



Caché Transact-SQL (TSQL) Migration Guide

Version 2017.2
2020-06-25

Caché Transact-SQL (TSQL) Migration Guide
Caché Version 2017.2 2020-06-25
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
1 Overview	3
1.1 Getting Started	3
1.1.1 Configuring TSQL	3
1.1.2 Migrating Source Code	3
1.1.3 Migrating the Data	4
1.2 TSQL Language Implementation	4
2 Caché TSQL Constructs	7
2.1 Commonly Used Constructs	7
2.1.1 Table References	7
2.1.2 Temporary Tables	7
2.1.3 Transactions	8
2.1.4 Cursor Name Management	8
2.1.5 System Stored Procedures	8
2.1.6 System Tables	9
2.1.7 SYSOBJECTS References	9
3 Caché TSQL Language Elements	11
3.1 Literals	11
3.1.1 String Literals	11
3.1.2 Empty Strings	11
3.1.3 NULL	12
3.1.4 Hexadecimal	12
3.1.5 Reserved Words	12
3.1.6 Comments, Blank Lines, and Semicolons	12
3.2 Identifiers	13
3.2.1 Delimited and Quoted Identifiers	13
3.3 Data Types	14
3.4 Operators	15
3.4.1 Arithmetic and Equality Operators	15
3.4.2 Concatenation Operator	15
3.4.3 Comparison Operators	16
3.4.4 NOT Logical Operator	16
3.4.5 Bitwise Logical Operators	16
4 TSQL Commands	17
4.1 Data Definition Language (DDL) Statements	17
4.1.1 CREATE TABLE	17
4.1.2 ALTER TABLE	18
4.1.3 DROP TABLE	19
4.1.4 CREATE INDEX	19
4.1.5 DROP INDEX	20
4.1.6 CREATE TRIGGER	20
4.1.7 DROP TRIGGER	21
4.1.8 CREATE VIEW	21
4.1.9 DROP VIEW	21
4.1.10 CREATE DATABASE	21

4.1.11 DROP DATABASE	21
4.2 Data Management Language (DML) Statements	22
4.2.1 DELETE	22
4.2.2 INSERT	22
4.2.3 UPDATE	23
4.2.4 READTEXT, UDATETEXT, WRITETEXT	23
4.2.5 TRUNCATE TABLE	24
4.3 Query Statements	24
4.3.1 SELECT	24
4.3.2 JOIN	26
4.3.3 UNION	27
4.3.4 FETCH Cursor	27
4.3.5 UPDATE STATISTICS	27
4.4 Flow of Control Statements	27
4.4.1 IF	27
4.4.2 WHILE	29
4.4.3 CASE	29
4.4.4 GOTO and Labels	29
4.4.5 WAITFOR	30
4.5 Assignment Statements	30
4.5.1 DECLARE	30
4.5.2 SET	30
4.6 Transaction Statements	31
4.6.1 SET TRANSACTION ISOLATION LEVEL	31
4.6.2 BEGIN TRANSACTION	31
4.6.3 COMMIT TRANSACTION	32
4.6.4 ROLLBACK TRANSACTION	32
4.6.5 CHECKPOINT	32
4.6.6 LOCK TABLE	32
4.7 Procedure Statements	33
4.7.1 CREATE PROCEDURE / CREATE FUNCTION	33
4.7.2 ALTER FUNCTION	34
4.7.3 DROP FUNCTION	34
4.7.4 DROP PROCEDURE	34
4.7.5 RETURN	35
4.7.6 EXECUTE	35
4.7.7 CALL	35
4.8 Other Statements	36
4.8.1 CREATE USER	36
4.8.2 CREATE ROLE	36
4.8.3 GRANT and REVOKE	36
4.8.4 PRINT	36
4.8.5 RAISERROR	36
4.8.6 UPDATE STATISTICS	37
4.8.7 USE database	37
4.9 InterSystems Extensions	37
4.9.1 CACHE	37
4.9.2 IMPORTASQUERY	38
5 TSQL Settings	39
5.1 DIALECT	39

5.2 ANSI_NULLS	40
5.3 CASEINSCOMPARE	40
5.4 QUOTED_IDENTIFIER	41
5.5 TRACE	41
6 TSQL Functions	43
6.1 Supported Functions	43
6.1.1 ABS	43
6.1.2 ACOS	43
6.1.3 ASCII	43
6.1.4 ASIN	43
6.1.5 ATAN	43
6.1.6 AVG	44
6.1.7 CAST	44
6.1.8 CEILING	44
6.1.9 CHAR	44
6.1.10 CHAR_LENGTH / CHARACTER_LENGTH	44
6.1.11 CHARINDEX	45
6.1.12 COALESCE	45
6.1.13 COL_NAME	45
6.1.14 CONVERT	45
6.1.15 COS	46
6.1.16 COT	46
6.1.17 COUNT	46
6.1.18 CURRENT_DATE	46
6.1.19 CURRENT_TIME	47
6.1.20 CURRENT_TIMESTAMP	47
6.1.21 CURRENT_USER	47
6.1.22 DATALENGTH	47
6.1.23 DATEADD	47
6.1.24 DATEDIFF	48
6.1.25 DATENAME	49
6.1.26 DATEPART	50
6.1.27 DAY	50
6.1.28 DB_NAME	50
6.1.29 DEGREES	50
6.1.30 ERROR_MESSAGE	50
6.1.31 ERROR_NUMBER	50
6.1.32 EXEC	50
6.1.33 EXP	51
6.1.34 FLOOR	51
6.1.35 GETDATE	51
6.1.36 GETUTCDATE	51
6.1.37 HOST_NAME	51
6.1.38 INDEX_COL	51
6.1.39 ISNULL	52
6.1.40 ISNUMERIC	52
6.1.41 LEFT	52
6.1.42 LEN	52
6.1.43 LOG	52
6.1.44 LOG10	52

6.1.45 LOWER	52
6.1.46 LTRIM	52
6.1.47 MAX	53
6.1.48 MIN	53
6.1.49 MONTH	53
6.1.50 NCHAR	53
6.1.51 NEWID	53
6.1.52 NOW	54
6.1.53 NULLIF	54
6.1.54 OBJECT_ID	54
6.1.55 OBJECT_NAME	54
6.1.56 PATINDEX	54
6.1.57 PI	55
6.1.58 POWER	55
6.1.59 QUOTENAME	55
6.1.60 RADIANS	55
6.1.61 RAND	56
6.1.62 REPLACE	56
6.1.63 REPLICATE	56
6.1.64 REVERSE	56
6.1.65 RIGHT	56
6.1.66 ROUND	56
6.1.67 RTRIM	57
6.1.68 SCOPE_IDENTITY	57
6.1.69 SIGN	57
6.1.70 SIN	57
6.1.71 SPACE	57
6.1.72 SQRT	57
6.1.73 SQUARE	58
6.1.74 STR	58
6.1.75 STUFF	58
6.1.76 SUBSTRING	58
6.1.77 SUM	58
6.1.78 SUSER_NAME	59
6.1.79 SUSER_SNAME	59
6.1.80 TAN	59
6.1.81 TEXTPTR	59
6.1.82 TEXTVALID	59
6.1.83 UNICODE	59
6.1.84 UPPER	59
6.1.85 USER	60
6.1.86 USER_NAME	60
6.1.87 YEAR	60
6.2 Unsupported Functions	60
7 TSQL Variables	61
7.1 Local Variables	61
7.1.1 Declaring a Local Variable	61
7.1.2 Setting a Local Variable	61
7.1.3 Initial and Default Values	62
7.1.4 Plain Local Variables	62

7.2 @@ Special Variables	62
7.2.1 @@ERROR	62
7.2.2 @@FETCH_STATUS	63
7.2.3 @@IDENTITY	63
7.2.4 @@LOCK_TIMEOUT	63
7.2.5 @@NESTLEVEL	63
7.2.6 @@ROWCOUNT	64
7.2.7 @@SERVERNAME	64
7.2.8 @@SPID	64
7.2.9 @@SQLSTATUS	64
7.2.10 @@TRANCOUNT	65
7.2.11 @@VERSION	65
8 Using the TSQL Shell	67
8.1 Other Ways to Execute Transact-SQL Code	67

About This Book

This book describes how to migrate schemas and stored procedures from Sybase or SQL Server and it will provide you with an understanding of the TSQL (Transact-SQL) implementation in Caché.

The book addresses a number of topics:

- An [Overview](#), which includes configuring TSQL and migrating source code and data.
- [Caché TSQL Constructs](#) including temporary tables, stored procedures, and transaction management.
- [Caché TSQL Language Elements](#): data types, operators, literals, reserved words.
- [Caché TSQL Commands](#)
- [Caché TSQL Settings](#)
- [Caché TSQL Functions](#)
- [Caché TSQL Variables](#)
- [The Caché TSQL Shell](#)

For a detailed outline, see the [Table of Contents](#).

