$LENGTH** returns the number of substrings in a string, based on a delimiter. Use **\$LENGTH** to determine the number of substrings in a string, and then use **\$PIECE** to extract individual substrings.

\$PIECE and \$LIST

The data storage techniques used by **\$PIECE** and the **\$LIST** functions are incompatible and should not be combined. For example, attempted to use **\$PIECE** on a list created using **\$LISTBUILD** yields unpredictable results and should be avoided. This is true for both SQL functions and the corresponding ObjectScript functions.

The **\$LIST** functions specify substrings without using a designated delimiter. If setting aside a delimiter character or character sequence is not appropriate to the type of data (for example, bitstring data), you should use the **\$LISTBUILD** and **\$LIST** SQL functions to store and retrieve substrings.

Null Values

\$PIECE does not distinguish between a delimited substring with a null string value (NULL), and a nonexistent substring. Both return <null>, the null string value. For example, the following examples both return the null string for a *from* value of 7:

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black', ',', 7)
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black', ',', 7)
```

In the first case, there is no seventh substring; a null string is returned. In the second case there is a seventh substring, as indicated by the delimiter at the end of the *plist* string; the value of this seventh substring is the null string.

The following example shows null values within a *plist*. It extracts substrings 3. This substring exists, but contains a null string:

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black', ',', 3)
```

The following examples also returns a null string, because the specified substrings do not exist:

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black', ',', 0)
```

```
SELECT $PIECE('Red,Green,,Blue,Yellow,Orange,Black',' ',8,20)
```

In the following example, the **\$PIECE** function returns the entire *plist* string, because there are no occurrences of *delimiter* in the *plist* string:

```
SELECT $PIECE('Red,Green,Blue,Yellow,Orange,Black',' #')
```

Nested \$PIECE Operations

To perform complex extractions, you can nest **\$PIECE** references within each other. The inner **\$PIECE** returns a substring that is operated on by the outer **\$PIECE**. Each **\$PIECE** uses its own delimiter. For example, the following returns the state abbreviation 'MA':

```
SELECT $PIECE($PIECE('John Jones/29 River St./Boston MA 02095','/',3),' ',2)
```

The following is another example of nested **\$PIECE** operations, using a hierarchy of delimiters. First, the inner **\$PIECE** uses the caret (^) delimiter to find the second piece, 'A,B,C', of the string. Then the outer **\$PIECE** uses the comma (,) delimiter to return the first and second pieces ('A,B') of the substring 'A,B,C':

```
SELECT $PIECE($PIECE('1,2,3^A,B,C^@#!','^',2),' ',1,2)
```

See Also

- SQL functions: [\\$EXTRACT](#) [\\$FIND](#) [\\$LENGTH](#) [\\$LIST](#)
- ObjectScript functions: [\\$EXTRACT](#) [\\$FIND](#) [\\$LENGTH](#) [\\$LIST](#) [\\$PIECE](#)

%PLUS

A collation function that converts numbers to canonical collation format.

```
%PLUS(expression)
%PLUS expression
```

Arguments

<i>expression</i>	An expression, which can be the name of a column, a number or a string literal, an arithmetic expression, or the result of another function, where the underlying data type can be represented as any character type.
-------------------	---

Description

%PLUS converts numbers or numeric strings to canonical form, then returns these *expression* values in numeric collation sequence.

A number can contain leading and trailing zeros, multiple leading plus and minus signs, a single decimal point indicator (.), and the E exponent indicator. In canonical form, all arithmetic operations are performed, exponents are expanded, signs are resolved to either a single leading minus sign or no sign, and leading and trailing zeros are stripped.

A numeric literal can be specified with or without enclosing string delimiters. If a string contains non-numeric characters, **%PLUS** truncates the number at the first non-numeric character, and returns the numeric part in canonical form. A non-numeric string (any string that begins with a non-numeric character) is returned as 0. **%PLUS** also returns NULLs as 0.

%PLUS is a Caché SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

```
WRITE $SYSTEM.Util.Collation("++007.500",3)
```

Compare **%PLUS** to **%MVR** collation, which sorts a string based on the numeric substrings within the string.

Examples

The following examples uses **%PLUS** to return Home_Street addresses in numeric order:

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %PLUS(Home_Street)
```

Note that the above example orders the integer part of the street address in ascending numerical order. Compare this with the following **ORDER BY** example, which orders records by street addresses in collation sequence:

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY Home_Street
```

See Also

- [%EXACT](#) collation function
- [%MINUS](#) collation function
- [%MVR](#) collation function
- [Collation](#) chapter in *Using Caché SQL*

POSITION

A string function that returns the position of a substring within a string.

```
POSITION(substring IN string)
```

Arguments

<i>substring</i>	The substring to search for. It can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
<i>string</i>	The string expression within which to search for <i>substring</i> .

Description

POSITION returns the position of the first location of *substring* within *string*. The position is returned as an integer. If *substring* is not found, 0 (zero) is returned. **POSITION** returns NULL if passed a NULL value for either argument.

POSITION is case-sensitive. Use one of the case-conversion functions to locate both uppercase and lowercase instances of a letter or character string.

POSITION, INSTR, CHARINDEX, and \$FIND

POSITION, **INSTR**, **CHARINDEX**, and **\$FIND** all search a string for a specified substring and return an integer position corresponding to the first match. **CHARINDEX**, **POSITION**, and **INSTR** return the integer position of the first character of the matching substring. **\$FIND** returns the integer position of the first character after the end of the matching substring. **CHARINDEX**, **\$FIND**, and **INSTR** support specifying a starting point for substring search. **INSTR** also support specifying the substring occurrence from that starting point.

The following example demonstrates these four functions, specifying all optional arguments. Note that the positions of *string* and *substring* differ in these functions:

```
SELECT POSITION('br' IN 'The broken brown briefcase') AS Position,
       CHARINDEX('br','The broken brown briefcase',6) AS Charindex,
       $FIND('The broken brown briefcase','br',6) AS Find,
       INSTR('The broken brown briefcase','br',6,2) AS Inst
```

For a list of functions that search for a substring, refer to [String Manipulation](#).

Examples

The following example returns 11, because “b” is the 11th character in the string:

```
SELECT POSITION('b' IN 'The quick brown fox') AS PosInt
```

The following example returns the length of the last name (surname) for each name in the Sample.Person table. It locates the comma used to separate the last name from the rest of the name field, then subtracts 1 from that position:

```
SELECT Name,
       POSITION(',') IN Name)-1 AS LNameLen
FROM Sample.Person
```

The following example returns the position of the first instance of the letter “B” in each name in the Sample.Person table. Because **POSITION** is case-sensitive, the **%SQLUPPER** function is used to convert all name values to uppercase before performing the search. Because **%SQLUPPER** adds a blank space at the beginning of a string, this example subtracts 1 to get the actual letter position. Searches that do not locate the specified string return zero (0); in this example, because of the subtraction of 1, the value displayed for these searches is -1:

```
SELECT Name,  
POSITION('B' IN %SQLUPPER(Name))-1 AS BPos  
FROM Sample.Person
```

See Also

- [CHARINDEX](#) function
- [\\$FIND](#) function
- [INSTR](#) function
- [String Manipulation](#)

POWER

A numeric function that returns the value of a given expression raised to the specified power.

```
POWER(numeric-expression,power)
{fn POWER(numeric-expression,power)}
```

Arguments

<i>numeric-expression</i>	The base number. Can be a positive or negative integer or fractional number.
<i>power</i>	The exponent, which is the power to which to raise <i>numeric-expression</i> . Can be a positive or negative integer or fractional number.

Description

POWER calculates one number raised to the power of another. It returns a value of data type [DECIMAL](#) with a precision of 36 and a scale of 18.

Note that **POWER** can be invoked as an ODBC scalar function (with the curly brace syntax) or as an SQL general scalar function.

POWER interprets a non-numeric string as 0 for either argument. For further details, refer to [Strings as Numbers](#). **POWER** returns NULL if passed a NULL value for either argument.

All combinations of *numeric-expression* and *power* are valid except:

- `POWER(0, -m)`: a 0 *numeric-expression* and a negative *power* results in an SQLCODE -400 error.
- `POWER(-n, .m)`: a negative *numeric-expression* and a fractional *power* results in an SQLCODE -400 error.

Examples

The following example raises 5 to the 3rd power:

```
SELECT POWER(5,3) AS Cubed
```

returns 125.

The following embedded SQL example returns the first 16 powers of 2:

```
SET a=1
WHILE a<17 {
&sql(SELECT {fn POWER(2,:a)}
INTO :b)
IF SQLCODE!=0 {
WRITE !,"Error code ",SQLCODE
QUIT }
ELSE {
WRITE !,"2 to the ",a," = ",b
SET a=a+1 }
}
```

See Also

- SQL functions: [EXP](#) [LOG10](#) [SQRT](#) [SQUARE](#)
- ObjectScript function: [\\$ZPOWER](#)
- ObjectScript [Exponentiation Operator \(**\)](#)

QUARTER

A date function that returns the quarter of the year as an integer for a date expression.

```
{fn QUARTER(date-expression)}
```

Arguments

<i>date-expression</i>	An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

QUARTER returns an integer from 1 to 4. The quarter is calculated for a Caché date integer, a [\\$HOROLOG](#) or [\\$ZTIMESTAMP](#) value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted. The *date-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

The time periods for the four quarters are as follows:

Quarter	Period (inclusive)
1	January 1 to March 31 (90 or 91 days)
2	April 1 to June 30 (91 days)
3	July 1 to September 30 (92 days)
4	October 1 to December 31 (92 days)

QUARTER evaluates only the month portion of a datetime string. **QUARTER** does not perform value or range checking for user-supplied values. Invalid month values are returned as follows: month=0 returns 1; month > 12 returns 4.

The same quarter information can be returned by using the [DATEPART](#) or [DATENAME](#) function. **DATEPART** and **DATENAME** performs value and range checking on the full date string. You can use the [DATEADD](#) or [TIMESTAMPADD](#) function to increment a date by a specified number of quarters.

This function can also be invoked from ObjectScript using the **QUARTER()** method call:

```
$SYSTEM.SQL.QUARTER(date-expression)
```

Examples

The following examples both return the number 1 because the date (February 25) is in the first quarter of the year:

```
SELECT {fn QUARTER('2004-02-25')} AS Q_Given
```

```
SELECT {fn QUARTER(59590)} AS Q_Given
```

The following examples all return the current quarter:

```
SELECT {fn QUARTER({fn NOW()})} AS Q_Now,
       {fn QUARTER(CURRENT_DATE)} AS Q_CurrD,
       {fn QUARTER(CURRENT_TIMESTAMP)} AS Q_CurrTstamp,
       {fn QUARTER($ZTIMESTAMP)} AS Q_ZTstamp,
       {fn QUARTER($HOROLOG)} AS Q_Horolog
```

See Also

- SQL functions: [DATENAME](#), [DATEPART](#), [DATEADD](#), [MONTH](#), [TO_DATE](#)
- ObjectScript function: [\\$ZDATE](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

RADIANS

A numeric function that converts degrees to radians.

```
RADIANS(numeric-expression)
```

Arguments

<i>numeric-expression</i>	The measure of an angle in degrees. An expression that resolves to a numeric value.
---------------------------	---

Description

RADIANS takes an angle measurement in degrees and returns the corresponding angle measurement in radians. **RADIANS** returns NULL if passed a NULL value.

The returned data type is NUMERIC unless *numeric-expression* is data type DOUBLE, in which case the returned data type is DOUBLE. The default precision is 36. The default scale is 18.

You can use the [DEGREES](#) function to convert radians to degrees.

Example

The following Embedded SQL example returns the radians equivalents corresponding to the degree values from 0 through 365 in 30-degree increments:

```
SET a=0
WHILE a<366 {
&sql(SELECT RADIANS(:a) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"degrees ",a," = radians ",b
  SET a=a+30 }
}
```

See Also

- SQL functions: [CONVERT DEGREES TO_NUMBER](#)

REPEAT

A string function that repeats a string a specified number of times.

```
REPEAT(expression,repeat-count)
{fn REPEAT(expression,repeat-count)}
```

Arguments

<i>expression</i>	The string expression to be repeated.
<i>repeat-count</i>	The number of times to repeat, expressed as an integer.

Description

REPEAT returns a string of *repeat-count* instances of *expression*, concatenated together.

If *expression* is NULL, **REPEAT** returns NULL. If *expression* is the empty string, **REPEAT** returns an empty string.

If *repeat-count* is a fractional number, only the integer part is used. If *repeat-count* is 0, **REPEAT** returns an empty string.

If *repeat-count* is a negative number, NULL, or a non-numeric string, **REPEAT** returns NULL.

Examples

The following examples show the two forms of **REPEAT**. Both examples return the string 'BANGBANGBANG':

```
SELECT REPEAT('BANG',3) AS Tripled
```

```
SELECT {fn REPEAT('BANG',3)} AS Tripled
```

See Also

- [REPLICATE](#)

REPLACE

A string function that replaces a substring within a string.

```
REPLACE(string,oldsubstring,newsubstring)
```

Arguments

<i>string</i>	A string expression that is the target for the substring search.
<i>oldsubstring</i>	The substring to match within <i>string</i> .
<i>newsubstring</i>	The substring used to replace <i>oldsubstring</i> .

Description

REPLACE searches a string for a substring and replaces all matches. Matching is case-sensitive. If a match is found, it replaces every instance of *oldsubstring* with *newsubstring*. The replacement substring may be longer or shorter than the substring it replaces. If the substring cannot be found, **REPLACE** returns the original *string* unchanged.

The value returned by **REPLACE** is always of data type VARCHAR, regardless of the data type of *string*. This allows for replacement operations such as `REPLACE(12.3, '.', '_')`.

The empty string is a string value. You can, therefore, use the empty string for any argument value. However, note that the ObjectScript empty string is passed to Caché SQL as NULL.

NULL is not a data value in Caché SQL. For this reason, specifying NULL for any of the **REPLACE** arguments returns NULL, regardless of whether or not a match occurs.

This function provides compatibility with Transact-SQL implementations.

REPLACE, STUFF, and \$TRANSLATE

Both **REPLACE** and **STUFF** perform substring replacement. **REPLACE** searches for a substring by data value. **STUFF** searches for a substring by string position and length.

REPLACE performs a single string-for-string matching and replacement. **\$TRANSLATE** performs character-for-character matching and replacement; it can replace all instances of one or more specified single characters with corresponding specified replacement single characters. It can also remove all instances of one or more specified single characters from a string.

By default, all three functions are case-sensitive and replace all matching instances.

For a list of functions that search for a substring, refer to [String Manipulation](#) in the Concepts section of this manual.

Examples

The following example searches for every instance of the substring 'K' and replaces it with the substring 'P':

```
SELECT REPLACE('KING KONG','K','P')
```

The following embedded SQL example searches for every instance of the substring 'KANSAS' and replaces it with the substring 'NEBRASKA':

```
SET str="KANSAS, ARKANSAS, NEBRASKA"
&sql(SELECT REPLACE(:str,'KANSAS','NEBRASKA') INTO :x)
WRITE !,"SQLCODE=",SQLCODE
WRITE !,"Output string=",x
```

The following example show that **REPLACE** handles the empty string (") just like any other string value:

```
SELECT REPLACE('',' ','Nothing'),
       REPLACE('KING KONG',' ','P'),
       REPLACE('KING KONG','K','')
```

The following example shows that **REPLACE** handles any NULL argument by returning NULL. All of the following **REPLACE** functions return NULL, including the last, in which no match occurs:

```
SELECT REPLACE(NULL,'K','P'),
       REPLACE(NULL,NULL,'P'),
       REPLACE('KING KONG',NULL,'P'),
       REPLACE('KING KONG','K',NULL),
       REPLACE('KING KONG','Z',NULL)
```

The following Embedded SQL example is identical to the previous NULLs example. It shows how the ObjectScript empty string host variable is treated as NULL within SQL:

```
SET a=""
&sql(SELECT
REPLACE(:a,'K','P'),
REPLACE(:a,:a,'P'),
REPLACE('KING KONG',:a,'P'),
REPLACE('KING KONG','K',:a),
REPLACE('KING KONG','Z',:a)
INTO :v,:w,:x,:y,:z)
WRITE !,"SQLCODE=",SQLCODE
WRITE !,"Output string=",v
WRITE !,"Output string=",w
WRITE !,"Output string=",x
WRITE !,"Output string=",y
WRITE !,"Output string=",z
```

See Also

- [CHARINDEX](#) function
- [\\$FIND](#) function
- [STUFF](#) function
- [\\$TRANSLATE](#) function
- [String Manipulation](#)

REPLICATE

A string function that repeats a string a specified number of times.

```
REPLICATE(expression, repeat-count)
```

Arguments

<i>expression</i>	The string expression to be repeated.
<i>repeat-count</i>	The number of times to repeat, expressed as an integer.

Description

Note: The **REPLICATE** function is an alias for the **REPEAT** function. **REPLICATE** is provided for TSQL compatibility. Refer to [REPEAT](#) for further details.

See Also

- [REPEAT](#)

REVERSE

A scalar string function that returns a character string in reverse character order.

```
REVERSE(string-expression)
```

Arguments

<i>string-expression</i>	The string expression to be reversed. The expression can be the name of a column, a string literal, a numeric, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
--------------------------	---

Description

REVERSE returns *string-expression* with its character order reversed. For example, 'Hello World!' is returned as '!dlroW olleH'. This is a simple string-order reversal, with no additional processing.

The string returned is data type VARCHAR, regardless of the data type of the input value. Numbers are converted to canonical form, numeric strings are not converted to canonical form before reversing.

Leading and trailing blanks are unaffected by reversing.

Reversing a NULL value results in a NULL.

Note: Because **REVERSE** always returns a VARCHAR string, some types of data become invalid when reversed:

- A reversed list is no longer a valid list and cannot be converted from storage format to display format.
- A reversed date is no longer a valid date, and cannot be converted from storage format to display format.

Examples

The following example reverses the Name field values. In this case, this results in names sorted by middle initial:

```
SELECT Name, REVERSE(Name) AS RevName
FROM Sample.Person
ORDER BY RevName
```

Note that because Name and RevName are just different representations of the same field, ORDER BY RevName and ORDER BY RevName, Name perform the same ordering.

The following example reverses a number and a numeric string:

```
SELECT REVERSE(+007.10) AS RevNum,
       REVERSE('+007.10') AS RevNumStr
```

The following Embedded SQL example reverses a \$DOUBLE number:

```
SET dnum=$DOUBLE(1.1)
&sql(SELECT REVERSE(:dnum) INTO :drevnum)
WRITE dnum, !
WRITE drevnum, !
```

The following example shows what happens when you reverse a list:

```
SELECT FavoriteColors, REVERSE(FavoriteColors) AS RevColors
FROM Sample.Person
```

The following example shows what happens when you reverse a date:

```
SELECT DOB,%INTERNAL(DOB) AS IntDOB,REVERSE(DOB) AS RevDOB  
FROM Sample.Person
```

See Also

- [CHAR](#)
- [STRING](#)
- [SUBSTRING](#)

RIGHT

A scalar string function that returns a specified number of characters from the end (rightmost position) of a string expression.

```
{fn RIGHT(string-expression,count)}
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
<i>count</i>	An integer that specifies the number of characters to return from the ending (rightmost) position of <i>string-expression</i> .

Description

RIGHT returns *count* number of characters from the end (rightmost position) of *string-expression*. **RIGHT** returns NULL if passed a NULL value for either argument.

RIGHT can only be used as an ODBC scalar function (with the curly brace syntax).

Examples

The following example returns the two rightmost characters of each name in the Sample.Person table:

```
SELECT Name,{fn RIGHT(Name,2)}AS MiddleInitial
FROM Sample.Person
```

The following embedded SQL example shows how **RIGHT** handles a *count* that is longer than the string itself:

```
&sql(SELECT Name,{fn RIGHT(Name,40)}
INTO :a,:b
FROM Sample.Person)
IF SQLCODE'=0 {
WRITE !,"Error code ",SQLCODE }
ELSE {
WRITE !,a,"=original",!,b,"=RIGHT 40" }
```

No padding is performed.

See Also

[LEFT](#) [LTRIM](#) [RTRIM](#)

ROUND

A numeric function that rounds or truncates a number at a specified number of digits.

```
ROUND(numeric-expr, scale[, flag])
{fn ROUND(numeric-expr, scale[, flag])}
```

Arguments

<i>numeric-expr</i>	The number to be rounded. A numeric expression.
<i>scale</i>	An expression that evaluates to an integer that specifies the number of places to round to, counting from the decimal point. Can be zero, a positive integer, or a negative integer. If <i>scale</i> is a fractional number, Caché rounds it to the nearest integer.
<i>flag</i>	<i>Optional</i> — A boolean flag that specifies whether to round or truncate the <i>numeric-expr</i> : 0=round, 1=truncate. The default is 0.

ROUND returns the same [data type](#) as *numeric-expr*. See [\\$DOUBLE Numbers](#) below.

Description

This function can be used to either round or truncate a number to the specified number of decimal digits.

ROUND rounds or truncates *numeric-expr* to *scale* places, counting from the decimal point. When rounding, the number 5 is always rounded up. Trailing zeroes are removed after a **ROUND** round or truncate operation. Leading zeros are not returned.

- If *scale* is a positive number, rounding is performed at that number of digits to the right of the decimal point. If *scale* is equal to or larger than the number of decimal digits, no rounding or zero filling occurs.
- If *scale* is zero, rounding is to the closest whole integer. In other words, rounding is performed at zero digits to the right of the decimal point; all decimal digits and the decimal point itself are removed.
- If *scale* is a negative number, rounding is performed at that number of digits to the left of the decimal point. If *scale* is equal to or larger than the number of integer digits in the rounded result, zero is returned.
- If *numeric-expr* is zero (however expressed: 00.00, -0, etc.) **ROUND** returns 0 (zero) with no decimal digits, regardless of the *scale* value.
- If *numeric-expr* or *scale* is NULL, **ROUND** returns NULL.

Note that the **ROUND** return value is always normalized, removing trailing zeros.

ROUND, TRUNCATE, and \$JUSTIFY

ROUND and **TRUNCATE** are numeric functions that perform similar operations; they both can be used to decrease the number of significant decimal or integer digits of a number. **ROUND** allows you to specify either rounding (the default), or truncation; **TRUNCATE** does not perform rounding. **ROUND** returns the same data type as *numeric-expr*; **TRUNCATE** returns *numeric-expr* as data type NUMERIC, unless *numeric-expr* is data type DOUBLE, in which case it returns data type DOUBLE.

ROUND rounds (or truncates) to a specified number of fractional digits, but its return value is always normalized, removing trailing zeros. For example, `ROUND(10.004, 2)` returns 10, not 10.00.

TRUNCATE truncates to a specified number of fractional digits. If the truncation results in trailing zeros, these trailing zeros are preserved. However, if *scale* is larger than the number of fractional decimal digits in the canonical form of *numeric-expr*, **TRUNCATE** does not zero-pad.

Use **\$JUSTIFY** when rounding to a fixed number of fractional digits is important — for example, when representing monetary amounts. **\$JUSTIFY** returns the specified number of trailing zeros following the rounding operation. When the number of digits to round is larger than the number of fractional digits, **\$JUSTIFY** zero-pads. **\$JUSTIFY** also right-aligns the numbers, so that the DecimalSeparator characters align in a column of numbers. **\$JUSTIFY** does not truncate.

\$DOUBLE Numbers

\$DOUBLE IEEE floating point numbers are encoded using binary notation. Most decimal fractions cannot be exactly represented in this binary notation. When a **\$DOUBLE** value is input to **ROUND** with a *scale* value and the rounding *flag* (*flag*=0, the default), the return value frequently contains more fractional digits than specified in *scale* because the fractional decimal result is not representable in binary, so the return value must be rounded to the nearest representable **\$DOUBLE** value, as shown in the following example:

```
SET x=1234.5678
SET y=$DOUBLE(1234.5678)
&sql(SELECT ROUND(:x,2),ROUND(:y,2) INTO :decnum,:dblnum)
WRITE "Decimal: ",x," rounded ",decnum,!
WRITE "Double: ",y," rounded ",dblnum
```

If you are using **ROUND** to truncate a **\$DOUBLE** value (*flag*=1), the return value for the **\$DOUBLE** is truncated to the number of fractional digits specified by *scale*. The **TRUNCATE** function also truncates a **\$DOUBLE** to the number of fractional digits specified by *scale*.

If you are using **ROUND** to round a **\$DOUBLE** value and wish to return a specific *scale*, you should convert the **\$DOUBLE** value to decimal representation before rounding the result.

ROUND with *flag*=0 (round, the default) returns **\$DOUBLE("INF")** and **\$DOUBLE("NAN")** as the empty string.

ROUND with *flag*=1 (truncate) returns **\$DOUBLE("INF")** and **\$DOUBLE("NAN")** as INF and NAN.

Examples

The following example uses a *scale* of 0 (zero) to round several fractions to integers. It shows that 5 is always rounded up:

```
SELECT ROUND(5.99,0) AS RoundUp,
       ROUND(5.5,0) AS Round5,
       {fn ROUND(5.329,0)} AS Roundoff
```

The following example truncates the same fractional numbers as the previous example:

```
SELECT ROUND(5.99,0,1) AS Trunc1,
       ROUND(5.5,0,1) AS Trunc2,
       {fn ROUND(5.329,0,1)} AS Trunc3
```

The following **ROUND** functions round and truncate a negative fractional number:

```
SELECT ROUND(-0.987,2,0) AS Round1,
       ROUND(-0.987,2,1) AS Trunc1
```

The following example rounds off pi to four decimal digits:

```
SELECT {fn PI()} AS ExactPi, ROUND({fn PI()},4) AS ApproxPi
```

The following example specifies a *scale* larger than the number of decimal digits:

```
SELECT {fn ROUND(654.98700,9)} AS Rounded
```

it returns 654.987 (Caché removed the trailing zeroes before the rounding operation; no rounding or zero padding occurred).

The following example rounds off the value of Salary to the nearest thousand dollars:

```
SELECT Salary,ROUND(Salary, -3) AS PayBracket
FROM Sample.Employee
ORDER BY Salary
```

Note that if `Salary` is less than five hundred dollars, it is rounded to 0 (zero).

In the following example each **ROUND** specifies a negative *scale* as large or larger than the number to be rounded:

```
SELECT {fn ROUND(987,-3)} AS Round1,
       {fn ROUND(487,-3)} AS Round2,
       {fn ROUND(987,-4)} AS Round3,
       {fn ROUND(987,-5)} AS Round4
```

The first **ROUND** function returns 1000, because the rounded result has more digits than the *scale*. The other three **ROUND** functions return 0 (zero).

See Also

- [\\$JUSTIFY](#) function
- [TRUNCATE](#) function
- [CEILING](#) function
- [FLOOR](#) function
- [MOD](#) function
- ObjectScript functions: [\\$DOUBLE](#), [\\$NORMALIZE](#), [\\$NUMBER](#)

RPAD

A string function that returns a string right-padded to a specified length.

```
RPAD(string-expression,length[,padstring])
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, a host variable, or the result of another scalar function. Can be of any data type convertible to a VARCHAR data type. <i>string-expression</i> cannot be a stream.
<i>length</i>	An integer specifying the number of characters in the returned string.
<i>padstring</i>	<i>Optional</i> — A string consisting of a character or a string of characters used to pad the input <i>string-expression</i> . The <i>padstring</i> character or characters are appended to the right of <i>string-expression</i> to supply as many characters as need to create an output string of <i>length</i> characters. <i>padstring</i> may be a string literal, a column, a host variable, or the result of another scalar function. If omitted, the default is a blank space character.

Description

RPAD pads a string expression with trailing pad characters. It returns a copy of the string padded to *length* number of characters. If the string expression is longer than *length* number of characters, the return string is truncated to *length* number of characters.

If *string-expression* is NULL, **RPAD** returns NULL. If *string-expression* is the empty string (") **RPAD** returns a string consisting entirely of pad characters. The returned string is type VARCHAR.

RPAD does not remove leading or trailing blanks; it pads the string including any leading or trailing blanks. To remove leading or trailing blanks before padding a string, use **LTRIM**, **RTRIM**, or **TRIM**.

Examples

The following example right pads column values with ^ characters (when needed) to return strings of length 16. Note that some Name strings are right padded, some Name strings are right truncated to return strings of length 16.

```
SELECT TOP 15 Name,RPAD(Name,16,'^') AS Name16
FROM Sample.Person
```

The following example right pads column values with the ^=^ pad string (when needed) to return strings of length 20. Note that the pad name string is repeated as many times as needed, and that some return strings contain partial pad strings:

```
SELECT TOP 15 Name,RPAD(Name,20,'^=^') AS Name20
FROM Sample.Person
```

See Also

[LPAD](#) [LTRIM](#) [RTRIM](#) [TRIM](#)

RTRIM

A string function that returns a string with the trailing blanks removed.

```
RTRIM(string-expression)
{fn RTRIM(string-expression)}
```

Arguments

<i>string-expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
--------------------------	--

Description

RTRIM removes the trailing blanks from a string expression, and returns the string as type VARCHAR. If *string-expression* is NULL, **RTRIM** returns NULL. If *string-expression* is a string consisting entirely of blank spaces, **RTRIM** returns the empty string ("").

RTRIM leave leading blanks; to remove leading blanks, use **LTRIM**. To remove leading and/or trailing characters of any type, use **TRIM**. To pad a string with trailing blanks or other characters, use **RPAD**. To create a string of blanks, use **SPACE**.

Note that **RTRIM** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Example

The following Embedded SQL example removes the five trailing blanks from the string. It leaves the five leading blanks:

```
SET a="      Test string with 5 leading and 5 trailing spaces.      "
&sql(SELECT {fn RTRIM(:a)} INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Before RTRIM",!,"start:",a,"end"
  WRITE !,"After RTRIM",!,"start:",b,"end" }
```

Returns:

```
Before RTRIM
start:      Test string with 5 leading and 5 trailing spaces.      :end
After RTRIM
start:      Test string with 5 leading and 5 trailing spaces.:end
```

See Also

[LTRIM](#) [TRIM](#) [RPAD](#) [SPACE](#)

SEARCH_INDEX

A function that returns a set of values from the index's Find() method.

```
SEARCH_INDEX([ [schema-name.]table-name.]index-name[ ,findparam[ ,... ]])
```

Arguments

<i>table-name</i>	<i>Optional</i> — The name of an existing table for which <i>index-name</i> is defined. Cannot be a view. The table's <i>schema_name</i> is optional. If omitted, all tables specified in the FROM clause are searched.
<i>index-name</i>	The index to be searched. The SqlName of the index map of an existing index.
<i>findparam</i>	<i>Optional</i> — An parameter or a comma-separated list of parameters to be passed to the index's Find() method.

Description

SEARCH_INDEX invokes the *index-name***Find()** method and returns a set of values. You can optionally pass parameters to this **Find()** method. For example, `SEARCH_INDEX(Sample.Person.NameIDX)` invokes the `Sample.Person.NameIDXFind()` method.

SEARCH_INDEX can be used with the [%FIND](#) predicate in a **WHERE** clause to supply the *oref* of an object that provides an abstract representation encapsulating a set of values. These values are commonly row IDs returned by a method called at query run time. **SEARCH_INDEX** invokes the index's **Find()** method to return this *oref*. This usage is shown in the following example:

```
SELECT Name FROM Sample.Person AS P
WHERE P.Name %FIND SEARCH_INDEX(Sample.Person.NameIDX)
```

The index must be found within the tables referenced by the SQL statement. An `SQLCODE -151` error is generated if the specified *index-name* does not exist within the tables used by the SQL statement. An `SQLCODE -152` error is generated if the specified *index-name* is not fully qualified, and is therefore ambiguous (could refer to more than one existing index) within the tables used by the SQL statement.

If the index exists, but it has no corresponding **Find()** method, a runtime `SQLCODE -149` error is generated “SQL Function encountered an error”, the error being `<METHOD DOES NOT EXIST>`.

For further details on the use of **SEARCH_INDEX**, refer to the [iFind Search Tool](#) chapter of *Using iKnow*.

See Also

- [CREATE INDEX](#)
- [%FIND](#) predicate
- [%INSET](#) predicate
- “[Defining and Building Indices](#)” chapter in *Caché SQL Optimization Guide*
- “[Using Indices](#)” in the “[Optimizing Query Performance](#)” chapter in *Caché SQL Optimization Guide*

SECOND

A time function that returns the second for a datetime expression.