When using Caché TSQL, you may find the following additional sources useful:

- [The Caché SQL Reference](#) provides details on individual SQL commands and functions, as well as information on the Caché SQL configuration settings, error codes, data types, and reserved words.
- “[Using the Caché SQL Gateway](#)” in *Using Caché SQL* describes how to use the Caché SQL Gateway, which enables you to treat external tables as if they were native Caché tables.
- *Using Caché with ODBC* describes how to use Caché ODBC, which enables you to access Caché tables via ODBC from external applications.
- *Using Caché with JDBC* describes how to use the Caché JDBC driver, which enables you to access Caché tables via JDBC from external applications.

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

1

Overview

Caché TSQL is an implementation of Transact-SQL which supports many of the features of both the Microsoft and Sybase implementations. Transact-SQL is used with Microsoft SQL Server (MSSQL) and Sybase Adaptive Server.

Caché TSQL also contains a few proprietary extensions not found in either of these implementations. These are described in the [Commands](#) chapter.

This document will help you to quickly migrate schemas and stored procedures from Microsoft or Sybase databases and it will provide you with an understanding of the TSQL (Transact-SQL) implementation in Caché.

1.1 Getting Started

To migrate existing TSQL applications to Caché TSQL, you need to perform three operations: configure Caché for TSQL, migrate the TSQL source code, and migrate the data.

1.1.1 Configuring TSQL

To configure your system for TSQL:

- Go into the Caché Management Portal. Select **System Administration, Configuration, SQL and Object Settings**, then select **TSQL Compatibility Settings**. Here you can specify the dialect (Sybase or MSSQL), and turn on or off the **ANSI_NULLS**, **CASEINSCOMPARE**, and **QUOTED_IDENTIFIER** settings. The default for all three is “off”. These values are used to set the `^%SYS(“tsql”,”SET”,...)` global array values.
- From the Management Portal, select **System Administration, Configuration, SQL and Object Settings**, then **General SQL Settings**. From here, you can set the **Default SQL Schema Name**. This is the [system-wide default schema name](#) (which maps to a package) for all unqualified DDL entities.
- From the Management Portal, select **System Administration, Configuration, SQL and Object Settings**, then **User-defined DDL Mappings**. You can use this option to map any needed user-defined data types.

1.1.2 Migrating Source Code

The initial application migration is simple:

1. *Migrate the DDL*: Import table and view definitions using either the `%SYSTEM.SQL.DDLImport()` method (for single files) or the `%SYSTEM.SQL.DDLImportDir()` method (for multiple files in a directory). Set the `DDLMode` parameter to either `"MSSQLServer"` or `"Sybase"`. For further details, see the *InterSystems Class Reference*.

Alternatively, you can invoke `$$SYSTEM.SQL.TSQL()`, `$$SYSTEM.SQL.Sybase()` or `$$SYSTEM.SQL.MSSQLServer()` method to import the schema. For further details, see the *InterSystems Class Reference*.

If the TSQL source contains CREATE PROC statements, then a class method containing the CREATE PROC source is created. Caché places this class method in either an existing class or in a new class whose name is based on the schema and procedure name. If the procedure already exists, then the existing version is replaced by the new version. If a class matching the class name generated from the schema and procedure already exists, then this class name is used — if it was previously generated by the TSQL utility. If not, then a unique class name is generated, based on the schema and procedure name. The resulting class is compiled once the procedure has been successfully created. If logging is requested then the source statements are logged along with the name of the containing class, class method, and the formal arguments generated. Any errors encountered by the process are also reported in the log. If an error is detected during CREATE PROC processing, Caché deletes any new class that was generated for that procedure.

2. *Inspect the log file for errors:* Search by Error #. A summary count of errors and successful imports will appear at the end of the log. In most cases, errors can be worked around or addressed by using information found in this document.
3. *Compile:* When you import DDL, table and view definition compilation is automatically performed. To compile other TSQL source code, it is best to use the command as follows:

```
DO $$SYSTEM.OBJ.CompileAll("-l")
```

The lowercase “L” qualifier flag specifies that locking *is not* applied for the duration of the compile. For a full list of flag qualifiers, call `DO $$SYSTEM.OBJ.ShowFlags()`.

1.1.3 Migrating the Data

In the Management Portal select **System Explorer**, **SQL**, then select the **Data Migration Wizard**.

1.2 TSQL Language Implementation

TSQL procedures are converted to Caché methods or queries with a Language type equal to TSQL. Use the following command:

```
DO ##class(%TSQL.Manager).load("sybase", <filename>, <logname>)
```

When compiling TSQL methods, ObjectScript code is generated. There is no system-level support for native TSQL. It is best to maintain the methods in TSQL to retain the familiar look of the original stored procedures.

- *Using TSQL in Studio*

You can write and maintain TSQL stored procedures (SPs) in Studio. A TSQL SP can be either a class method or a query. A class method takes parameters and returns a single scalar result, a query takes parameters and returns rows. If you put plain SELECT statements into a class method they will be executed but you won't be able to get the rows.

- *Writing a TSQL class method*

Create a class method stored procedure and enter the language as tsql. You can use the following template as a starting point:

```
ClassMethod Example() As %Integer
    [ Language = tsql, ReturnResultSets, SqlName=name, SqlProc ]
{
}
```

- *Using the TSQL Shell*

A TSQL interpreter shell is useful for debugging and experimentation. See “[Using the TSQL Shell](#)” for more information.

- *Using the Caché SQL Shell*

The Caché SQL Shell can be used to execute lines of TSQL code by setting the DIALECT parameter to Sybase or MSSQL. You can execute a TSQL script file from the Caché SQL Shell by using the Shell’s RUN command, See “[Using the SQL Shell Interface](#)” in the *Using Caché SQL* manual.

- *Using Dynamic SQL*

Caché Dynamic SQL can be used to execute TSQL code queries and a limited subset of other DML and DDL statements. In Dynamic SQL you set the %Dialect property to Sybase or MSSQL. See “[Using Dynamic SQL](#)” in the *Using Caché SQL* manual.

- *Using Triggers*

You can write and maintain triggers, which are sets of instructions that appear in TSQL code and that are executed in response to certain SQL events. See “[Using Triggers](#)” in the *Using Caché SQL* manual.

- *TSQL Language Reference*

Microsoft has good TSQL reference material at:

<http://msdn.microsoft.com/en-us/library/bb545450.aspx>

2

Caché TSQL Constructs

2.1 Commonly Used Constructs

2.1.1 Table References

Caché TSQL supports table references with the Caché SQL format:

```
schema.table
```

The only mandatory table reference component is `table`.

Other forms of Transact-SQL may use table references with up to four components, separated by dots. Here is how a Transact-SQL table reference is processed:

- The `server.` prefix, if present, is ignored.
- The `database.` prefix, if present, is removed. Caché only supports one database name: 'master'.
- The `user.` prefix, if present, is mapped to the *schema* name.

For the purposes of name translation, a field name has the field suffix removed while translation is performed and then replaced afterwards.

2.1.2 Temporary Tables

Caché TSQL supports `#tablename` temporary tables. A `#tablename` temporary table is visible to the current procedure of the current process. It is also visible to any procedure called from the current procedure. `#tablename` syntax is only supported in TSQL procedures (class methods projected as procedures with language `tsql`).

A temporary table is defined by using **CREATE TABLE** with a table name starting with "#". The temporary table is created at runtime. A `#tablename` table definition goes out of scope when you exit the procedure. All temporary table definitions go out of scope when the connection is dropped. You can also explicitly delete a temporary table using **DROP TABLE**.

However, if a temporary table is referenced by an active result set, the temporary table may become invisible to the process, but the data and definition are retained until the result set goes out of scope.

A `#tablename` temporary table is visible both to the creating procedure and to any procedures called from that procedure. Temporary tables are visible to nested procedure calls. It is not necessary to declare the temporary table in the called procedure. If the called procedure also creates a temporary table with the same name, Caché uses the most recently created

table definition. Because a temporary table is defined using a Caché local variable, the creation, modification, and deletion of these tables are not journaled transaction events; rolling back the transaction has no effect on these operations.

2.1.3 Transactions

Code generated for **BEGIN TRAN**, **COMMIT** and **ROLLBACK** uses explicit transaction mode, but following a transaction TSQL always restores the mode which was active before the **BEGIN TRAN** statement. TSQL restores this mode when the procedure is exited from, or when a **COMMIT** or **ROLLBACK** is issued, whichever comes first.

2.1.4 Cursor Name Management

You can declare the same cursor more than once, so long as only one version of the cursor is open at runtime. If the same cursor is declared more than once in a stored procedure, all but the first declaration are associated with renamed cursors. **OPEN**, **FETCH**, **CLOSE**, and **DEALLOCATE** statements are assumed to refer to the most recent **DECLARE** for the given cursor. Note that the lexical position of a statement within a stored procedure is all that is used to match up a cursor name with its **DECLARE** — no account is taken of runtime paths through the code.

Cursors inside queries are named using an extension of the scheme used in Caché SQL queries. For example:

```
DECLARE C CURSOR FOR SELECT A FROM B
--
OPEN C
FETCH C
CLOSE C
DEALLOCATE C
--
DECLARE C CURSOR FOR SELECT D FROM E
--
OPEN C
FETCH C
CLOSE C
DEALLOCATE C
```

Would be effectively translated to:

```
DECLARE C CURSOR FOR SELECT A FROM B
--
OPEN C
FETCH C
CLOSE C
DEALLOCATE C
--
DECLARE Cv2 CURSOR FOR SELECT D FROM E
--
OPEN Cv2
FETCH Cv2
CLOSE Cv2
DEALLOCATE Cv2
```

2.1.5 System Stored Procedures

sp_addtype

Supported, with the following limitations:

- No unquoted names. For example, you can't use `EXEC sp_addtype ssn, ...` — you must use `EXEC sp_addtype 'ssn', ...`
- No spaces in physical type.
- Only the physical type is used for validation.

2.1.6 System Tables

System tables exist per Caché namespace.

Systypes

Partially supported.

2.1.7 SYSOBJECTS References

Commonly, an application will have setup procedures that create tables, views, and the metadata for the application environment. Such procedures will have expressions like:

```
IF EXISTS (SELECT * FROM SYSOBJECTS
WHERE ID = OBJECT_ID('People'))
```

This determines if a table exists, in this example. It's usually followed by a DROP and CREATE statement to reestablish the table metadata.

TSQL procedures and triggers can reference the SYSOBJECTS system table. Caché TSQL supports the following columns in the SYSOBJECTS table (%TSQL.sys.objects class properties):

<i>name</i>	Object name.
<i>id</i>	Object Id.
<i>type</i>	Object type: can be one of the following values: K=PRIMARY KEY or UNIQUE constraint; P=stored procedure; RI=FOREIGN KEY constraint; S=system table; TR=trigger; U=user table; V=view.
<i>deltrig</i>	Object ID of a delete trigger if the entry is a table. Table ID of a table if the entry is a trigger.
<i>instrig</i>	Object ID of a table's insert trigger if the entry is a table.
<i>updtrig</i>	Object ID of a table's update trigger if the entry is a table.
<i>parent_obj</i>	Object identification number of parent object. For example, the table ID if a trigger or constraint.
<i>schema</i>	Name of the schema in which the object resides.
<i>parent_obj_name</i>	Object name of parent_obj. If <i>parent_obj</i> =0, <i>parent_obj_name</i> is the same as <i>name</i> .

The SYSOBJECTS table is read-only. The SYSOBJECTS table may be referenced from outside a TSQL procedure or trigger by the name %TSQL_sys.objects. SYSOBJECTS is not supported for tables mapped across namespaces.

Note: Caché provides the %Dictionary package of class objects that can perform the same operations as SYSOBJECTS references. For further details, refer to the %Dictionary package in the *InterSystems Class Reference*.

3

Caché TSQL Language Elements

This chapter describe the following TSQL language elements:

- [Literals, Reserved Words, and Comments](#)
- [Identifiers](#)
- [Data Types](#)
- [Arithmetic, Comparison, String, Logical, and Bitwise Operators](#)

3.1 Literals

3.1.1 String Literals

A string literal must be delimited by quote characters. The preferred delimiter characters are single quote characters. You can also use double quote characters as string delimiters if you specify `SET DELIMITED_IDENTIFIER OFF`. Otherwise, double quote characters are parsed as delimiting an identifier.

If you delimit a string literal with single quote characters, you can include literal double quote characters within the string. To include a literal single quote character within the string, double it by typing two single quotes.

A string containing literal single quotes, such as `'this is an ''embedded'' string'`, is compiled by Caché to single quotes within double quotes: `"this is an 'embedded' string"`.

3.1.2 Empty Strings

When migrating Transact-SQL code to Caché TSQL, it may be necessary to redefine the empty string. You can do this by setting the following Caché system global:

```
^%SYS("sql", "sys", "namespace", nspace, "empty string")
```

All of these specified values are keyword literals, except `nspace`, which is a namespace name specified as a quoted string.

CAUTION: Changing the empty string definition should be done with extreme caution. It can result in data containing different representations for an empty string. It can also cause existing programs to fail when executed in this namespace. After defining the empty string, you must purge all cached queries and recompile all classes and routines for that namespace that use the former empty string definition.

The following ObjectScript example changes the empty string definition for the SAMPLES namespace. It first sets the empty string value to a single blank space. It then sets the empty string value to the non-printing character represented by the ASCII code 0. (This example then immediately resets the empty string value to the Caché default):

```
SET ^%SYS("sql","sys","namespace","SAMPLES","empty string")=" "
WRITE !,"Empty string set to:"
ZZDUMP ^%SYS("sql","sys","namespace","SAMPLES","empty string")
SET ^%SYS("sql","sys","namespace","SAMPLES","empty string")=$CHAR(0)
WRITE !,"Empty string set to:"
ZZDUMP ^%SYS("sql","sys","namespace","SAMPLES","empty string")
SET ^%SYS("sql","sys","namespace","SAMPLES","empty string")=""
WRITE !,"Empty string reset to:"
ZZDUMP ^%SYS("sql","sys","namespace","SAMPLES","empty string")
WRITE !,!, "End of sample program"
```

3.1.3 NULL

In TSQL a NULL supplied to a boolean operation returns as FALSE, as shown in the following example:

```
DECLARE @var BINARY(1)
SELECT @var=NULL
IF @var PRINT "true" ELSE PRINT "false"
```

In Sybase dialect, NULL is equal to NULL. A NULL=NULL comparison returns TRUE, and a NULL != NULL comparison returns FALSE.

In MSSQL dialect, a comparison of NULL with any value returns FALSE. Thus NULL=NULL and NULL != NULL comparisons both return FALSE.

```
DECLARE @var BINARY(1)
SELECT @var=NULL
IF @var=NULL PRINT "true" ELSE PRINT "false"
```

In Sybase dialect, NULL is not equal to any value. Therefore, Not Equals (!=) comparison involving NULL and any boolean, numeric, or string value (including the empty string ("")) returns TRUE. All Equals (=), Greater Than (>) or Less Than (<) comparisons return FALSE.

In MSSQL dialect, NULL cannot be compared to a value. Thus all Equals (=), Not Equals (!=), Greater Than (>) or Less Than (<) comparisons return FALSE.

In a TSQL string concatenation operation, NULL is equivalent to an empty string. In a TSQL arithmetic operation, NULL is equivalent to 0.

3.1.4 Hexadecimal

Caché TSQL automatically converts hexadecimal numeric literals in TSQL source code to the corresponding decimal (base-10) numeric literals.

3.1.5 Reserved Words

Caché TSQL cannot use as identifiers the SQL Server reserved words. Caché TSQL can use Caché SQL reserved words (that are not also SQL Server reserved words) if the **QUOTED_IDENTIFIER** SQL configuration setting is set to Yes.

3.1.6 Comments, Blank Lines, and Semicolons

Caché TSQL supports both single-line and multi-line comments.

- A single line comment continues to the rest of the line. When used in the TSQL shell, a comment does not encompass the end-of-line qualifier, such as /x or /c. Caché TSQL supports both — and // as single-line comment delimiters.
- A multi-line comment begins with /* and ends with */. A comment can include nested /* ... */ comments.

```
PRINT 'these are comments'
-- this is a single-line comment
// this is a single-line comment
/* This is a multi-line comment
The command
PRINT 'do not print'
is part of the comment and is not executed */
```

3.1.6.1 TSQL-only Statements

Caché TSQL provides the means to include executable statements within Caché TSQL code which are parsed as nonexecutable comments in Transact-SQL. A statement prefixed with two hyphens and a vertical bar is parsed by Caché as an executable statement. Sybase Adaptive Server and Microsoft SQL Server consider this to be a Transact-SQL comment.

```
PRINT 'any context'
-- PRINT 'commented out'
--| PRINT 'InterSystems only'
```

3.1.6.2 Semicolons

You can specify a blank line by using either two hyphens or a semicolon.

A semicolon either before or after a TSQL statement is ignored. They are supported for compatibility with Transact-SQL code, such as stored procedures, that ends statements with a semicolon.

```
PRINT 'no semicolon'
--
PRINT 'trailing semicolon';
;
;PRINT 'leading semicolon'
```

3.2 Identifiers

An identifier is a name for a TSQL object, such as a table, column, view, key, index, trigger, or stored procedure. Naming conventions for identifiers are as follows:

- The first character of an identifier must be a letter, an underscore (_) or a percent (%) character.
- Subsequent characters of an identifier may be letters, numbers, underscores (_), dollar signs (\$), or pound signs (#).
- Identifiers can be of any length, but must be unique within their first 30 characters.
- Identifiers are not case-sensitive.
- An identifier cannot be an SQL reserved word.
- A pound sign (#) prefix to an identifier specifies that it is the name of a temporary table.
- An at sign (@) prefix to an identifier specifies that it is the name of a variable.