```
{fn SECOND(time-expression)}
```

Arguments

<i>time-expression</i>	An expression that is the name of a column, the result of another scalar function, or a string or numeric literal. It must resolve either to a timestamp string or a \$HOROLOGY string, where the underlying data type can be represented as %Time or %TimeStamp.
------------------------	---

Description

SECOND returns an integer from 0 to 59, and may return fractional seconds as well. The seconds are calculated for a [\\$HOROLOGY](#) or [\\$ZTIMESTAMP](#) value, an ODBC format date string (with no time value), or a timestamp.

A *time-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

To change the default time format, use the [SET OPTION](#) command.

You must supply either a timestamp string (yyyy-mm-dd hh:mm:ss) or a **\$HOROLOGY** string. A **\$HOROLOGY** string may be a full datetime string (63274,37279) or only the time integer portion of **\$HOROLOGY** (37279). You cannot supply a time string (hh:mm:ss); this always returns 0, regardless of the actual number of seconds. The *time-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

The seconds (ss) portion should be an integer in the range from 0 through 59. There is, however, no range checking for user-supplied values. Numbers greater than 59, negative numbers, and fractions are returned as specified. Leading zeros are optional on input. Leading and trailing zeros are suppressed on output.

SECOND returns 0 seconds when the seconds portion is '0', '00', or a nonnumeric value. Zero seconds is also returned if an ODBC date with no time expression is supplied, if the seconds portion of the time expression is omitted entirely ('hh', 'hh:mm', 'hh:mm:', or 'hh:.'), or the time expression is invalid.

The same time information can be returned using [DATEPART](#) or [DATENAME](#).

This function can also be invoked from ObjectScript using the **SECOND()** method call:

```
$SYSTEM.SQL.SECOND(time-expression)
```

Fractional Seconds

SECOND returns fractions of a second if supplied in *time-expression*. Trailing zeros are truncated. If no fractional seconds are specified (for example: 38.00) the decimal separator is also truncated.

The standard Caché internal representation of time values ([\\$HOROLOGY](#)) does not support fractional seconds. Timestamps do support fractional seconds.

The following SQL functions support fractional seconds: **SECOND**, **CURRENT_TIMESTAMP**, **DATENAME**, **DATEPART**, and **GETDATE**. **CURTIME**, **CURRENT_TIME**, and **NOW** do not support fractional seconds.

The SQL [SET OPTION](#) statement permits you to set the default precision (number of decimal digits) for fractional seconds.

The ObjectScript [\\$ZTIMESTAMP](#) special variable can be used to represent fractional seconds. The ObjectScript functions [\\$ZDATETIME](#), [\\$ZDATETIMEH](#), [\\$ZTIME](#), and [\\$ZTIMEH](#) support fractional seconds.

Examples

The following examples both return the number 38 because it is the thirty-eighth second of the time expression:

```
SELECT {fn SECOND('2000-02-16 18:45:38')} AS Seconds_Given
```

```
SELECT {fn SECOND(67538)} AS Seconds_Given
```

The following example returns .9 seconds. The leading and trailing zeros are truncated:

```
SELECT {fn SECOND('2000-02-16 18:45:00.9000')} AS Seconds_Given
```

The following example returns 0 seconds because the seconds portion of the datetime string has been omitted:

```
SELECT {fn SECOND('2000-02-16 18:45')} AS Seconds_Given
```

The following example returns 0 seconds because the time expression has been omitted from the datetime string:

```
SELECT {fn SECOND('2000-02-16')} AS Seconds_Given
```

The following examples all return the seconds portion of the current time, in whole seconds:

```
SELECT {fn SECOND(CURRENT_TIME)} AS Sec_CurrentT,
       {fn SECOND({fn CURTIME()})} AS Sec_CurT,
       {fn SECOND({fn NOW()})} AS Sec_Now,
       {fn SECOND($HOROLOG)} AS Sec_Horolog,
       {fn SECOND($ZTIMESTAMP)} AS Sec_ZTS
```

The following example shows that leading zeros are suppressed. The first **SECOND** function returns a length 2, the others return a length of 1. An omitted time is considered to be 0 seconds, which has a length of 1:

```
SELECT LENGTH({fn SECOND('2004-02-05 11:45:22')}),
       LENGTH({fn SECOND('2004-02-15 03:05:06')}),
       LENGTH({fn SECOND('2004-02-15 3:5:6')}),
       LENGTH({fn SECOND('2004-02-15')})
```

The following Embedded SQL example shows that the **SECOND** function recognizes the TimeSeparator character specified for the locale:

```
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeSeparator", ".")
&sql(SELECT {fn SECOND('2000-02-16 18.45.38')} INTO :a)
WRITE "seconds=",a
```

See Also

- SQL concepts: [Data Type](#), [Date and Time Constructs](#)
- SQL functions: [HOUR](#), [MINUTE](#), [CURRENT_TIME](#), [CURTIME](#), [NOW](#), [DATEPART](#), [DATENAME](#)
- ObjectScript function: [\\$ZTIME](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

SIGN

A numeric function that returns the sign of a given numeric expression.

```
SIGN(numeric-expression)
{fn SIGN(numeric-expression)}
```

Arguments

<i>numeric-expression</i>	A number for which the sign is to be returned.
---------------------------	--

Description

SIGN returns the following:

- -1 if *numeric-expression* is less than zero.
- 0 (zero) if *numeric-expression* is zero: 0, +0, or -0.
- 1 if *numeric-expression* is greater than zero.
- NULL if *numeric-expression* is NULL, or if it is a non-numeric string.

SIGN can be used as either an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

SIGN converts *numeric-expression* to [canonical form](#) before determining its value. For example, `SIGN(---+3)` and `SIGN(-3+5)` both return 1, indicating a positive number.

Note: In Caché SQL, two negative signs (hyphens) are the in-line [comment](#) indicator. For this reason, a **SIGN** argument specifying two successive negative signs must be presented as a numeric string enclosed in quotes.

Examples

The following examples shows the effects of **SIGN**:

```
SELECT SIGN(-49) AS PosNeg
```

returns -1.

```
SELECT {fn SIGN(-0.0)} AS PosNeg
```

returns 0.

```
SELECT SIGN(---16.748) AS PosNeg
```

returns 1.

```
SELECT {fn SIGN(NULL)} AS PosNeg
```

returns <null>.

See Also

- [+\(Positive\)](#) and [-\(Negative\)](#) unary operators
- [ABS](#) function
- [ISNUMERIC](#) function

- [%PLUS](#) and [%MINUS](#) collation functions

%SIMILARITY

Returns a number indicating the similarity of a field value to a text.

```
%SIMILARITY( field, document )
```

Arguments

<i>field</i>	A data column whose values are being compared with the <i>document</i> text. Must be of data type %TEXT. Cannot be a view field.
<i>document</i>	An alphabetic string to match with values in <i>field</i> . A <i>document</i> consists of a series of words separated by a delimiter (commonly, the space character).

Description

The %SIMILARITY function returns a numeric value indicating the similarity between each value of *field* and the text specified in *document*. The fractional values returned range from 0 (no similarity at all) to 1 (identical). The returned value is of type NUMERIC with a precision of 19 and a scale of 18.

You can use %SIMILARITY as a *select-item* or in a predicate in a **WHERE** clause. You can use %SIMILARITY to determine order of ranking, for example in an **ORDER BY** clause.

%SIMILARITY can be used on a %Text string or a character stream field.

To use %SIMILARITY on a string, change the %String property to %Text, and set LANGUAGECLASS and MAXLEN [property parameters](#). For example:

```
Property MySentences As %Text(LANGUAGECLASS = "%Text.English",MAXLEN = 1000);
```

Specifying a MAXLEN value (in bytes) is required for %Text properties.

To use %SIMILARITY on a [character stream field](#), the stream field must be defined as type %Stream.GlobalCharacterSearchable. For example:

```
Property MyTextStream As %Stream.GlobalCharacterSearchable(LANGUAGECLASS = "%Text.English");
```

The available languages are English, French, German, Italian, Japanese, Portuguese, and Spanish. See the %Text package class documentation (in %SYS) in the *InterSystems Class Reference* for further details.

If *field* is neither data type %Text nor %Stream.GlobalCharacterSearchable, the system generates an SQLCODE -309 error.

SIMILARITYINDEX

%SIMILARITY has both an indexed and a non-indexed implementation. For both %Text and %Stream.GlobalCharacterSearchable, you can, optionally, set the SIMILARITYINDEX property parameter. If no SIMILARITYINDEX is specified, Caché uses a non-indexed (and much slower) implementation that takes the maximum similarity of any 32k chunk of the document. Since the similarity metric takes document length into account, the similarity calculated in this way is different (and usually larger) than it would be for the document as a whole. Also, since chunks do not overlap, similar terms that appear across the chunk boundary do not contribute as much to similarity as they would for the document as a whole, which acts to reduce the similarity value. In contrast, a %SIMILARITY value that is based on a SIMILARITYINDEX is not chunked, and is therefore based on the document as a whole. For both performance and consistency it is recommended that you should set up a similarity index if you need to use %SIMILARITY on streams.

Note: If text is represented as a stream that is greater than the maximum length of a string and a search on the text uses **%SIMILARITY** on a non-indexed field, the document is broken up into chunks of characters. If non-indexed fields span boundaries between chunks, they may not be properly referenced. To avoid this issue, only use **%SIMILARITY** on indexed fields. For information on the maximum length of a string, see the section “[Support for Long String Operations](#)” in the chapter “[Server Configuration Options](#)” in the *Caché Programming Orientation Guide*.

For further details on SIMILARITYINDEX see the %Library.Text class.

Similarity Analysis

A returned value of equality (1.00000) means that the *field* value and *document* string consist of the same words. Two words are considered identical if they have the same stem form; for example, dog=dogs and jump=jumped=jumping. The words in *field* and *document* may be in a completely different order. By default, word comparison is not case-sensitive.

Note: The similarity of two identical strings may be very slightly less than or very slightly more than exactly 1.

A returned value of highly similar generally means that most or all of the words (or other delimited data items) in *field* are also found in *document*, though not necessarily in the same stem form or order. The *document* text may also contain words not present in *field*. Extra words in *document* that are not present in *field* have less effect on similarity than words missing from *document* that are present in *field*. One or two duplicates in *document* of a word present in *field* generally add to the degree of significance, but large numbers of duplicates in *document* diminish significance. One-letter and two-letter words have less effect on significance than longer words.

%SIMILARITY comparison is governed by the class parameters of the %Text.Text system class, found in the %SYS namespace. These parameters allow you to specify, among other things, whether comparison is to be case-sensitive or not case-sensitive, and the treatment of numbers, punctuation characters, and multi-word phrases.

Caché can use specific language analysis rules, including common word analysis (“noise word” lists) and stemming rules, to determine similarity. The available languages are English, French, German, Italian, Japanese, Portuguese, and Spanish.

For a much more detailed treatment of **%SIMILARITY** and %Text, refer to the %Text package class documentation in the *InterSystems Class Reference*.

iKnow and iFind

The Caché [iKnow text analysis tool](#) and [iFind text search tool](#) also provide similarity analysis. These facilities are entirely separate from %Text classes. They provide a substantially different and significantly more sophisticated form of textual analysis.

Example

The following example returns the top 10 records that match the **%SIMILARITY** string value. Because the most similar matches have the highest similarity value, the ORDER BY clause here is DESC (in descending order):

```
SELECT TOP 10 MySentences FROM Sample.MyTexts ORDER BY %SIMILARITY(MySentences,'the quick brown fox jumped') DESC
```

See Also

- [SELECT](#) statement, [ORDER BY](#) clause, [WHERE](#) clause
- [%CONTAINS](#) operator
- [%CONTAINSTERM](#) operator
- “[Queries Invoking Free-text Search](#)” in the “[Querying the Database](#)” chapter of *Using Caché SQL*.

SIN

A scalar numeric function that returns the sine, in radians, of an angle.

```
{fn SIN(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type FLOAT. This is an angle expressed in radians.
-------------------------	---

Description

SIN takes any numeric value and returns its sine as a floating point number. **SIN** returns NULL if passed a NULL value. **SIN** treats nonnumeric strings as the numeric value 0.

SIN returns a value of data type FLOAT with a precision of 19 and a scale of 18.

SIN can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the [DEGREES](#) function to convert radians to degrees. You can use the [RADIANS](#) function to convert degrees to radians.

Example

The following example shows the effect of **SIN**:

```
SELECT {fn SIN(0.52)} AS Sine
```

returns 0.496880.

See Also

- SQL functions: [ACOS](#) [ASIN](#) [ATAN](#) [COS](#) [COT](#) [TAN](#)
- ObjectScript function: [\\$ZSIN](#)

SPACE

A string function that returns a string of spaces.

```
SPACE(count)
{fn SPACE(count)}
```

Arguments

<i>count</i>	An integer expression specifying the number of blank spaces to return.
--------------	--

Description

SPACE returns a string of blank spaces *count* spaces long. If *count* is a numeric string, a decimal number, or a mixed numeric string, Caché resolves it to its integer portion. If *count* is a negative number or a nonnumeric string, Caché resolves it to 0.

To remove blank spaces from a string, use **LTRIM** (leading blanks) or **RTRIM** (trailing blanks).

Note: The **SPACE** function should not be confused with the **SPACE collation type**. **SPACE** collation appends a single space to a value, forcing it to be evaluated as a string. To establish **SPACE** collation, **CREATE TABLE** provides a **%SPACE** collation keyword, and ObjectScript provides the **Collation()** method of the **%SYSTEM.Util** class.

Examples

The following embedded SQL example returns a string of spaces the length of the *name* field:

```
&sql(SELECT SPACE(LENGTH(name))
INTO :a
FROM Sample.Person)
IF SQLCODE'=0 {
WRITE !,"Error code ",SQLCODE }
ELSE {
WRITE !,"Leave this much space:",a,"for names" }
```

See Also

[LTRIM](#) [RTRIM](#) [TRIM](#)

%SQLSTRING

A collation function that sorts values as strings.

```
%SQLSTRING(expression[,maxlen])
%SQLSTRING expression
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2). <i>expression</i> can be a subquery.
<i>maxlen</i>	<i>Optional</i> — A positive integer, which specifies that the collated value will be truncated to the value of <i>maxlen</i> . Note that <i>maxlen</i> includes the appended leading blank space. You can enclose <i>maxlen</i> with double parentheses to suppress literal substitution : <code>((maxlen))</code> .

Description

%SQLSTRING converts *expression* to format that is sorted as a (case-sensitive) string. **%SQLSTRING** strips trailing whitespace (spaces, tabs, and so on) from the string, then adds one leading blank space to the beginning of the string. This appended blank space forces NULL and numeric values to be collated as strings. Leading and trailing zeros are removed from numbers.

Because **%SQLSTRING** appends a blank space to all values, it collates a **NULL** value as a blank space, with a string length of 1. **%SQLSTRING** collates any value containing only whitespace (spaces, tabs, and so on) as the SQL [empty string](#) ("). When **%SQLSTRING** appends a blank space to an empty (zero-length) string, it collates as a blank space plus the internal representation of an empty string, \$CHAR(0), resulting in a string length of 2.

The optional *maxlen* argument truncates the *expression* string to the specified number of characters when indexing or collating. For example, if you insert a long string with *maxlen* truncation, the full string is inserted and can be retrieved by a **SELECT** statement; the index global for this string is truncated to the specified length. This means that **ORDER BY** and comparison operations only evaluate the truncated index string. Such truncation is especially useful for indexing on strings that exceed the 255-character limit for Caché subscripts. When converting from non-Caché systems, some users encountered problems when they indexed on a VARCHAR(255) field and then tried to insert data into the table. With the *maxlen* argument, if you need to index on a long field, you can use the truncation length parameter. **%SQLSTRING** performs *maxlen* truncation after converting *expression*; if *maxlen* exceeds the length of the converted *expression* no padding is added.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",8)
```

This function can also be invoked from ObjectScript using the **SQLSTRING()** method call:

```
WRITE $SYSTEM.SQL.SQLSTRING("The quick, BROWN fox.")
```

Both of these methods support truncation after SQLSTRING conversion. Note that the truncation length must include the appended blank:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",8,6),!
WRITE $SYSTEM.SQL.SQLSTRING("The quick, BROWN fox.",6)
```

For a not case-sensitive string conversion, refer to [%SQLUPPER](#).

Note: To change the system-wide default collation from %SQLUPPER (which is not case-sensitive) to %SQLSTRING (which is case-sensitive), use the following command:

```
WRITE $$SetEnvironment^%apiOBJ("collation", "%Library.String", "SQLSTRING")
```

After issuing this command, you must purge indexes, recompile all classes, then rebuild indexes. Do not rebuild indices while the table's data is being accessed by other users. Doing so may result in inaccurate query results.

Examples

The following query uses %SQLSTRING in the **WHERE** clause to perform a case-sensitive select:

```
SELECT Name FROM Sample.Person
WHERE %SQLSTRING Name %STARTSWITH %SQLSTRING 'Al'
ORDER BY Name
```

By default, %STARTSWITH string comparisons are not case-sensitive. This example uses the %SQLSTRING format to make this comparison case-sensitive. It returns all names that begin with “Al” (such as Allen, Alton, etc.). Note when using %STARTSWITH, you should apply %SQLSTRING collation to both sides of the statement.

The following example uses %SQLSTRING with a string truncation to return the first two characters of each name. Note that the string truncation is 3 (not 2) because of the leading blank added by %SQLSTRING. The **ORDER BY** clause uses this two-character field to put the rows in a rough collation sequence:

```
SELECT Name, %SQLSTRING(Name,3) AS FirstTwo
FROM Sample.Person
ORDER BY FirstTwo
```

This example returns the truncated values without changing the case of letters. To return truncated values in converted to uppercase, use %STRING.

The following example applies %SQLSTRING to a subquery:

```
SELECT TOP 5 Name, %SQLSTRING((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

See Also

- [CREATE TABLE](#)
- [%STARTSWITH](#) predicate
- [%SQLUPPER](#) collation function
- [%TRUNCATE](#) collation function
- [Collation](#) chapter in *Using Caché SQL*

%SQLUPPER

A collation function that sorts values as uppercase strings.

```
%SQLUPPER(expression[,maxlen])
%SQLUPPER expression
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2). <i>expression</i> can be a subquery.
<i>maxlen</i>	<i>Optional</i> — An integer, which specifies that the collated value will be truncated to the value of <i>maxlen</i> . Note that <i>maxlen</i> includes the appended leading blank space. You can enclose <i>maxlen</i> with double parentheses to suppress literal substitution : ((<i>maxlen</i>)).

Description

SQLUPPER is the default collation.

%SQLUPPER converts *expression* to a format that is sorted as a (not case-sensitive) uppercase string. **%SQLUPPER** converts all alphabetic characters to uppercase, strips trailing whitespace (spaces, tabs, and so on) from the string, then adds one leading blank space to the beginning of the string. This appended blank space causes NULL and numeric values to be collated as strings.

SQL converts numeric values to [canonical form](#) (removing leading and trailing zeros, expanding exponents, etc.) before passing the number to the function. SQL does not convert numeric strings to canonical form.

Because **%SQLUPPER** appends a blank space to all values, it collates a **NULL** value as a blank space, with a string length of 1. **%SQLUPPER** collates any value containing only whitespace (spaces, tabs, and so on) as the SQL [empty string](#) ("). When **%SQLUPPER** appends a blank space to an empty (zero-length) string, it collates as a blank space plus the internal representation of an empty string, \$CHAR(0), resulting in a string length of 2.

The optional *maxlen* argument truncates the converted *expression* string to the specified number of characters when indexing or collating. For example, if you insert a long string with *maxlen* truncation, the full string is inserted and can be retrieved by a **SELECT** statement; the index global for this string is truncated to the specified length. This means that **ORDER BY** and comparison operations only evaluate the truncated index string. Such truncation is especially useful for indexing on strings that exceed the 255-character limit for Caché subscripts. When converting from non-Caché systems, some users encountered problems when they indexed on a VARCHAR(255) field and then tried to insert data into the table. With the *maxlen* argument, if you need to index on a long field, you can use the truncation length parameter. **%SQLUPPER** performs *maxlen* truncation after converting *expression*; if *maxlen* exceeds the length of the converted *expression* no padding is added.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",7)
```

This function can also be invoked from ObjectScript using the **SQLUPPER()** method call:

```
WRITE $SYSTEM.SQL.SQLUPPER("The quick, BROWN fox.")
```

Both of these methods support truncation after SQLUPPER conversion. Note that the truncation length must include the appended blank:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",7,6),!
WRITE $SYSTEM.SQL.SQLUPPER("The quick, BROWN fox.",6)
```

For a case-sensitive string conversion, refer to [%SQLSTRING](#).

Note: To change the system-wide default collation from %SQLUPPER (which is not case-sensitive) to %SQLSTRING (which is case-sensitive), use the following command:

```
WRITE $$SetEnvironment^%apiOBJ("collation", "%Library.String", "SQLSTRING")
```

After issuing this command, you must purge indexes, recompile all classes, then rebuild indexes. Do not rebuild indices while the table's data is being accessed by other users. Doing so may result in inaccurate query results.

Other Case Conversion Functions

The **%SQLUPPER** function is the preferred way in SQL to convert a data value for not case-sensitive comparison or collation. **%SQLUPPER** adds a leading blank space to the beginning of the data, which forces numeric data and the NULL value to be interpreted as strings.

The following are other functions for converting the case of a data value:

- **%ALPHAUP**: converts letters to uppercase, has no effect on number characters, deletes all punctuation characters except commas and question marks (including deleting the minus sign and a period used as a decimal separator from numbers), deletes all blank spaces (leading, trailing, and embedded). Does not force numerics to be interpreted as a string.
- **UPPER**, **%UPPER**, and **UCASE**: converts letters to uppercase, has no effect on number characters, punctuation characters, embedded spaces, and leading and trailing blank spaces. Does not force numerics to be interpreted as a string.
- **LOWER** and **LCASE**: converts letters to lowercase, has no effect on number characters, punctuation characters, embedded spaces, and leading and trailing blank spaces. Does not force numerics to be interpreted as a string.
- **%STRING**: converts letters to uppercase, deletes all punctuation except commas (including deleting the minus sign and a period used as a decimal separator from numbers), deletes all blank spaces (leading, trailing, and embedded). It adds a leading blank space to the beginning of the data, which forces numeric data and the NULL value to be interpreted as strings.
- **%SQLSTRING**: does not convert letter case. However, it adds a leading blank space to the beginning of the data, which forces numeric data and the NULL value to be interpreted as strings.

Alphanumeric Collation Order

The case conversion functions collate data values that begin with a number using different algorithms, as follows:

%ALPHAUP and %STRING	%SQLUPPER, %SQLSTRING, and all other case conversion functions
5988 Clinton Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 641 First Place, 6572 First Avenue, 6643 First Street, 665 Ash Drive, 66 Main Street, 672 Main Court, 6754 Oak Court, 6986 Madison Blvd, 6 Oak Avenue, 7000 Ash Court, 709 Oak Avenue	5988 Clinton Avenue, 6 Oak Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 641 First Place, 6572 First Avenue, 66 Main Street, 6643 First Street, 665 Ash Drive, 672 Main Court, 6754 Oak Court, 6986 Madison Blvd, 7000 Ash Court, 709 Oak Avenue

Examples

The following query uses **%SQLUPPER** with a string truncation to return the first two characters of each name in uppercase. Note that the string truncation is 3 (not 2) because of the leading blank added by **%SQLUPPER**. The **ORDER BY** clause uses this two-character field to put the rows in a rough collation sequence:

```
SELECT Name, %SQLUPPER(Name,3) AS FirstTwo
FROM Sample.Person
ORDER BY FirstTwo
```

The following example applies **%SQLUPPER** to a subquery:

```
SELECT TOP 5 Name, %SQLUPPER((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

See Also

- [CREATE TABLE](#)
- [%STARTSWITH](#) predicate
- [%SQLSTRING](#) collation function
- [%TRUNCATE](#) collation function
- [Collation](#) chapter in *Using Caché SQL*

SQRT

A numeric function that returns the square root of a given numeric expression.

```
SQRT(numeric-expression)
{fn SQRT(numeric-expression)}
```

Arguments

<i>numeric-expression</i>	An expression that resolves to a positive number from which the square root is calculated.
---------------------------	--

Description

SQRT returns the square root of *numeric-expression*. The *numeric-expression* must be a positive number. A negative *numeric-expression* (other than -0) generates an SQLCODE -400 error. **SQRT** returns NULL if passed a NULL value.

SQRT returns a value of [data type](#) NUMERIC, unless *numeric-expression* is data type DOUBLE, in which case the returned data type is DOUBLE. The returned value has a precision of 36 and a scale of 18.

SQRT can be specified as a regular scalar function or as an ODBC scalar function (with the curly brace syntax).

Examples

The following example shows the two **SQRT** syntax forms. Both return the square root of 49:

```
SELECT SQRT(49) AS SRoot, {fn SQRT(49)} AS ODBCRoot
```

The following embedded SQL example returns the square roots of the integers 0 through 10:

```
SET a=0
WHILE a<11 {
&sql(SELECT SQRT(:a) INTO :b)
IF SQLCODE'=0 {
WRITE !,"Error code ",SQLCODE
QUIT }
ELSE {
WRITE !,"The square root of ",a," = ",b
SET a=a+1 }
}
```

See Also

- SQL functions: [POWER ROUND SQUARE](#)
- ObjectScript function: [\\$ZSQ](#)

SQUARE

A scalar numeric function that returns the square of a number.

```
SQUARE(numeric-expression)
```

Arguments

<i>numeric-expression</i>	An expression that resolves to a numeric value.
---------------------------	---

Description

SQUARE returns the square of *numeric-expression*. **SQUARE** returns NULL if passed a NULL value.

The precision and scale returned by **SQUARE** are the same as those returned by the [SQL multiplication operator](#).

Examples

The following Embedded SQL example returns the squares of the integers 0 through 10:

```
SET a=0
WHILE a<11 {
&sql(SELECT SQUARE(:a) INTO :b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE
  QUIT }
ELSE {
  WRITE !,"The square of ",a," = ",b
  SET a=a+1 }
}
```

See Also

- SQL functions: [POWER](#) [ROUND](#) [SQRT](#)
- ObjectScript function: [\\$ZPOWER](#)

STR

A function that converts a numeric to a string.

```
STR(number[,length[,decimals]])
```

Arguments

<i>number</i>	An expression that resolves to a numeric. It can be a field name, a numeric, or the result of another function. If a field name is specified, the logical value is used.
<i>length</i>	<i>Optional</i> — An integer specifying the total length of the desired output string, including all characters (digits, decimal point, sign, blank spaces). The default is 10.
<i>decimals</i>	<i>Optional</i> — An integer specifying the number of places to the right of the decimal point to include. The default is 0.

Description

STR converts a numeric to the STRING format, truncating the numeric based on the values of *length* and *decimals*. The *length* argument must be large enough to include the entire integer portion of the number, and, if *decimals* is specified, that number of decimal digits plus 1 (for the decimal point). If *length* is not large enough, **STR** returns a string of asterisks (*) equal to *length*.

STR converts numerics to their [canonical form](#) before string conversion. It therefore performs arithmetic operations, removes leading and trailing zeros and leading plus signs from numbers.

If the *number* argument is **NULL**, **STR** returns NULL. If the *number* argument is the empty string ("), **STR** returns the empty string. **STRING** retains whitespace.

Example

In the following Embedded SQL example, **STR** converts numerics into a string:

```
&sql( SELECT STR(123),
           STR(123,4),
           STR(+00123.45,3),
           STR(+00123.45,3,1),
           STR(+00123.45,5,1)
      INTO :v,:w,:x,:y,:z)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Resulting STR:",v," string"
  WRITE !,"Resulting STR:",w," string"
  WRITE !,"Resulting STR:",x," string"
  WRITE !,"Resulting STR:",y," string"
  WRITE !,"Resulting STR:",z," string" }
```

The first **STR** function returns a string consisting of 7 leading blanks and the number 123; the seven leading blanks are because the default string length is 10. The second **STR** function returns the string " 123"; note the leading blank needed to return a string of length 4. The third **STR** function returns the string "123"; the numeric is put into canonical form, and *decimals* defaults to 0. The fourth **STR** function returns "***" because the string length is not long enough to encompass the entire number as specified; the number of asterisks indicates the string length. The fifth **STR** function returns "123.4"; note that the *length* must be 5 to include the decimal digit.

See Also

[STRING %STRING %SQLUPPER %SQLSTRING](#)

STRING

A function that converts and concatenates expressions into a string.

```
STRING(string1[,string2][,...][,stringn])
```

Arguments

<i>string</i>	An expression, which can be a field name, a string literal, a numeric, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2). If a field name is specified, the logical value is used.
---------------	--

Description

STRING converts one or more *strings* to the **STRING** format, and then concatenates these strings into a single string. No case transformation is performed.

STRING converts numerics to their [canonical form](#) before string conversion. It therefore performs arithmetic operations, removes leading and trailing zeros and leading plus signs from numbers.

If one of the *string* arguments is **NULL**, **STRING** returns **NULL**. If one of the *string* arguments is the [empty string](#) ("), **STRING** concatenates the other arguments. **STRING** retains whitespace.

You can use the [%SQLSTRING](#) function to convert a data value for case-sensitive string comparison, or the [%SQLUPPER](#) function to convert a data value for not case-sensitive string comparison.

Examples

In the following Embedded SQL example, **STRING** concatenates three substrings into a single string. The example shows the handling of blank spaces, the empty string, and **NULL**:

```
&sql(SELECT STRING('a','b','c'),
           STRING('a',' ','c'),
           STRING('a','','c'),
           STRING('a',NULL,'c')
      INTO :w,:x,:y,:z)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Resulting string is:",w
  WRITE !,"Resulting string is:",x
  WRITE !,"Resulting string is:",y
  WRITE !,"Resulting string is:",z }
```

In the following Embedded SQL example, **STRING** converts numerics into a string. All of these **STRING** functions return the string '123':

```
&sql(SELECT STRING(123),
           STRING(+00123.00),
           STRING('1',23),
           STRING(1,(10*2)+3)
      INTO :w,:x,:y,:z)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Resulting string is:",w
  WRITE !,"Resulting string is:",x
  WRITE !,"Resulting string is:",y
  WRITE !,"Resulting string is:",z }
```

In the following Embedded SQL example, **STRING** retrieves sample data from fields and concatenates it into a string:

```
&sql (SELECT STRING(Name, Age)
      INTO :x
      FROM Sample.Person)
IF SQLCODE'=0 {
  WRITE !, "Error code ", SQLCODE }
ELSE {
  WRITE !, "Resulting string is:", x }
```

See Also

[%STRING](#) [%SQLUPPER](#) [%SQLSTRING STR](#)

%STRING

Deprecated. A collation function that converts characters to the STRING collation format.

```
%STRING(expression[, maxlen])
%STRING expression
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2). <i>expression</i> can be a subquery.
<i>maxlen</i>	<i>Optional</i> — An integer, which specifies that the collated value will be truncated to the value of <i>maxlen</i> . Note that <i>maxlen</i> includes the appended leading blank space. You can enclose <i>maxlen</i> with double parentheses to suppress literal substitution : ((<i>maxlen</i>)).

Description

This is a deprecated collation function. Please refer to [%SQLSTRING](#) (for case-sensitive string comparison) or [%SQLUPPER](#) (for not case-sensitive string comparison) for new development.

%STRING converts *expression* to the STRING format:

- Converts all letters to uppercase.
- Removes all punctuation characters, except the comma. ([%ALPHAUP](#) removes all punctuation characters, except the comma and the question mark.)
- Removes all blank spaces (leading, trailing, and embedded).
- Then adds a single space to the beginning of the value, forcing numeric data and the NULL value to be collated as strings.

SQL converts a number to [canonical form](#) before passing the number to the function. **%STRING** then removes the period (used as the decimal separator character in many locales) and the minus sign, as well as other punctuation and blanks. For this reason, **%STRING** must be used with caution on any expression containing non-alphabetic information.

Because **%STRING** appends a blank space to all values, it collates a [NULL](#) value as a blank space, with a string length of 1. **%STRING** collates any value containing only whitespace (spaces, tabs, and so on) as the SQL [empty string](#) ("). When **%STRING** appends a blank space to an empty (zero-length) string, it collates as a blank space plus the internal representation of an empty string, \$CHAR(0), resulting in a string length of 2.

The optional *maxlen* truncation argument is especially useful for indexing on strings that exceed the 255-character limit for Caché subscripts. When converting from non-Caché systems, some users encountered problems when they indexed on a VARCHAR(255) field and then tried to insert data into the table. With the *maxlen* argument, if you need to index on a long field, you can use STRING collation with a truncation length parameter. **%STRING** performs *maxlen* truncation after converting *expression*; if *maxlen* exceeds the length of the converted *expression* no padding is added.

%STRING is a Caché SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",9)
```

This function can also be invoked from ObjectScript using the **STRING()** method call:

```
WRITE $SYSTEM.SQL.STRING("The quick, BROWN fox.")
```

Both of these methods support truncation after `STRING` conversion. Note that the truncation length must include the appended blank:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",9,6),!
WRITE $SYSTEM.SQL.STRING("The quick, BROWN fox.",6)
```

The `%SQLUPPER` function is the preferred way in `SQL` to convert a data value for not case-sensitive comparison or collation. Refer to `%SQLUPPER` for further information on case transformation functions.

Examples

The following example selects `Name` values that begin with “od”:

```
SELECT Name
FROM Sample.Person
WHERE %STRING(Name) %STARTSWITH %STRING 'od'
```

Because `%STRING` removes punctuation and blank spaces and performs not case-sensitive collation, this example returns names such as “Odem”, “O'Donnell”, “ODonnell”, and “O Donnell”.

The following example uses `%STRING` with a string truncation to return the first two characters of each name in uppercase. Note that the string truncation is 3 (not 2) because of the leading blank added by `%STRING`. The `ORDER BY` clause uses this two-character field to put the rows in a rough collation sequence:

```
SELECT Name, %STRING(Name,3) AS FirstTwo
FROM Sample.Person
ORDER BY FirstTwo
```

The following example applies `%STRING` to a subquery:

```
SELECT TOP 5 Name, %STRING((SELECT Name FROM Sample.Company),10) AS Company
FROM Sample.Person
```

Alphanumeric Collation Order

The case conversion functions collate data values that begin with a number using different algorithms, as follows:

%ALPHAUP and %STRING	%SQLUPPER, %SQLSTRING, and all other case conversion functions
5988 Clinton Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 641 First Place, 6572 First Avenue, 6643 First Street, 665 Ash Drive, 66 Main Street, 672 Main Court, 6754 Oak Court, 6986 Madison Blvd, 6 Oak Avenue, 7000 Ash Court, 709 Oak Avenue	5988 Clinton Avenue, 6 Oak Avenue, 6023 Washington Court, 6090 Elm Court, 6185 Clinton Drive, 6209 Clinton Street, 6284 Oak Drive, 6310 Franklin Street, 6406 Maple Place, 641 First Place, 6572 First Avenue, 66 Main Street, 6643 First Street, 665 Ash Drive, 672 Main Court, 6754 Oak Court, 6986 Madison Blvd, 7000 Ash Court, 709 Oak Avenue

See Also

- [CREATE TABLE](#)
- [%STARTSWITH](#) predicate
- [STRING](#) function

- [%SQLUPPER](#) collation function
- [%SQLSTRING](#) collation function
- [%TRUNCATE](#) collation function
- [Collation](#) chapter in *Using Caché SQL*

STUFF

A string function that replaces a substring within a string.