Some identifiers are qualified with a schema name. For example, schema.tablename or schema.storedprocedure. If the schema name is omitted, the identifier is unqualified. TSQL resolves unqualified identifiers by using the schemaPath property, which provides a search path of schemas to check for the specified table name or stored procedure name.

3.2.1 Delimited and Quoted Identifiers

A delimited identifier is not restricted by the naming conventions of ordinary identifiers. For example, a delimited identifier can be the same word as an SQL reserved word; a delimited identifier can contain space characters.

By default, both square brackets and double quotation marks can be used to delimit an identifier. These delimiters are interchangeable; you can define a delimited identifier by enclosing it with square brackets, and invoke the same delimited identifier by specifying it enclosed with double quotation marks.

You can specify a quoted identifier if the **QUOTED_IDENTIFIER** SQL configuration setting is set to Yes. You specify a quoted identifier by enclosing it in double quotation marks. When **QUOTED_IDENTIFIER** is on, double quotes are parsed as delimiting an identifier. When **QUOTED_IDENTIFIER** is off, double quotes are parsed as alternative delimiters for string literals. The preferable delimiters for string literals are single quotes. A quoted identifier can contain any characters, including blank spaces.

3.3 Data Types

The following data types are supported for local variables and table columns. These data types are supported in that they are parsed as valid data types; however, no range or value validation is performed.

BINARY(*n*) and **VARBINARY**(*n*). The (*n*) size specification is mandatory.

BIT

BOOLEAN

CHAR and **VARCHAR**

CHAR(*n*), **NCHAR**(*n*), **VARCHAR**(*n*), and **NVARCHAR**(*n*)

VARCHAR(MAX), and **NVARCHAR**(MAX). By default, these map to %Stream.GlobalCharacter.

DATETIME and **SMALLDATETIME**

DECIMAL, **DECIMAL**(*p*), and **DECIMAL**(*p*,*s*). Where *p* and *s* are integers specifying precision (total digits) and scale (decimal digits).

DOUBLE and **DOUBLE PRECISION**

FLOAT and **FLOAT**(*n*)

INT, **BIGINT**, **SMALLINT**, and **TINYINT**

MONEY and **SMALLMONEY**

NATIONAL

NUMERIC, **NUMERIC**(*p*), and **NUMERIC**(*p*,*s*). Where *p* and *s* are integers specifying precision (total digits) and scale (decimal digits).

REAL

TIMESTAMP

Note: The Microsoft SQL Server **TIMESTAMP** data type is *not* used for date or time information. It is an integer counter of the number of times a record is inserted or updated in a table. It should not be confused with the Caché SQL and ODBC **TIMESTAMP** data type, which represents a date and time in YYYY-MM-DD HH:MM:SS.nnnnnnnn format. In TSQL, use **DATETIME** and **SMALLDATETIME** for date and time values.

ROWVERSION

SQL_VARIANT

The following SQL Server data types are supported in a specific context:

CURSOR

NTEXT, TEXT By default, these map to %Stream.GlobalCharacter.

IMAGE

TABLE

The following are not implemented:

- UNIQUEIDENTIFIER stored as a 16-byte binary string. Instead use VARCHAR(32) as the data type for a globally unique ID.
- SQL92 and TSQL options
- UPDATE OF

3.4 Operators

3.4.1 Arithmetic and Equality Operators

Caché TSQL supports + (addition), – (subtraction), * multiplication, / division, and % modulo arithmetic operators.

Caché TSQL supports the following equality and comparison operators:

- = (equal to)
- <> (not equal to) and != (not equal to)
- < (less than), !< (not less than), <= (less than or equal to)
- > (greater than), !> (not greater than), >= (greater than or equal to)

When performing equality comparisons (= or <>) between date values with different data types, all date and time values are compared using the TIMESTAMP data type. Thus two dates in different formats can be meaningfully compared. A date value declared as a STRING data type can be compared to a date value declared as a DATETIME data type.

3.4.2 Concatenation Operator

Caché TSQL supports the + (plus sign) as both a concatenation operator and the addition operator. The plus sign functions as a concatenation operator with strings. You can concatenate several strings together using this operator. If all items are strings, TSQL performs concatenation; however, if one of the items is a number, TSQL performs addition, treating non-numeric strings as 0.

'world'+ 'wide'+ 'web' concatenates to 'worldwideweb'

'world'+ '33'+ 'web' concatenates to 'world33web'

'world'+33+'web' performs addition (0+33+0=33)

In a TSQL string concatenation operation, NULL is equivalent to an empty string. In a TSQL arithmetic operation, NULL is equivalent to 0. Note that because the plus sign (+) is used for both concatenation and addition, the data type declaration of the NULL variable is critical. The following examples all return “bigdeal”:

```
DECLARE @var1 BINARY(1)
DECLARE @var2 VARCHAR(10)
SELECT @var1=NULL,@var2=NULL
PRINT "big"+NULL+"deal"
PRINT "big"+@var1+"deal"
PRINT "big"+@var2+"deal"
```

The following example returns 0; it treats the + as an arithmetic operator and interprets the argument as $0 + 0 + 0 = 0$:

```
DECLARE @var1 INT
SELECT @var1=NULL
PRINT "big"+@var1+"deal"
```

Caché TSQL also supports || as a concatenation operator.

3.4.3 Comparison Operators

3.4.3.1 BETWEEN

Caché TSQL supports the **BETWEEN** range check operator of the form: `BETWEEN num1 AND num2`. **BETWEEN** is inclusive of the specified range limits.

3.4.3.2 IS NULL

Caché TSQL supports the IS NULL match operator. A variable is NULL if it has been declared but not assigned a value, or if it has been explicitly specified as NULL. The empty string is not NULL.

3.4.3.3 LIKE

Caché TSQL supports the LIKE pattern match operator. LIKE performs not case-sensitive matching of letters. Caché TSQL also supports NOT LIKE.

3.4.4 NOT Logical Operator

The NOT logical operator inverts the truth value of the statement that follows it. For example, `IF NOT EXISTS(. . .)`. NOT is not case-sensitive.

3.4.5 Bitwise Logical Operators

Caché TSQL supports the AND (&), OR (|), XOR (^), and NOT (~) bitwise operators for the integer data type. The decimal integers are converted to binary, the logical operation is performed, and the resulting binary is converted to a decimal integer value. The NOT (~) operator is a unary operator that inverts bits.

4

TSQL Commands

This chapter lists the supported TSQL commands in the following groups:

- [Data Definition Language \(DDL\) statements](#)
- [Data Management Language \(DML\) statements](#)
- [Query statements](#)
- [Flow of control statements](#)
- [Assignment statements](#)
- [Transaction statements](#)
- [Procedure statements](#)
- [Other statements](#)
- [Caché extensions](#)

4.1 Data Definition Language (DDL) Statements

The following DDL statements are supported.

4.1.1 CREATE TABLE

The **CREATE TABLE** statement defines a table, its fields, and their data types and constraints.

```
CREATE TABLE [schema. | #]tablename (fieldname datatype constraint [,...])
```

A **CREATE TABLE** can create a temporary table by prefixing a # character to the table name. A temporary table can only be defined from a stored procedure; you cannot define a temporary table from Dynamic SQL outside of a stored procedure. To create a fully-qualified temporary table name, use quotes around each name element such as the following:

```
"SQLUser" . "#mytemp".
```

A valid table name must begin with a letter, an underscore character (`_`), or a # character (for a local temporary table). Subsequent characters of a table name may be letters, numbers, or the #, \$, or `_` characters. Table names are not case-sensitive.

A field name must be a valid [TSQL identifier](#). A field name can be delimited using square brackets. This is especially useful when defining a field that has the same name as a reserved word. The following example defines two fields named Check and Result:

```
CREATE TABLE mytest ([Check] VARCHAR(50),[Result] VARCHAR(5))
```

The optional **CONSTRAINT** keyword can be used to specify a user-defined constraint name for a column constraint or a table constraint. You can specify multiple **CONSTRAINT name type** statements for a column.

Caché SQL does not retain constraint names. Therefore these names cannot be used by a subsequent **ALTER TABLE** statement.

The table column constraints **DEFAULT**, **IDENTITY**, **NULL**, **NOT NULL**, **PRIMARY KEY**, **[FOREIGN KEY] REFERENCES** (the keywords **FOREIGN KEY** are optional), **UNIQUE**, **CLUSTERED**, and **NONCLUSTERED** are supported. The table constraint **FOREIGN KEY REFERENCES** is supported.

The column definition **DEFAULT** values can include the following TSQL functions: **CURRENT_TIMESTAMP**, **CURRENT_USER**, **GETDATE**, **HOST_NAME**, **ISNULL**, **NULLIF**, and **USER**.

The column definition **IDENTITY** constraint is supported and assigned a system-generated sequential integer. The **IDENTITY** arguments *seed* and *increment* are parsed, but ignored.

The table constraint clauses **WITH**, **ON**, and **TEXTIMAGE ON** are parsed for compatibility, but are ignored. The <index options> clause for the **UNIQUE** or **PRIMARY KEY** constraint is parsed for compatibility, but is ignored.

The following SQL Server parenthesized **WITH** options in a table constraint are parsed but ignored: **ALLOW_PAGE_LOCKS**, **ALLOW_ROW_LOCKS**, **DATA_COMPRESSION**, **FILLFACTOR**, **IGNORE_DUP_KEY**, **PAD_INDEX**, and **STATISTICS_NORECOMPUTE**.

The column constraints **CLUSTERED** and **NONCLUSTERED** are parsed for compatibility, but are ignored.

The **CHECK** column constraint is not supported. If a **CHECK** constraint is encountered while compiling TSQL source Caché generates an error message indicating that **CHECK** constraints are not supported. This error is logged in the compile log (if active), and the source is placed in the unsupported log (if active).

If the table already exists, an **SQLCODE -201** error is issued.

The following Dynamic SQL example creates a temporary table named **#mytest** with four fields, populates it with data, then displays the results. The **LastName** field has multiple constraints. The **FirstName** field takes a default. The **DateStamp** field takes a system-defined default:

```
SET sql=9
SET sql(1)="CREATE TABLE #mytest (MyId INT PRIMARY KEY,"
SET sql(2)="LastName VARCHAR(20) CONSTRAINT unq_lname UNIQUE "
SET sql(3)=" CONSTRAINT nonull_lname NOT NULL,"
SET sql(4)="FirstName VARCHAR(20) DEFAULT '***TBD***',"
SET sql(5)="DateStamp DATETIME DEFAULT CURRENT_TIMESTAMP)"
SET sql(6)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1224,'Smith','John') "
SET sql(7)="INSERT INTO #mytest(MyId,LastName) VALUES (1225,'Jones') "
SET sql(8)="SELECT MyId,FirstName,LastName,DateStamp FROM #mytest"
SET sql(9)="DROP TABLE #mytest"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
WRITE status,!
SET result=statement.%Execute()
DO result.%Display()
```

4.1.2 ALTER TABLE

The **ALTER TABLE** statement allows you to modify the definition of a table, its fields, and their data types and constraints.

The following syntactical forms are supported:

```
ALTER TABLE <tablename> ADD <colname> <datatype> [DEFAULT <value>] [{UNIQUE | NOT
NULL} | CONSTRAINT <constraintname> {UNIQUE | NOT NULL} ]
ALTER TABLE <tablename> ALTER COLUMN <colname> <newdatatype>
ALTER TABLE <tablename> REPLACE <colname> DEFAULT { <constant> | USER | NULL }
ALTER TABLE <tablename> DROP COLUMN <colname>
ALTER TABLE <tablename> ADD <tableconstraint> FOR <colname>
ALTER TABLE <tablename> DROP <tableconstraint>
ALTER TABLE <tablename> DROP FOREIGN KEY <role>
ALTER TABLE <tablename> ADD CONSTRAINT <constraint> DEFAULT <defaultvalue> FOR
<colname>
ALTER TABLE <tablename> ADD CONSTRAINT <constraint> FOREIGN KEY
ALTER TABLE <tablename> DROP CONSTRAINT <constraint>
ALTER TABLE <tablename> DISABLE
```

ALTER TABLE...ADD can only specify the NOT NULL constraint if it also specifies a DEFAULT value. The full supported syntax for **ALTER TABLE...ADD** is as follows:

```
ALTER TABLE <tablename>
[ WITH CHECK | WITH NOCHECK ]
ADD <colname> <datatype> [DEFAULT <value>] [{UNIQUE | NOT NULL} | CONSTRAINT
<constraintname> {UNIQUE | NOT NULL} ]
[ FOREIGN KEY (col1[,col2[,...]])
REFERENCES table(col1[,col2[,...]]) ]
```

WITH CHECK | WITH NOCHECK is parsed by Caché, but is ignored. In Transact-SQL, WITH CHECK | WITH NOCHECK provides an execution time check of existing data for a new or newly enabled constraint. Caché TSQL does not specifically support that, although Caché SQL will check existing data against a new constraint.

ALTER TABLE...ADD CONSTRAINT...DEFAULT syntax does not create a column constraint. Instead, it performs the equivalent of an **ALTER TABLE...ALTER COLUMN...DEFAULT** statement. This means that Caché establishes the specified column default as the field property's initial expression. Because no column constraint is defined, this "constraint" cannot be subsequently dropped or changed.

CHECK | NOCHECK CONSTRAINT is not supported by Caché TSQL. Specifying this CHECK or NOCHECK keyword generates an error message.

ALTER TABLE...DROP COLUMN: the keyword DELETE is a synonym for the keyword DROP.

4.1.3 DROP TABLE

Deletes a table definition.

```
DROP TABLE [IF EXISTS] tablename [,tablename2 [,...]]
```

Deletes a table definition. You can delete a single table or a comma-separated list of tables. You can delete both regular tables and temporary tables. (Temporary table names begin with a '#' character.) **DROP TABLE** ignores a nonexistent temporary table name and completes without error.

The optional IF EXISTS clause suppresses errors if you specify a non-existent *tablename* that is not a temporary table.

4.1.4 CREATE INDEX

Creates an index for a specified table or view.

```
CREATE INDEX indexname ON tablename(fieldname)
```

You can create an index on the IDKEY (which is treated as a clustered index), on an IDENTITY column (which create an index on the %%ID column), on the Primary Key, or on other columns.

The following Transact-SQL features are parsed, but ignored:

- The CLUSTERED/NONCLUSTERED keywords. Other than the IDKEY, which is implicitly treated as a clustered index, Caché TSQL does not support clustered indexes.
- The ASC/DESC keywords.
- The INCLUDE clause.
- All WITH clause index options. The comma-separated list of WITH clause options can optionally be enclosed in parentheses, and can include flag options set =ON or =OFF.
- The ON filegroup or IN dbspace-name clause.

The following Transact-SQL features are not currently supported:

- The DATA clause.
- The VIRTUAL keyword.
- Using a function name as an alternative to a column name.

4.1.5 DROP INDEX

Deletes an index definition. You can delete a single index or a comma-separated list of indices, using either of the following syntax forms:

```
DROP INDEX table.index [,table.index]
DROP INDEX index ON table [WITH (...)] [,index ON table [WITH (...)] ]
```

Where *table* is the name of the table containing the indexed field, and *index* is the name of the index.

The WITH (...) clause, with any value within the parentheses, is accepted by syntax checking for compatibility, but is not validated and performs no operation.

4.1.6 CREATE TRIGGER

Creates a statement-level trigger.

```
CREATE TRIGGER trigger_name {BEFORE | AFTER}
{INSERT | DELETE | UPDATE [OF col1[,col2[,...]]]}
[ORDER integer]
ON table-name
[ REFERENCING OLD [ROW] [AS] name |
NEW [ROW] [AS] name |
OLD TABLE [AS] identifier |
NEW TABLE [AS] identifier ][,...]
[FOR EACH {ROW | STATEMENT}]
[WHEN (condition)]
[LANGUAGE {TSQL | SQL | OBJECTSCRIPT}]
triggered statement
[WITH EXECUTE]
```

Caché TSQL does not support row-level triggers.

Caché TSQL supports BEFORE triggers. This is a Caché-specific extension to TSQL which is not provided in Transact-SQL software from other vendors.

4.1.7 DROP TRIGGER

Deletes a trigger definition. You can delete a single trigger or a comma-separated list of triggers.

4.1.8 CREATE VIEW

```
CREATE VIEW [owner.]view_name [(colname1 [,colname2 [,...]])]
  [WITH ENCRYPTION | SCHEMABINDING | VIEW_METADATA]
  AS select_statement
  [WITH CHECK OPTION]
```

A *view_name* must be a unique [TSQL identifier](#). If the view already exists, an SQLCODE -201 error is issued. A *view_name* can be a [delimited identifier](#). For example, `CREATE VIEW "Name/Age View"`

There are two ways to specify the names of the view columns:

- You can use the optional *colname* comma-separated list to specify view column names for the corresponding table columns returned by the SELECT statement.

```
CREATE VIEW NameAgeV (FullName,Years)
AS SELECT Name,Age FROM Sample.Person
```

- You can specify column aliases in the SELECT statement, and these aliases are used as the view column names.

```
CREATE VIEW NameAgeV
AS SELECT Name AS FullName,Age AS Years FROM Sample.Person
```

If neither is specified, the table column names are used as the view column names.

The WITH ENCRYPTION, SCHEMABINDING, and VIEW_METADATA keywords are ignored.

The *select_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 pair an ORDER BY clause with 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.

The *select_statement* can contain a UNION or UNION ALL.

The optional WITH CHECK OPTION clause prevents an update through the view that makes the record inaccessible to that view. It does this by checking the WITH clause in the SELECT statement. WITH CHECK OPTION binds to Caché SQL using the default of CASCADE.

4.1.9 DROP VIEW

Deletes a view definition. You can delete a single view, or a comma-separated list of views. **DROP VIEW** is not an all-or-nothing operation. It deletes existing views from the list of views, and ignores nonexistent views in the list of views.