```
STUFF(string, start, length, substring)
```

Arguments

<i>string</i>	A string expression that is the target for the substring replacement.
<i>start</i>	The starting point for replacement, specified as a positive integer. A character count from the beginning of <i>string</i> , counting from 1. Permitted values are 0 through the length of <i>string</i> . To append characters, specify a <i>start</i> of 0 and a <i>length</i> of 0. The empty string or a nonnumeric value is treated as 0.
<i>length</i>	The number of characters to replace, specified as a positive integer. To insert characters, specify a <i>length</i> of 0. To replace all characters after <i>start</i> , specify a <i>length</i> greater than the number of existing characters. The empty string or a nonnumeric value is treated as 0.
<i>substring</i>	A string expression used to replace the substring identified by its starting point and length. Can be longer or shorter than the substring it replaces. Can be the empty string.

Description

STUFF replaces a substring with another substring. It identifies the substring to be replaced by location and length, and replaces it with *substring*.

This function provides compatibility with Transact-SQL implementations.

The replacement *substring* may be longer or shorter than the original value. To delete the original value, *substring* can be the empty string ("").

The *start* value must be within the current length of *string*. You can append a *substring* to the beginning of *string* by specifying a *start* value of 0. The empty string or a nonnumeric value is treated as 0.

Specifying NULL for the *start*, *length*, or *substring* argument returns NULL.

REPLACE and STUFF

Both **REPLACE** and **STUFF** perform substring replacement. **REPLACE** searches for a substring by data value. **STUFF** searches for a substring by string position and length.

For a list of functions that search for a substring, refer to [String Manipulation](#) in the Concepts section of this manual.

Examples

The following example shows a single-character substitution, turning KING into KONG:

```
SELECT STUFF('KING', 2, 1, 'O')
```

The following examples replace an 8-character substring (Kentucky) with a longer 12-character substring and a shorter 2-character substring:

```
SELECT STUFF('In my old Kentucky home', 11, 8, 'Rhode Island'),
       STUFF('In my old Kentucky home', 11, 8, 'KY')
```

The following example inserts a substring:

```
SELECT STUFF('In my old Kentucky home', 19, 0, ' (KY)')
```

The following example appends a substring to the beginning of the string:

```
SELECT STUFF('In my old Kentucky home',0,0,'The sun shines bright ')
```

The following example deletes an 8-character substring by replacing it with the empty string:

```
SELECT STUFF('In my old Kentucky home',11,8,'')
```

See Also

- [REPLACE](#) function
- [\\$EXTRACT](#) function
- [SUBSTRING](#) function
- [SUBSTR](#) function
- [String Manipulation](#)

SUBSTR

A string function that returns a substring that is derived from a specified string expression.

```
SUBSTR(string-expression,start[,length])
```

Arguments

<i>string-expression</i>	The string expression from which the substring is to be derived. The expression can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
<i>start</i>	An integer that specifies where in <i>string-expression</i> the substring will begin. A positive starting position specifies the number of characters from the beginning of the string. The first character in <i>string-expression1</i> is at position 1. A negative starting position specifies the number of characters from the end of the string. If <i>start</i> is 0 (zero), it is treated as 1.
<i>length</i>	<i>Optional</i> — A positive integer that specifies the length of the substring to return. This value specifies that the substring ends <i>length</i> characters to the right of the starting position. If omitted, substring goes from <i>start</i> to the end of <i>string-expression</i> . If <i>length</i> is 0 or a negative number, Caché returns NULL.

Description

Because *start* can be negative, you can obtain a substring from either the beginning or end of the original string.

Floating-point numbers passed as arguments to **SUBSTR** are converted to integers by truncating the fractional portion.

- If *start* is 0, -0, or 1, the returned substring begins with the first character of the string.
- If *start* is a negative number the returned substring begins that number of characters from the end of the string, with -1 representing the last character of the string. If the negative number is so large that its value counted backwards from the end of the string would position before the beginning of the string, the returned substring begins with the first character of the string.
- If *start* is past the end of the string, NULL is returned.
- If *length* larger than the remaining characters in the string, the substring from *start* to the end of the string is returned.
- If *length* is less than 1, NULL is returned.
- If either *start* or *length* is NULL, NULL is returned.

SUBSTR cannot be used with stream data. If *string-expression* is a stream field, **SUBSTR** generates an SQLCODE -37. Use **SUBSTRING** to extract a substring from stream data.

SUBSTR is supported for Oracle compatibility.

Examples

The following example returns the substring CDEFG because it specifies that the substring begin at the third character (C) and continue to the end of the string:

```
SELECT SUBSTR('ABCDEFG',3) AS Sub
```

The following example returns the substring CDEF because it specifies that the substring begin at the third character (C) and continue for four characters (until F):

```
SELECT SUBSTR('ABCDEFGH', 3, 4) AS Sub
```

The following example returns the substring CDEF because it specifies that Caché should first count five characters backwards from the end of the original string, and then return the next four characters:

```
SELECT SUBSTR('ABCDEFGH', -5, 4) AS Sub
```

See Also

- SQL function: [SUBSTRING](#)
- ObjectScript functions: [\\$EXTRACT](#) [\\$PIECE](#)

SUBSTRING

A string function that returns a substring from a larger character string.

```
SUBSTRING(string-expression, start[, length])
SUBSTRING(string-expression FROM start [FOR length])
{fn SUBSTRING(string-expression, start[, length])}
```

Arguments

<i>string-expression</i>	The string expression from which the substring is to be derived. An expression, which can be the name of a column, a string literal, or the result of another scalar function. The underlying data type can be a character type (such as CHAR or VARCHAR), a numeric, or a data stream.
<i>start</i>	An integer that specifies the position in <i>string-expression</i> to begin the substring. The first character in <i>string-expression</i> is at position 1. If the start position is higher than the length of the string, SUBSTRING returns an empty string (""). If the start position is lower than 1 (zero, or a negative number) the substring begins at position 1, but the <i>length</i> of the substring is reduced by the start position.
<i>length</i>	<i>Optional</i> — An integer that specifies the length of the substring to return. If <i>length</i> is not specified, the default is to return the rest of the string.

Description

The value of *start* controls the starting point of the substring:

- If *start* is less than 1, the value of *length* is decremented by a corresponding amount. Thus, if *start* is 0, the value of *length* is diminished by 1; if *start* is -1, the value of *length* is diminished by 2.

The value of *length* controls the size of the substring:

- If *length* is a positive value (1 or greater), the substring ends *length* number of characters to the right of the starting position. (This effective length may be diminished if the *start* number is less than 1.)
- If *length* is larger than the number of character remaining in the string, all characters to the right of the starting position through the end of *string-expression* are returned.
- If *length* is zero, NULL is returned.
- If *length* is a negative number, Caché issues an SQLCODE -140 error.

Floating-point numbers passed as arguments to **SUBSTRING** are converted to integers by truncating the fractional portion.

SUBSTRING extracts a substring from the beginning of a string. **SUBSTR** can extract a substring from either the beginning or the end of a string. Note that these two SQL functions handle argument values differently. **SUBSTRING** can be used with [character stream data](#); **SUBSTR** cannot be used with stream data.

SUBSTRING can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

Return Value

If any **SUBSTRING** argument value is NULL, **SUBSTRING** returns NULL.

If *string-expression* is any %String data type, the **SUBSTRING** return value is the same data type as the *string-expression* data type. This allows **SUBSTRING** to handle user-defined string data types with special encoding.

If *string-expression* is *not* a %String data type (for example, %Float), the **SUBSTRING** return value is %String.

Examples

This example returns the string “forward”:

```
SELECT {fn SUBSTRING( 'forward pass',1,7 )} AS SubText
```

This example returns the string “pass”:

```
SELECT {fn SUBSTRING( 'forward pass',9,4 )} AS SubText
```

The following example returns the first four characters of each name:

```
SELECT Name,SUBSTRING(Name,1,4) AS FirstFour  
FROM Sample.Person
```

The following example demonstrates another syntactical form of **SUBSTRING**. This example is functionally the same as the previous example:

```
SELECT Name,SUBSTRING(Name FROM 1 FOR 4) AS FirstFour  
FROM Sample.Person
```

The following example shows how the *length* is reduced by a *start* value of less than 1. (A *start* value of 0 reduces *length* by 1, a *start* value of -1 reduces *length* by 2, and so forth.) In this case, *length* is reduced by 3, so only one character (“A”) is returned:

```
SELECT {fn SUBSTRING( 'ABCDEFG',-2,4 )} AS SubText
```

See Also

- SQL function: [SUBSTR](#)
- ObjectScript functions: [\\$EXTRACT \\$PIECE](#)

SYSDATE

A date/time function that returns the current local date and time.

SYSDATE

Description

SYSDATE takes no arguments and returns the current local date and time as a [timestamp](#) in %TimeStamp data type format (yyyy-mm-dd hh:mm:ss.fff). **SYSDATE** returns the current local date and time for this [timezone](#); it adjusts for local time variants, such as [Daylight Saving Time](#).

By default, **SYSDATE** returns time in whole second increments. This default can be configured.

Note: **SYSDATE** is a synonym for the argumentless [CURRENT_TIMESTAMP](#) function. The [CURRENT_TIMESTAMP](#) function is preferred for use in Caché SQL. The **SYSDATE** function is provided for compatibility with other versions of SQL.

See Also

- [CURRENT_TIMESTAMP](#)

TAN

A scalar numeric function that returns the tangent, in radians, of an angle.

```
{fn TAN(float-expression)}
```

Arguments

<i>float-expression</i>	An expression of type FLOAT. This is an angle expressed in radians.
-------------------------	---

Description

TAN takes any numeric value and returns its tangent. **TAN** returns NULL if passed a NULL value. **TAN** treats nonnumeric strings as the numeric value 0.

TAN returns a value of data type FLOAT with a precision of 36 and a scale of 18.

TAN can only be used as an ODBC scalar function (with the curly brace syntax).

You can use the [DEGREES](#) function to convert radians to degrees. You can use the [RADIANS](#) function to convert degrees to radians.

Example

The following example shows the effect of **TAN**.

```
SELECT {fn TAN(0.52)} AS Tangent
```

returns 0.572561.

See Also

- SQL functions: [ACOS](#) [ASIN](#) [ATAN](#) [COS](#) [COT](#) [SIN](#)
- ObjectScript function: [\\$ZTAN](#)

TIMESTAMPADD

A scalar date/time function that returns a new timestamp calculated by adding a number of intervals of a specified date part to a timestamp.

```
{fn TIMESTAMPADD(interval-type, integer-exp, timestamp-exp) }
```

Arguments

<i>interval-type</i>	The type of time/date interval that <i>integer-exp</i> represents, specified as a keyword.
<i>integer-exp</i>	An integer value expression that is to be added to <i>timestamp-exp</i> .
<i>timestamp-exp</i>	A timestamp value expression, which will be increased by the value of <i>integer-exp</i> .

Description

The **TIMESTAMPADD** function modifies a date/time expression by incrementing the specified date part by the specified number of units. For example, if *interval-type* is `SQL_TSI_MONTH` and *integer-exp* is 5, **TIMESTAMPADD** increments *timestamp-exp* by five months. You can also decrement a date part by specifying a negative integer for *integer-exp*.

TIMESTAMPADD returns a timestamp of the same data type as the input *timestamp-exp*:%Library.TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff).

Note that **TIMESTAMPADD** can only be used as an ODBC scalar function (with the curly brace syntax).

Similar time/date modification operations can be performed on a timestamp using the [DATEADD](#) general function.

Interval Types

The *interval-type* argument can be one of the following timestamp intervals:

- `SQL_TSI_FRAC_SECOND`
- `SQL_TSI_SECOND`
- `SQL_TSI_MINUTE`
- `SQL_TSI_HOUR`
- `SQL_TSI_DAY`
- `SQL_TSI_WEEK`
- `SQL_TSI_MONTH`
- `SQL_TSI_QUARTER`
- `SQL_TSI_YEAR`

These timestamp intervals may be specified with or without enclosing quotation marks, using single quotes or double quotes. They are not case-sensitive.

Incrementing or decrementing a timestamp interval causes other intervals to be modified appropriately. For example, incrementing the hour past midnight automatically increments the day, which may in turn increment the month, and so forth. **TIMESTAMPADD** always returns a valid date, taking into account the number of days in a month, and calculating for leap year. For example, incrementing January 31 by one month returns February 28 (the highest valid date in the month), unless the specified year is a leap year, in which case it returns February 29.

You can increment or decrement by fractional seconds of three digits of precision. Specify fractional seconds as an integer count of thousandths of a second (001 through 999).

DATEADD and **TIMESTAMPADD** handle quarters (3-month intervals); **DATEDIFF** and **TIMESTAMPDIFF** do not handle quarters.

%TimeStamp Format

If the *timestamp-exp* argument is in %Library.TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) the following rules apply:

- If *timestamp-exp* specifies only a time value, the date portion of *timestamp-exp* is set to '1900-01-01' before calculating the resulting timestamp.
- If *timestamp-exp* specifies only a date value, the time portion of *timestamp-exp* is set to '00:00:00' before calculating the resulting timestamp.
- The *timestamp-exp* can include or omit fractional seconds. The *timestamp-exp* can include any number of digits of precision, but *interval-type* SQL_TSI_FRAC_SECOND specifies exactly three digits of precision. Attempting to specify a SQL_TSI_FRAC_SECOND of less than or more than three digits can have unpredictable results.

Range and Value Checking

TIMESTAMPADD performs the following checks on %Library.TimeStamp input values:

- All specified parts of the *timestamp-exp* must be valid before any **TIMESTAMPADD** operation can be performed.
- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits. An invalid date value results in an SQLCODE -400 error.
- Date values must be within a valid range. Years: 1841 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 00 through 23. Minutes: 0 through 59. Seconds: 0 through 59. The number of days in a month must match the month and year. For example, the date '02-29' is only valid if the specified year is a leap year. An invalid date value results in an SQLCODE -400 error.
- The incremented (or decremented) year value returned must be within the range 1841 through 9999. Incrementing or decrementing beyond this range returns <null>.
- Date values less than 10 may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid. Date values less than 10 are always returned with a leading zero.
- Time values may be wholly or partially omitted. If *timestamp-exp* specifies an incomplete time, zeros are supplied for the unspecified parts.
- An hour value less than 10 must include a leading zero. Omitting this leading zero results in an SQLCODE -400 error.

Examples

The following example adds 1 week to the original timestamp:

```
SELECT {fn TIMESTAMPADD(SQL_TSI_WEEK,1,'2003-12-20 12:00:00')}
```

it returns 2003-12-27 12:00:00, because adding 1 week adds 7 days.

The following example adds 5 months to the original timestamp:

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MONTH,5,'1999-12-20 12:00:00')}
```

returns 2000-05-20 12:00:00 because in this case adding 5 months also increments the year.

The following example also adds 5 months to the original timestamp:

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MONTH,5,'1999-01-31 12:00:00')}
```

it returns 1999-06-30 12:00:00. Here **TIMESTAMPADD** modified the day value as well as the month, because simply incrementing the month would result in June 31, which is an invalid date.

The following example increments the original timestamp by 45 minutes:

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MINUTE,45,'1999-12-20 00:00:00')}
```

returns 1999-12-20 00:45:00.

The following example decrements the original timestamp by 45 minutes:

```
SELECT {fn TIMESTAMPADD(SQL_TSI_MINUTE,-45,'1999-12-20 00:00:00')}
```

it returns 1999-12-19 23:15:00. Note that in this case decrementing the time also decremented the day.

See Also

- [TIMESTAMPDIFF](#), [DATEADD](#), [DATENAME](#), [DATEPART](#), [TO_TIMESTAMP](#)

TIMESTAMPDIFF

A scalar date/time function that returns an integer count of the difference between two timestamps for a specified date part.

```
{fn TIMESTAMPDIFF(interval-type, startdate, enddate)}
```

Arguments

<i>interval-type</i>	The type of time/date interval that the returned value will represent.
<i>startdate</i>	A timestamp value expression.
<i>enddate</i>	A timestamp value expression that will be compared to <i>startdate</i> .

Description

The **TIMESTAMPDIFF** function returns the difference between two given timestamps (that is, one timestamp is subtracted from the other) for the specified date part interval (seconds, days, weeks, etc.). The value returned is an **INTEGER**, the number of these intervals between the two timestamps. (If *enddate* is earlier than *startdate*, **TIMESTAMPDIFF** returns a negative **INTEGER** value.)

The *startdate* and *enddate* are timestamps. These timestamps are in %Library.TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff).

The *interval-type* argument can be one of the following timestamp intervals:

- SQL_TSI_FRAC_SECOND
- SQL_TSI_SECOND
- SQL_TSI_MINUTE
- SQL_TSI_HOUR
- SQL_TSI_DAY
- SQL_TSI_WEEK
- SQL_TSI_MONTH
- SQL_TSI_YEAR

These timestamp intervals may be specified with or without enclosing quotation marks, using single quotes or double quotes. They are not case-sensitive.

TIMESTAMPDIFF and **DATEDIFF** do not handle quarters (3-month intervals).

Note that **TIMESTAMPDIFF** can only be used as an ODBC scalar function (with the curly brace syntax). Similar time/date comparison operations can be performed on a timestamp using the **DATEDIFF** general function.

%TimeStamp Format

If the *startdate* or *enddate* argument is in %Library.TimeStamp data type format (yyyy-mm-dd hh:mm:ss.ffff) the following rules apply:

- If either timestamp expression specifies only a time value and *interval-type* specifies a date interval (days, weeks, months, or years), the missing date portion of the timestamp defaults to '1900-01-01' before calculating the resulting interval count.

- If either timestamp expression specifies only a date value and *interval-type* specifies a time interval (hours, minutes, seconds, fractional seconds), the missing time portion of the timestamp defaults to '00:00:00.000' before calculating the resulting interval count.
- You can include or omit fractional seconds of any number of digits of precision. `SQL_TSI_FRAC_SECOND` returns a difference of fractional seconds as an integer count of thousandths of a second (three digits of precision).

Range and Value Checking

`TIMESTAMPDIFF` performs the following checks on input values.

- All specified parts of the *startdate* and *enddate* must be valid before any `TIMESTAMPDIFF` operation can be performed.
- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits. An invalid date value results in an `SQLCODE -8` error.
- Date values must be within a valid range. Years: 0001 through 9999. Months: 1 through 12. Days: 1 through 31. Hours: 00 through 23. Minutes: 0 through 59. Seconds: 0 through 59. The number of days in a month must match the month and year. For example, the date '02-29' is only valid if the specified year is a leap year. An invalid date value results in an `SQLCODE -8` error.
- Date values less than 10 (month and day) may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.
- Time values may be wholly or partially omitted. If *startdate* or *enddate* specifies an incomplete time, zeros are supplied for the unspecified parts.
- An hour value less than 10 must include a leading zero. Omitting this leading zero results in an `SQLCODE -8` error.

Examples

The following example returns 7 because the second timestamp (1999-12-20 12:00:00) is 7 months greater than the first one:

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MONTH,
    '1999-5-19 00:00:00', '1999-12-20 12:00:00')}
```

The following example returns 566 because the second timestamp ('12:00:00') is 566 minutes greater than the first one (02:34:12):

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MINUTE, '02:34:12', '12:00:00')}
```

The following example returns -1440 because the second timestamp is one day (1440 minutes) lesser than the first one:

```
SELECT {fn TIMESTAMPDIFF(SQL_TSI_MINUTE, '2017-04-06', '2017-04-05')}
```

See Also

- [TIMESTAMPADD](#), [DATEDIFF](#), [TO_TIMESTAMP](#)

TO_CHAR

A string function that converts a date, timestamp, or number to a formatted character string.

```
TO_CHAR(tochar-expression[,format])
```

```
TOCHAR(tochar-expression[,format])
```

Arguments

<i>tochar-expression</i>	A logical date, timestamp, or number expression to be converted.
<i>format</i>	<i>Optional</i> — A character code that specifies a date, timestamp, or number format for the <i>tochar-expression</i> conversion. If omitted, TO_CHAR returns <i>tochar-expression</i> as a canonical number.

Description

The names **TO_CHAR** and **TOCHAR** are interchangeable and are supported for Oracle compatibility.

The **TO_CHAR** function with *format* has five uses:

- To convert a date integer to a formatted date string.
- To convert a date before 1840 to a Julian date integer.
- To convert a time integer to a formatted time string.
- To convert a date and time to a formatted datetime string.
- To convert a number to a formatted numeric string.

This function can also be invoked from ObjectScript using the **TOCHAR()** method call:

```
$SYSTEM.SQL.TOCHAR(tochar-expression,format)
```

Valid and Invalid Arguments

- For *tochar-expression* to be interpreted as a timestamp, it must be of the format YYYY-MM-DD HH:MI:SS, or one of the following valid variants: Month and date values that are less than 10 may include or omit a leading zero; if the leading zero is omitted, it is also omitted in the returned date. The seconds value may be omitted, though the colon indicating its place must be specified (HH:MI:); in the returned time the seconds default to 00. The seconds value may include fractional seconds (HH:MM:SS.nnn); in the returned time these fractional seconds are truncated. A timestamp must include a time portion, even if *format* does not specify time formatting.
- If *tochar-expression* is not a valid timestamp format, **TO_CHAR** interprets it as an integer, ending interpretation when it encounters the first non-integer character. If *format* is a date or timestamp format, **TO_CHAR** interprets *tochar-expression* as a **\$HOROLOG** date integer. Thus 2010-03-23 12-15:23 (note erroneous hyphen in time value) is interpreted as the **\$HOROLOG** date 2010 (1846-07-03 12:00:00 AM).
- If a *tochar-expression* date or time is not a valid date or time value, Caché issues an SQLCODE -400 error. This can occur with a nonexistent date, such as February 30, or a date before 12/31/1840.
- If you specify a *format* with an invalid date, time, or timestamp code element (for example, YYYYYY, MIN, HH48), **TO_CHAR** returns the format code literal for the invalid code element; it returns date, time, or timestamp conversion values for valid code elements, if any.

- If **TO_CHAR** cannot recognize any *format* code elements (for example, *format* is an empty string) or if a number format has fewer digits than the *tochar-expression* value, **TO_CHAR** returns pound sign (#) characters. (This is true when *tochar-expression* begins with at least two integer digits; otherwise **TO_CHAR** returns NULL.)
- If you omit *format*, **TO_CHAR** returns the numeric portion of *tochar-expression* as a canonical number, truncating when it encounters a nonnumeric character. If *tochar-expression* is nonnumeric, **TO_CHAR** returns 0. If *tochar-expression* is null, **TO_CHAR** returns null.

TO_CHAR and TO_DATE

- **TO_CHAR** converts a date integer to a formatted date string, or a time integer to a formatted time string. If you erroneously supply **TO_CHAR** with a formatted date or time string, it returns erroneous data.
- **TO_DATE** converts a formatted date string to the corresponding date integer. If you erroneously supply **TO_DATE** with a date integer, it returns this integer unmodified.
- Note that for Julian dates these operations are reversed.

These correct and erroneous uses of **TO_DATE** and **TO_CHAR** are shown in the following examples.

The following Embedded SQL example uses **TO_DATE** to perform a date conversion. **TO_DATE** takes a date string and returns the corresponding date integer (59832). The **\$ZDATE** function is used to display this date integer as the formatted date 10/24/2004. In this example, **TO_DATE** is also erroneously supplied a date integer; it simply returns this integer.

```
&sql (SELECT
        TO_DATE('2004-10-24','YYYY-MM-DD'), /* correct */
        TO_DATE(59832,'YYYY-MM-DD')      /* ERROR! */
    INTO :a,:b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,a
    WRITE !,$ZDATE(a)
    WRITE !,b
}
```

The following Embedded SQL example shows date conversions using **TO_CHAR**. The first **TO_CHAR** converts a date integer to the corresponding formatted date string, as expected. However, the second **TO_CHAR** gives unexpected results. Since **TO_CHAR** expects a numeric input, it treats the date separators in the input as minus signs and performs the subtractions. It therefore formats a date corresponding to the date integer 1970 (2004 minus 10 minus 24): 1846–5–24. Obviously, this was not the programmer's intent.

```
&sql (SELECT
        TO_CHAR(59832,'YYYY-MM-DD'),      /* correct */
        TO_CHAR(2004-10-24,'YYYY-MM-DD') /* ERROR! */
    INTO :a,:b)
IF SQLCODE'=0 {
    WRITE !,"Error code ",SQLCODE }
ELSE {
    WRITE !,a
    WRITE !,b }
}
```

Related SQL Functions

- **TO_CHAR** converts a date integer, a timestamp, or a number to a string.
- **TO_DATE** performs the reverse operation for dates; it converts a formatted date string to a date integer.
- **TO_TIMESTAMP** performs the reverse operation for timestamps; it converts a formatted date and time string to a standard timestamp.
- **TO_NUMBER** performs the reverse operation for numbers; it converts a numeric string to a number.
- **CAST** and **CONVERT** perform DATE, TIMESTAMP, and NUMBER data type conversions.

Date-to-String Conversion

\$HOROLOG format is the Caché SQL Logical format for representing dates and times. It is a string containing two comma-separated integers: the first is the number of days since December 31, 1840; the second is the number of seconds since midnight of the current day.

You can use **TO_CHAR** to convert a **\$HOROLOG** date integer or a **\$HOROLOG** string of two comma-separated integers to a formatted date string, or a formatted date and time string. The value for *tochar-expression* must be a valid **\$HOROLOG** value.

The following table lists the valid date *format* codes for this version of **TO_CHAR**.

Format Code	Meaning
D	Day of week (1-7). By default, 1 is Sunday (the first day of the week), but this designation is configurable; refer to the DAYOFWEEK function.
DD	Two-digit day of month (01-31).
DY	Abbreviated name of the day, as specified by the WeekdayAbbr property of the current locale. The defaults are: Sun Mon Tue Wed Thu Fri Sat
DAY	Name of day, as specified by the WeekdayName property in the current locale. The defaults are: Sunday Monday Tuesday Wednesday Thursday Friday Saturday
MM	Two-digit month number (01-12; 01 = JAN).
MON	Abbreviated name of month, as specified by the MonthAbbr property in the current locale. The defaults are: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec. Not case-sensitive.
MONTH	Full name of the month, as specified by the MonthName property in the current locale. The defaults are: January February March April May June July August September October November December. Not case-sensitive.
YYYY	Four-digit year.
YYY	Last 3 digits of the year.
YY	Last 2 digits of the year.
Y	Last digit of the year.
RRRR	Four-digit year.
RR	Last 2 digits of the year.
DDD	Day of the Year (see below).
J	Julian date (number of days since January 1, 4712 BC (BCE)).

Separator characters are required between the date format elements, with the exception of the following format strings: **YYYYMMDD**, **DDMMYYYY**, and **YYYYMM**. The last of these returns the year and month values and ignores the day of the month.

Note that locales mentioned in the *format* code definitions refer to the same locales described in the ObjectScript **\$ZDATE** and **\$ZDATEH** documentation.

Date Conversion Examples

The following are all valid uses of **TO_CHAR** with a **\$HOROLOG** date integer or a full **\$HOROLOG** string value to return a formatted date string or a date and time string:

```
SELECT TO_CHAR(63012,'YYYY-MM-DD') AS DateFD,
       TO_CHAR(63012,'YYYY-MM-DD HH24:MI:SS') AS DateFDT,
       TO_CHAR('63012,50278','YYYY-MM-DD') AS DateTimeFD,
       TO_CHAR('63012,50278','YYYY-MM-DD HH24:MI:SS') AS DateTimeFDT
```

In the following example each **TO_CHAR** takes a date integer and returns a date string formatted according to the *format* string argument:

```
SELECT TO_CHAR(63012,'MM/DD/YYYY'),           /* returns 07/09/2013 */
       TO_CHAR(63012,'DAY MONTH DD, YYYY') /* returns Tuesday July 09, 2013 */
```

The following example takes a date integer and returns a formatted date string. Characters that are not *format* characters are passed through to the output string as literals:

```
SELECT TO_CHAR(63012,'The date MM/DD/YYYY should be noted')
```

returns the string: The date 07/09/2013 should be noted

Day of the Year

You can use DDD to convert a date expression to the day of the year (number of days elapsed since January 1) and the year. The format string DDD, YYYY must be paired with a date expression in **\$HOROLOG** format. (The **\$HOROLOG** time value, if specified, is ignored.) The DDD and YYYY (or YY) format elements can be specified in any order; a separator character between them is mandatory and is returned as a literal. The following examples show this use of Day of the Year:

```
SELECT TO_CHAR('60871','YYYY:DDD')
```

```
SELECT TO_CHAR('60871,12345','DDD YY')
```

TO_CHAR permits you to return the day of the year corresponding to a date expression. **TO_DATE** permits you to return a date expression corresponding to a day of the year.

Julian Date Conversion

The “Julian” date format is provided to allow for dates before the year 1841. **TO_CHAR** converts a date value for data type %Date or %TimeStamp to a seven-digit Julian date integer.

Note: By default, the %Date data type does not represent dates prior to [December 31, 1840](#). However, you can redefine the MINVAL parameter for this data type to permit representation of earlier dates as negative integers, with the limit of January 1, Year 1. This representation of dates as negative integers is not compatible with the “Julian” date format described here. For further details refer to the [Data Types](#) reference page in this manual.

If you specify a *format* that consists of a string containing the letter 'J', the date value returned will be a “Julian” date—that is, a count of days from January 1, 4712BCE. Only the letter 'J' may be specified in the *format* string; the inclusion of any other characters causes 'J' to be treated as a literal, and the date to be translated as a standard date.

The maximum *tochar-expression* value for Julian dates is '9999-12-31' which corresponds to Julian day count 5373484. The minimum value is '-4712-01-01' which corresponds to Julian day count 0000001. A Julian day count is always represented as a seven-digit integer, with leading zeros when necessary.

The following example returns 2369916 (signing of the Declaration of Independence of the United States) and 1709980 (battle of Actium marks beginning of Roman Empire under Augustus Caesar):

```
SELECT TO_CHAR('1776-07-04','J') AS UnitedStatesStart,
       TO_CHAR('-0031-09-02','J') AS RomanEmpireStart
```

Note: The following consideration should not affect the interconversion of dates and Julian day counts using **TO_CHAR** and **TO_DATE**. It may affect some calculations made using Julian day counts.

Julian day counts prior to 1721424 (1/1/1) are compatible with other software implementations, such as Oracle. They are *not* identical to BCE dates in ordinary usage. In ordinary usage, there is no Year 0; dates go from 12/31/-1 to 1/1/1. In Oracle usage, the Julian dates 1721058 through 1721423 are simply invalid, and return an error. In Caché these Julian dates return the non-existent Year 0 as a place holder. Thus calculations involving BCE dates must be adjusted by one year to correspond to common usage.

Also be aware that these date counts do not take into account changes in date caused by the Gregorian calendar reform (enacted October 15, 1582, but not adopted in Britain and its colonies until 1752).

TO_CHAR permits you to return a Julian day count corresponding to a date expression. **TO_DATE** permits you to return a date expression corresponding to a Julian day count, as shown in the following example:

```
SELECT TO_CHAR('1776-07-04','J') AS JulianCount,
       TO_DATE(2369916,'J') AS JulianDate
```

For further details on using Julian dates, see the **TO_DATE** function.

Time-to-String Conversion

You can use **TO_CHAR** to convert the following *tochar-expression* time values to a formatted time string:

- A **\$HOROLOGY** time integer (the time component of **\$HOROLOGY**). The value for *tochar-expression* must be a valid Logical time (an integer in the range 0 through 86399). *Do not* supply a full **\$HOROLOGY** value with both date and time components (such as 62556 , 42152); **TO_CHAR** time conversion would incorrectly convert the first (date) component of **\$HOROLOGY** to a formatted time string, and ignore the second (time) component.
- A Logical timestamp value. The value for *tochar-expression* must be of the %TimeStamp data type (not a string data type) in the format YYYY-MM-DD hh:mm:ss. The date component of the timestamp is ignored and the time component converted. For example, **SYSDATE** is a Logical timestamp.
- A time value in standard ODBC time format. The value for *tochar-expression* must be in the format hh:mm:ss and can be a string.
- A time value in local time format (using the current NLS locale settings). For example, if the NLS TimeSeparator is set to “^”, the value for *tochar-expression* can be in the format hh^mm^ss and can be a string.

In all of these cases, the value for *format* must be a string that contains only time format codes:

Format Code	Meaning
HH	Hour of Day (1 through 12)
HH12	Hour of Day (1 through 12)
HH24	Hour of Day (0 through 23)
MI	Minute (0 through 59)
SS	Second (0 through 59)
SSSSS	Seconds since midnight (0 through 86388)
AM / PM	Meridian Indicator (AM = before noon, PM = after noon). Converts a time value to 12-hour format with the appropriate AM or PM suffix. The returned AM or PM suffix is derived from the time value, not from which format code you specified. In the <i>format</i> you can use either AM or PM; they are functionally identical.

Inclusion of any other *format* code values causes the *tochar-expression* integer to be interpreted as a date.