4.1.10 CREATE DATABASE

CREATE DATABASE syntax is parsed to provide compatibility with MSSQL. No functionality is provided.

The MSSQL attach a database and create a database snapshot syntax options are not supported.

4.1.11 DROP DATABASE

DROP DATABASE syntax is parsed to provide compatibility with MSSQL. No functionality is provided.

4.2 Data Management Language (DML) Statements

- TSQL can resolve an unqualified table name using a [schema search path](#) for a single DML statement in Dynamic SQL.
- TSQL *cannot* resolve an unqualified table name using a schema search path for multiple DML statements in Dynamic SQL. This includes multiple statements such as an explicit BEGIN TRANSACTION followed by a single DML statement.

4.2.1 DELETE

Deletes rows of data from a table. Both **DELETE** and **DELETE FROM** are supported. Most other options are supported, with the following exceptions:

- FROM table hints
- rowset functions
- OPTION clause
- join hints
- only very simple theta joins are supported (the FROM table clause is transformed into nested subqueries)
- primary table hints

The following table_hints are parsed but ignored: FASTFIRSTROW, HOLDINDEX, INDEX(name), NOLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, REPEATABLEREAD, ROWLOCK, SERIALIZABLE, SHARED, TABLOCK, TABLOCKX, UPDLOCK, XLOCK. Table hints can be optionally preceded by the WITH keyword, and, if WITH is specified, optionally enclosed in parentheses. A list of table hints can be separated by either commas or blank spaces.

DELETE sets the @@ROWCOUNT system variable to the number of rows deleted, and the @@IDENTITY system variable to the IDENTITY value of the last row deleted.

4.2.2 INSERT

Inserts rows of data into a table. The following MSSQL syntactic forms are supported:

```
INSERT column VALUES list_of_values
INSERT column SELECT select_statement
```

The corresponding Sybase list_of_values and SELECT syntactic forms are supported. (Sybase does not use the VALUES keyword.)

Most options are supported, with the following exceptions:

- EXECUTE as a value clause
- rowset functions
- DEFAULT VALUES

The following table_hints are parsed but ignored: FASTFIRSTROW, HOLDINDEX, INDEX(name), NOLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, REPEATABLEREAD, ROWLOCK, SERIALIZABLE, SHARED, TABLOCK, TABLOCKX, UPDLOCK, XLOCK. Table hints can be optionally preceded by the WITH keyword, and, if WITH is specified, optionally enclosed in parentheses. A list of table hints can be separated by either commas or blank spaces.

INSERT sets the @@ROWCOUNT system variable to the number of rows inserted, and the @@IDENTITY system variable to the IDENTITY value of the last row inserted.

4.2.3 UPDATE

Updates values of existing rows of data in a table.

```
UPDATE tablename SET fieldname=value[,fieldname2=value2[,...]]
WHERE [tablename.]fieldname=value
```

Most **UPDATE** options are supported, with the following exceptions:

- rowset
- OPTION
- only very simple theta joins are supported (the FROM table clause is transformed into nested subqueries)
- table hints

UPDATE supports the use of a local variable on the left-hand-side of a SET clause. This local variable can be either instead of a column name or in addition to a column name. The following example shows a SET to a column name, a SET to a local variable, and a SET to both a column name and a local variable:

```
UPDATE table SET x=3,@v=b,@c=Count=Count+1
```

If a local variable is specified on the left-hand-side of a SET, the right-hand-side cannot be DEFAULT.

The following table_hints are parsed but ignored: FASTFIRSTROW, HOLDINDEX, INDEX(name), NOLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, REPEATABLE_READ, ROWLOCK, SERIALIZABLE, SHARED, TABLOCK, TABLOCKX, UPDLOCK, XLOCK. Table hints can be optionally preceded by the WITH keyword, and, if WITH is specified, optionally enclosed in parentheses. A list of table hints can be separated by either commas or blank spaces.

UPDATE sets the @@ROWCOUNT system variable to the number of rows updated, and the @@IDENTITY system variable to the IDENTITY value of the last row updated.

The following Dynamic SQL example shows a simple **UPDATE** operation:

```
SET sql=9
SET sql(1)="CREATE TABLE #mytest (MyId INT PRIMARY KEY,"
SET sql(2)="LastName VARCHAR(20) CONSTRAINT nonull_lname NOT NULL,"
SET sql(3)="FirstName VARCHAR(20) DEFAULT '***TBD***')"
```

```
SET sql(4)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1224,'Smith','John')"
```

```
SET sql(5)="INSERT INTO #mytest(MyId,LastName) VALUES (1225,'Jones')"
```

```
SET sql(6)="INSERT INTO #mytest(MyId,LastName) VALUES (1226,'Brown')"
```

```
SET sql(7)="UPDATE #mytest SET FirstName='Fred' WHERE #mytest.LastName='Jones'"
```

```
SET sql(8)="SELECT FirstName,LastName FROM #mytest ORDER BY LastName"
```

```
SET sql(9)="DROP TABLE #mytest"
```

```
SET statement=##class(%SQL.Statement).%New()
```

```
SET statement.%Dialect="MSSQL"
```

```
SET status=statement.%Prepare(.sql)
```

```
WRITE status,!
```

```
SET result=statement.%Execute()
```

```
DO result.%Display()
```

4.2.4 READTEXT, UDATETEXT, WRITETEXT

READTEXT returns stream data from a column of a table. It requires a valid text pointer value, which can be retrieved using the **TEXTPTR** function.

UPDATETEXT updates stream data from a column of a table. It requires a valid text pointer value, which can be retrieved using the **TEXTPTR** function.

WRITETEXT writes data to a stream column of a table. It requires a valid text pointer value, which can be retrieved using the **TEXTPTR** function.

4.2.5 TRUNCATE TABLE

```
TRUNCATE TABLE tablename
```

Deletes all rows from the specified table. Supported to the extent that it is a synonym for `DELETE FROM table` with no `WHERE` clause. However, **TRUNCATE TABLE** does not reset the `RowId (ID)`, `IDENTITY`, or `SERIAL (%Counter)` row counters. The InterSystems SQL [TRUNCATE TABLE](#) command does reset these counters.

4.3 Query Statements

4.3.1 SELECT

```
SELECT [DISTINCT | ALL]
  [TOP [(]{ int | @var | ? | ALL}{ )}]
  select-item { ,select-item }
  [INTO #temptable]
  [FROM table [[AS] t-alias] [,table2 [[AS] t-alias2]] ]
  [[WITH] [(] tablehint=val [,tablehint=val] [)] ]
  [WHERE condition-expression]
  [GROUP BY scalar-expression]
  [HAVING condition-expression]
  [ORDER BY item-order-list [ASC | DESC] ]
```

The above `SELECT` syntax is supported. The following features are not supported:

- `TOP nn PERCENT` or `TOP WITH TIES`
- `OPTION`
- `WITH CUBE`
- `WITH ROLLUP`
- `GROUP BY ALL`
- `GROUP WITH`
- `COMPUTE` clause
- `FOR BROWSE`

`TOP nn` specifies the number of rows to retrieve. Caché TSQL supports `TOP nn` with a integer, `?`, local variable, or the keyword `ALL`. The `TOP` argument can be enclosed in parentheses `TOP (nn)`. These parentheses are retained, preventing preparer substitution. If `SET ROWCOUNT` specifies fewer rows than `TOP nn`, the `SET ROWCOUNT` value is used. The following Dynamic SQL example shows the use of `TOP` with a local variable:

```
SET sql=3
SET sql(1)="DECLARE @var INT"
SET sql(2)="SET @var=4"
SET sql(3)="SELECT TOP @var Name,Age FROM Sample.Person"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()
```

The *select-item* list can contain the following:

- field names, functions, and expressions
- the **\$IDENTITY** pseudo-column name, which always returns the [RowID](#) value, regardless of the field name assigned to the RowID.
- an asterisk: **SELECT *** is supported. The asterisk means to select all columns in the specified table. You can qualify the asterisk with the table name or table alias: **SELECT mytable.***.
- a subquery
- stream fields. A **SELECT** on a stream field returns the *oref* (object reference) of the opened stream object.

An INTO clause can be used to select values into a table. This table can be a permanent table, or a temporary table, as shown in the following example:

```
SELECT name,age INTO #MyTemp FROM Sample.Person
SELECT name,age FROM #MyTemp WHERE name LIKE '[A-D]%'
```

If the INTO table does not already exist, **SELECT** creates it.

An INTO clause cannot be used when the **SELECT** is a subquery or is part of a UNION.

The FROM clause is not required. A **SELECT** without a FROM clause can be used to assign a value to a local variable, as follows:

```
DECLARE @myvar INT
SELECT @myvar=1234
PRINT @myvar
```

The FROM clause supports table hints with either of the following syntactic forms:

```
FROM tablename (INDEX=indexname)
FROM tablename INDEX (indexname)
```

Table hints can be optionally preceded by the **WITH** keyword, and optionally enclosed in parentheses. A list of table hints can be separated by either commas or blank spaces. The following table hints are parsed but ignored: FASTFIRSTROW, HOLDINDEX, NOLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, REPEAT-ABLEREAD, ROWLOCK, SERIALIZABLE, SHARED, TABLOCK, TABLOCKX, UPDLOCK, XLOCK.