The following example causes '60871' to be interpreted as the time value 04:54:31 PM:

```
SELECT TO_CHAR('60871', 'HH12:MI:SS PM')
```

The following example converts the time portions of two Logical timestamps to formatted time strings. Note that *format* does not support fractional seconds; fractional seconds in *tochar-expression* are truncated.

```
SELECT TO_CHAR(SYSDATE, 'HH12:MI:SS PM'),
       TO_CHAR(CURRENT_TIMESTAMP(6), 'HH12:MI:SS PM')
```

The following Embedded SQL example converts time values specified in both ODBC standard format and current NLS locale format:

```
SET restore=##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator")
WRITE "Time Separator is = ", restore, !
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeSeparator", "^")
WRITE "Time Separator is now = ", ##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator"), !
&sql(SELECT TO_CHAR('15:35:43.99', 'HH12:MI:SS PM'),
        TO_CHAR('15^35^43.99', 'HH12:MI:SS PM')
        INTO :standard, :local)
WRITE "Converted standard-format time: ", standard, !
WRITE "Converted locale-format time: ", local, !
DO ##class(%SYS.NLS.Format).SetFormatItem("TimeSeparator", restore)
WRITE "Time Separator is = ", ##class(%SYS.NLS.Format).GetFormatItem("TimeSeparator")
```

Timestamp to Formatted Datetime String Conversion

You can use **TO_CHAR** to convert a timestamp to a formatted datetime string. The value for *tochar-expression* must be a valid Logical timestamp value.

The date portion of the timestamp is formatted using the date-to-string conversion *format* codes. The following table lists additional *format* codes for the time portion of the timestamp.

Format Code	Meaning
HH	Hour of Day (1 through 12)
HH12	Hour of Day (1 through 12)
HH24	Hour of Day (0 through 23)
MI	Minute (0 through 59)
SS	Second (0 through 59)
SSSSS	Seconds since midnight (0 through 86388)
AM	Meridian Indicator (before noon)
PM	Meridian Indicator (after noon)

The following example returns the current system date (a timestamp), and the current system date converted for display with two different formats:

```
SELECT SYSDATE,
       TO_CHAR(SYSDATE, 'MM/DD/YYYY HH:MI:SS'),
       TO_CHAR(SYSDATE, 'DD MONTH YYYY at SSSSS seconds')
```

Note that any characters used in the *format* string which are not format codes are just returned in place in the resulting string.

Number-to-String Conversion

You can use **TO_CHAR** to convert a number to a formatted numeric string. The following table lists the valid format codes for the *format* argument for this use of **TO_CHAR**.

If you omit the *format* argument, the input numeric value is evaluated as an integer: leading zeros and a leading plus sign are deleted, a leading minus sign is retained, and the numeric value is truncated at the first nonnumeric character, such as a comma or period. No leading blanks or other formatting is provided.

Format Code	Example	Description
9	9999	Return value with the specified number of digits, with a leading space if positive or with a minus sign if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.
0	09999 99990	Return leading zeros. Return trailing zeros.
\$	\$9999	Return value with a leading dollar sign. Note that the dollar sign is preceded by a blank for positive numbers.
B	B9999	Return blanks for the integer part of a fixed-point number when the integer part is zero (regardless of 0' in the <i>format</i> argument).
S	S9999 9999S	Return negative value with a leading minus sign "-". Return positive value with a leading plus sign "+". Return negative value with a trailing minus sign "-". Return positive value with a trailing plus sign "+".
D	99D99	Return a decimal separator character in the specified position. The DecimalSeparator used is the one defined for the locale. The default is a period ".". Only one "D" is allowed in the <i>format</i> argument.
G	9G999	Return a numeric group separator character in the specified position(s). The NumericGroupSeparator used is the one defined for the locale. The default is a comma ",". No numeric group separators may appear to the right of the decimal separator.
FM	FM90.9	Return a value with no leading or trailing blanks.
,	9,999	Return a comma in the specified position. No comma may appear to the right of the decimal. The <i>format</i> argument may not begin with a comma.
.	99.99	Return a decimal point (that is, a period ".") in the specified position. Only one "." is allowed in the <i>format</i> argument.

Your *format* can specify the decimal separator and the numeric group separator either as a literal character, or as the current value of the locale's DecimalSeparator and NumericGroupSeparator. You can determine the current locale values as follows:

```
WRITE ##class(%SYS.NLS.Format).GetFormatItem("DecimalSeparator"),!  
WRITE ##class(%SYS.NLS.Format).GetFormatItem("NumericGroupSeparator")
```

If the *format* argument contains fewer integer digits than the input numeric expression, **TO_CHAR** does not return a number; instead, it returns a string of two or more pound signs (##). The number of pound signs represents the length of the current *format* argument, plus one.

If the *format* argument contains fewer decimal digits than the input numeric expression, **TO_CHAR** rounds the number to the specified number of decimal digits, or to an integer, if no decimal format is provided.

If *tochar-expression* is null, **TO_CHAR** returns null.

Number-to-String Examples

The following embedded SQL example shows basic number-to-string conversions:

```
&sql(SELECT
  TO_CHAR(1000,'9999'),
  TO_CHAR(10,'9999')
INTO :numfull,:numshort)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Formatted number:",numfull
  WRITE !,"Formatted number:",numshort
  WRITE !,"Note leading blanks" }
```

Returns the specified number with the appropriate number of leading blanks. An unsigned positive number is always preceded by a blank character. Additional leading blanks are provided if the specified number has fewer digits than the *format* argument.

The following embedded SQL example shows the use of separator characters:

```
&sql(SELECT
  TO_CHAR(1000,'9,999.99'),
  TO_CHAR(1000,'9G999D99')
INTO :comma,:groupsep)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Formatted number:",comma
  WRITE !,"Formatted number:",groupsep
  WRITE !,"Note leading blank" }
```

The first **TO_CHAR** returns the string: '1,000.00'. The second **TO_CHAR** may also return this value, but the separator characters displayed depend upon the locale setting.

The following embedded SQL example shows the use of positive and negative signs:

```
&sql(SELECT
  TO_CHAR(10,'99.99'),
  TO_CHAR(-10,'99.99'),
  TO_CHAR(10,'S99.99'),
  TO_CHAR(-10,'S99.99'),
  TO_CHAR(10,'99.99S'),
  TO_CHAR(-10,'99.99S')
INTO :pos,:neg,:poslead,:neglead,:postrail,:negtrail)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Formatted number:",pos
  WRITE !,"Formatted number:",neg
  WRITE !,"Formatted number:",poslead
  WRITE !,"Formatted number:",neglead
  WRITE !,"Formatted number:",postrail
  WRITE !,"Formatted number:",negtrail
  WRITE !,"Note use of leading blank" }
```

Note that a leading blank only appears before a positive number with no sign formatting. No leading blank appears before a negative number, or before any signed number, regardless of the placement of the sign.

The following embedded SQL example show the use of the “FM” format to override the default leading blank for unsigned positive numbers:

```
&sql(SELECT
  TO_CHAR(12345678.90,'99,999,999.99'),
  TO_CHAR(12345678.90,'FM99,999,999.99')
INTO :num,:fmnum)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"Formatted number:",num
  WRITE !,"Formatted number:",fmnum
  WRITE !,"Note leading blank" }
```

The first **TO_CHAR** returns the string: '12,345,678.90'. The second **TO_CHAR** returns the string: '12,345,678.90' (with no leading blank).

The following embedded SQL example show the use of the leading dollar sign:

```
&sql(SELECT
  TO_CHAR(1234567890, '$9G999G999G999'),
  TO_CHAR(1234567890, 'S$9G999G999G999'),
  TO_CHAR(12345678.90, '$99G999G999D99')
INTO :d, :sd, :dD)
IF SQLCODE'=0 {
  WRITE !, "Error code ", SQLCODE }
ELSE {
  WRITE !, "Formatted number:", d
  WRITE !, "Formatted number:", sd
  WRITE !, "Formatted number:", dD
  WRITE !, "Note leading blanks" }
```

The dollar sign is always preceded either by a sign or by a blank character.

The following embedded SQL example shows what happens when the *format* argument contain fewer integer digits than the input numeric value:

```
&sql(SELECT
  TO_CHAR(1234567.89, '9'),
  TO_CHAR(1234567.89, '99'),
  TO_CHAR(1234567.89, '99D99')
INTO :a, :b, :c)
IF SQLCODE'=0 {
  WRITE !, "Error code ", SQLCODE }
ELSE {
  WRITE !, "Formatted number:", a
  WRITE !, "Formatted number:", b
  WRITE !, "Formatted number:", c }
```

Each **TO_CHAR** returns a string of pound signs: “###”, “####”, and “#####”, respectively.

The following embedded SQL example shows what happens when the *format* argument contains fewer decimal (fractional) digits than the input numeric expression:

```
&sql(SELECT
  TO_CHAR(1234567.4999, '9999999.9'),
  TO_CHAR(1234567.91, '9999999')
INTO :a, :b)
IF SQLCODE'=0 {
  WRITE !, "Error code ", SQLCODE }
ELSE {
  WRITE !, "Formatted number:", a
  WRITE !, "Formatted number:", b }
```

The returned numbers are rounded to “1234567.5” and “1234568”, respectively.

See Also

- SQL functions: [CONVERT TO_DATE TO_NUMBER](#)
- ObjectScript functions: [\\$FNUMBER \\$ZDATE](#)

TO_DATE

A date function that converts a formatted string to a date.

```
TO_DATE(date_string[,format])
TODATE(date_string[,format])
```

Arguments

<i>date_string</i>	The string to be converted to a date. A string date expression where the underlying data type is CHAR or VARCHAR2.
<i>format</i>	<i>Optional</i> — A date format string corresponding to <i>date_string</i> . If <i>format</i> is omitted, 'DD MON YYYY' is the default value; this default is configurable.

Description

The names **TO_DATE** and **TODATE** are interchangeable and are supported for Oracle compatibility.

The **TO_DATE** function converts date strings in various formats to a date integer value, with data type DATE. It is used to input dates in various string formats, storing them in a standard internal representation. **TO_DATE** returns a date with the following format:

```
nnnnn
```

Where *nnnnn* is a positive integer between 0 (December 31, 1840) and 2980013 (December 31, 9999), inclusive. This represents a count of days. Time values are ignored.

The default earliest date is December 31, 1840. You can change the DATE data type MINVAL parameter to permit negative integers representing dates prior to December 31, 1840, as described in the [Data Types](#) reference page in this manual. Dates before December 31, 1840 can also be represented using Julian dates, as described below.

This function can also be invoked from ObjectScript using the **TODATE()** method call:

```
$SYSTEM.SQL.TODATE(date_string,format)
```

The **TO_DATE** function can be used in data definition when supplying a default value to a field. For example:

```
CREATE TABLE mytest
(ID NUMBER(12,0) NOT NULL,
End_Year DATE DEFAULT TO_DATE('31-12-2007','DD-MM-YYYY') NOT NULL)
```

For further details on this use of **TO_DATE**, refer to the [CREATE TABLE](#) command.

Related SQL Functions

- **TO_DATE** converts a formatted date string to a date integer.
- **TO_CHAR** performs the reverse operation; it converts a date integer to a formatted date string.
- **TO_TIMESTAMP** converts a formatted date and time string to a standard timestamp.
- **CAST** and **CONVERT** perform DATE data type conversion.

Note: An earlier version of **TO_DATE** supported date integer-to-string conversions that now must be done using the **TO_CHAR** function. Older applications may have to be modified.

Date String

The first argument specifies a date string literal. You can supply a date string of any kind for the input *date_string*. Each character must correspond to the *format* string, with the following exceptions:

- Leading zeros may be included or omitted (with the exception of a *date_string* without separator characters).
- Years may be specified with two digits or four digits.
- Month names may be specified in full or as the first three letters of the name. Only the first three letters must be correct. Month names are not case-sensitive.
- Time values appended to a date are ignored.

Format

The second argument specifies a date format as a string of code characters.

Default Date Format

If you specify no *format*, **TO_DATE** parses the date string using the default format. The supplied default format is DD MON YYYY. For example, '11 Nov 1993'.

This default format is configurable, using the ObjectScript **\$\$SYSTEM.SQL.SetToDateDefaultFormat()** class method. To determine the current setting, call **\$\$SYSTEM.SQL.CurrentSettings()**, which displays the `TO_DATE() Default Format` setting.

Format Elements

A *format* is a string of one or more format elements specified according to the following rules:

- Format elements are not case-sensitive.
- Almost any sequence or number of format elements is permitted.
- Format strings separate their elements with non-alphanumeric separator characters (for example, a space, slash, or hyphen) that match the separator characters in the *date_string*. This use of specified date separator characters does not depend on the DateSeparator defined for your NLS locale.
- The following date format strings do not require separator characters: MMDDYYYY, DDMMYYYY, YYYYMMDD, and YYYYDDMM. The incomplete date format YYYYMM is also supported, and assume a DD value of 01. Note that in these cases leading zeros must be provided for MM and DD values.

The following table lists the valid date format elements for the *format* argument:

Element	Meaning
DD	Two-digit day of month (01-31). Leading zeros are not required, unless <i>format</i> contains no date separator characters.
MM	Two-digit month number (01-12; 01 = January). Leading zeros are not required, unless <i>format</i> contains no date separator characters. In Japanese and Chinese, a month number consists of a numeric value followed by the ideogram for “month”.
MON	Abbreviated name of month, as specified by the MonthAbbr property in the current locale. By default, in English this is the first three letters of the month name. In other locales, month abbreviations may be more than three letters long and/or may not consist of the first letters of the month name. A period character is not permitted. Not case-sensitive.

Element	Meaning
MONTH	Full name of the month, as specified by the MonthName property in the current locale. Not case-sensitive.
YYYY	Four-digit year.
YY	Last two digits of the year. The first 2 digits of a 2-digit year default to 19.
RR / RRRR	Two-digit year to four-digit year conversion. (See below.)
DDD	Day of the year. The count of days since January 1. (See below.)
J	Julian date. (See below.)

A `TO_DATE` *format* can also include a D (day of week number), DY (day of week abbreviation), or DAY (day of week name) element. However, these format elements are not validated or used to determine the return value. For further details on these *format* elements, refer to [TO_CHAR](#).

Date Formats for Single Date Elements

You can specify DD, DDD, MM, or YYYY as a complete date format. Because these *format* strings omit the month, year, or both the month and year, Caché interprets them as referring to the current month and year:

- DD returns the date for the specified day in the current month of the current year.
- DDD returns the date for the specified day of the year in the current year.
- MM returns the date for the first day of the specified month in the current year.
- YYYY - returns the date for the first day of the current month of the specified year.

The following Embedded SQL examples show these formats:

```

NEW SQLCODE
&sql(
SELECT
    TO_DATE('300','DDD'),
    TO_DATE('24','DD')
INTO :a,:b)
IF SQLCODE=0 {
WRITE "DDD format: ",a," = ",$ZDATE(a,1,,4),!
WRITE "DD format: ",b," = ",$ZDATE(b,1,,4) }
ELSE { WRITE "error:",SQLCODE }

NEW SQLCODE
&sql(
SELECT
    TO_DATE('8','MM'),
    TO_DATE('2004','YYYY')
INTO :a,:b)
IF SQLCODE=0 {
WRITE "MM format: ",a," = ",$ZDATE(a,1,,4),!
WRITE "YYYY format: ",b," = ",$ZDATE(b,1,,4),!
WRITE "done" }
ELSE { WRITE "error:",SQLCODE }

```

Two-Digit Year Conversion (RR and RRRR formats)

The YY format converts a two-digit year value to four digits by simply appending 19. Thus 07 becomes 1907 and 93 becomes 1993.

The RR format provides more flexible two-digit to four-digit year conversion. This conversion is based on the current year. If the current year is in the first half of a century (for example, 2000 through 2050), two-digit years from 00 through 49 are expanded to a four-digit year in the current century, and two-digit years from 50 through 99 are expanded to a four-digit year in the previous century. If the current year is in the second half of a century (for example, 2050 through 2099), all two-digit years are expanded to a four-digit year in the current century. This expansion of two-digit years to four-digit years is shown in the following Embedded SQL example:

```

NEW SQLCODE
&sql(SELECT
  TO_DATE('29 September 00','DD MONTH RR'),
  TO_DATE('29 September 08','DD MONTH RR'),
  TO_DATE('29 September 49','DD MONTH RR'),
  TO_DATE('29 September 50','DD MONTH RR'),
  TO_DATE('29 September 77','DD MONTH RR')
  INTO :a,:b,:c,:d,:e)
IF SQLCODE=0 {
  WRITE a," = ",$ZDATE(a,1,,4),!
  WRITE b," = ",$ZDATE(b,1,,4),!
  WRITE c," = ",$ZDATE(c,1,,4),!
  WRITE d," = ",$ZDATE(d,1,,4),!
  WRITE e," = ",$ZDATE(e,1,,4) }
ELSE { WRITE "error:",SQLCODE }

```

The RRRR format permits you to input a mix of two-digit and four-digit years. Four-digit years are passed through unchanged (the same as YYYY). Two-digit years are converted to four-digit years, using the RR format algorithm. This is shown in the following Embedded SQL example:

```

NEW SQLCODE
&sql(SELECT
  TO_DATE('29 September 2008','DD MONTH RRRR'),
  TO_DATE('29 September 08','DD MONTH RRRR'),
  TO_DATE('29 September 1949','DD MONTH RRRR'),
  TO_DATE('29 September 49','DD MONTH RRRR'),
  TO_DATE('29 September 1950','DD MONTH RRRR'),
  TO_DATE('29 September 50','DD MONTH RRRR')
  INTO :a,:b,:c,:d,:e,:f)
IF SQLCODE=0 {
  WRITE a," 4-digit = ",$ZDATE(a,1,,4),!
  WRITE b," 2-digit = ",$ZDATE(b,1,,4),!
  WRITE c," 4-digit = ",$ZDATE(c,1,,4),!
  WRITE d," 2-digit = ",$ZDATE(d,1,,4),!
  WRITE e," 4-digit = ",$ZDATE(e,1,,4),!
  WRITE f," 2-digit = ",$ZDATE(f,1,,4) }
ELSE { WRITE "error:",SQLCODE }

```

Day of the Year (DDD format)

You can use DDD to convert the day of the year (number of days elapsed since January 1) to an actual date. The format string DDD YYYY must be paired with a corresponding *date_string* consisting of an integer number of days and a four-digit year. (Two-digit years must be specified as RR (not YY) when used with DDD.) The format string DDD defaults to the current year. The number of elapsed days must be a positive integer in the range 1 through 365 (366 if YYYY is a leap year). The four-digit year must be within the standard Caché date range: 1841 through 9999. The DDD and YYYY format elements can be specified in any order; a separator character between them is mandatory. The following example shows this use of Day of the Year:

```

NEW SQLCODE
&sql(SELECT TO_DATE('2008:60','YYYY:DDD')
  INTO :a)
IF SQLCODE=0 {WRITE a," = ",$ZDATE(a,1,,4) }
ELSE { WRITE "error:",SQLCODE }

```

If a format string contains both a DD and a DDD element, the DDD element is dominant. This is shown in the following example, which returns 2/29/2008 (not 12/31/2008):

```

NEW SQLCODE
&sql(SELECT TO_DATE('2008-12-31-60','YYYY-MM-DD-DDD')
  INTO :a)
IF SQLCODE=0 {WRITE a," = ",$ZDATE(a,1,,4) }
ELSE { WRITE "error:",SQLCODE }

```

TO_DATE permits you to return a date expression corresponding to a day of the year. **TO_CHAR** permits you to return the day of the year corresponding to a date expression.

Julian Dates (J format)

In Caché SQL, a Julian date can be used for any date before December 31, 1840. Because Caché represents this date internally as 0, special syntax is needed to represent earlier dates. **TO_DATE** provides a *format* of 'J' (or 'j') for this purpose.

Julian date conversion converts a seven-digit internal numeric value (a Julian day count) to a display-format or ODBC-format date. For example:

```
NEW SQLCODE
&sql(SELECT TO_DATE(2300000,'J')
      INTO :a)
IF SQLCODE=0 {WRITE a }
ELSE { WRITE "error:",SQLCODE }
```

returns the following date: 1585-01-31 (ODBC format) or 01/31/1585 (display format). Julian day count 1721424 returns January 1st of the Year 1 (1-01-01). Julian day counts such as 1709980 (battle of Actium marks beginning of Roman Empire under Augustus Caesar) return BCE (BC) dates, which are displayed with the year preceded by a minus sign.

Note: By default, the %Date data type does not represent dates prior to [December 31, 1840](#). However, you can redefine the MINVAL parameter for this data type to permit representation of earlier dates as negative integers, with the limit of January 1, Year 1. This representation of dates as negative integers is not compatible with the “Julian” date format described here. For further details refer to the [Data Types](#) reference page in this manual.

A Julian day count is always represented internally as a seven-digit number, with leading zeros when necessary. **TO_DATE** allows you to input a Julian day count without the leading zeros. The highest permitted Julian date is 5373484, it returns 12/31/9999. The lowest permitted Julian date is 0000001, it returns 01/01/-4712 (which is BCE date 01/01/-4713). Any value outside this range generates an SQLCODE -400 error, with a %msg value of “Invalid Julian Date value. Julian date must be between 1 and 5373484”.

Note: The following consideration should not affect the interconversion of dates and Julian day counts using **TO_CHAR** and **TO_DATE**. It may affect some calculations made using Julian day counts.

Julian day counts prior to 1721424 (1/1/1) are compatible with other software implementations, such as Oracle. They are *not* identical to BCE dates in ordinary usage. In ordinary usage, there is no Year 0; dates go from 12/31/-1 to 1/1/1. In Oracle usage, the Julian dates 1721058 through 1721423 are simply invalid, and return an error. In Caché these Julian dates return the non-existent Year 0 as a place holder. Thus calculations involving BCE dates must be adjusted by one year to correspond to common usage.

Also be aware that these date counts do not take into account changes in date caused by the Gregorian calendar reform (enacted October 15, 1582, but not adopted in Britain and its colonies until 1752).

TO_DATE permits you to return a date expression corresponding to a Julian day count. **TO_CHAR** permits you to return a Julian day count corresponding to a date expression, as shown in the following example:

```
SELECT
  TO_CHAR('1776-07-04','J') AS JulianCount,
  TO_DATE(2369916,'J') AS JulianDate
```

Examples

Default Date Format Examples

The following embedded SQL example specifies date strings that are parsed using the default date format. Both of these are converted to the DATE data type internal value of 60537:

```
NEW SQLCODE
&sql(SELECT
      TO_DATE('29 September 2006'),
      TO_DATE('29 SEP 2006')
      INTO :a,:b)
IF SQLCODE=0 {WRITE a,! ,b }
ELSE { WRITE "error:",SQLCODE }
```

The following embedded SQL example specifies date strings with two-digit years with *format* default. Note that two-digit years default to 1900 through 1999. Thus, the internal DATE value is 24012:

```

NEW SQLCODE
&sql(SELECT
    TO_DATE('29 September 06'),
    TO_DATE('29 SEP 06')
    INTO :a,:b)
IF SQLCODE=0 {WRITE a,! ,b }
ELSE { WRITE "error:",SQLCODE }

```

Specified Date Format Examples

The following embedded SQL example specifies date strings in various formats. All of these are converted to the DATE data type internal value of 60810.

```

NEW SQLCODE
&sql(SELECT
    TO_DATE('2007 Jun 29','YYYY MON DD'),
    TO_DATE('JUNE 29, 2007','month dd, YYYY'),
    TO_DATE('2007**06**29','YYYY**MM**DD'),
    TO_DATE('06/29/2007','MM/DD/YYYY')
    INTO :a,:b,:c,:d)
IF SQLCODE=0 {WRITE !,a,! ,b,! ,c,! ,d }
ELSE { WRITE "error:",SQLCODE }

```

The following embedded SQL example specifies date formats that do not require element separators. They return the date internal value of 60810:

```

NEW SQLCODE
&sql(SELECT
    TO_DATE('06292007','MMDYYYY'),
    TO_DATE('29062007','DDMMYYYY'),
    TO_DATE('20072906','YYYYDDMM'),
    TO_DATE('20070629','YYYYMMDD')
    INTO :a,:b,:c,:d)
IF SQLCODE=0 {WRITE !,a,! ,b,! ,c,! ,d }
ELSE { WRITE "error:",SQLCODE }

```

The following example specifies the YYYYMM date format. It does not require element separators. It supplies 01 for the missing day element, returning the date 60782 (June 1, 2007):

```

NEW SQLCODE
&sql(SELECT TO_DATE('200706','YYYYMM')
    INTO :a )
IF SQLCODE=0 {WRITE a," = ",$ZDATE(a,1,,4) }
ELSE { WRITE "error:",SQLCODE }

```

See Also

- SQL functions: [CAST](#), [CONVERT](#), [TO_CHAR](#), [TO_TIMESTAMP](#)
- ObjectScript functions: [\\$ZDATE](#), [\\$ZDATEH](#)

TO_NUMBER

A string function that converts a string expression to a value of NUMERIC data type.

```
TO_NUMBER(string-expression)
```

```
TONUMBER(string-expression)
```

Arguments

<i>string-expression</i>	The string expression to be converted. The expression can be the name of a column, a string literal, or the result of another function, where the underlying data type is of type CHAR or VARCHAR2.
--------------------------	---

Description

The names **TO_NUMBER** and **TONUMBER** are interchangeable. They are supported for Oracle compatibility.

TO_NUMBER converts *string-expression* to a number of data type NUMERIC. However, if *string-expression* is of data type DOUBLE, **TO_NUMBER** returns a number of data type DOUBLE.

TO_NUMBER conversion takes a numeric string and converts it to a canonical number by resolving plus and minus signs, expanding exponential notation ("E" or "e"), and removing leading zeros. **TO_NUMBER** halts conversion when it encounters a nonnumeric character (such as a letter or a numeric group separator). Thus the string '7dwarves' converts to 7. If the first character of *string-expression* is a nonnumeric string, **TO_NUMBER** returns 0. If *string-expression* is an empty string ("), **TO_NUMBER** returns 0. **TO_NUMBER** resolves -0 to 0. **TO_NUMBER** does not resolve arithmetic operations. Thus the string '2+4' converts to 2. If NULL is specified for *string-expression*, **TO_NUMBER** returns null.

The NUMERIC data type has a default SCALE of 2. Therefore, when selecting this value in DISPLAY mode, **TO_NUMBER** always displays the return value with 2 decimal places. Additional fractional digits are rounded to two decimal places; trailing zeros are resolved to two decimal places. When **TO_NUMBER** is used via xDBC, it also returns the type as NUMERIC with a SCALE of 2. In LOGICAL mode or ODBC mode, the returned value is a canonical number; no scale is imposed on fractional digits and trailing zeros are omitted.

Related SQL Functions

- **TO_NUMBER** converts a string to a number of data type NUMERIC.
- **TO_CHAR** performs the reverse operation; it converts a number to a string.
- **CAST** and **CONVERT** can be used to convert a string to a number of any data type. For example, you can convert a string to a number of data type INTEGER.
- **TO_DATE** converts a formatted date string to a date integer.
- **TO_TIMESTAMP** converts a formatted date and time string to a standard timestamp.

Examples

The following two examples show how **TO_NUMBER** converts a string to a number, then returns it as data type NUMERIC with appropriate SCALE. The first example returns the number in Display mode, the second example returns the number in Logical mode:

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT TO_NUMBER('+-0123.0093degrees')"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=2
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display() // Display mode value: 123.01
```

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT TO_NUMBER('+-0123.0093degrees')"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display() // Logical mode value: 123.0093
```

The following examples show that when *string-expression* is of data type **DOUBLE**, **TO_NUMBER** returns the value as data type **DOUBLE**:

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT TO_NUMBER(CAST('+-0123.0093degrees' AS DOUBLE))"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=2
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display() // Display mode value
```

```
ZNSPACE "SAMPLES"
SET myquery = "SELECT TO_NUMBER(CAST('+-0123.0093degrees' AS DOUBLE))"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display() // Logical mode value
```

The following example shows how to use **TO_NUMBER** to list street addresses ordered in ascending numerical order:

```
SELECT Home_Street,Name
FROM Sample.Person
ORDER BY TO_NUMBER(Home_Street)
```

Compare the results with the same data ordered in ascending string order:

```
SELECT Home_Street,Name
FROM Sample.Person
ORDER BY Home_Street
```

See Also

- [TO_CHAR](#)
- [TO_DATE](#)

TO_TIMESTAMP

A date function that converts a formatted string to a timestamp.

```
TO_TIMESTAMP(date_string[,format])
```

Arguments

<i>date_string</i>	A string expression to be converted to a timestamp. This expression may contain a date value, a time value, or a date and time value.
<i>format</i>	<i>Optional</i> — A date and time format string corresponding to <i>date_string</i> . If omitted, defaults to <code>DD MON YYYY HH:MI:SS</code> .

Description

The **TO_TIMESTAMP** function converts date and time strings in various formats to a standard timestamp, with data type **TIMESTAMP**. **TO_TIMESTAMP** returns a timestamp with the following format:

```
YYYY-mm-dd hh:mm:ss
```

with leading zeroes always included. Time is specified using a 24-hour clock. By default, a returned timestamp does not include fractional seconds.

If *date_string* omits components of the timestamp, **TO_TIMESTAMP** supplies the missing components. If both *date_string* and *format* omit the year, *yyyy* defaults to the current year; if only *date_string* omits the year, it defaults to 00, which is expanded to a four-digit year according to the year *format* element. If a day or month value is omitted, *dd* defaults to 01; *mm-dd* defaults to 01-01. A missing time component defaults to 00. Fractional seconds are supported, but must be explicitly specified; no fractional seconds are provided by default.

TO_TIMESTAMP supports conversion of two-digit years to four digits. **TO_TIMESTAMP** supports conversion of 12-hour clock time to 24-hour clock time. It provides range validation of date and time element values, including leap year validation. Range validation violations generate an **SQLCODE -400** error.

This function can also be invoked from ObjectScript using the **TOTIMESTAMP()** method call:

```
$SYSTEM.SQL.TOTIMESTAMP(date_string,format)
```

The **TO_TIMESTAMP** function can be used in data definition when supplying a default value to a timestamp field. For example:

```
CREATE TABLE Sample.MyEmpReviews
(EmpNum INTEGER UNIQUE NOT NULL,
 ReviewDate TIMESTAMP DEFAULT TO_TIMESTAMP(365,'DDD'))
```

In this example, the user inserting a record can either supply a *ReviewDate* value, supply no *ReviewDate* value and get the default timestamp of the 365th day of the current year, or supply a *ReviewDate* of **NULL** and get **NULL**.

TO_TIMESTAMP can be used with the **CREATE TABLE** or **ALTER TABLE ADD COLUMN** statements. Only a literal value for *date_string* can be used in this context. For further details, refer to the [CREATE TABLE](#) command.

Related SQL Functions

- **TO_TIMESTAMP** converts a formatted date and time string to a standard timestamp.
- **TO_CHAR** performs the reverse operation; it converts a standard timestamp to a formatted date and time string.
- **TO_DATE** converts a formatted date string to a date integer.
- **CAST** and **CONVERT** perform **TIMESTAMP** data type conversion.

Date and Time String

The *date_string* argument specifies a date and time string literal. If you supply a date string with no time component, **TO_TIMESTAMP** supplies the time value 00:00:00. If you supply a time string with no date component, **TO_TIMESTAMP** supplies the date of 01-01 (January 1) of the current year.

You can supply a date and time string of any kind for the input *date_string*. Each *date_string* character must correspond to the *format* string, with the following exceptions:

- Leading zeros may be included or omitted (with the exception a *date_string* without separator characters).
- Years may be specified with two digits or four digits.
- Month abbreviations (with *format* MON) must match the month abbreviation for that locale. For some locales, a month abbreviation may not be the initial sequential characters of the month name. Month abbreviations are not case-sensitive.
- Month names (with *format* MONTH) should be specified as full month names. However, **TO_TIMESTAMP** does not require full month names with *format* MONTH; it accepts the initial character(s) of the full month name and selects the first month in the month list that corresponds to that initial letter sequence. Therefore, in English, “J” = “January”, “Ju” = “June”, “Jul” = “July”. All characters specified must match the sequential characters of the full month name; characters beyond the full month name are not checked. For example, “Fe”, “Febru”, and “FebruaryLeap” are all valid values; “Febs” is not a valid value. Month names are not case-sensitive.
- Time values can be input with the time separator characters defined for the locale. The output timestamp always represents the time value with the ODBC standard time separator characters: colon (:) for hours, minutes, and seconds, and period (.) for fractional seconds. An omitted time element defaults to zeroes. By default, a timestamp is returned without fractional seconds.

Format

A *format* is a string of one or more format elements specified according to the following rules:

- Format elements are not case-sensitive.
- Almost any sequence or number of format elements is permitted.
- Format strings separate their elements with non-alphanumeric separator characters (for example, a space, slash, or hyphen) that match the separator characters in the *date_string*. These separator characters do not appear in the output string, which uses standard timestamp separators: hyphens for date values, colons for time values, and a period (when required) for fractional seconds. This use of separator characters does not depend on the DateSeparator defined for your NLS locale.
- The following date format strings do not require separator characters: MMDDYYYY, DDMYYYYY, YYYYMMDDHHMISS, YYYYMMDDHHMI, YYYYMMDDHH, YYYYMMDD, YYYYDDMM, HHMISS, and HHMI. The incomplete date format YYYYMM is also supported, and assume a DD value of 01. Note that in these cases leading zeros must be provided for all elements (such as MM and DD), with the exception of the final element.
- Characters in *format* that are not valid format elements are ignored.

Format Elements

The following table lists the valid date format elements for the *format* argument:

Element	Meaning
DD	Two-digit day of month (01-31). Leading zeros are not required, unless <i>format</i> contains no date separator characters.

Element	Meaning
MM	Two-digit month number (01-12; 01 = January). Leading zeros are not required, unless <i>format</i> contains no date separator characters. In Japanese and Chinese, a month number consists of a numeric value followed by the ideogram for “month”.
MON	Abbreviated name of month, as specified by the MonthAbbr property in the current locale. By default, in English this is the first three letters of the month name. In other locales, month abbreviations may be more than three letters long and/or may not consist of the first letters of the month name. A period character is not permitted. Not case-sensitive.
MONTH	Full name of the month, as specified by the MonthName property in the current locale. Not case-sensitive.
YYYY	Four-digit year.
YY	Last two digits of the year. The first 2 digits of a YY 2-digit year default to 19.
RR / RRRR	Two-digit year to four-digit year conversion. (See below.)
DDD	Day of the year. The number of days since January 1. (See below.)
HH	Hour, specified as either 01–12 or 00–23, depending on whether a meridian indicator (AM or PM) is specified. Can be specified as HH12 or HH24.
MI	Minute, specified as 00–59.
SS	Second, specified as 00–59.
FF	Fractions of a second. The two-letter element FF indicates that one or more fractional digits are returned. The actual number of fractional digits returned is specified in <i>date_string</i> , which can specify any number of fractional digits. TO_TIMESTAMP returns exactly the fractional value explicitly supplied in <i>date_string</i> ; trailing zeroes are neither padded nor truncated. You must provide a decimal separator format character. Fractional seconds are ignored if this format element is not the two-letter element FF. By default, a Caché timestamp does not include fractional seconds.
AM / PM	Meridian indicator, specifies a 12-hour clock. (See below.)
A.M. / P.M.	Meridian indicator (with periods), specifies a 12-hour clock. (See below.)

A **TO_TIMESTAMP** *format* can also include a D (day of week number), DY (day of week abbreviation), or DAY (day of week name) element to match the input *date_string*. However, these format elements are not validated or used to determine the return value. For further details on these *format* elements, refer to [TO_CHAR](#).

Two-Digit Year Conversion (RR and RRRR formats)

The RR format provides two-digit to four-digit year conversion. This conversion is based on the current year. If the current year is in the first half of a century (for example, 2000 through 2050), two-digit years from 00 through 49 are expanded to a four-digit year in the current century, and two-digit years from 50 through 99 are expanded to a four-digit year in the previous century. If the current year is in the second half of a century (for example, 2050 through 2099), all two-digit years are expanded to a four-digit year in the current century. This expansion of two-digit years to four-digit years is shown in the following example:

```
SELECT TO_TIMESTAMP('29 September 00', 'DD MONTH RR'),
       TO_TIMESTAMP('29 September 08', 'DD MONTH RR'),
       TO_TIMESTAMP('29 September 49', 'DD MONTH RR'),
       TO_TIMESTAMP('29 September 50', 'DD MONTH RR'),
       TO_TIMESTAMP('29 September 77', 'DD MONTH RR')
```

The RRRR format permits you to input a mix of two-digit and four-digit years. Four-digit years are passed through unchanged (the same as YYYY). Two-digit years are converted to four-digit years, using the RR format algorithm. This is shown in the following example:

```
SELECT TO_TIMESTAMP('29 September 2008','DD MONTH RRRR')AS FourDigit,
       TO_TIMESTAMP('29 September 08','DD MONTH RRRR') AS TwoDigit,
       TO_TIMESTAMP('29 September 1949','DD MONTH RRRR') AS FourDigit,
       TO_TIMESTAMP('29 September 49','DD MONTH RRRR') AS TwoDigit,
       TO_TIMESTAMP('29 September 1950','DD MONTH RRRR') AS FourDigit,
       TO_TIMESTAMP('29 September 50','DD MONTH RRRR') AS TwoDigit
```

Day of the Year (DDD format)

You can use DDD to convert the day of the year (number of days elapsed since January 1) to an actual date. The format string DDD YYYY must be paired with a corresponding *date_string* consisting of an integer number of days and a four-digit year. (Two-digit years must be specified as RR (not YY) when used with DDD.) The format string DDD defaults to the current year. The number of elapsed days must be a positive integer in the range 1 through 365 (366 if YYYY is a leap year). The four-digit year must be within the standard Caché date range: 1841 through 9999. (If you omit the year, it defaults to the current year.) The DDD and year (YYYY, RRRR, or RR) format elements can be specified in any order; a separator character between them is mandatory; this separator can be a blank space. The following example shows this use of Day of the Year:

```
SELECT TO_TIMESTAMP('2008:160','YYYY:DDD')
```

If a format string contains both a DD and a DDD element, the DDD element is dominant. This is shown in the following example, which returns 2008-02-29 00:00:00 (not 2008-12-31 00:00:00):

```
SELECT TO_TIMESTAMP('2008-12-31-60','YYYY-MM-DD-DDD')
```

TO_TIMESTAMP permits you to return a date expression corresponding to a day of the year. **TO_CHAR** permits you to return the day of the year corresponding to a date expression.

Dates Before 1841

TO_TIMESTAMP cannot represent a date before December 31, 1840. Attempted to input such a date results in an SQL-CODE -400 error.

TO_DATE provides a Julian date format than can represent dates back to January 1, 4712 BCE. Julian date conversion converts a seven-digit internal positive integer value (a Julian day count) to a display-format or ODBC-format date. Time values are not supported for Julian dates.

12-Hour Clock Time

A **TIMESTAMP** always represents time using a 24-hour clock. A *date_string* may represent time using a 12-hour clock or a 24-hour clock. **TO_TIMESTAMP** assumes a 24-hour clock, unless one of the following applies:

- The *date_string* time value is followed by 'am' or 'pm' (with no periods). These meridian indicators are not case-sensitive, and may be appended to the time value, or be separated from it by one or more spaces.
- The *format* follows the time format with an 'a.m.' or 'p.m.' element (either one), separated from the time format by one or more spaces. For example: DD MON YYYY HH:MI:SS.FF P.M. This *format* supports 12-hour clock *date_string* values such as 2:23pm, 2:23:54.6pm, 2:23:54 pm, 2:23:54 p.m., and 2:23:54 (assumed to be AM). Meridian indicators are not case-sensitive. When using a meridian indicator with periods, it must be separated from the time value by one or more spaces.

Examples

The following embedded SQL example specifies date strings in various formats. The first one uses the default format, the others specify a *format*. All of these convert *date_string* to the timestamp value of 2007-06-29 00:00:00:

```

&sql(SELECT
  TO_TIMESTAMP('29 JUN 2007'),
  TO_TIMESTAMP('2007 Jun 29','YYYY MON DD'),
  TO_TIMESTAMP('JUNE 29, 2007','month dd, YYYY'),
  TO_TIMESTAMP('2007**06**29','YYYY**MM**DD'),
  TO_TIMESTAMP('06/29/2007','MM/DD/YYYY'),
  TO_TIMESTAMP('29/6/2007','DD/MM/YYYY')
  INTO :a,:b,:c,:d,:e,:f)
IF SQLCODE=0 { WRITE !,a!,b!,c!,d!,e!,f }
ELSE { WRITE "SQLCODE error:",SQLCODE }

```

The following example specifies the YYYYMM date format. It does not require element separators. **TO_TIMESTAMP** supplies the missing day and time values:

```
SELECT TO_TIMESTAMP('200706','YYYYMM')
```

This example returns the timestamp 2007-06-01 00:00:00.

The following example specifies just the HH:MI:SS.FF time format. **TO_TIMESTAMP** supplies the missing date value. In each case, this example returns the date of 2017-01-01 (where 2017 is the current year):

```

SELECT TO_TIMESTAMP('11:34','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.00','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.7','HH:MI:SS.FF'),
  TO_TIMESTAMP('11:34:22.7000000','HH:MI:SS.FF')

```

Note that fractional seconds are passed through exactly as specified, with no padding or truncation.

The following example shows another way to specify a time format with fractional seconds:

```
SELECT TO_TIMESTAMP('113422.9678','HHMISS.FF')
```

See Also

- SQL commands: [CREATE TABLE](#), [ALTER TABLE](#)
- SQL functions: [CAST](#), [CONVERT](#), [TO_CHAR](#), [TO_DATE](#), [TO_NUMBER](#)
- ObjectScript functions: [\\$ZDATE](#) [\\$ZDATEH](#)

\$TRANSLATE

A string function that performs character-for-character replacement.

```
$TRANSLATE(string, identifier[, associator])
```

Arguments

<i>string</i>	The target string. It can be a field name, a literal, a host variable, or an SQL expression.
<i>identifier</i>	The character(s) to search for in <i>string</i> . It can be a string or numeric literal, a host variable, or an SQL expression.
<i>associator</i>	<i>Optional</i> — The replacement character(s) corresponding to each character in the <i>identifier</i> . It can be a string or numeric literal, a host variable, or an SQL expression.

Description

The **\$TRANSLATE** function performs character-for-character replacement within a return value string. It processes the *string* argument one character at a time. It compares each character in *string* with each character in the *identifier* argument. If **\$TRANSLATE** finds a match, it makes note of the position of that character.

- The two-argument form of **\$TRANSLATE** removes all instances of the characters in the *identifier* argument from the output string.
- The three-argument form of **\$TRANSLATE** replaces all instances of each *identifier* character found in the *string* with the positionally corresponding *associator* character. Replacement is performed on a character, not a string, basis. If the *identifier* argument contains more characters than the *associator* argument, the excess characters in the *identifier* argument are deleted from the output string. If the *identifier* argument contains fewer characters than the *associator* argument, the excess character(s) in the *associator* argument are ignored.

SQLCODE -380 is issued if you specify too few arguments. SQLCODE -381 is issued if you specify too many arguments.

\$TRANSLATE is case-sensitive.

\$TRANSLATE cannot be used to replace NULL with a character.

\$TRANSLATE and REPLACE

\$TRANSLATE performs character-for-character matching and replacement. **REPLACE** performs string-for-string matching and replacement. **REPLACE** can replace a single specified substring of one or more characters with another substring, or remove multiple instances of a specified substring. **\$TRANSLATE** can replace multiple specified characters with corresponding specified replacement characters.

By default, both functions are case-sensitive, start at the beginning of *string*, and replace all matching instances. **REPLACE** has arguments that can be used to change these defaults.

Examples

In the following example, a two-argument **\$TRANSLATE** modifies Name values by removing punctuation (commas, spaces, periods, apostrophes, hyphens), returning names that consist of only alphabetic characters. Note that the *identifier* doubles the apostrophe to escape it as a literal character, rather than a string delimiter:

```
SELECT TOP 20 Name, $TRANSLATE(Name, ', . '' -') AS AlphaName
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

In the following example, a three-argument **\$TRANSLATE** modifies Name values by replacing commas and spaces with caret (^) characters, returning names delimited in three pieces (surname, first name, middle initial). Note that the *associator* must specify “^” as many times as the number of characters in *identifier*:

```
SELECT TOP 20 Name,$TRANSLATE(Name,', ','^^^^') AS PiecesNamePunc
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

In the following example, a three-argument **\$TRANSLATE** modifies Name values by both replacing commas and spaces with caret (^) characters (specified in the *identifier* and *associator*) and removing periods, apostrophes, and hyphens (specified in the *identifier*, omitted from the *associator*):

```
SELECT TOP 20 Name,$TRANSLATE(Name,', .''-','^^^^') AS PiecesNameNoPunc
FROM Sample.Person
WHERE Name %STARTSWITH 'O'
```

See Also

- [REPLACE](#) function
- [STUFF](#) function
- [String Manipulation](#)

TRIM

A string function that returns a character string with leading and/or trailing characters removed.

```
TRIM([end_keyword] [string-expression-1] FROM string-expression-2)
```

Arguments

<i>end_keyword</i>	<i>Optional</i> — A keyword specifying the which end of <i>string-expression-2</i> to strip. Available values are LEADING, TRAILING, BOTH. The default is BOTH.
<i>string-expression-1</i>	<i>Optional</i> — The string expression specifying the characters to strip from <i>string-expression-2</i> . Every instance of the specified character is stripped. Thus 'abc' strips 'bbbaacaaa'. If not specified, TRIM strips spaces.
<i>string-expression-2</i>	The string expression which will be stripped. Both string expressions can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2). The FROM keyword is required if either <i>end_keyword</i> or <i>string-expression-1</i> is specified. If neither of these arguments are specified, no FROM keyword is specified.

Description

TRIM strips the specified characters from the beginning or end of a supplied value. By default, stripping of letters is case-sensitive.

The optional *end_keyword* argument can take the following values:

LEADING	A keyword that specifies that the characters in <i>string-expression-1</i> are to be removed from the beginning of <i>string-expression-2</i> .
TRAILING	A keyword that specifies that the characters in <i>string-expression-1</i> are to be removed from the end of <i>string-expression-2</i> .
BOTH	A keyword that specifies that the characters in <i>string-expression-1</i> are to be removed from both the beginning and end of <i>string-expression-2</i> . BOTH is the default and is used if no <i>end_keyword</i> is specified.

You can use **LTRIM** to trim leading blanks, or **RTRIM** to trim trailing blanks.

To pad a string with leading or trailing blanks or other characters, use **LPAD** or **RPAD**.

NULL and Empty String

TRIM returns NULL if either string expression is NULL.

TRIM returns an empty string if *string-expression-2* is the empty string, or if **TRIM** strips all of the characters from *string-expression-2*.

Examples

The following example uses the *end_keyword* and *string-expression-1* defaults; it removes leading and trailing blanks from "abc":

```
SELECT TRIM('  abc  ') AS Trimmed
```

The following example removes the character "x" from the beginning of the string "xxxabcxxx", resulting in "abcxxx":

```
SELECT TRIM(LEADING 'x' FROM 'xxxabcxxx') AS Trimmed
```

The following example removes the character "x" from the beginning and end of "xxxabcxxx", resulting in "abc":

```
SELECT TRIM(BOTH 'x' FROM 'xxxabcxxx') AS Trimmed
```

The following example removes all instances of the characters "xyz" from the end of "abcxyz", resulting in "abc":

```
SELECT TRIM(TRAILING 'xyz' FROM 'abczzxyyyz') AS Trimmed
```

The following example removes the leading letters "B" or "R" from the FavoriteColors values. Note that you must convert a list to a string in order to apply **TRIM**:

```
SELECT TOP 15 Name, FavoriteColors,  
       TRIM(LEADING 'BR' FROM $LISTTOSTRING(FavoriteColors)) AS Trimmed  
FROM Sample.Person
```

See Also

- SQL functions: [LTRIM](#) [RTRIM](#) [LPAD](#) [RPAD](#)
- ObjectScript function: [\\$ZSTRIP](#)

TRUNCATE

A scalar numeric function that truncates a number at a specified number of digits.

```
{fn TRUNCATE(numeric-expr, scale)}
```

Arguments

<i>numeric-expr</i>	The number to be truncated. A number or numeric expression.
<i>scale</i>	An expression that evaluates to an integer that specifies the number of places to truncate, counting from the decimal point. Can be zero, a positive integer, or a negative integer. If <i>scale</i> is a fractional number, Caché rounds it to the nearest integer.

Description

TRUNCATE truncates *numeric-expr* by truncating at the *scale* number of digits from the decimal point. It does not round numbers or add padding zeroes. Leading and trailing zeroes are removed before the **TRUNCATE** operation. **TRUNCATE** returns the same data type as *numeric-expr*.

- If *scale* is a positive number, truncation is performed at that number of digits to the right of the decimal point. If *scale* is equal to or larger than the number of decimal digits, no truncation or zero filling occurs.
- If *scale* is zero, the number is truncated to a whole integer. In other words, truncation is performed at zero digits to the right of the decimal point; all decimal digits and the decimal point itself are truncated.
- If *scale* is a negative number, truncation is performed at that number of digits to the left of the decimal point. If *scale* is equal to or larger than the number of integer digits in the number, zero is returned.
- If *numeric-expr* is zero (however expressed: 00.00, -0, etc.) **TRUNCATE** returns 0 (zero) with no decimal digits, regardless of the *scale* value.
- If *numeric-expr* or *scale* is NULL, **TRUNCATE** returns NULL.

TRUNCATE can only be used as an ODBC scalar function (with the curly brace syntax).

ROUND can be used to perform a similar truncation operation on numbers. **TRIM** can be used to perform a similar truncation operation on strings.

TRUNCATE, ROUND, and \$JUSTIFY

TRUNCATE and **ROUND** are numeric functions that perform similar operations; they both can be used to decrease the number of significant decimal or integer digits of a number. **ROUND** allows you to specify either rounding (the default), or truncation; **TRUNCATE** does not perform rounding. **ROUND** returns the same data type as *numeric-expr*; **TRUNCATE** returns *numeric-expr* as data type NUMERIC, unless *numeric-expr* is data type DOUBLE, in which case it returns data type DOUBLE.

TRUNCATE truncates to a specified number of fractional digits. If the truncation results in trailing zeros, these trailing zeros are preserved. However, if *scale* is larger than the number of fractional decimal digits in the canonical form of *numeric-expr*, **TRUNCATE** does not zero-pad.

ROUND rounds (or truncates) to a specified number of fractional digits, but its return value is always normalized, removing trailing zeros. For example, `ROUND(10.004, 2)` returns 10, not 10.00.

Use **\$JUSTIFY** when rounding to a fixed number of fractional digits is important — for example, when representing monetary amounts. **\$JUSTIFY** returns the specified number of trailing zeros following the rounding operation. When the

number of digits to round is larger than the number of fractional digits, **\$JUSTIFY** zero-pads. **\$JUSTIFY** also right-aligns the numbers, so that the DecimalSeparator characters align in a column of numbers. **\$JUSTIFY** does not truncate.

Examples

The following two examples both truncate a number to two decimal digits. The first (using Dynamic SQL) specifies *scale* as an integer; the second (using Embedded SQL) specifies *scale* as a host variable that resolves to an integer:

```
SET myquery = "SELECT {fn TRUNCATE(654.321888,2)} AS trunc"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()

SET x=2
&sql(SELECT {fn TRUNCATE(654.321888,:x)}
INTO :a)
IF SQLCODE'=0 {
WRITE !,"Error code ",SQLCODE }
ELSE {
WRITE !,"truncated value is: ",a }
```

both examples return 654.32 (truncation to two decimal places).

The following Dynamic SQL example specifies a *scale* larger than the number of decimal digits:

```
SET myquery = "SELECT {fn TRUNCATE(654.321000,9)} AS trunc"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

it returns 654.321 (Caché removed the trailing zeroes before the truncation operation; no truncation or zero padding occurred).

The following Dynamic SQL example specifies a *scale* of zero:

```
SET myquery = "SELECT {fn TRUNCATE(654.321888,0)} AS trunc"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

it returns 654 (all decimal digits and the decimal point are truncated).

The following Dynamic SQL example specifies a negative *scale*:

```
SET myquery = "SELECT {fn TRUNCATE(654.321888,-2)} AS trunc"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

it returns 600 (two integer digits have been truncated and replaced by zeroes; note that no rounding has been done).

The following Dynamic SQL example specifies a negative *scale* as large as the integer portion of the number:

```
SET myquery = "SELECT {fn TRUNCATE(654.321888,-3)} AS trunc"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

it returns 0.

See Also

- SQL functions: [\\$JUSTIFY](#), [ROUND](#), [RTRIM](#), [TRIM](#),
- ObjectScript function: [\\$NORMALIZE](#)

%TRUNCATE

A collation function that truncates a string to the specified length and applies EXACT collation.

```
%TRUNCATE(expression[, length])
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2). <i>expression</i> can be a subquery.
<i>length</i>	<i>Optional</i> — The truncation length, specified as an integer. The initial <i>length</i> characters of <i>expression</i> are returned. If you omit <i>length</i> , %TRUNCATE collation is identical to %EXACT collation. You can enclose <i>length</i> with double parentheses to suppress literal substitution : ((<i>length</i>)).

Description

%TRUNCATE truncates *expression* to the specified length, then returns it in the EXACT collation sequence.

EXACT collation orders pure numeric values (values for which $x=+x$) in numeric order first, followed by all other characters in string collation sequence. The EXACT string collation sequence is the same as the ANSI-standard ASCII collation sequence: digits are collated before uppercase alphabetic characters and uppercase alphabetic characters are collated before lowercase alphabetic characters. Punctuation characters occur at several places in the sequence.

%TRUNCATE passes through NULLs unchanged.

%TRUNCATE is a Caché SQL extension and is intended for SQL lookup queries.

Examples

The following example uses %TRUNCATE to return the first four characters of Name values:

```
SELECT TOP 5 Name, %TRUNCATE(Name, 4) AS ShortName
FROM Sample.Person
```

The following example applies %TRUNCATE to a subquery:

```
SELECT TOP 5 Name, %TRUNCATE((SELECT Name FROM Sample.Company), 10) AS Company
FROM Sample.Person
```

The following example uses %TRUNCATE in the **GROUP BY** clause to create an alphabet list that returns the number of names that begin with each letter:

```
SELECT Name AS FirstLetter, COUNT(Name) AS NameCount
FROM Sample.Person GROUP BY %TRUNCATE(Name, 1) ORDER BY Name
```

The following two examples show how %TRUNCATE performs EXACT collation. The ORDER BY in the first example truncates Home_Street to two characters. Because the first two characters of a street address are almost always numbers, the Home_Street fields are ordered in the numeric sequence of their first two numbers.

```
SELECT Name, Home_Street
FROM Sample.Person
ORDER BY %TRUNCATE(Home_Street, 2)
```

The ORDER BY in the second example truncates Home_Street to four characters. Because the fourth character of some street addresses is not a number (a blank space, for example), the Home_Street values that begin with four (or more) numbers

are ordered first in numeric sequence, then the Home_Street values that contain a non-numeric character within the first four characters are ordered in string sequence:

```
SELECT Name,Home_Street
FROM Sample.Person
ORDER BY %TRUNCATE(Home_Street,4)
```

See Also

[ASCII](#) [%EXACT](#) [%MVR](#) [%STRING](#) [%UPPER](#)

\$TSQL_NEWID

A function that returns a globally unique ID.

```
$TSQL_NEWID( )
```

Description

\$TSQL_NEWID returns a [globally unique ID \(GUID\)](#). A GUID is used to synchronize databases on occasionally connected systems. A GUID is a 36-character string consisting of 32 hexadecimal digits separated into five groups by hyphens. Its data type is `%Library.UniqueIdentifier`.

\$TSQL_NEWID is provided in Caché SQL to support [Caché Transact-SQL \(TSQL\)](#). The corresponding TSQL function is [NEWID](#).

The **\$TSQL_NEWID** function takes no arguments. Note that the argument parentheses are required.

The `%Library.GUID` abstract class provides support for globally unique IDs, including the **AssignGUID()** method, which can be used to assign a globally unique ID to a class.

Examples

The following example returns a GUID:

```
SELECT $TSQL_NEWID( )
```

See Also

- TSQL: [NEWID function](#)

UCASE

A case-transformation function that converts all lowercase letters in a string to uppercase letters.

```
UCASE(string-expression)
{fn UCASE(string-expression)}
```

Arguments

<i>string-expression</i>	The string whose characters are to be converted to uppercase. The expression can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR).
--------------------------	--

Description

UCASE converts lowercase letters to uppercase for display purposes. It has no effects on non-alphabetic characters; it leaves unchanged numbers, punctuation, and leading or trailing blank spaces.

Note that **UCASE** can be used as an ODBC scalar function (with the curly brace syntax) or as an SQL general function.

UCASE does not force a numeric to be interpreted as a string. Caché SQL removes leading and trailing zeros from numerics. A numeric specified as a string retains leading and trailing zeros.

UCASE does not affect **collation**. The **%SQLUPPER** function is the preferred way in SQL to convert a data value for not case-sensitive collation. Refer to **%SQLUPPER** for further information on case transformation for collation.

This function can also be invoked from ObjectScript using the **UPPER()** method call:

```
$SYSTEM.SQL.UPPER(expression)
```

Examples

The following example returns each person's name in uppercase letters:

```
SELECT Name, {fn UCASE(Name)} AS CapName
FROM Sample.Person
```

UCASE also works on Unicode (non-ASCII) alphabetic characters, as shown in the following Embedded SQL example, which converts Greek letters from lowercase to uppercase:

```
IF $SYSTEM.Version.IsUnicode() {
  SET a=$CHAR(950,949,965,963)
  &sql(SELECT UCASE(:a)
  INTO :b
  FROM Sample.Person)
  IF SQLCODE'=0 {WRITE !,"Error code ",SQLCODE }
  ELSE {WRITE !,a,!,b }
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- SQL functions: [%ALPHAUP](#) [LCASE](#) [%SQLUPPER](#) [UPPER](#) [%UPPER](#)
- ObjectScript function: [\\$ZCONVERT](#)

UNIX_TIMESTAMP

A date/time function that converts a date expression to a UNIX timestamp.

```
UNIX_TIMESTAMP([date-expression])
```

Arguments

<i>date-expression</i>	<i>Optional</i> — An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal. UNIX_TIMESTAMP does not convert from one timezone to another. If <i>date-expression</i> is omitted, defaults to the current UTC timestamp.
------------------------	--

Description

UNIX_TIMESTAMP returns a UNIX® timestamp, the count of seconds (and fractional seconds) since '1970-01-01 00:00:00'.

If you do not specify *date-expression*, *date-expression* defaults to the current UTC timestamp. Therefore, `UNIX_TIMESTAMP()` is equivalent to `UNIX_TIMESTAMP(GETUTCDATE(3))`, assuming the system-wide default precision of 3.

If you specify *date-expression*, **UNIX_TIMESTAMP** converts the specified *date-expression* value to a UNIX timestamp, calculating the count of seconds to that timestamp. **UNIX_TIMESTAMP** can return a positive or negative count of seconds.

UNIX_TIMESTAMP returns its value as data type %Library.Numeric. It can return fractional seconds of precision. If you do not specify *date-expression*, it takes the currently configured system-wide precision. If you specify *date-expression* it takes its precision from *date-expression*.

date-expression Values

The optional *date-expression* can be specified as:

- An ODBC timestamp value (data type %Library.TimeStamp): YYYY-MM-DD HH:MI:SS.FFF
- A \$HOROLOG date value (data type %Library.Date): a count of the number of days since December 31, 1840, where day 1 is January 1, 1841.
- A \$HOROLOG timestamp, with or without fractional seconds: 64412,54736.

UNIX_TIMESTAMP does not perform timezone conversion: if *date-expression* is in UTC time, UTC UnixTime is returned; if *date-expression* is local time, a local UnixTime value is returned.

Fractional Seconds Precision

Fractional seconds are always truncated, not rounded, to the specified precision. A *date-expression* in %Library.TimeStamp data type format can have a maximum precision of nine. The actual number of digits supported is determined by the *date-expression precision* argument, the configured default time precision, and the system capabilities. If you specify a *precision* larger than the configured default time precision, the additional digits of precision are returned as trailing zeros.

Configuring Precision

The default precision can be configured using the following:

- **SET OPTION** with the TIME_PRECISION option.
- The `$$SYSTEM.SQL.SetDefaultTimePrecision()` method call.
- Go to the Management Portal, select [System] > [Configuration] > [General SQL Settings]. View and edit the current setting of **Default time precision for GETDATE(), CURRENT_TIME, and CURRENT_TIMESTAMP**.

Specify an integer 0 through 9 (inclusive) for the default number of decimal digits of precision to return. The default is 0. The actual precision returned is platform dependent; *precision* digits in excess of the precision available on your system are returned as zeroes.

Date and Time Functions Compared

UNIX_TIMESTAMP returns date and time expressed as a number of elapsed seconds from an arbitrary date.

GETUTCDATE returns a universal (independent of time zone) date and time as a %TimeStamp (ODBC timestamp) data type value.

You can also use the ObjectScript **\$ZTIMESTAMP** special variable to return a universal (time zone independent) timestamp.

The ObjectScript **\$ZDATETIME** function *dformat -2* takes a Caché \$HOROLOGY date and returns a UNIX timestamp; **\$ZDATETIMEH** *dformat -2* takes a UNIX timestamp and returns a Caché %HOROLOGY date. These ObjectScript functions convert local time to UTC time. **UNIX_TIMESTAMP** does not convert local time to UTC time.

Examples

The following example returns a UTC UNIX timestamp. The first *select-item* takes the *date-expression* default, the second specifies an explicit UTC timestamp:

```
SELECT
  UNIX_TIMESTAMP() AS DefaultUTC,
  UNIX_TIMESTAMP(GETUTCDATE(3)) AS ExplicitUTC
```

The following example returns a local UNIX timestamp for the current local date and time, and a UTC UNIX timestamp for a UTC date and time value. The first *select-item* specifies the local **CURRENT_TIMESTAMP**, the second specifies **\$HOROLOGY** (local date and time), the third specifies the current UTC date and time:

```
SELECT
  UNIX_TIMESTAMP(CURRENT_TIMESTAMP(2)) AS CurrTSLocal,
  UNIX_TIMESTAMP($HOROLOGY) AS HorologLocal,
  UNIX_TIMESTAMP(GETUTCDATE(3)) AS ExplicitUTC
```

The following example compares **UNIX_TIMESTAMP** (which does not convert local time) and **\$ZDATETIME** (which does convert local time):

```
ZNSPACE "SAMPLES"
SET unixutc=$ZDATETIME($HOROLOGY,-2)
SET myquery = "SELECT UNIX_TIMESTAMP($HOROLOGY) AS UnixLocal,? AS UnixUTC"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(unixutc)
DO rset.%Display()
```

See Also

- SQL concepts: [Data Type](#), [Date and Time Constructs](#)
- SQL timestamp functions: [CAST](#), [CONVERT](#), [GETDATE](#), [GETUTCDATE](#), [NOW](#), [SYSDATE](#), [TIMESTAMPADD](#), [TIMESTAMPDIFF](#), [TO_TIMESTAMP](#)
- ObjectScript: [\\$ZDATETIME](#) and [\\$ZDATETIMEH](#) functions, [\\$HOROLOGY](#) special variable, [\\$ZTIMESTAMP](#) special variable

UPPER

A case-transformation function that converts all lowercase letters in a string expression to uppercase letters.

```
UPPER(expression)
UPPER expression
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
-------------------	--

Description

The **UPPER** function converts all alphabetic characters to uppercase letters. This is the inverse of the **LOWER** function. **UPPER** leaves unchanged numbers, punctuation, and leading or trailing blank spaces.

UPPER does not force a numeric to be interpreted as a string. Caché SQL removes leading and trailing zeros from numerics. A numeric specified as a string retains leading and trailing zeros.

This function can also be invoked from ObjectScript using the **UPPER()** method call:

```
$SYSTEM.SQL.UPPER(expression)
```

UPPER is a standard function for alphabetic case conversion. This should not be confused with the **%UPPER** collation function or UPPER collation, which are deprecated and should not be used for new development. For uppercase collation use **%SQLUPPER**, which provides superior collation of numerics, NULL values and empty strings.

Examples

The following example returns all names, selecting those where the uppercase form of the name starts with “JO”:

```
SELECT Name
FROM Sample.Person
WHERE UPPER(Name) %STARTSWITH UPPER('JO')
```

The following example returns all names in uppercase, selecting those where the name starts with “JO”:

```
SELECT UPPER(Name) AS CapName
FROM Sample.Person
WHERE Name %STARTSWITH UPPER('JO')
```

The following Embedded SQL example converts the lowercase Greek letter Delta to uppercase. This example uses the **UPPER** syntax that uses a space, rather than parentheses, to separate keyword from argument:

```
IF $SYSTEM.Version.IsUnicode() {
  &sql(SELECT UPPER {fn CHAR(948)},{fn CHAR(948)})
  INTO :a,:b
  FROM Sample.Person)
  IF SQLCODE'=0 {WRITE !,"Error code ",SQLCODE }
  ELSE {WRITE !,a!,b }
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- [%SQLUPPER](#) collation function
- [%UPPER](#) collation function
- [%STARTSWITH](#) predicate condition

- [Collation](#) chapter in *Using Caché SQL*

%UPPER

Deprecated. A collation function that converts alphabetic characters to the UPPER collation format.

```
%UPPER(expression)
%UPPER expression
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
-------------------	--

Description

This is a deprecated collation function. Please refer to [%SQLUPPER](#) for new development. **%SQLUPPER** provides superior collation of numerics, NULL values and empty strings.

%UPPER converts all alphabetic characters to uppercase (i.e., the UPPER format). It leaves unchanged numbers, punctuation, and leading and trailing blank spaces.

%UPPER does not force a numeric to be interpreted as a string. Caché SQL removes leading and trailing zeros from numerics. A numeric specified as a string retains leading and trailing zeros.

%UPPER is a Caché SQL extension and is intended for SQL lookup queries.

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",5)
```

This function can also be invoked from ObjectScript using the **UPPER()** method call:

```
WRITE $SYSTEM.SQL.UPPER("The quick, BROWN fox.")
```

Examples

The following example returns all names, selecting those where the uppercase form of the name starts with “JO”:

```
SELECT Name
FROM Sample.Person
WHERE %UPPER(Name) %STARTSWITH %UPPER('JO')
```

The following example returns all names in uppercase, selecting those where the name starts with “JO”:

```
SELECT %UPPER(Name) AS CapName
FROM Sample.Person
WHERE Name %STARTSWITH 'Jo'
```

The following Embedded SQL example converts the lowercase Greek letter Delta to uppercase. This example uses the **%UPPER** syntax that uses a space, rather than parentheses, to separate keyword from argument:

```
IF $SYSTEM.Version.IsUnicode() {
  &sql(SELECT %UPPER {fn CHAR(948)},{fn CHAR(948)})
  INTO :a,:b
  FROM Sample.Person)
  IF SQLCODE'=0 {WRITE !,"Error code ",SQLCODE }
  ELSE {WRITE !,a!,b }
}
ELSE {WRITE "This example requires a Unicode installation of Caché"}
```

See Also

- [CREATE TABLE](#)
- [%STARTSWITH](#) predicate
- [%SQLUPPER](#) collation function
- [%TRUNCATE](#) collation function
- [UPPER](#) function
- [Collation](#) chapter in *Using Caché SQL*

USER

A function that returns the user name of the current user.