A WHERE clause can use AND, OR, and NOT logic keywords. It can group multiple search conditions using parentheses. The WHERE clause supports the following search conditions:

- Equality comparisons: = (equals), <> (not equals), < (less than), > (greater than), <= (less than or equals), >= (greater than or equals)
- IS NULL and IS NOT NULL comparisons
- BETWEEN comparisons: Age BETWEEN 21 AND 65 (inclusive of 21 and 65); Age NOT BETWEEN 21 AND 65 (exclusive of 21 and 65). 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. It uses the same collation type as the column it is matching against. By default, string data types collate as not case-sensitive.
- IN comparisons: Home_State IN ('MA' , 'RI' , 'CT')
- LIKE and NOT LIKE comparisons, specified as a quoted string. The comparison string can contain wildcards: _ (any single character); % (any string); [abc] (any value in the set specified as a list of items); [a-c] (any value in the set specified as a range of items). Caché TSQL does not support the ^ wildcard. A LIKE comparison can include an ESCAPE clause, such as the following: WHERE CategoryName NOT LIKE 'D_%' ESCAPE '\'
- EXISTS comparison check: used with a subquery to test whether the subquery evaluates to the empty set. For example
SELECT Name FROM Sample.Person WHERE EXISTS (SELECT LastName FROM Sample.Employee

WHERE LastName= 'Smith'). In this example, all Names are returned from Sample.Person if a record with LastName='Smith' exists in Sample.Employee. Otherwise, no records are returned from Sample.Person.

- ANY and ALL comparison check: used with a subquery and an equality comparison operator. The SOME keyword is a synonym for ANY.

WHERE clause and HAVING clause comparisons are not case-sensitive.

A HAVING clause can be specified after a GROUP BY clause. The HAVING clause is like a WHERE clause that can operate on groups, rather than on the full data set. HAVING and WHERE use the same comparisons. This is shown in the following example:

```
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 Dynamic SQL example selects table data into a result set:

```
SET sql=7
SET sql(1)="CREATE TABLE #mytest (MyId INT PRIMARY KEY, "
SET sql(2)="LastName VARCHAR(20), "
SET sql(3)="FirstName VARCHAR(20))"
SET sql(4)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1224,'Smith','John')"
SET sql(5)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1225,'Jones','Wilber')"
SET sql(6)="SELECT FirstName,LastName FROM #mytest"
SET sql(7)="DROP TABLE #mytest"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()
```

The following Dynamic SQL example selects a single column value into a local variable:

```
SET sql=9
SET sql(1)="CREATE TABLE #mytest (MyId INT PRIMARY KEY, "
SET sql(2)="LastName VARCHAR(20), "
SET sql(3)="FirstName VARCHAR(20))"
SET sql(4)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1224,'Smith','John')"
SET sql(5)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1225,'Jones','Wilber')"
SET sql(6)="DECLARE @nam VARCHAR(20)"
SET sql(7)="SELECT @nam=LastName FROM #mytest"
SET sql(8)="PRINT @nam"
SET sql(9)="DROP TABLE #mytest"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
DO statement.%Execute()
```

An ORDER BY clause can specify ascending (ASC) or descending (DESC) order. The default is ascending. Unlike Caché SQL, an ORDER BY may be used in subqueries and in queries that appear in expressions. For example:

```
SET @var = (SELECT TOP 1 name FROM mytable ORDER BY name)
```

4.3.2 JOIN

JOIN (equivalent to INNER JOIN), INNER JOIN, and LEFT JOIN supported. Parentheses can be used to rationalize parsing of multiple joins.

Note: Caché TSQL uses the following symbolic representations for outer joins:

=* Left Outer Join
 *= Right Outer Join

These correspond to Caché SQL usage. They are the *exact opposite* of the SQL Server and Sybase join syntax (where =* is a Right Outer Join). It is strongly recommended that you represent outer joins using ANSI standard keyword syntax, rather than this symbolic syntax.

4.3.3 UNION

A union of two (or more) **SELECT** statements is supported. Caché TSQL supports **UNION** and **UNION ALL**. If you specify **UNION ALL**, only the first **SELECT** can specify an INTO table. This INTO table can be a defined table, or a temporary table generated from the **SELECT** column list.

4.3.4 FETCH Cursor

The **OPEN**, **FETCH**, **CLOSE**, and **DEALLOCATE** commands are mainly supported. The following features are not supported:

- OPEN/FETCH/CLOSE @local
- FETCH followed by any qualifier other than NEXT (the qualifier can be omitted).
- Note that DEALLOCATE is supported, but that, by design, it generates no code.

4.3.5 UPDATE STATISTICS

Optimizes query access for a specified table. Caché passes the specified table name argument to the **\$SYSTEM.SQL.TuneTable()** method for optimization. UPDATE STATISTICS calls **\$SYSTEM.SQL.TuneTable()** with update=1 and display=0. The returned %msg is ignored and KeepClassUpToDate defaults to 'false'. All other UPDATE STATISTICS syntax is parsed for compatibility only and ignored. In a batch or stored procedure, only the first UPDATE STATISTICS statement for a given table generates a call to **\$SYSTEM.SQL.TuneTable()**.

4.4 Flow of Control Statements

4.4.1 IF

The **IF** command is supported with four syntactic forms:

IF...ELSE syntax:

```
IF condition
statement
[ELSE statement]
```

IF...THEN...ELSE single-line syntax:

```
IF condition THEN statement [ELSE statement]
```

ELSEIF...END IF syntax:

```
IF condition THEN
statements
{ELSEIF condition THEN statements}
[ELSE statements]
END IF
```

ELSE IF (SQL Anywhere) syntax:

```
IF condition THEN statement
{ELSE IF condition THEN statement}
[ELSE statement]
```

The first syntactic form is the TSQL standard format. No THEN keyword is used. You may use white space and line breaks freely. To specify more than one *statement* in a clause you must use BEGIN and END keywords to demarcate the block of statements. The ELSE clause is optional. This syntax is shown in the following example:

```
SET sql=4
SET sql(1)="DECLARE @var INT"
SET sql(2)="SET @var=RAND()"
SET sql(3)="IF @var<.5 PRINT 'The Oracle says No'"
SET sql(4)="ELSE PRINT 'The Oracle says Yes' "
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()
```

The second syntactic form is single-line syntax. The THEN keyword is required. A line break restriction requires that IF condition THEN statement all be on the same line, though only the first keyword of the *statement* must be on that line. Otherwise, you may use white space and line breaks freely. To specify more than one *statement* in a clause you must use BEGIN and END keywords to demarcate the block of statements. The ELSE clause is optional. This syntax is shown in the following example:

```
SET sql=3
SET sql(1)="DECLARE @var INT "
SET sql(2)="SET @var=RAND() "
SET sql(3)="IF @var<.5 THEN PRINT 'No' ELSE PRINT 'Yes' "
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()
```

The third syntactic form provides an ELSEIF clause. You can specify zero, one, or more than one ELSEIF clauses, each with its own *condition* test. Within an IF, ELSEIF, or ELSE clause you can specify multiple statements. BEGIN and END keywords are permitted but not required. A line break restriction requires a line break between IF condition THEN and the first *statement*. Otherwise, you may use white space and line breaks freely. The ELSE clause is optional. The END IF keyword clause is required. This syntax is shown in the following example:

```

SET sql=14
SET sql(1)="DECLARE @var INT "
SET sql(2)="SET @var=RAND() "
SET sql(3)="IF @var<.2 THEN "
SET sql(4)="PRINT 'The Oracle' "
SET sql(5)="PRINT 'says No' "
SET sql(6)="ELSEIF @var<.4 THEN "
SET sql(7)="PRINT 'The Oracle' "
SET sql(8)="PRINT 'says Possibly' "
SET sql(9)="ELSEIF @var<.6 THEN "
SET sql(10)="PRINT 'The Oracle' "
SET sql(11)="PRINT 'says Probably' "
SET sql(12)="ELSE PRINT 'The Oracle' "
SET sql(13)="PRINT 'says Yes' "
SET sql(14)="END IF"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()

```

The fourth syntactic form is compatible with SQL Anywhere. It provides an **ELSE IF** clause (note space between keywords). You can specify zero, one, or more than one **ELSE IF** clauses, each with its own *condition* test. To specify more than one *statement* in a clause you must use **BEGIN** and **END** keywords to demarcate the block of statements. You may use white space and line breaks freely. The **ELSE** clause is optional. This syntax is shown in the following example:

```

SET sql=6
SET sql(1)="DECLARE @var INT "
SET sql(2)="SET @var=RAND() "
SET sql(3)="IF @var<.2 THEN PRINT 'The Oracle says No'"
SET sql(4)="ELSE IF @var<.4 THEN PRINT 'The Oracle says Possibly'"
SET sql(5)="ELSE IF @var<.6 THEN PRINT 'The Oracle says Probably'"
SET sql(6)="ELSE PRINT 'The Oracle says Yes'"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()

```

4.4.2 WHILE

The **WHILE** command is supported. It executes code while a condition is true.