```
USER
{fn USER}
{fn USER( )}
```

Description

USER takes no arguments and returns the user name (also referred to as the authorization ID) of the current user. The general function does not allow parentheses; the ODBC scalar function can specify or omit the empty parentheses.

A user name is defined with the **CREATE USER** command.

Typical uses for **USER** are in the **SELECT** statement select list or in the **WHERE** clause of a query. In designing a report, **USER** can be used to print the current user for whom the report is being produced.

Examples

The following example returns the current user name:

```
SELECT USER AS CurrentUser
```

The following example selects the name and record creation date information for those records created by the current user:

```
SELECT Name FROM Sample.Person WHERE Name = USER
```

See Also

[CREATE USER GRANT](#)

WEEK

A date function that returns the week of the year as an integer for a date expression.

```
{fn WEEK(date-expression)}
```

Arguments

<i>date-expression</i>	An expression that is the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

WEEK takes a *date-expression*, and returns the number of weeks from the beginning of the year for that date.

By default, weeks are calculated using the **\$HOROLOG** date (number of days since Dec. 31, 1840). Therefore, weeks are counted from year to year, such that Week 1 is the days that complete the seven-day period begun by the last week of the previous year. A week always begins with a Sunday; therefore, the first Sunday of the calendar year marks the changing from Week 1 to Week 2. If the first Sunday of the year is January 1, then that Sunday is in Week 1; if the first Sunday of the year is later than January 1, then that Sunday is the first day of Week 2. For this reason, Week 1 is commonly less than seven days in length. You can determine the day of the week by using the **DAYOFWEEK** function. The total number of weeks in a year is commonly 53, and can be 54 in leap years.

Caché also supports the ISO 8601 standard for determining the week of the year. This standard is principally used in European countries. When Caché is configured for ISO 8601, **WEEK** begins counting a week with Monday, and assigns the week to the year that contains that week's Thursday. For example, Week 1 of 2004 ran from Monday 29 December 2003 to Sunday 4 January 2004, because this week's Thursday was 1 January 2004, which was the first Thursday of 2004. Week 1 of 2005 ran from Monday 3 January 2005 to Sunday 9 January 2005, because its Thursday was 6 January 2005, which was the first Thursday of 2005. The total number of weeks in a year is commonly 52, but can occasionally be 53. To activate ISO 8601 counting, SET ^%SYS("sql", "sys", "week ISO8601")=1.

The *date-expression* can be a Caché date integer, a **\$HOROLOG** or **\$ZTIMESTAMP** value, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The time portion of the timestamp is not evaluated and can be omitted. The *date-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

The same week information can be returned by using the **DATEPART** or **DATENAME** function.

This function can also be invoked from ObjectScript using the **WEEK()** method call:

```
$$SYSTEM.SQL.WEEK(date-expression)
```

Date Validation

WEEK performs the following checks on input values. If a value fails a check, the null string is returned.

- A date string must be complete and properly formatted with the appropriate number of elements and digits for each element, and the appropriate separator character. Years must be specified as four digits.
- Date values must be within a valid range. Years: 1841 through 9999. Months: 1 through 12. Days: 1 through 31.
- The number of days in a month must match the month and year. For example, the date '02-29' is only valid if the specified year is a leap year.
- Date values less than 10 may include or omit a leading zero. Other non-canonical integer values are not permitted. Therefore, a Day value of '07' or '7' is valid, but '007', '7.0' or '7a' are not valid.

Examples

The following Embedded SQL example returns the day of week and week of year for January 2, 2005 (which is a Sunday) and January 1, 2006 (which is a Sunday).

```
SET x="2005-1-2"
SET y="2006-1-1"
&sql(SELECT {fn DAYOFWEEK(:x)},{fn WEEK(:x)},
      {fn DAYOFWEEK(:y)},{fn WEEK(:y)}
INTO :a,:b,:c,:d)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"2005 Day of Week is: ",a," (Sunday=1)"
  WRITE " Week of Year is: ",b
  WRITE !,"2006 Day of Week is: ",c," (Sunday=1)"
  WRITE " Week of Year is: ",d }
```

The following examples return the number 9 because the date is the ninth week of the year 2004:

```
SELECT {fn WEEK('2004-02-25')} AS Wk_Date,
      {fn WEEK('2004-02-25 08:35:22')} AS Wk_Tstamp,
      {fn WEEK(59590)} AS Wk_DInt
```

The following example returns the number 54 because this particular date is in a leap year that began with Week 2 starting on the second day, as demonstrated by the example immediately following it:

```
SELECT {fn WEEK('2000-12-31')} AS Week

SELECT {fn WEEK('2000-01-01')}||{fn DAYNAME('2000-01-01')} AS WeekofDay1,
      {fn WEEK('2000-01-02')}||{fn DAYNAME('2000-01-02')} AS WeekofDay2
```

The following examples all return the current week:

```
SELECT {fn WEEK({fn NOW()})} AS Wk_Now,
      {fn WEEK(CURRENT_DATE)} AS Wk_CurrD,
      {fn WEEK(CURRENT_TIMESTAMP)} AS Wk_CurrTS,
      {fn WEEK($HOROLOG)} AS Wk_Horolog,
      {fn WEEK($ZTIMESTAMP)} AS Wk_ZTS
```

The following Embedded SQL example shows the Caché default week of the year and the week of the year with the ISO 8601 standard applied:

```
TestISO
SET def=$DATA(^%SYS("sql","sys","week ISO8601"))
IF def=0 {SET ^%SYS("sql","sys","week ISO8601")=0}
ELSE {SET isoval=^%SYS("sql","sys","week ISO8601")}
      IF isoval=1 {GOTO UnsetISO }
      ELSE {SET isoval=0 GOTO WeekOfYear }
UnsetISO
SET ^%SYS("sql","sys","week ISO8601")=0
WeekOfYear
&sql(SELECT {fn WEEK($HOROLOG)} INTO :a)
WRITE "For Today:",!
WRITE "default week of year is ",a,!
SET ^%SYS("sql","sys","week ISO8601")=1
&sql(SELECT {fn WEEK($HOROLOG)} INTO :b)
WRITE "ISO8601 week of year is ",b,!
ResetISO
SET ^%SYS("sql","sys","week ISO8601")=isoval
```

See Also

- SQL functions: [DATENAME](#), [DATEPART](#), [DAYOFWEEK](#), [MONTH](#), [QUARTER](#), [TO_DATE](#), [YEAR](#)
- ObjectScript special variables: [\\$HOROLOG](#), [\\$ZTIMESTAMP](#)

XMLCONCAT

A function that concatenates XML elements.

```
XMLCONCAT(XmlElement1,XmlElement2[,XmlElementN])
```

Arguments

<i>XmlElement</i>	An XMLEMENT function. Specify two or more <i>XmlElement</i> to concatenate.
-------------------	--

Description

The **XMLCONCAT** function returns the values from several **XMLEMENT** functions as a single string. **XMLCONCAT** can be used in a **SELECT** query or subquery that references either a table or a view. **XMLCONCAT** can appear in a **SELECT** list alongside ordinary field values.

Examples

The following query concatenates the values from two **XMLEMENT** functions:

```
SELECT Name,XMLCONCAT(XMLELEMENT("Para",Name),
                      XMLELEMENT("Para",Home_City)) AS ExportString
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
ExportString
<Para>Emerson,Molly N.</Para><Para>Boston</Para>
```

The following query nests an **XMLCONCAT** within an **XMLEMENT** function:

```
SELECT XMLELEMENT("Item",Name,
                  XMLCONCAT(
                      XMLELEMENT("Para",Home_City,' ',Home_State),
                      XMLELEMENT("Para",'is residence')))
AS ExportString
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
ExportString
<Item>Emerson,Molly N.<Para>Boston MA</Para><Para>is residence</Para></Item>
```

See Also

[SELECT](#) statement

[XMLAGG](#) function

[XMLELEMENT](#) function

XMLLEMENT

A function that formats an XML markup tag to enclose one or more expression values.

```
XMLLEMENT([NAME] tag,expression[,expression])
```

```
XMLLEMENT([NAME] tag,XMLATTRIBUTES(expression [AS alias]),expression[,expression])
```

Arguments

<p>NAME <i>tag</i></p>	<p>The name of an XML markup tag. The NAME keyword is optional. This argument has three syntactical forms: <code>NAME "tag"</code>, <code>"tag"</code>, and <code>NAME</code>. The first two are functionally identical. If specified, <i>tag</i> must be enclosed in double quotes. The case of letters in <i>tag</i> is preserved.</p> <p>XMLLEMENT performs no validation of <i>tag</i> values. However, the XML standard requires that a valid <i>tag</i> name cannot contain any of the characters <code>! "#\$%&'()*+,-/;=>?@[\\]^_`{ }~</code>, nor a space character, and cannot begin with <code>"</code>, <code>.</code>, or a numeric digit.</p> <p>If you specify the NAME keyword without a <i>tag</i> value, Caché supplies the default tag value: <code><Name> ... </Name></code>.</p>
<p><i>expression</i></p>	<p>Any valid expression. Usually the name of a column that contains the data values to be tagged. You can specify a comma-separated list of columns or other expressions, all of which will be enclosed within the same <i>tag</i>. The first comma-separated element can be an XMLATTRIBUTES function. Only one XMLATTRIBUTES element can be specified.</p>

Description

The **XMLLEMENT** function returns the values of *expression* tagged with the XML (or HTML) markup start-tag and end-tag specified in *tag*. For example, `XMLLEMENT(NAME "Para",Home_City)` returns values such as the following: `<Para>Chicago</Para>`. **XMLLEMENT** cannot be used to generate an empty-element tag.

XMLLEMENT can be used in a **SELECT** query or subquery that references either a table or a view. **XMLLEMENT** can appear in a **SELECT** list alongside ordinary field values.

The *tag* argument uses double quotes to enclose a literal string. In nearly all other contexts, Caché SQL uses single quotes to enclose a literal string; it uses double quotes to specify a [delimited identifier](#). Therefore, delimited identifier support must be enabled to use this feature.

When SQL code is specified as a string delimited by double quotes, such as in a [Dynamic SQL %Prepare\(\)](#) method, you must escape the *tag* double quotes by specifying two double quotes, as follows:

```
SET myquery = "SELECT XMLLEMENT( ""Para"",Name) FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
```

Commonly, *expression* is the name of a field, (or an expression containing one or more field names) in the multiple rows returned by a query. An *expression* can be a field of any type, including a [data stream field](#). The specified *expression* value is returned enclosed by a start tag and an end tag, as shown in the following format:

```
<tag>value</tag>
```

If the value to be tagged is either the empty string ("") value or a NULL, the following is returned:

```
<tag></tag>
```

If the *expression* contains multiple comma-separated elements, the results are concatenated, as shown in the following format:

```
<tag>value1value2</tag>
```

XMLEMENT functions can be nested. **XMLEMENT** and **XMLFOREST** functions may be nested in any combination. **XMLEMENT** functions can be concatenated using [XMLCONCAT](#). However, **XMLEMENT** does not do XML type resolution of entire expressions. For example, **XMLEMENT** cannot perform character conversion within a clause of a **CASE** statement (see example below).

XMLATTRIBUTES Function

The **XMLATTRIBUTES** function can only be used within an **XMLEMENT** function. If an element of *expression* is an **XMLATTRIBUTES** function, the specified expression becomes an attribute of the tag, as shown in the following format:

```
<tag ID='63' >value</tag>
```

You can only specify one **XMLATTRIBUTES** function within an **XMLEMENT** function. By convention it is the first *expression* element, though it can be any element in *expression*. Caché encloses attribute values with single quotes and inserts a space between the attribute value and the closing angle bracket (>) for the tag.

XMLEMENT and XMLFOREST Compared

- **XMLEMENT** concatenates the values of its *expression* list within a single tag. **XMLFOREST** assigns a separate tag for each *expression* item.
- **XMLEMENT** requires that you specify a tag value. **XMLFOREST** allows you to either take default tag values or specify individual tag values. **XMLEMENT** cannot specify an empty (nameless) tag: `<value</>`; **XMLFOREST** can.
- **XMLEMENT** allows you to specify a tag attribute using **XMLATTRIBUTES**. **XMLFOREST** does not allow you to specify a tag attribute.
- **XMLEMENT** returns a tag string for NULL. **XMLFOREST** does not return a tag string for NULL.

Punctuation Character Values

If a data value contains a punctuation character that XML/HTML might interpret as a tag or other coding, **XMLEMENT** and **XMLFOREST** convert this character to the corresponding encoded form:

ampersand (&) becomes `&` ;

apostrophe (') becomes `'` ;

quotation mark (") becomes `"` ;

open angle bracket (<) becomes `<` ;

close angle bracket (>) becomes `>` ;

To represent an apostrophe in a supplied text string, specify two apostrophes, as in the following example: 'can' 't'. Doubling apostrophes is not necessary for column data.

Examples

The following example returns each person's Name field value in Sample.Person as ordinary data and as xml tagged data:

```
SELECT Name,
       XMLLEMENT("Para",Name) AS ExportName
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
Name                ExportName
Emerson,Molly N.    <Para>Emerson,Molly N.</Para>
```

The following example returns every distinct Home_City and Home_State pair value in Sample.Person as xml tagged data with the tag <Address> ... </Address>. A blank space *expression* is specified to prevent concatenation of the city name and the state name:

```
SELECT DISTINCT
       XMLLEMENT(NAME "Address",Home_City,' ',Home_State) AS CityState
FROM Sample.Person
ORDER BY Home_City
```

Note that in the above example the optional NAME keyword is supplied. In the next example, the NAME keyword is provided without the *tag* value:

```
SELECT DISTINCT
       XMLLEMENT(NAME,Home_City,' ',Home_State) AS CityState
FROM Sample.Person
ORDER BY Home_City
```

In this case the same data is returned, but is tagged with the default tag: <Name> ... </Name>.

The following example returns a character stream field as xml tagged data:

```
SELECT Name,Notes,
       XMLLEMENT("Para",Notes)
FROM Sample.Employee WHERE Notes IS NOT NULL
```

A sample row of the data returned would appear as follows:

```
Emerson,Molly N. 5%Stream.GlobalCharacter^Sample.PersonS
<Para>5%Stream.GlobalCharacter^Sample.PersonS</Para>
```

The following example shows that **XMLLEMENT** functions can be nested:

```
SELECT XMLLEMENT("Para",Home_State,
                XMLLEMENT("Emphasis",Name),Age)
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
<Para>CA<Emphasis>Emerson,Molly N.</Emphasis>24</Para>
```

The following example shows that **XMLLEMENT** *can not* tag a value within a CASE statement clause:

```
SELECT XMLLEMENT("Para",Home_State,
                XMLLEMENT("Para",Name),
                CASE WHEN Age < 21 THEN NULL
                     ELSE XMLLEMENT("Para",Age) END )
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
<Para>CA<Para>Emerson,Molly N.</Para>&lt;Para&gt;24&lt;/Para&gt;</Para>
```

The following query returns the Name field values in Sample.Person as XML-tagged data in a tag that uses the RowID field as a tag attribute:

```
SELECT XMLELEMENT("Para",XMLATTRIBUTES(%ID),Name) AS ExportName
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
ExportName
<Para ID='101' >Emerson,Molly N.</Para>
```

You can specify an alias for an attribute, as shown in the following example:

```
SELECT XMLELEMENT("Para",XMLATTRIBUTES(%ID AS ItemKey),Name)
FROM Sample.Person
```

A sample row of the data returned would appear as follows:

```
<Para ItemKey='101' >Emerson,Molly N.</Para>
```

See Also

[XMLAGG](#) function

[XMLCONCAT](#) function

[XMLFOREST](#) function

[SELECT](#) statement

XMLFOREST

A function that formats multiple XML markup tags to enclose expression values.

```
XMLFOREST(expression [AS tag][,expression [AS tag]])
```

Arguments

<i>expression</i>	Any valid expression. Usually the name of a column that contains the data values to be tagged. When specified as a comma-separated list, each expression in the list will be enclosed in its own XML markup tag.
AS <i>tag</i>	<p><i>Optional</i> — The name of an XML markup tag. The AS keyword is mandatory if <i>tag</i> is specified. The case of letters in <i>tag</i> is preserved.</p> <p>Enclosing <i>tag</i> with double quotes is optional. If you omit the double quotes, <i>tag</i> must follow XML naming standards. Enclosing <i>tag</i> with double quotes removes these naming restrictions.</p> <p>XMLFOREST enforces XML naming standards for a valid <i>tag</i> name. It cannot contain any of the characters !"#\$\$%&'()*+,-/;<=>?@[\\]^_{ }~, nor a space character, and cannot begin with "-", ".", or a numeric digit.</p> <p>If you specify an <i>expression</i> without the AS <i>tag</i> clause, the tag value is the name of the <i>expression</i> column (in capital letters): <HOME_CITY>Chicago</HOME_CITY>.</p>

Description

The **XMLFOREST** function returns the values of each *expression* tagged with its own XML markup start-tag and end-tag, as specified in *tag*. For example, `XMLFOREST(Home_City AS City,Home_State AS State)` returns values such as the following: `<City>Chicago</City><State>IL</State>`. **XMLFOREST** cannot be used to generate an empty-element tag.

XMLFOREST can be used in a [SELECT](#) query or subquery that references either a table or a view. **XMLFOREST** can appear in a **SELECT** list alongside ordinary column values.

The specified *expression* value is returned enclosed by a start tag and an end tag, as shown in the following format:

```
<tag>value</tag>
```

Commonly, *expression* is the name of a column, or an expression containing one or more column names. An *expression* can be a field of any type, including a [data stream field](#). **XMLFOREST** tags each *expression* as follows:

- If AS *tag* is specified, **XMLFOREST** tags the resulting values with the specified tag. The *tag* value is case-sensitive.
- If AS *tag* is omitted, and *expression* is a column name, **XMLFOREST** tags the resulting values with the column name. Column name default tags are always uppercase.
- If *expression* is not a column name (for example, an aggregate function, a literal, or a concatenation of two columns) the AS *tag* clause is required.

XMLFOREST provides a separate tag for each item in a comma-separated list. **XMLELEMENT** concatenates all of the items in a comma-separated list within a single tag.

XMLFOREST functions can be nested. Any combination of nested **XMLFOREST** and **XMLELEMENT** functions is permitted. **XMLFOREST** functions can be concatenated using [XMLCONCAT](#).

NULL Values

The **XMLFOREST** function only returns a tag for actual data values. It does not return a tag when the *expression* value is NULL. The empty string (") is considered a data value. If the value to be tagged is the empty string ("), **XMLFOREST** returns:

```
<tag></tag>
```

XMLFOREST differs from **XMLELEMENT** in the handling of NULL. **XMLELEMENT** always returns a tag value, even when the field value is NULL.

Punctuation Character Values

If a data value contains a punctuation character that XML/HTML might interpret as a tag or other coding, **XMLFOREST** and **XMLELEMENT** convert this character to the corresponding encoded form:

ampersand (&) becomes `&`;

apostrophe (') becomes `'`;

quotation mark (") becomes `"`;

open angle bracket (<) becomes `<`;

close angle bracket (>) becomes `>`;

To represent an apostrophe in a supplied text string, specify two apostrophes, as in the following example: 'can' 't'. Doubling apostrophes is not necessary for column data.

Examples

The following query returns the Name column values in Sample.Person as ordinary data and as xml tagged data:

```
SELECT Name,XMLFOREST(Name) AS ExportName
FROM Sample.Person
```

A sample row of the data returned would appear as follows. Here the tag defaults to the name of the column:

```
Name                ExportName
Emerson,Molly N.    <NAME>Emerson,Molly N.</NAME>
```

The following example specifies multiple columns:

```
SELECT XMLFOREST(Home_City,
                Home_State AS Home_State,
                AVG(Age) AS AvAge) AS ExportData
FROM Sample.Person
```

The Home_City field specifies no tag; the tag is generated from the column name in all capital letters: `<HOME_CITY>`. The Home_State field's AS clause is optional. It is specified here because specifying the tag name allows you to control the case of the tag: `<Home_State>`, rather than `<HOME_STATE>`. The AVG(Age) AS clause is mandatory, because the value is an aggregate, not a column value, and thus has no column name. A sample row of the data returned would appear as follows.

```
ExportData
<HOME_CITY>Chicago</HOME_CITY><Home_State>IL</Home_State>
<AvAge>48.0198019801980198</AvAge>
```

See Also

[XMLAGG](#) function

[XMLELEMENT](#) function

[XMLCONCAT](#) function

[SELECT](#) statement

YEAR

A date function that returns the year for a date expression.

```
YEAR(date-expression)
{fn YEAR(date-expression)}
```

Arguments

<i>date-expression</i>	An expression that evaluates to either a Caché date integer, an ODBC date, or a timestamp. This expression can be the name of a column, the result of another scalar function, or a date or timestamp literal.
------------------------	--

Description

YEAR takes as input a Caché date integer, an ODBC format date string, or a timestamp.

A *date-expression* timestamp is data type %Library.TimeStamp (yyyy-mm-dd hh:mm:ss.fff).

The *date-expression* can also be specified as data type %Library.FilemanDate, %Library.FilemanTimestamp, or %MV.Date.

The year (yyyy) portion should be a four-digit integer in the range 1840 through 9999. There is, however, no validation or range checking for user-supplied dates. **YEAR** returns the year portion of invalid dates (such as 2005-02-31) and out-of-range dates (such as 1830-02-20). Year values outside the range 1840 through 9999, negative numbers, and fractions are returned as specified. Two digit years are *not* expanded to four digits.

YEAR returns the corresponding year as a four-digit integer.

Note: For compatibility with Caché internal representation of dates, it is *strongly* recommended that all year values be expressed as four-digit integers within the range of 1840 through 9999.

The **TO_DATE** and **TO_CHAR** SQL functions support “Julian dates,” which can be used to represent years before 1840. ObjectScript provides method calls that support such Julian dates.

YEAR returns zero when the year portion is a string of one or more zeroes (for example '0' or '0000'), or a nonnumeric value. **YEAR** interprets the initial numeric string encountered as the year value, so omitting the year portion of the date string ('-mm-dd hh:mm:ss'), or omitting the date portion ('hh:mm:ss') results in the first number encountered ('-mm' or 'hh') being treated as the year value. Thus, some placeholder should be supplied for an unknown year value; for compatibility with Caché, 9999 is generally the preferred value.

The year format default is four-digit years. To change this year display default, use the **SET OPTION** command with the YEAR_OPTION option.

The elements of a datetime string can be returned using the following SQL scalar functions: **YEAR**, **MONTH**, **DAY**, **DAYOFMONTH**, **HOUR**, **MINUTE**, **SECOND**. The same elements can be returned by using the **DATEPART** or **DATENAME** function. **DATEPART** and **DATENAME** perform value and range checking on year values.

This function can also be invoked from ObjectScript using the **YEAR()** method call:

```
$SYSTEM.SQL.YEAR(date-expression)
```

Examples

The following examples return the integer 2004. No validation is performed:

```
SELECT YEAR('2004-02-16 12:45:37') AS Year_Given
```

```
SELECT {fn YEAR(59590)} AS Year_Given
```

The following examples return the year as 0 because the year field contains a nonnumeric placeholder. The separator character (-) must be preceded by a some character(s); otherwise the month is returned as the year value:

Asterisk as year placeholder:

```
SELECT {fn YEAR('*-02-16')} AS Year_Given
```

Space character as year placeholder:

```
SELECT YEAR(' -02-16') AS Year_Given
```

The following example returns the current year:

```
SELECT YEAR(GETDATE()) AS Year_Now
```

The following Embedded SQL example returns the current year from two functions. The **CURRENT_DATE** function returns data type DATE; the **NOW** function returns data type TIMESTAMP. **YEAR** returns a four-digit year integer for both input data types:

```
&sql(SELECT {fn YEAR(CURRENT_DATE)},
        {fn YEAR({fn NOW()})} INTO :a,:b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"CURRENT_DATE year is: ",a
  WRITE !,"NOW year is: ",b }
```

The following Embedded SQL example shows that **YEAR** returns the year portion of an invalid date (the year 2001 was not a leap year) or an out-of-range date:

```
SET testdate1="2001-02-29"
SET testdate2="1835-02-19"
&sql(SELECT YEAR(:testdate1),
        YEAR(:testdate2) INTO :a,:b)
IF SQLCODE'=0 {
  WRITE !,"Error code ",SQLCODE }
ELSE {
  WRITE !,"invalid date returns: ",a
  WRITE !,"out-of-range date returns: ",b }
QUIT
```

See Also

- SQL functions: [DATENAME](#), [DATEPART](#), [DAYOFYEAR](#), [QUARTER](#), [WEEK](#), [TO_DATE](#)
- ObjectScript function: [\\$ZDATE](#)

SQL MultiValue Support Functions

\$MVFMT

A MultiValue formatting function for a string.

```
$MVFMT(string, format)
```

Arguments

<i>string</i>	A MultiValue string expression to be formatted for display.
<i>format</i>	A quoted string consisting of positional letter and number codes specifying the display format for <i>string</i> .

Description

The **\$MVFMT** function returns the *string* value formatted as specified by *format*. This formatting may include padding or rounding/truncating of *string*. The most common use for **\$MVFMT** is to provide a uniform display format for decimal numbers.

The *format* string has the following format:

```
wfRn
```

<i>w</i>	<i>Optional</i> — The overall width of the display field, specified as a positive integer. Used to impose a uniform width (number of characters) on <i>string</i> . Different operations are performed if <i>w</i> is larger or smaller than the length of <i>string</i> , as described below.
<i>f</i>	<i>Optional</i> — A fill character, specified as a single character. (Certain fill characters, as described below, must be specified as a quoted string.) You must specify <i>w</i> to use <i>f</i> . If you specify <i>w</i> , but do not specify <i>f</i> , it defaults to the space character.
R	<i>Optional</i> — The letter 'R' or 'L' specifying right or left justification. This letter code is not case-sensitive. If you do not specify a letter code, \$MVFMT defaults to left justification. (The letters 'T' and 'U' are synonyms for 'L').
<i>n</i>	<i>Optional</i> — The number of fractional digits to the right of the decimal place, specified as a positive integer. If you specify <i>n</i> , it must either be the only code in <i>format</i> , or it must be preceded by the letter 'R' or 'L'. If you do not specify <i>n</i> , \$MVFMT defaults to number of fractional digits in <i>string</i> .

There are two basic uses of *format*:

- To return fractional numbers in a standard form.
- To return strings in a standard form.

For fractional numbers, the most basic *format* is 'Rn', where R is either the letter 'R' specifying right justification or the letter 'L' specifying left justification, and *n* is the number of digits to the right of the decimal point to display. If *string* is an integer or has fewer fractional digits than *n*, zero padding is added. If *string* has more digits than *n*, the number is rounded to the specified number of fractional digits. If *n* is zero, the number is rounded to an integer and the decimal point is removed. If *string* is less than 1, specifying *n* supplies a zero (0) to the left of the decimal point. If *string* contains any character other than a number, the decimal point character, or a minus sign, **\$MVFMT** does no zero padding or rounding.

For strings, the most basic *format* is 'wE', where "w" is an integer specifying width and f is a literal fill character (for example '9^'). You can use *w* (width) and *f* (fill) formatting to make a display field a standard width. By default, the justification is 'L' (left); you can, of course, specify 'R' for right justification.

A more complex example of *format* is '10#R5', where "10" is the overall width of the display field, "#" is the fill character to use to fill out the display field. Because 'R' indicates right justification, these fill characters will appear to the left of the *string* value. The "5" indicates that the *string* value is to have 5 digits to the right of the decimal place.

The *w* (width) value may be larger than, equal to, or smaller than the number of characters (including the decimal point) of *string*. If *string* is a fractional number, *w* is applied after \$MVFMT adjusts the number of fractional digits (by rounding or zero padding).

- If *w* is greater than the length of *string*, \$MVFMT appends *f* fill characters to *string* to make the resulting string *w* characters in length. If 'L' (left justification) fill characters are applied to the end of the string; if 'R' (right justification) fill characters are applied to the beginning of string.
- If *w* is equal to the length of *string* (after rounding or zero padding of fractional digits), no operation is performed.
- If *w* is less than the length of *string*, \$MVFMT inserts a Text Mark (@TM, CHAR(251)) character after every *w* count of characters. If "L" (left justification), characters are counted forward from the beginning of the string; if "R" (right justification), characters are counted backward from the end of the string. \$MVFMT then appends *f* fill characters so that all Text Mark delimited substrings are *w* characters long (the Text Mark itself is not counted). If 'L' (left justification) fill characters are applied to the end of the string; if 'R' (right justification) fill characters are applied to the beginning of string.

The fill character is optional; if omitted, filling is done with blank spaces. The fill character cannot be the same as the *format string delimiter character*. If the fill character is a number, the backslash (\), or the letters L, R, T, or U, it must be enclosed in string delimiter quotes that are different than the *format string*. For example: '10"0"R2'. You cannot use the backslash as a string delimiter for the fill character.

Examples

The following example uses 'Rn' formatting to format numeric values so that they display four decimal digits. Note that both zero padding and rounding are performed as needed:

```
SELECT $MVFMT(1.2, 'R4'),      /* Returns 1.2000 */
       $MVFMT(1.77777, 'R4'), /* Returns 1.7778 */
       $MVFMT(.4, 'R4'),     /* Returns 0.4000 */
       $MVFMT(0, 'R4')       /* Returns 0.0000 */
```

See Also

- [\\$MVFMTS](#) function
- The MultiValue Basic [FMT](#) function, as described in the *Caché MVBasic Reference*

\$MVFMTS

A MultiValue formatting function for dynamic array elements.

```
$MVFMTS(dynarray, format)
```

Arguments

<i>dynarray</i>	A MultiValue dynamic array to be formatted for display.
<i>format</i>	A quoted string consisting of positional letter and number codes specifying the display format for the elements of <i>dynarray</i> .

Description

The **\$MVFMTS** function returns the *dynarray* value with each element formatted as specified by *format*. This formatting may include justification, character filling, and the rounding or zero padding of numeric element values. The most common use for **\$MVFMTS** is to provide a uniform display format for fractional numbers.

The *format* string has the following format:

```
wfRn
```

<i>w</i>	<i>Optional</i> — The overall width of the display field, specified as a positive integer. Used to impose a uniform width (number of characters) for each element of <i>dynarray</i> . Different operations are performed if <i>w</i> is larger or smaller than the length of an element, as described in the \$MVFMT function.
<i>f</i>	<i>Optional</i> — A fill character, specified as a single character. If the fill character is a number, the backslash (\), or the letters “L”, “R”, or “T” it must be enclosed in string delimiter quotes. You must specify <i>w</i> to use <i>f</i> . If you specify <i>w</i> , but do not specify <i>f</i> , it defaults to the space character.
R	<i>Optional</i> — The letter “R” or “L” specifying right or left justification. This letter code is not case-sensitive. If you do not specify a letter code, \$MVFMTS defaults to left justification.
<i>n</i>	<i>Optional</i> — The number of fractional digits to the right of the decimal place, specified as a positive integer. If you specify <i>n</i> , it must either be the only code in <i>format</i> , or it must be preceded by the letter “R” or “L”. If you do not specify <i>n</i> , \$MVFMTS defaults to number of fractional digits in <i>string</i> .

There are two basic uses of *format*:

- To return fractional numbers in a standard form. **\$MVFMTS** can be used to round a fractional number to an integer or to a specified number of fractional digits. If the specified number of fractional digits is larger than the number of fractional digits in the element value, **\$MVFMTS** zero pads the additional digits.
- To return strings in a standard form. **\$MVFMTS** can left justify or right justify a string and add a fill character before or after to make each element contain the same number of characters.

For further details on *format* codes, refer to the [\\$MVFMT](#) function.

See Also

- [\\$MVFMT](#) function

- The MultiValue Basic [FMTS](#) function, as described in the *Caché MVBasic Reference*

\$MVICONV

A MultiValue external-to-internal conversion function.

```
$MVICONV(expression, code)
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
<i>code</i>	A character code, specified as a quoted string, that specifies the type of conversion to perform. Conversion is from external format to internal format.

Description

The **\$MVICONV** function is a MultiValue conversion function used to convert a string from external (output) format to internal (storage) format. The type of conversion is specified by a character *code* string that is specific to the type of data to be converted.

\$MVICONV is a Caché SQL extension and is intended for MultiValue data compatibility. It is functionally identical to the MultiValue **ICONV** function, as described in the *Caché MVBasic Reference*. Refer to **ICONV** for further information.

The **\$MVICONV** function converts from external format to internal format. The **\$MVOCONV** function converts from internal format to external format. You can use the **\$MVICONVS** function to convert the elements of a dynamic array from external format to internal format.

Example

The following example converts a MultiValue date from external to internal format using various *code* values. Note that MultiValue internal date 0 is December 31, 1967, which corresponds to the Caché internal (**\$HOROLOG**) date of 46385:

```
NEW SQLCODE
SET indate="12/31/2008"
&sql(SELECT $MVICONV(:indate,'D') INTO :a)
IF SQLCODE=0 {
  WRITE a," = ",$ZDATE((a+46385),1,,4) }
ELSE { WRITE "error:",SQLCODE }
```

See Also

- **\$MVICONVS** function
- **\$MVOCONV** function
- **\$MVOCONVS** function
- The MultiValue **ICONV** function in the *Caché MVBasic Reference*

\$MVICONVS

A MultiValue external-to-internal conversion function for dynamic arrays.

```
$MVICONVS (dynarray, code)
```

Arguments

<i>dynarray</i>	A MultiValue dynamic array , each element of which specifies a value in external (display) format.
<i>code</i>	A character code, specified as a quoted string, that specifies the type of conversion to perform. Conversion is from external format to internal format.

Description

The **\$MVICONVS** function is a MultiValue conversion function used to convert the element values in a MultiValue dynamic array from external (output) format to internal (storage) format. The type of conversion is specified by a character *code* string that is specific to the type of data to be converted.

\$MVICONVS is a Caché SQL extension and is intended for MultiValue data compatibility. It is functionally identical to the MultiValue [ICONVS](#) function, as described in the *Caché MVBasic Reference*. Refer to [ICONV](#) for further information.

The **\$MVICONVS** function converts dynamic array elements from external format to internal format. The [\\$MVOCONVS](#) function converts dynamic array elements from internal format to external format. You can use the [\\$MVICONV](#) function to convert a string from external format to internal format.

See Also

- [\\$MVICONV](#) function
- [\\$MVOCONV](#) function
- [\\$MVOCONVS](#) function
- The MultiValue [ICONVS](#) function in the *Caché MVBasic Reference*

\$MVOCONV

A MultiValue internal-to-external conversion function.

```
$MVOCONV ( expression , code )
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
<i>code</i>	A character code, specified as a quoted string, that specifies the type of conversion to perform. Conversion is from internal format to external format.

Description

The **\$MVOCONV** function is a MultiValue conversion function used to convert a string from internal (storage) format to external (output) format. The type of conversion is specified by a character *code* string that is specific to the type of data to be converted.

\$MVOCONV is a Caché SQL extension and is intended for MultiValue data compatibility. It is functionally identical to the MultiValue **CONV** function, as described in the *Caché MVBASIC Reference*. Refer to **CONV** for further information.

The following types of conversions are supported:

- Character conversions: character-to-code, code-to-character.
- Numeric conversions: decimal-to-hex, hex-to-decimal, masked decimal conversion, range extraction.
- String conversions: case conversion, Soundex conversion, string length conversion, uniform string length adjustment, delimited substring extraction, pattern match extraction, string extraction by length.
- Time and Date conversions: time internal-to-display format, date internal-to-display format, date display-to-internal format, date element extraction, date day-of-week, day-of-year, and quarter calculation.

The **\$MVOCONV** function converts from internal format to external format. The **\$MVICONV** function converts from external format to internal format. You can use the **\$MVOCONVS** function to convert the elements of a dynamic array from internal format to external format.

Example

The following example converts a MultiValue date from internal to external format using various *code* values. Note that MultiValue internal date 0 is December 31, 1967, which corresponds to the Caché internal (\$HOROLOG) date of 46385:

```
NEW SQLCODE
SET indate=15000
&sql (SELECT
    $MVOCONV (:indate, 'D'),
    $MVOCONV (:indate, 'D2'),
    $MVOCONV (:indate, 'D/'),
    $MVOCONV (:indate, 'D2-')
    INTO :a, :b, :c, :d)
IF SQLCODE=0 {
WRITE a, " = ", $ZDATE ((indate+46385), 1, , 4), !
WRITE b, " = ", $ZDATE ((indate+46385), 1, , 4), !
WRITE c, " = ", $ZDATE ((indate+46385), 1, , 4), !
WRITE d, " = ", $ZDATE ((indate+46385), 1, , 4) }
ELSE { WRITE "error:", SQLCODE }
```

See Also

- [\\$MVICONV](#) function
- [\\$MVOCONVS](#) function
- The MultiValue [OCONV](#) function in the *Caché MVBasic Reference*

\$MVOCONVS

A MultiValue internal-to-external conversion function for dynamic arrays.

```
$MVOCONVS (dynarray, code)
```

Arguments

<i>dynarray</i>	A MultiValue dynamic array , each element of which specifies a value in internal (storage) format.
<i>code</i>	A character code, specified as a quoted string, that specifies the type of conversion to perform. Conversion is from internal format to external format.

Description

The **\$MVOCONVS** function is a MultiValue conversion function used to convert the element values in a MultiValue dynamic array from internal (storage) format to external (output) format. The type of conversion is specified by a character *code* string that is specific to the type of data to be converted.

\$MVOCONVS is a Caché SQL extension and is intended for MultiValue data compatibility. It is functionally identical to the MultiValue [OCONVS](#) function, as described in the *Caché MVBasic Reference*. Refer to [OCONVS](#) for further information.

The following types of conversions are supported:

- Character conversions: character-to-code, code-to-character.
- Numeric conversions: decimal-to-hex, hex-to-decimal, masked decimal conversion, range extraction.
- String conversions: case conversion, Soundex conversion, string length conversion, uniform string length adjustment, delimited substring extraction, pattern match extraction, string extraction by length.
- Time and Date conversions: time internal-to-display format, date internal-to-display format, date display-to-internal format, date element extraction, date day-of-week, day-of-year, and quarter calculation.

The **\$MVOCONVS** function converts dynamic array elements from internal format to external format. The [\\$MVICONVS](#) function converts dynamic array elements from external format to internal format. You can use the [\\$MVOCONV](#) function to convert a string from internal format to external format.

See Also

- [\\$MVICONVS](#) function
- [\\$MVOCONV](#) function
- The MultiValue [OCONVS](#) function in the *Caché MVBasic Reference*

%MVR

A MultiValue collation sequence function.

```
%MVR(expression)
```

Arguments

<i>expression</i>	A string expression, which can be the name of a column, a string literal, or the result of another function, where the underlying data type can be represented as any character type (such as CHAR or VARCHAR2).
-------------------	--

Description

%MVR is a Caché SQL extension and is intended for MultiValue data compatibility.

%MVR returns *expression* converted to the MultiValue collation sequence. It is used when a string contains both numeric and non-numeric characters. **%MVR** divides the *expression* string into substrings, each substring containing either all numeric or all non-numeric characters. The numeric substrings are sorted in signed numeric order. The non-numeric substrings are sorted in case-sensitive ASCII collation sequence.

If *expression* is all numeric, the returned value is the same as numeric collation. If *expression* is all non-numeric, the returned value is the same as [%SPACE collation](#).

You can perform the same collation conversion in ObjectScript using the **Collation()** method of the %SYSTEM.Util class:

```
WRITE $SYSTEM.Util.Collation("The quick, BROWN fox.",2)
```

This function can also be invoked from ObjectScript using the **MVR()** method call:

```
WRITE $SYSTEM.SQL.MVR("The quick, BROWN fox.")
```

You can reverse **%MVR** collation using the **RevCollation()** method of the %SYSTEM.Util class.

Examples

The following examples show how **%MVR** collation differs from default string collation in the handling of strings containing a numeric component. The first example orders the values in string sequence: 1027 appears before 107. The first example orders the values in %MVR sequence: 107 appears before 1027.

```
SELECT TOP 20 Name,Home_Street
FROM Sample.Person
ORDER BY Home_Street
```

```
SELECT TOP 20 Name,Home_Street
FROM Sample.Person
ORDER BY %MVR(Home_Street)
```

See Also

- [ASCII](#) function
- [%EXACT](#) collation function
- [%SQLSTRING](#) collation function
- [%SQLUPPER](#) collation function
- [%TRUNCATE](#) collation function
- [Collation](#) chapter in *Using Caché SQL*

SQL Unary Operators

- (Negative)

A unary operator that returns an expression as a negative, numeric value.

```
-expression
```

Arguments

<i>expression</i>	A numeric expression.
-------------------	-----------------------

Description

Unary operators perform an operation on only one expression of any of the data types of the numeric data type category.

– (Negative) is a Caché SQL extension.

Examples

The following example returns three numeric fields: the Age column from Sample.Person; the – (Negative) value of the average of Age; and the Age minus the average age:

```
SELECT Age,  
       -(AVG(age)) AS NegAvg,  
       Age-AVG(Age) AS AgeRelAvg  
FROM Sample.Person
```

See Also

[+ \(Positive\)](#)

+ (Positive)

A unary operator that returns an expression as a positive, numeric value.

```
+expression
```

Arguments

<i>expression</i>	A numeric expression.
-------------------	-----------------------

Description

Unary operators perform an operation on only one expression. This expression can be any of the data types of the numeric data type category.

+ (Positive) is a Caché SQL extension.

See Also

[- \(Negative\)](#)

SQL Reference Material

Data Types

Specifies the kind of data that an SQL entity (such as a column) can contain.

Description

The following topics are described here:

- A table of the supported [DDL data types and their class property mappings](#)
- [Data type precedence](#) used to select the most inclusive data type from data values having different data types
- [Date, Time, and TimeStamp data types](#)
 - Usage in SqlCategory of standard and user-defined logical values
 - Configurable support for dates prior to December 31, 1840
- Support for [long strings](#), the [List data type](#), and the [Stream data type](#)
- Support for the [ROWVERSION data type](#)
- Data types exposed by Caché [ODBC / JDBC](#)
- Determining a column's data type using [query metadata](#) methods and [data type integer codes](#)
- Creating [user-defined data types](#)
- Handling of [undefined data types](#)
- Data type [conversion functions](#)

A data type specifies the kind of value that a column can hold. You specify the data type when defining a field with **CREATE TABLE**. When defining SQL expressions, you can specify the DDL data types listed in the following table (left-hand column). When you specify one of these DDL data types, it maps to the Caché data type listed in the right-hand column.

To view the current system data type mappings, go to the Management Portal, select **[System] > [Configuration] > [System-defined DDL Mappings]**.

You can also define additional user data types. To create or view the user data type mappings, go to the Management Portal, select **[System] > [Configuration] > [User-defined DDL Mappings]**.

Table of DDL Data Types

DDL Data Type	Corresponding Caché Class Property Data Type
BIGINT	%Library.BigInt (MAXVAL=9223372036854775807, MINVAL=-9223372036854775807) If a BIGINT column can contain both NULLs and extremely small negative numbers, you may need to redefine the index null marker to support standard index collation. For further details refer to " Indexing a NULL " in the <i>SQL Optimization Guide</i> .
BIGINT(%1)	%Library.BigInt The %1 is ignored. Equivalent to BIGINT. Provided for MySQL compatibility.
BINARY	%Library.Binary(MAXLEN=1)

DDL Data Type	Corresponding Caché Class Property Data Type
BINARY(%1)	%Library.Binary(MAXLEN=%1)
BINARY VARYING	%Library.Binary(MAXLEN=1)
BINARY VARYING(%1)	%Library.Binary(MAXLEN=%1)
BIT	%Library.Boolean
CHAR	%Library.String(MAXLEN=1)
CHAR(%1)	%Library.String(MAXLEN=%1)
CHAR VARYING	%Library.String(MAXLEN=1)
CHAR VARYING(%1)	%Library.String(MAXLEN=%1)
CHARACTER	%Library.String(MAXLEN=1)
CHARACTER VARYING	%Library.String(MAXLEN=1)
CHARACTER VARYING(%1)	%Library.String(MAXLEN=%1)
CHARACTER(%1)	%Library.String(MAXLEN=%1)
DATE	%Library.Date
DATETIME	%Library.DateTime
DATETIME2	%Library.DateTime
DEC	%Library.Numeric MAXVAL=9999999999999999, MINVAL=-9999999999999999, SCALE=0.
DEC(%1)	%Library.Numeric A 64-bit signed integer. If %1 is less than 19, MAXVAL and MINVAL are the %1 number of digits. For example, DEC(8) MAXVAL=99999999, MINVAL=-99999999, SCALE=0. The largest meaningful value for %1 is 19; %1 values larger than 19 do not issue an error, but default to 19. If %1 is 19 or greater: MAXVAL=9223372036854775807, MINVAL=-9223372036854775808, SCALE=0.
DEC(%1,%2)	%Library.Numeric (MAXVAL=< '\$maxval'^%apiSQL(%1,%2)'>, MINVAL=< '\$minval'^%apiSQL(%1,%2)'>, SCALE=%2)
DECIMAL	%Library.Numeric MAXVAL=9999999999999999, MINVAL=-9999999999999999, SCALE=0.
DECIMAL(%1)	%Library.Numeric A 64-bit signed integer. If %1 is less than 19, MAXVAL and MINVAL are the %1 number of digits. For example, DECIMAL(8) MAXVAL=99999999, MINVAL=-99999999, SCALE=0. The largest meaningful value for %1 is 19; %1 values larger than 19 do not issue an error, but default to 19. If %1 is 19 or greater: MAXVAL=9223372036854775807, MINVAL=-9223372036854775808, SCALE=0.

DDL Data Type	Corresponding Caché Class Property Data Type
DECIMAL(%1,%2)	%Library.Numeric (MAXVAL=< '\$maxval'^%apiSQL(%1,%2)'>, MINVAL=< '\$minval'^%apiSQL(%1,%2)'>, SCALE=%2)
DOUBLE	%Library.Double This is the IEEE floating point standard. An SQL column with this data type returns a default precision of 20. For further details (including important max/min value limits), refer to the \$DOUBLE function in the <i>ObjectScript Reference</i> .
DOUBLE PRECISION	%Library.Double This is the IEEE floating point standard. An SQL column with this data type returns a default precision of 20. For further details (including important max/min value limits), refer to the \$DOUBLE function in the <i>ObjectScript Reference</i> .
IMAGE	%Stream.GlobalBinary
INT	%Library.Integer (MAXVAL=2147483647, MINVAL=-2147483648)
INT(%1)	%Library.Integer (MAXVAL=2147483647, MINVAL=- 2147483648). The %1 is ignored. Equivalent to INT. Provided for MySQL compatibility.
INTEGER	%Library.Integer (MAXVAL=2147483647, MINVAL=-2147483648)
LONG	%Stream.GlobalCharacter
LONG BINARY	%Stream.GlobalBinary
LONG RAW	%Stream.GlobalBinary
LONGTEXT	%Stream.GlobalCharacter Equivalent to LONG. Provided for MySQL compatibility.
LONG VARCHAR	%Stream.GlobalCharacter
LONG VARCHAR(%1)	%Stream.GlobalCharacter
LONGVARBINARY	%Stream.GlobalBinary
LONGVARBINARY(%1)	%Stream.GlobalBinary
LONGVARCHAR	%Stream.GlobalCharacter
LONGVARCHAR(%1)	%Stream.GlobalCharacter
MEDIUMINT	%Library.Integer(MAXVAL=8388607,MINVAL=- 8388608) Provided for MySQL compatibility.

DDL Data Type	Corresponding Caché Class Property Data Type
MEDIUMINT(%1)	%Library.Integer(MAXVAL=8388607,MINVAL=-8388608) The %1 is ignored. Provided for MySQL compatibility.
MONEY	%Library.Currency(MAXVAL=922337203685477.5807, MINVAL=-922337203685477.5808, SCALE=4)
NATIONAL CHAR	%Library.String(MAXLEN=1)
NATIONAL CHAR(%1)	%Library.String(MAXLEN=%1)
NATIONAL CHAR VARYING	%Library.String(MAXLEN=1)
NATIONAL CHAR VARYING(%1)	%Library.String(MAXLEN=%1)
NATIONAL CHARACTER	%Library.String(MAXLEN=1)
NATIONAL CHARACTER(%1)	%Library.String(MAXLEN=%1)
NATIONAL CHARACTER VARYING	%Library.String(MAXLEN=1)
NATIONAL CHARACTER VARYING(%1)	%Library.String(MAXLEN=%1)
NATIONAL VARCHAR	%Library.String(MAXLEN=1)
NATIONAL VARCHAR(%1)	%Library.String(MAXLEN=%1)
NCHAR	%Library.String(MAXLEN=1)
NCHAR(%1)	%Library.String(MAXLEN=%1)
NTEXT	%Stream.GlobalCharacter
NUMBER	%Library.Numeric A 64-bit signed integer. (MAXVAL=9223372036854775807, MINVAL=-9223372036854775808, SCALE=0)
NUMBER(%1)	%Library.Numeric A 64-bit signed integer. If %1 is less than 19, MAXVAL and MINVAL are the %1 number of digits. For example, NUMBER(8) MAXVAL=99999999, MINVAL=-99999999, SCALE=0. The largest meaningful value for %1 is 19; %1 values larger than 19 do not issue an error, but default to 19. If %1 is 19 or greater: MAXVAL=9223372036854775807, MINVAL=-9223372036854775808, SCALE=0.
NUMBER(%1,%2)	%Library.Numeric (MAXVAL=< '\$maxval^%apiSQL(%1,%2) >, MINVAL=< '\$minval^%apiSQL(%1,%2) >, SCALE=%2)
NUMERIC	%Library.Numeric MAXVAL=9999999999999999, MINVAL=-9999999999999999, SCALE=0.

DDL Data Type	Corresponding Caché Class Property Data Type
NUMERIC(%1)	%Library.Numeric A 64-bit signed integer. If %1 is less than 19, MAXVAL and MINVAL are the %1 number of digits. For example, NUMERIC(8) MAXVAL=99999999, MINVAL=-99999999, SCALE=0. The largest meaningful value for %1 is 19; %1 values larger than 19 do not issue an error, but default to 19. If %1 is 19 or greater: MAXVAL=9223372036854775807, MINVAL=-9223372036854775808, SCALE=0.
NUMERIC(%1,%2)	%Library.Numeric (MAXVAL=< '\$\$maxval'^%apiSQL(%1,%2)' >, MINVAL=< '\$\$minval'^%apiSQL(%1,%2)' >, SCALE=%2)
NVARCHAR	%Library.String(MAXLEN=1)
NVARCHAR(%1)	%Library.String(MAXLEN=%1)
NVARCHAR(%1,%2)	%Library.String(MAXLEN=%1)
NVARCHAR(MAX)	%Stream.GlobalCharacter Equivalent to LONGVARCHAR. Provided for TSQL compatibility.
RAW(%1)	%Library.Binary(MAXLEN=%1)
ROWVERSION	%Library.RowVersion(MAXVAL=9223372036854775807, MINVAL=1) A system-assigned sequential integer. See ROWVERSION Data Type for details.
SERIAL	%Library.Counter System-generated: (MAXVAL=2147483647, MINVAL=1). User-supplied: (MAXVAL=2147483647, MINVAL=-2147483648)
SMALLDATETIME	%Library.DateTime MAXVAL='2079-06-06-23:59:59'; MINVAL='1900-01-01 00:00:00')
SMALLINT	%Library.SmallInt (MAXVAL=32767, MINVAL=-32768)
SMALLINT(%1)	%Library.SmallInt The %1 is ignored. Equivalent to SMALLINT. Provided for MySQL compatibility.
SMALLMONEY	%Library.Currency SCALE=4
SYSNAME	%Library.String(MAXLEN=128)
TEXT	%Stream.GlobalCharacter
TIME	%Library.Time
TIMESTAMP	%Library.TimeStamp

DDL Data Type	Corresponding Caché Class Property Data Type
TINYINT	%Library.TinyInt (MAXVAL=127, MINVAL=-128)
TINYINT(%1)	%Library.TinyInt The %1 is ignored. Equivalent to TINYINT. Provided for MySQL compatibility.
UNIQUEIDENTIFIER	%Library.UniqueIdentifier
VARBINARY	%Library.Binary(MAXLEN=1)
VARBINARY(%1)	%Library.Binary(MAXLEN=%1)
VARCHAR	%Library.String(MAXLEN=1)
VARCHAR(%1)	%Library.String(MAXLEN=%1)
VARCHAR(%1,%2)	%Library.String(MAXLEN=%1)
VARCHAR2(%1)	%Library.String(MAXLEN=%1)
VARCHAR(MAX)	%Stream.GlobalCharacter Equivalent to LONGVARCHAR. Provided for TSQL compatibility only.

Important: Each of the DDL or Caché data type expressions shown above is actually one continuous string. These strings may contain space characters, but generally do not contain white space of any kind. Some white space appears in this table for readability.

MAXLEN and Space Usage

Caché SQL should not be affected by an overly large MAXLEN value. When specifying a %Library.String data type, the MAXLEN value you specify does not have to correspond closely to the actual size of the data. If the field value is "ABC", Caché only uses that much space on disk, in the global buffers, and in private process memory. Even if the field is declared with MAXLEN=1000, the private process memory does not allocate that much space for the field. Caché only allocates memory for the actual size of the field value, regardless of the declared length.

ODBC applications may be affected by an overly large MAXLEN value. ODBC applications try to make decisions about the size of a field needed based on metadata from the server, so the application may allocate more buffer space than is actually needed. For this reason, Caché supplies a system-wide default ODBC VARCHAR maximum length of 4096; this default is configurable. To determine the current setting, call `$$SYSTEM.SQL.CurrentSettings()`. The Caché ODBC driver takes the data from the TCP buffer and converts it into the applications buffer, so MAXLEN size does not affect our ODBC client.

JDBC applications should not be affected by an overly large MAXLEN value. Java and .Net do not have the application allocate buffers. The clients only allocated what is needed to hold the data as a native type.

Precision and Scope

Numeric data types such as NUMERIC(6,2) have two integer values (*p,s*) precision and scope. These are mapped to Caché %Library class data types, as described in “Understanding DDL Data Type Mappings”. When specified in an SQL data type, the following apply on Windows systems (maximums may differ on other systems):

- Precision: an integer between 0 and 19+*s* (inclusive). This value determines the maximum and minimum permitted value. This is, commonly, the total number of digits in the number; however, its exact value is determined by the %Library class data type mapping. The maximum integer value is 9223372036854775807. A precision larger than 19+*s* defaults to 19+*s*.

- **Scope:** an integer that specifies the maximum number of decimal (fractional) digits permitted. Can be a positive integer, 0, or a negative integer. If *s* is larger than or equal to *p*, only a fractional value is permitted, the actual *p* value is ignored. The largest permitted scope is 18, which corresponds to .9999999999999999. A scope larger than 18 defaults to 18.

The following example shows the maximum values for different combinations of precision and scope:

```
FOR i=0:1:6 {
  WRITE "Max for (" , i , ", 2)=", $$maxval^%apiSQL(i,2), !}
```

SQL System Data Type Mappings

The syntax shown for DDL and Caché data type expressions in the above table are the default mappings configured for the SQL.SystemDataTypes. There are separate mapping tables available for supplied system data types, and user data types.

To view and modify the current data type mappings, Go to the Management Portal, select **[System] > [Configuration] > [System-defined DDL Mappings]**.

Understanding DDL Data Type Mappings

When mapping data types from DDL to Caché, regular parameters and function parameters follow these rules:

- **Regular Parameters** — These are identified in the DDL data type and the Caché data type in the format `%#`. For example:

```
VARCHAR(%1)
```

maps to:

```
%String(MAXLEN=%1)
```

Hence, a DDL data type of:

```
VARCHAR(10)
```

maps to:

```
%String(MAXLEN=10)
```

- **Function Parameters** — These are used when a parameter in the DDL data type has to undergo some transformation before it can be put into the Caché data type. An example of this is the transformation of a DDL data type's numeric precision and scale parameters into a Caché data type's *MAXVAL*, *MINVAL*, and *SCALE* parameters. For example:

```
DECIMAL(%1,%2)
```

maps to:

```
%Numeric(MAXVAL=<| '$$maxval^%apiSQL(%1,%2)' |> ,
          MINVAL=<| '$$minval^%apiSQL(%1,%2)' |> ,
          SCALE=%2)
```

The DDL data type `DECIMAL` takes parameters Precision (*%1*) and Scale (*%2*), but the Caché data type `%Numeric` does not have a precision parameter. Therefore, to convert `DECIMAL` to `%Numeric`, the Precision parameter must be converted to appropriate `%Numeric` parameters, in this case by applying the Caché functions *format*, *maxval*, and *minval* to the parameters supplied by `DECIMAL`. The special `<| 'xxx' |>` syntax (as shown above) signals the DDL processor to do parameter replacement and then call the function with the values supplied. The `<| 'xxx' |>` expression is then replaced with the value returned from the function call.

Considering this example with actual values, there might be a `DECIMAL` data type with a precision of 4 digits and a scale of 2:

```
DECIMAL(4,2)
```

This maps to:

```
%Numeric(MAXVAL=<| '$$maxval^%apiSQL(4,2)' |>,
          MINVAL=<| '$$minval^%apiSQL(4,2)' |>,
          SCALE=2)
```

which evaluates to:

```
%Numeric(MAXVAL=99.99,MINVAL=-99.99,SCALE=2)
```

For information about numeric formatting, refer to the [\\$FNUMBER](#) function in the *Caché ObjectScript Reference*.

For more information about the *maxval* and *minval* functions, see the next topic.

Data Type Precedence

When an operation can return several different values, and these values may have different data types, Caché assigns the return value whichever data type has the highest precedence. For example, a NUMERIC data type can contain all possible INTEGER data type values, but an INTEGER data type cannot contain all possible NUMERIC data type values. Thus NUMERIC has the higher precedence (is more inclusive).

For example, if a [CASE](#) statement has a possible result value of data type INTEGER, and a possible result value of data type NUMERIC, the actual result is always of type NUMERIC, regardless of which of these two cases are taken.

The precedence for data types is as follows, from highest (most inclusive) to lowest:

```
LONGVARIABLE
LONGVARIABLE
VARIABLE
VARIABLE
GUID
TIMESTAMP
DOUBLE
NUMERIC
BIGINT
INTEGER
DATE
TIME
SMALLINT
TINYINT
BIT
```

Normalize and Validate

The `%Library.DataType` superclass contains classes for the specific data types. These data type classes provide a **Normalize()** method to normalize an input value to the data type format and an **IsValid()** method to determine if an input value is valid for that data type, as well as various mode conversion methods such as **LogicalToDisplay()** and **DisplayToLogical()**.

The following examples show the **Normalize()** method for the `%TimeStamp` data type:

```
SET indate=63445
SET tsdate=##class(%Library.TimeStamp).Normalize(indate)
WRITE "%TimeStamp date: ",tsdate

SET indate="2014-2-2"
SET tsdate=##class(%Library.TimeStamp).Normalize(indate)
WRITE "%TimeStamp date: ",tsdate
```

The following examples show the **IsValid()** method for the `%TimeStamp` data type:

```
SET datestr="July 4, 2014"
SET stat=##class(%Library.TimeStamp).IsValid(datestr)
IF stat=1 {WRITE datestr," is a valid %TimeStamp",! }
ELSE {WRITE datestr," is not a valid %TimeStamp",! }
```

```

SET leapdate="2004-02-29 00:00:00"
SET noleap="2005-02-29 00:00:00"
SET stat=##class(%Library.TimeStamp).IsValid(leapdate)
  IF stat=1 {WRITE leapdate," is a valid %TimeStamp",!}
  ELSE {WRITE leapdate," is not a valid %TimeStamp",!}
SET stat=##class(%Library.TimeStamp).IsValid(noleap)
  IF stat=1 {WRITE noleap," is a valid %TimeStamp",!}
  ELSE {WRITE noleap," is not a valid %TimeStamp",!}

```

Date, Time, and TimeStamp Data Types

You can define date, time, and timestamp data types, and interconvert dates and timestamps through standard Caché SQL date and time functions. For example, you can use **CURRENT_DATE** or **CURRENT_TIMESTAMP** as input to a field defined with that data type, or use **DATEADD**, **DATEDIFF**, **DATENAME**, or **DATEPART** to manipulate date values stored with this data type.

The data type classes %Library.Date, %Library.Time, %Library.TimeStamp, %MV.Date, %Library.FilemanDate, and %Library.FilemanTimeStamp are treated as follows with regard to [SqlCategory](#):

1. %Library.Date classes, and any user-defined data type class that has a logical value of +\$HOROLOG (the date portion of \$HOROLOG) should use DATE as the SqlCategory.
2. %Library.Time classes, and any user-defined data type class that has a logical value of \$PIECE(\$HOROLOG, ",", 2) (the time portion of \$HOROLOG) should use TIME as the SqlCategory. TIME supports fractional seconds, so this data type can also be used for HH:MI:SS.FF to a user-specified number of digits of precision. To support fractional seconds set the Precision parameter: a value of 0 rounds to the nearest second; a value of "" (the default) retains whatever precision is specified in the data value.

A field using the %Time datatype reports precision and scale to xDBC as follows: If a *precision* argument is not specified, or specified as 0 or "", xDBC sets precision=8 and scale=0; fractional seconds are truncated. If a *precision* argument is specified, xDBC sets precision=8+*precision* and scale=*precision*.

3. %Library.TimeStamp classes, and any user-defined data type class that has a logical value of YYYY-MM-DD HH:MI:SS.FF should use TIMESTAMP as the SqlCategory. Note that %Library.TimeStamp derives its maximum precision from the system platform's precision.
4. %Library.DateTime is a subclass of %Library.TimeStamp. It defines a type parameter named DATEFORMAT and it overrides the **DisplayToLogical()** and **OdbcToLogical()** methods to handle imprecise datetime input that TSQL applications are accustomed to.
5. %MV.Date classes, or any user-defined data type class that has a logical date value of \$HOROLOG-46385, should use MVDATE as the SqlCategory.
6. %Library.FilemanDate classes, or any user-defined data type class that has a logical date value of CYYMMDD, should use FMDATE as the SqlCategory.
7. %Library.FilemanTimeStamp classes, or any user-defined data type class that has a logical date value of CYYM-MDD.HHMMSS, should use FMTIMESTAMP as the SqlCategory.
8. A user-defined date data type that does not fit into any of the preceding logical values should define the SqlCategory of the data type as DATE and provide in the data type class a **LogicalToDate()** method to convert a user-defined logical date value to a %Library.Date logical value, and a **DateToLogical()** method to convert a %Library.Date logical value to the user-defined logical date value.
9. A user-defined time data type that does not fit into any of the preceding logical values should define the SqlCategory of the data type as TIME and provide in the data type class a **LogicalToTime()** method to convert a user-defined logical time value to a %Library.Time logical value, and a **TimeToLogical()** method to convert a %Library.Time logical value to the user-defined logical time value.
10. A user-defined timestamp data type that does not fit into any of the preceding logical values should define the SqlCategory of the data type as TIMESTAMP and provide in the data type class a **LogicalToTimeStamp()** method to convert

a user-defined logical timestamp value to a %Library.TimeStamp logical value, and a **TimeStampToLogical()** method to convert a %Library.TimeStamp logical value to the user-defined logical timestamp value.

When comparing FMTIMESTAMP category values with DATE category values, Caché no longer strips the time from the FMTIMESTAMP value before comparing it to the DATE. This is identical behavior to comparing TIMESTAMP with DATE values, and comparing TIMESTAMP with MVDATA values. It is also compatible with how other SQL vendors compare timestamps and dates. This means a comparison of a FMTIMESTAMP 320110202.12 and DATE 62124 are equal when compared using the SQL equality (=) operator. Applications must convert the FMTIMESTAMP value to a DATE or FMDATE value to compare only the date portions of the values.

Dates Prior to December 31, 1840

A date is commonly represented by the DATE data type. This data type stores a date in \$HOROLOG format, as a positive integer count of days from the arbitrary starting date of [December 31, 1840](#).

By default, dates can only be represented by a positive integer (MINVAL=0). However, you can change the MINVAL type parameter to enable storage of dates prior to December 31, 1840. By setting MINVAL to a negative number, you can store dates prior to December 31, 1840 as negative integers. The earliest allowed MINVAL value is -672045. This corresponds to January 1 of Year 1 (CE). DATE data type cannot represent BCE (also known as BC) dates.

Note: Be aware that these date counts do not take into account changes in date caused by the Gregorian calendar reform (enacted October 15, 1582, but not adopted in Britain and its colonies until 1752).

You can redefine the minimum date for your locale as follows:

```
SET oldMinDate = ##class(%SYS.NLS.Format).GetFormatItem("DATEMINIMUM")
IF oldMinDate=0 {
  DO ##class(%SYS.NLS.Format).SetFormatItem("DATEMINIMUM",-672045)
  SET newMinDate = ##class(%SYS.NLS.Format).GetFormatItem("DATEMINIMUM")
  WRITE "Changed earliest date to ",newMinDate
}
ELSE { WRITE "Earliest date was already reset to ",oldMinDate}
```

The above example sets the MINVAL for your locale to the earliest permitted date (1/1/01).

Note: Caché does not support using [Julian dates](#) with negative logical DATE values (%Library.Date values with MINVAL<0). Thus, these MINVAL<0 values are not compatible with the Julian date format returned by the [TO_CHAR](#) function.

Long Strings

Caché provides support for long strings. Long strings are strings greater than 32,767 characters (64K bytes) up to a maximum length of 3,641,144 characters. Commonly, such strings should be assigned one of the %Stream.GlobalCharacter data types. (Prior to version 2011.1, these were assigned the CStream%String (or %Library.GlobalCharacterStream) data type; these older data types continue to be fully supported.)

Long string support is enabled by default. You can disable or enable long string support for a Caché instance. If disabled, you can enable long strings using either the Management Portal or the ObjectScript *EnableLongStrings* property of the Config.Miscellaneous class. In the Management Portal select **[System] > [Configuration] > [Memory and Startup]**. To enable support for long strings system-wide, select the **Enable Long Strings** check box. Then click the Save button. After long strings are enabled, any future invoked process on that system will support long strings. For further details, refer to [Long Strings](#) in the “Data Types and Values” chapter of *Using Caché ObjectScript*.

With long strings enabled, you can assign a %Library.String data types a MAXLEN of greater than 16,374 Unicode characters (or 32K 8-bit characters). How such long strings are handled depends on your xDBC protocol:

- Protocol 46: When ODBC or JDBC accesses %Library.String data with a MAXLEN greater than 16,374 characters, only the first 16,374 characters are returned. If you need to support data in a single field that is longer than 16,374 characters, you should use a stream data type.

- Protocol 47: No ODBC or JDBC string length limit.

The protocol that is used is the highest protocol supported by both the Caché instance and the ODBC driver facilities. If all facilities on both host and client support Protocol 47, that protocol is used. If any one facility only supports Protocol 46 that protocol is used, regardless of Protocol 47 support in other facilities. The protocol that was actually used is recorded in the Caché ODBC log.

Note that, by default, Caché establishes a system-wide ODBC VARCHAR maximum length of 4096; this [ODBC maximum length is configurable](#).

List Structures

Caché supports the list structure data type %List (data type class %Library.List). This is a compressed binary format, which does not map to a corresponding native data type for Caché SQL. In its internal representation it corresponds to data type VARBINARY with a default MAXLEN of 32749.

For this reason, [Dynamic SQL](#) cannot use %List data in a **WHERE** clause comparison. You also cannot use **INSERT** or **UPDATE** to set a property value of type %List.

Dynamic SQL returns the data type of list structured data as VARCHAR.

If you use an ODBC or JDBC client, %List data is projected to VARCHAR string data, using LogicalToOdbc conversion. A list is projected as a string with its elements delimited by commas. Data of this type can be used in a **WHERE** clause, and in **INSERT** and **UPDATE** statements. Note that, by default, Caché establishes a system-wide ODBC VARCHAR maximum length of 4096; this [ODBC maximum length is configurable](#).

For further details on data type class %Library.List, refer to the *InterSystems Class Reference*. For further details on using lists in a **WHERE** clause, see the %INLIST predicate and the [FOR SOME %ELEMENT](#) predicate. For further details on handling list data as a string, see the %EXTERNAL function.

Caché SQL supports eight list functions: [\\$LIST](#), [\\$LISTBUILD](#), [\\$LISTDATA](#), [\\$LISTFIND](#), [\\$LISTFROMSTRING](#), [\\$LISTGET](#), [\\$LISTLENGTH](#), and [\\$LISTTOSTRING](#). ObjectScript supports three additional list functions: [\\$LISTVALID](#) to determine if an expression is a list, [\\$LISTSAME](#) to compare two lists, and [\\$LISTNEXT](#) to sequentially retrieve elements from a list.

Stream Data Types

The Stream data types correspond to the Caché class property data types %Stream.GlobalCharacter (for CLOBs) and %Stream.GlobalBinary (for BLOBs).

A field with a Stream data type cannot be used as an argument to most SQL scalar, aggregate, or unary functions. Attempting to do so generates an SQLCODE -37 error code. The few functions that are exceptions are listed in the [Storing and Using Stream Data \(BLOBs and CLOBs\)](#) chapter of *Using Caché SQL*.

A field with a Stream data type cannot be used as an argument to most SQL predicate conditions. Attempting to do so generates an SQLCODE -313 error code. The predicates that accept a stream field are listed in the [Storing and Using Stream Data \(BLOBs and CLOBs\)](#) chapter of *Using Caché SQL*.

The use of Stream data types in indices, and when performing inserts and updates are also restricted. For further details on Stream restrictions, refer to the [Storing and Using Stream Data \(BLOBs and CLOBs\)](#) chapter of *Using Caché SQL*.

SERIAL Data Type

A field with a SERIAL data type can take a user-specified positive integer value, or Caché can assign it a sequential positive integer value.

An **INSERT** operation specifies one of the following values for a SERIAL field:

- No value, 0 (zero), or a nonnumeric value: Caché ignores the specified value, and instead increments this field's current serial counter value by 1, and inserts the resulting integer into the field.

- A positive integer value: Caché inserts the user-specified value into the field, and changes the serial counter value for this field to this integer value.

Thus a SERIAL field contains a series incremental integer values. These values are not necessarily continuous or unique. For example, the following is a valid series of values for a SERIAL field: 1, 2, 3, 17, 18, 25, 25, 26, 27. Sequential integers are either Caché-generated or user-supplied; nonsequential integers are user-supplied. If you wish SERIAL field values to be unique, you must apply a UNIQUE constraint on the field.

An **UPDATE** operation can only change a serial field value if the field currently has no value (NULL), or its value is 0. Otherwise, an SQLCODE -105 error is generated.

Caché imposes no restriction on the number of SERIAL fields in a table.

ROWVERSION Data Type

The ROWVERSION data type defines a read-only field that contains a unique system-assigned positive integer, beginning with 1. Caché assigns sequential integers as part of each insert, update, or %Save operation. These values are not user-modifiable.

Caché maintains a single row version counter namespace-wide. All tables in a namespace that contain a ROWVERSION field share the same row version counter. Thus, the ROWVERSION field provides row-level version control, allowing you to determine the order in which changes were made to rows in one or more tables in a namespace.

You can only specify one field of ROWVERSION data type per table.

The ROWVERSION field should not be included in a unique key or primary key. The ROWVERSION field cannot be part of an IDKey index.

For details on using ROWVERSION, refer to [RowVersion Field](#) section of the “Defining Tables” chapter of *Using Caché SQL*.

ROWVERSION and %Counter

Both ROWVERSION and %Counter (%Library.Counter) assign a sequential integer to a field as part of an **INSERT** operation. But these two counters are significantly different and are used for different purposes:

- The ROWVERSION counter is at the namespace level. The %Counter counter is at the table level. These two counters are completely independent of each other and independent of the RowID counter.
- The ROWVERSION counter is incremented by insert, update, or %Save operations. The %Counter counter is only incremented by insert operations.
- A ROWVERSION field value cannot be user-specified; the value is always supplied from the ROWVERSION counter. A %Counter field value is supplied from the table’s %Counter counter during an insert if you do not specify a value for this field. If an insert supplies a %Counter integer value, that value is inserted rather than the current counter value:
 - If an insert supplies a %Counter field value greater than the current counter value, Caché inserts that value and resets the %Counter counter to the next sequential integer.
 - If an insert supplies a %Counter field value lesser than the current counter value, Caché does not reset the %Counter counter.
 - An insert can supply a %Counter field value as a negative integer or a fractional number. Caché truncates a fractional number to its integer component. If the supplied %Counter field value is 0 (or truncates to 0), Caché inserts the current counter value.

You cannot update an existing %Counter field value.

- A ROWVERSION value is always unique. Because you can insert a user-specified %Counter value, you must specify a UNIQUE field constraint to guarantee unique %Counter values.

- The ROWVERSION counter cannot be reset. A **TRUNCATE TABLE** resets the %Counter counter; performing a **DELETE** on all rows does not reset the %Counter counter.
- Only one ROWVERSION field is allowed per table. You can specify multiple %Counter fields in a table.

DDL Data Types Exposed by Caché ODBC / JDBC

Caché ODBC exposes a subset of the DDL data types, and maps other data types to this subset of data types. These mappings are not reversible. For example, the statement `CREATE TABLE mytable (f1 BINARY)` creates a Caché class that is projected to ODBC as `mytable (f1 VARBINARY)`. A Caché list data type is projected to ODBC as a VARCHAR string.

ODBC exposes the following data types: BIGINT, BIT, DATE, DOUBLE, GUID, INTEGER, LONGVARBINARY, LONGVARCHAR, NUMERIC, OREF, SMALLINT, TIME, TIMESTAMP, TINYINT, VARBINARY, VARCHAR. Note that, by default, Caché establishes a system-wide ODBC VARCHAR maximum length of 4096; this [ODBC maximum length is configurable](#).

When one of these ODBC/JDBC data type values is mapped to Caché SQL, the following operations occur: DOUBLE data is cast using `$DOUBLE`. NUMERIC data is cast using `$DECIMAL`.

The GUID data type corresponds to Caché SQL UNIQUEIDENTIFIER data type. Failing to specify a valid value to a GUID / UNIQUEIDENTIFIER field generates a #7212 General Error.

Query Metadata Returns Data Type

You can use Dynamic SQL to return metadata about a query, including the data type of a specified column in the query.

The following Dynamic SQL examples return the column name and the integer code for the ODBC data type for each of the columns in Sample.Person and Sample.Employee:

```
SET myquery="SELECT * FROM Sample.Person"
SET rset = ##class(%SQL.Statement).%New()
SET tStatus = rset.%Prepare(myquery)
SET x=rset.%Metadata.columns.Count()
WHILE x>0 {
  SET column=rset.%Metadata.columns.GetAt(x)
  WRITE !,x," ",column.colName," ",column.ODBCType
  SET x=x-1 }
WRITE !,"end of columns"
```

```
SET myquery="SELECT * FROM Sample.Employee"
SET rset = ##class(%SQL.Statement).%New()
SET tStatus = rset.%Prepare(myquery)
SET x=rset.%Metadata.columns.Count()
WHILE x>0 {
  SET column=rset.%Metadata.columns.GetAt(x)
  WRITE !,x," ",column.colName," ",column.ODBCType
  SET x=x-1 }
WRITE !,"end of columns"
```

List structured data, such as the FavoriteColors column in Sample.Person, returns a data type of 12 (VARCHAR) because ODBC represents a Caché %List data type value as a string of comma-separated values.

Streams data, such as the Notes and Picture columns in Sample.Employee, return the data types -1 (LONGVARCHAR) or -4 (LONGVARBINARY).

A ROWVERSION field returns data type -5 because %Library.RowVersion is a subclass of %Library.BigInt.

For further details, refer to the [Dynamic SQL](#) chapter of *Using Caché SQL* and the %SQL.Statement class in the *InterSystems Class Reference*.

Integer Codes for Data Types

In query metadata and other contexts, the defined data type for a column may be returned as an integer code. There are two sets of integer codes used to represent data types:

- Client data type codes are returned by the `%Library.ResultSet.GetColumnType()` method. These values are listed in the `%Library.ResultSet` class in the *InterSystems Class Reference*.
- xDBC data type codes (SQLType) are used by ODBC and JDBC. They are returned by `%SQL.Statement.%Metadata.columns.GetAt()` method, as shown in the example above. The JDBC codes are the same as the ODBC codes, except in the representation of time and date data types. These ODBC and JDBC values are listed below:

<i>ODBC</i>	<i>JDBC</i>	<i>Data Type</i>
-11	-11	GUID
-7	-7	BIT
-6	-6	TINYINT
-5	-5	BIGINT
-4	-4	LONGVARBINARY
-3	-3	VARBINARY
-2	-2	BINARY
-1	-1	LONGVARCHAR
0	0	Unknown type
1	1	CHAR
2	2	NUMERIC
3	3	DECIMAL
4	4	INTEGER
5	5	SMALLINT
6	6	FLOAT
7	7	REAL
8	8	DOUBLE
9	91	DATE
10	92	TIME
11	93	TIMESTAMP
12	12	VARCHAR

For further details, refer to the [Dynamic SQL](#) chapter of *Using Caché SQL*.

Caché also supports Unicode SQL types for ODBC applications working with multibyte character sets, such as in Chinese, Hebrew, Japanese, or Korean locales.

<i>ODBC</i>	<i>Data Type</i>
-10	WLONGVARCHAR
-9	WVARCHAR

To activate this functionality, refer to “Creating a DSN by Using the Control Panel” in *Using Caché with ODBC*.

Creating User-Defined DDL Data Types

You can modify the set of data types either by overriding the data type mapping for a system data type parameter value, or by defining a new user data type. You can modify system data types to override the InterSystems default mappings. You can create user-defined data types to provide additional data type mappings that InterSystems does not supply.

To view and modify or add to the current user data type mappings, Go to the Management Portal, select **[System] > [Configuration] > [User-defined DDL Mappings]**. To add a user data type, select **Create New User-defined DDL Mapping**. In the displayed box, input a **Name**, for example `VARCHAR(100)` and a **Datatype**, for example `MyString100(MAXLEN=100)`.

The result will be an entry in the list of user-defined DDL data types.

As shown in previous examples, there are several useful routines for entering user-defined DDL data types:

- **maxval^%apiSQL()** — Given a precision and scale, returns the maximum valid value (MAXVAL) for each of the Caché numeric data types. The syntax is:

```
maxval^%apiSQL(precision,scale)
```

where both precision and scale are required.

- **minval^%apiSQL()** — Given a precision and scale, returns the minimum valid value (MINVAL) for each of the Caché numeric data types. The syntax is:

```
minval^%apiSQL(precision,scale)
```

where both precision and scale are required.

If you need to map a DDL data type to a Caché property with a collection type of Stream, specify `%Stream.GlobalCharacter` for Character Stream data and `%Stream.GlobalBinary` for Binary Stream data. (Prior to version 2011.1, character stream data was assigned the `CStream%String` (or `%Library.GlobalCharacterStream`) data type, and binary stream data was assigned the `BStream%String` (or `%Library.GlobalBinaryStream`) data type. These older data types continue to be fully supported.)

Pass-through if No DDL Mapping is Found

If DDL encounters a data type not in the DDL data type column of the **SystemDataTypes** table, it next examines the **UserDataTypes** table. If no mapping appears for the data type in either table, no conversion of the data type occurs, and the data type passes directly to the class definition as specified in DDL.

For example, the following field definitions could appear in a DDL statement:

```
CREATE TABLE TestTable (
    Field1 %String,
    Field2 %String(MAXLEN=45)
)
```

Given the above definitions, if DDL finds no mappings for `%String` or `%String(MAXLEN=%1)` or `%String(MAXLEN=45)` in **SystemDataTypes** or **UserDataTypes**, then the `%String` and `%String(MAXLEN=45)` types are passed directly to the appropriate class definition.

Converting Data Types

To convert data from one data type to another, use the **CAST** or **CONVERT** function.

CAST supports conversion to several character string and numeric data types, as well as to **DATE**, **TIME**, and **TIMESTAMP** data types.

CONVERT has two syntactical forms. Both forms support conversion to and from **DATE**, **TIME**, and **TIMESTAMP** data types, as well as conversion between other data types.

CAST and CONVERT Handling of VARCHAR

The VARCHAR data type (with no specified size) is mapped to a MAXLEN of 1 character, as shown in the above table. However, when you **CAST** or **CONVERT** a value to VARCHAR, the default size mapping is 30 characters. This default size of 30 characters is provided for compatibility with non-Caché software requirements.

See Also

- [CAST, CONVERT](#)
- [TO_CHAR, TO_DATE, TO_NUMBER](#)

Date and Time Constructs

Validates and converts an ODBC date, time, or timestamp.

```
{d 'yyyy-mm-dd' }
{t 'hh:mm:ss[.fff]' }
{ts 'yyyy-mm-dd [hh:mm:ss.fff]' }
{ts 'mm/dd/yyyy [hh:mm:ss.fff]' }
{ts 'nnnnn' }
```

Description

These constructs take a string in ODBC date, time, or timestamp format and convert it to the corresponding Caché date, time, or timestamp format. They perform data typing and value and range checking.

{d 'string'}

The {d 'string'} date construct validates a date in ODBC format. If the date is valid, it stores it (logical mode) in Caché [\\$HOROLOG](#) date format as an integer count value from 1840-12-31. Caché does not append a default time value.

If you supply:

- An invalid date (such as a date not in ODBC format or the date 02-29 in a non-leap year): Caché generates an SQLCODE -146 error: “yyyy-mm-dd' is an invalid ODBC/JDBC Date value”.
- A date prior to 1840-12-31: Caché generates an SQLCODE -146 error.
- An ODBC timestamp value: Caché validates both the date and time portions of the timestamp. If both are valid, it converts the date portion only. If either date or time are invalid, the system generates an SQLCODE -146 error.

{t 'string'}

The {t 'string'} time construct validates a time in ODBC format. If the time is valid, it stores it (logical mode) in Caché [\\$HOROLOG](#) time format as an integer count of seconds from midnight, with the specified fractional seconds. Caché Display mode and ODBC mode do not display the fractional seconds; the fractional seconds are truncated from these display formats.

If you supply:

- An invalid time (such as a time not in ODBC format or a time with hour >23): Caché generates an SQLCODE -147 error: “hh:mi:ss.fff' is an invalid ODBC/JDBC Time value”.
- An ODBC timestamp value: Caché generates an SQLCODE -147 error.

{ts 'string'}

The {ts 'string'} timestamp construct validates a date/time and returns it in ODBC timestamp format; specified fractional seconds are always preserved and displayed. The {ts 'string'} timestamp construct also validates a date and returns it in ODBC timestamp format with a supplied time value of 00:00:00.

If you supply:

- A valid timestamp in ODBC format: Caché stores the supplied value unchanged. This is because Caché timestamp format is the same as ODBC timestamp format.
- A valid timestamp using the locale default date and time formats (for example, 2/29/2004 12:23:46.77): Caché stores and displays the supplied value in ODBC format.
- An invalid timestamp (such as a timestamp with the date portion specifying 02-29 in a non-leap year, or with the time portion specifying hour >23): Caché returns the string “error” as the value.

- A timestamp specifying a date prior to 1840-12-31: Caché returns the string “error” as the value.
- A valid date (in ODBC or locale format) with no time value: Caché appends a time value of 00:00:00, then stores the resulting timestamp in ODBC format. It supplies leading zeros where necessary. For example, 2/29/2004 returns 2004-02-29 00:00:00.
- A valid \$HOROLOG integer date (0 through 2980013): Caché appends a time value of 00:00:00, then stores the resulting timestamp in ODBC format. For example, 59594 returns 2004-02-29 00:00:00. Note that a \$HOROLOG date integer cannot have leading zeros.
- A correctly formatted, but invalid, date (in ODBC or locale format) with no time value: Caché appends a time value of 00:00:00. It then stores the date portion as supplied. For example, 02/29/2003 returns 02/29/2003 00:00:00. 1776-07-04 returns 1776-07-04 00:00:00.
- An incorrectly formatted and invalid, date (in ODBC, locale, or \$HOROLOG format) with no time value: Caché returns the string “error”. For example, 2/29/2003 (no leading zero and invalid date value) returns “error”. 00234 (\$HOROLOG with leading zeros) returns “error”

See the [\\$HOROLOG](#) special variable in the *Caché ObjectScript Reference* for further information.

Examples

The following Dynamic SQL example validates dates supplied in ODBC format (with or without leading zeros) and stores them as the equivalent \$HOROLOG value 59594. This example displays %SelectMode 0 (logical) values:

```
SET myquery = 2
SET myquery(1) = "SELECT {d '2004-02-29'} AS date1,"
SET myquery(2) = "{d '2004-2-29'} AS date2"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following Dynamic SQL example validates times supplied in ODBC format (with or without leading zeros) and stores them as the equivalent \$HOROLOG value 43469. This example displays %SelectMode 0 (logical) values:

```
SET myquery = 3
SET myquery(1) = "SELECT {t '12:04:29'} AS time1,"
SET myquery(2) = "{t '12:4:29'} AS time2,"
SET myquery(3) = "{t '12:04:29.00000'} AS time3"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following Dynamic SQL example validates times supplied in ODBC format with fractional seconds, and stores them as the equivalent \$HOROLOG value 43469 with the fractional seconds appended. Trailing zeros are truncated. This example displays %SelectMode 0 (logical) values:

```
SET myquery = 3
SET myquery(1) = "SELECT {t '12:04:29.987'} AS time1,"
SET myquery(2) = "{t '12:4:29.987'} AS time2,"
SET myquery(3) = "{t '12:04:29.987000'} AS time3"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The following Dynamic SQL example validates time and date values in several formats and stores them as the equivalent ODBC timestamp. A time value of 00:00:00 is supplied when necessary. This example displays %SelectMode 0 (logical) values:

```
SET myquery = 6
SET myquery(1) = "SELECT {ts '2011-02-14 01:43:38'} AS ts1,"
SET myquery(2) = "{ts '2011-02-14'} AS ts2,"
SET myquery(3) = "{ts '02/14/2011 01:43:38.999'} AS ts3,"
SET myquery(4) = "{ts '2/14/2011 01:43:38'} AS ts4,"
SET myquery(5) = "{ts '02/14/2011'} AS ts5,"
SET myquery(6) = "{ts '62136'} AS ts6"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=0
SET tStatus = tStatement.%Prepare(.myquery)
SET rset = tStatement.%Execute()
IF rset.%Next() {
WRITE rset.ts1,!
WRITE rset.ts2,!
WRITE rset.ts3,!
WRITE rset.ts4,!
WRITE rset.ts5,!
WRITE rset.ts6
}
```

Default user name and password

Provides default login identity.

Description

The default user name and password for Caché provide a basic way to log in to the database and get started. The default user name is “_SYSTEM” (uppercase) and “SYS” is its password.

Field constraint

Specifies rules about a field's contents.

Description

A field constraint specifies rules governing the data values permitted for a field. A field may have the following constraints:

- **NOT NULL:** You must specify a value for this field in every record (empty strings acceptable).
- **UNIQUE:** If you specify a value for this field in a record, it must be a unique value (one empty string acceptable). You can, however, create multiple records with no value (NULL) for the field.
- **DEFAULT:** You must either specify a value or Caché provides a default for this field in every record (empty strings acceptable). The default may be NULL, an empty string, or any other value appropriate for the data type.
- **UNIQUE NOT NULL:** You must specify a unique value for this field in every record (one empty string acceptable). Can be used as a primary key.
- **DEFAULT NOT NULL:** You must either specify a value or Caché provides a default value for this field in every record (empty strings acceptable).
- **UNIQUE DEFAULT:** *Not Recommended* — You must either specify a unique value or Caché provides a default value for this field in every record (one empty string acceptable). The default may be NULL, an empty string, or any other value appropriate for the data type. Use only if the default is a unique generated value (for example, CURRENT_TIMESTAMP), or if the default is intended to be used only once.
- **UNIQUE DEFAULT NOT NULL:** *Not Recommended* — You must either specify a unique value or Caché provides a default value for this field in every record (one empty string acceptable). The default may be an empty string or any other value appropriate for the data type; it cannot be NULL. Use only if the default is a unique generated value (for example, CURRENT_TIMESTAMP), or if the default is intended to be used only once. Can be used as a primary key.
- **IDENTITY:** Caché provides a unique, system-generated, non-modifiable integer value for this field in every record. Other field constraint keywords are ignored. Can be used as a primary key.

Data values must be appropriate for the field's data type. An empty string is not an acceptable value for a numeric field.

These field constraints are further described in the page for the [CREATE TABLE](#) command.

Reserved words

A list of SQL reserved words.

```
%AFTERHAVING | %ALLINDEX | %ALPHAUP | %ALTER | %BEGTRANS |
%CHECKPRIV | %CLASSNAME | %CLASSPARAMETER | %DEBUGFULL | %DELDATA |
%DESCRIPTION | %EXACT | %EXTERNAL | %FILE | %FIRSTTABLE | %FLATTEN |
%FOREACH | %FULL | %ID | %IDADDED | %IGNOREINDEX | %IGNOREINDICES |
%INLIST | %INORDER | %INTERNAL | %INTEXT | %INTRANS | %INTRANSACTION |
%KEY | %MATCHES | %MCODE | %MERGE | %MINUS | %MVR | %NOCHECK |
%NODELDATA | %NOFLATTEN | %NOFLPLAN | %NOINDEX | %NOLOCK |
%NOMERGE | %NOPARALLEL | %NOREDUCE | %NOSVSO | %NOTOPOPT |
%NOTRIGGER | %NOUNIONOPT | %NUMROWS | %ODBCIN | %ODBCOUT |
%PARALLEL | %PLUS | %PROFILE | %PROFILE_ALL | %PUBLICROWID | %ROUTINE |
%ROWCOUNT | %RUNTIMEIN | %RUNTIMEOUT | %STARTSWITH |
%STARTTABLE | %SQLSTRING | %SQLUPPER | %STRING | %TABLENAME |
%TRUNCATE | %UPPER | %VALUE | %VID
ABSOLUTE | ADD | ALL | ALLOCATE | ALTER | AND | ANY | ARE | AS |
ASC | ASSERTION | AT | AUTHORIZATION | AVG | BEGIN | BETWEEN |
BIT | BIT_LENGTH | BOTH | BY | CASCADE | CASE | CAST |
CHAR | CHARACTER | CHARACTER_LENGTH | CHAR_LENGTH |
CHECK | CLOSE | COALESCE | COLLATE | COMMIT | CONNECT |
CONNECTION | CONSTRAINT | CONSTRAINTS | CONTINUE | CONVERT |
CORRESPONDING | COUNT | CREATE | CROSS | CURRENT |
CURRENT_DATE | CURRENT_TIME | CURRENT_TIMESTAMP |
CURRENT_USER | CURSOR | DATE | DEALLOCATE | DEC | DECIMAL |
DECLARE | DEFAULT | DEFERRABLE | DEFERRED | DELETE | DESC |
DESCRIBE | DESCRIPTOR | DIAGNOSTICS | DISCONNECT | DISTINCT |
DOMAIN | DOUBLE | DROP | ELSE | END | ENDEXEC | ESCAPE | EXCEPT |
EXCEPTION | EXEC | EXECUTE | EXISTS | EXTERNAL | EXTRACT |
FALSE | FETCH | FIRST | FLOAT | FOR | FOREIGN | FOUND | FROM | FULL |
GET | GLOBAL | GO | GOTO | GRANT | GROUP | HAVING | HOUR |
IDENTITY | IMMEDIATE | IN | INDICATOR | INITIALLY |
INNER | INPUT | INSENSITIVE | INSERT | INT | INTEGER | INTERSECT |
INTERVAL | INTO | IS | ISOLATION | JOIN | LANGUAGE | LAST |
LEADING | LEFT | LEVEL | LIKE | LOCAL | LOWER | MATCH | MAX | MIN |
MINUTE | MODULE | NAMES | NATIONAL | NATURAL | NCHAR |
NEXT | NO | NOT | NULL | NULLIF | NUMERIC | OCTET_LENGTH | OF | ON |
ONLY | OPEN | OPTION | OR | OUTER | OUTPUT | OVERLAPS |
PAD | PARTIAL | PREPARE | PRESERVE | PRIMARY | PRIOR | PRIVILEGES |
PROCEDURE | PUBLIC | READ | REAL | REFERENCES | RELATIVE |
RESTRICT | REVOKE | RIGHT | ROLE | ROLLBACK | ROWS |
SCHEMA | SCROLL | SECOND | SECTION | SELECT | SESSION_USER |
SET | SMALLINT | SOME | SPACE | SQLERROR | SQLSTATE | STATISTICS |
SUBSTRING | SUM | SYSDATE | SYSTEM_USER | TABLE | TEMPORARY |
THEN | TIME | TIMEZONE_HOUR | TIMEZONE_MINUTE | TO | TOP |
TRAILING | TRANSACTION | TRIM | TRUE | UNION | UNIQUE |
UPDATE | UPPER | USER | USING | VALUES | VARCHAR | VARYING | WHEN |
WHenever | WHERE | WITH | WORK | WRITE
```

Description

Within SQL certain words are *reserved*. You cannot use an SQL reserved word as an SQL [identifier](#) (such as the name for a table, a column, an AS alias, or other entity), unless:

- The word is delimited with double quotes ("*word*"), and
- Delimited identifiers are supported. For further details, refer to the [Identifiers](#) in *Using Caché SQL*.

This list contains only those words that are reserved in this sense; it does not contain all SQL keywords. Several of the words listed above start with the "%" character, indicating that they are Caché SQL proprietary extension keywords. In general, it is not recommended to use words that begin with "%" as identifiers such as table and column names, because new Caché SQL extension keywords may be added in the future.

You can check if a word is an SQL reserved word by invoking the **IsReservedWord()** method, as shown in the following example. Specify the reserved word as a quoted string; reserved words are not case-sensitive.

\$SYSTEM.SQL.IsReservedWord() returns a boolean value.

```
WRITE !,"Reserved?: ", $SYSTEM.SQL.IsReservedWord("VARCHAR" )
WRITE !,"Reserved?: ", $SYSTEM.SQL.IsReservedWord("varchar" )
WRITE !,"Reserved?: ", $SYSTEM.SQL.IsReservedWord("VarChar" )
WRITE !,"Reserved?: ", $SYSTEM.SQL.IsReservedWord("FRED" )
```

This method can also be called as a stored procedure from ODBC or JDBC: %SYSTEM.SQL_IsReservedWord("nnnn").

Special Variables

System-supplied variables.

```
$HOROLOG
$JOB
$NAMESPACE
$TLEVEL
$USERNAME
$ZHOROLOG
$ZJOB
$ZPI
$ZTIMESTAMP
$ZTIMEZONE
$ZVERSION
```

Description

Caché SQL directly supports a number of the ObjectScript special variables. These variables contain system-supplied values. They can be used wherever a literal value can be specified in Caché SQL.

SQL special variable names are not case-sensitive. Most can be specified using an abbreviation.

Variable Name	Abbreviation	Data Type Returned	Use
\$HOROLOG	\$H	%String/VARCHAR	Local date and time for the current process
\$JOB	\$J	%String/VARCHAR	Job ID of the current process
\$NAMESPACE	none	%String/VARCHAR	Current namespace name
\$TLEVEL	\$TL	%Integer/INTEGER	Current transaction nesting level
\$USERNAME	none	%String/VARCHAR	User name for the current process
\$ZHOROLOG	\$ZH	%Numeric/NUMERIC(21,6)	Number of elapsed seconds since Caché startup
\$ZJOB	\$ZJ	%Integer/INTEGER	Job status for the current process
\$ZPI	none	%Numeric/NUMERIC(21,18)	The numeric constant PI
\$ZTIMESTAMP	\$ZTS	%String/VARCHAR	Current date and time in Coordinated Universal Time format
\$ZTIMEZONE	\$ZTZ	%Integer/INTEGER	Local time zone offset from GMT
\$ZVERSION	\$ZV	%String/VARCHAR	The current version of Caché

For further details, refer to the corresponding ObjectScript special variable, as described in the *Caché ObjectScript Reference*.

Examples

The following example returns a result set that includes the current date and time:

```
SELECT TOP 5 Name,$H  
FROM Sample.Person
```

The following example only returns a result set if the time zone is within the continental United States:

```
SELECT TOP 5 Name,Home_State  
FROM Sample.Person  
WHERE $ZTIMEZONE BETWEEN 300 AND 480
```

String Manipulation

String manipulation functions and operators.

Description

Caché SQL provides support for several types of string manipulation:

- Strings can be manipulated by length, character position, or substring value.
- Strings can be manipulated by a designated delimiter character or delimiter string.
- Strings can be tested by pattern matching and word-aware searches.
- Specially encoded strings, called lists, contain embedded substring identifiers without using a delimiter character. The various **\$LIST** functions operate on these encoded character strings, which are incompatible with standard character strings. The only exceptions are the **\$LISTGET** function and the one-argument and two-argument forms of **\$LIST**, which take an encoded character string as input, but output a single element value as a standard character string.

Caché SQL supports string functions, string condition expressions, and string operators.

Caché string manipulation is case-sensitive. Letters in strings can be converted to uppercase, to lowercase, or retained as mixed case. [String collation](#) can be case-sensitive, or not case-sensitive; by default, SQL string collation is `SQLUPPER` which is not case-sensitive. Caché SQL provides numerous letter case and [collation functions](#) and operators.

When a string is specified for a numeric argument, most Caché SQL functions perform the following string-to-number conversions: a nonnumeric string is converted to the number 0; a numeric string is converted to a canonical number; and a mixed-numeric string is truncated at the first nonnumeric character and then converted to a canonical number.

String Concatenation

The following functions concatenate substrings into a string:

- **CONCAT**: concatenates two substrings, returns a single string.
- **STRING**: concatenates two or more substrings, returns a single string.
- **XMLAGG**: concatenates all of the values of a column, returns a single string. For further details, see [Aggregate Functions](#).
- **LIST**: concatenates all of the values of a column, including a comma delimiter, returns a single string. For further details, see [Aggregate Functions](#).

The concatenate operator (`||`) can also be used to concatenate two strings.

String Length

The following functions can be used to determine the length of a string:

- **CHARACTER_LENGTH** and **CHAR_LENGTH**: return the number of characters in a string, including trailing blanks. `NULL` returns `NULL`.
- **LENGTH**: returns the number of characters in a string, excluding trailing blanks. `NULL` returns `NULL`.
- **\$LENGTH**: returns the number of characters in a string, including trailing blanks. `NULL` is returned as 0.

Truncation and Trimming

The following functions can be used to truncate or trim a string. Truncation limits the length of the string, deleting all characters beyond the specified length. Trimming deletes leading and/or trailing blank spaces from a string.

- Truncation: **CONVERT**, **%SQLSTRING**, **%SQLUPPER**, and **%STRING**.

- Trimming: [TRIM](#), [LTRIM](#), and [RTRIM](#).

Substring Search

The following functions search for a substring within a string and return a string position:

- [POSITION](#): searches by substring value, finds first match, returns position of beginning of substring.
- [CHARINDEX](#): searches by substring value, finds first match, returns position of beginning of substring. Starting point can be specified.
- [\\$FIND](#): searches by substring value, finds first match, returns position of end of substring. Starting point can be specified.
- [INSTR](#): searches by substring value, finds first match, returns position of beginning of substring. Both starting point and substring occurrence can be specified.

The following functions search for a substring by position or delimiter within a string and return the substring:

- [\\$EXTRACT](#): searches by string position, returns substring specified by start position, or start and end positions. Searches from beginning of string.
- [SUBSTRING](#): searches by string position, returns substring specified by start position, or start and length. Searches from beginning of string.
- [SUBSTR](#): searches by string position, returns substring specified by start position, or start and length. Searches from beginning or end of string.
- [\\$PIECE](#): searches by delimiter character, returns first delimited substring. Starting point can be specified or defaults to beginning of string.
- [\\$LENGTH](#): searches by delimiter character, returns the number of delimited substrings. Searches from beginning of string.
- [\\$LIST](#): searches by substring count on a specially encoded list string. It locates a substring by substring count and returns the substring value. Searches from beginning of string.

The contains operator (`()`) can also be used to determine if a substring appears in a string.

The [%STARTSWITH](#) comparison operator matches the specified character(s) against the beginning of a string.

Substring Search-and-Replace

The following functions search for a substring within a string and replace it with another substring.

- [REPLACE](#): searches by string value, replaces substring with new substring. Searches from beginning of string.
- [STUFF](#): searches by string position and length, replaces substring with new substring. Searches from beginning of string.

Character-Type and Word-Aware Comparisons

The [%PATTERN](#) comparison operator matches a string to a specified pattern of character types.

The [%CONTAINS](#) and [%CONTAINSTERM](#) comparison operators perform a word-aware search of a string for specified words or phrases.