```
WHILE condition BEGIN statements END
```

The **BEGIN** and **END** keywords are required if *statements* is more than one command.

4.4.3 CASE

The **CASE** statement is supported. The **ELSE** clause is optional. If no **WHEN** clause is satisfied and no **ELSE** clause is specified, the **CASE** statement sets the expression to **NULL**.

4.4.4 GOTO and Labels

Caché TSQL supports the **GOTO** command and labels. A label must be a valid [TSQL identifier](#) followed by a colon (:). A **GOTO** reference to a label does not include the colon.

4.4.5 WAITFOR

Caché TSQL supports both forms of the **WAITFOR** command:

```
WAITFOR DELAY timeperiod
WAITFOR TIME clocktime
```

timeperiod is the amount of time to wait before resuming execution, expressed as 'hh:mm[:ss[.fff]]. Thus `WAITFOR DELAY '00:00:03'` provides a time delay of 3 seconds. `WAITFOR DELAY '00:03'` provides a time delay of 3 minutes.

clocktime is the time at which to resume execution, expressed as 'hh:mm[:ss[.fff]]', using a 24-hour clock. Thus `WAITFOR TIME '00:00:03'` resumes execution at 3 seconds after midnight.

4.5 Assignment Statements

4.5.1 DECLARE

Used to declare the data type for a local variable.

```
DECLARE @var [AS] datatype [ = initval]
```

Only the form which declares local variables is supported; cursor variables are not supported.

@var can be any local variable name; local variable names are not case-sensitive. *datatype* can be any valid data type, such as `CHAR(12)` or `INT`. The optional *initval* argument allows you to specify an initial value for the local variable.

For further details on data types, refer to the [TSQL Constructs](#) chapter of this document.

4.5.2 SET

Used to assign a value to a local variable:

```
DECLARE @var CHAR(20)
SET @var='hello world'
```

Used to set a system setting:

These settings have immediate effect at parse time, whether inside a stored procedure or not. The change persists until another **SET** command alters it – even if the **SET** is made inside a stored procedure, and accessed outside the SP or in another SP.

The following **SET** variables are supported:

- `SET ANSI_NULLS` Permitted values are `SET ANSI_NULLS ON` and `SET ANSI_NULLS OFF`. If `ANSI_NULLS OFF`, $a=b$ is true if $(a=b \text{ OR } (a \text{ IS NULL}) \text{ AND } (b \text{ IS NULL}))$.
- `SET DATEFIRST` number
- `SET IDENTITY_INSERT` Permitted values are `SET IDENTITY_INSERT ON` and `SET IDENTITY_INSERT OFF`. If `ON`, an `INSERT` statement can specify an identity column value. This variable applies exclusively to the current process and cannot be set on [linked tables](#). Therefore, to use this option you should define a procedure in TSQL to perform both the `SET IDENTITY_INSERT` and the `INSERT`, then link the procedure and execute the procedure in Caché via the gateway.

- `SET NOCOUNT` Permitted values are `SET NOCOUNT ON` and `SET NOCOUNT OFF`. When set to `ON`, messages indicating the number of rows affected by a query are suppressed. This can have significant performance benefits.
- `SET QUOTED_IDENTIFIER` Permitted values are `SET QUOTED_IDENTIFIER ON` and `SET QUOTED_IDENTIFIER OFF`. When `SET QUOTED_IDENTIFIER` is on, double quotes are parsed as delimiting a quoted identifier. When `SET QUOTED_IDENTIFIER` is off, double quotes are parsed as delimiting a string literal. The preferable delimiters for string literals are single quotes.
- `SET ROWCOUNT` Set to an integer. Affects subsequent `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements to limit the number of rows affected. In a `SELECT` statement, `ROWCOUNT` takes precedence over `TOP`: if `ROWCOUNT` is less than `TOP`, the `ROWCOUNT` number of rows is returned; if `TOP` is less than `ROWCOUNT`, the `TOP` number of rows is returned. `ROWCOUNT` remains set for the duration of the process or until you revert it to default behavior. To revert to default behavior, `SET ROWCOUNT 0`. If you specify a fractional value, `ROWCOUNT` is set to the next larger integer.
- `SET TRANSACTION ISOLATION LEVEL` See Transaction Statements below.

The following **SET** variables are parsed, but ignored:

- `SET DATEFORMAT`
- `SET TEXTSIZE`

4.6 Transaction Statements

Caché TSQL provides support for transactions, including named transaction names. It does not support savepoints. Distributed transactions are not supported.

4.6.1 SET TRANSACTION ISOLATION LEVEL

Supported for the following forms only:

- `SET TRANSACTION ISOLATION LEVEL READ COMMITTED`
- `SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED.`

4.6.2 BEGIN TRANSACTION

```
BEGIN TRAN [name]
BEGIN TRANSACTION [name]
```

Initiates a transaction. The optional *name* argument can be used to specify a named transaction, also known as a savepoint. The *name* value must be supplied as a literal; it cannot be a variable.

Note: A [Data Management Language \(DML\) statement](#) that is within an explicit transaction *cannot* resolve an unqualified table name using a schema search path.

4.6.3 COMMIT TRANSACTION

```
COMMIT
COMMIT TRAN
COMMIT TRANSACTION
COMMIT WORK
```

These four syntactical forms are functionally identical. 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.

A transaction is defined as the operations since and including the **BEGIN TRANSACTION** statement. A **COMMIT** restores the transaction level counter to its state immediately prior to the **BEGIN TRANSACTION** statement that initialized the transaction.

A single **COMMIT** causes all named transactions to be committed.

4.6.4 ROLLBACK TRANSACTION

```
ROLLBACK [name]
ROLLBACK TRAN [name]
ROLLBACK TRANSACTION [name]
ROLLBACK WORK [name]
```

These four syntactical forms are functionally identical; the **ROLLBACK** keyword, as specified below, refers to any of these syntactical forms. The optional *name* argument specifies a named transaction, as specified by a **BEGIN TRANSACTION name** statement. The *name* value must be supplied as a literal; it cannot be a variable.

A **ROLLBACK** rolls back a transaction, undoing work performed but not committed, decrementing the transaction level counter, and releasing locks. It is used to restore the database to a previous consistent state.

- A **ROLLBACK** rolls back all work completed during the current transaction, resets the transaction level counter to zero and releases all locks. This restores the database to its state before the beginning of the transaction.
- A **ROLLBACK name** rolls back all work done since the specified named transaction (savepoint) and decrements the 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 named transaction does not exist, or has already been rolled back, **ROLLBACK** rolls back the entire current transaction.

4.6.5 CHECKPOINT

The **CHECKPOINT** statement is parsed but ignored in Caché TSQL. It performs no operation.

4.6.6 LOCK TABLE

```
LOCK TABLE tablename IN {SHARE | EXCLUSIVE} MODE [WAIT numsecs | NOWAIT]
```

The **LOCK TABLE** statement locks all of the records in the specified table. You can lock a table in **SHARE MODE** or in **EXCLUSIVE MODE**. The optional **WAIT** clause specifies the number of seconds to wait in attempting to acquire the table lock. The **LOCK TABLE** statement immediately releases any prior lock held by the current user on the specified table. **LOCK TABLE** locks the table for the duration of the current transaction.

4.7 Procedure Statements

The following standard Transact-SQL statements are supported.

4.7.1 CREATE PROCEDURE / CREATE FUNCTION

Creates a named executable procedure.

```
CREATE PROCEDURE procname [[@var [AS] datatype [= | DEFAULT value] [,...]] [RETURNS
datatype] [AS] code
CREATE PROC procname [[@var [AS] datatype [= | DEFAULT value] [,...]] [RETURNS
datatype] [AS] code
CREATE FUNCTION procname [[@var [AS] datatype [= | DEFAULT value] [,...]] [RETURNS
datatype] [AS] code
```

You can return a single scalar value result from either a `PROCEDURE` or a `FUNCTION`. `OUTPUT` parameters and default values are also supported. These commands convert the return type from a TSQL type declaration to a Caché type descriptor. Currently, result sets and tables can't be returned.

Supported as either **CREATE PROCEDURE** or **CREATE PROC**. **CREATE FUNCTION** is very similar to **CREATE PROCEDURE**, but the routine type argument value is "FUNCTION", rather than "PROCEDURE".

- Any statements can be used in a **CREATE FUNCTION**.
- The `RETURN` keyword is allowed in a **CREATE PROCEDURE**. If a procedure completes without invoking a **RETURN** or **RAISERROR** statement, it returns an integer value of 0.
- The `WITH EXECUTE` keyword clause is allowed in a **CREATE PROCEDURE** and **CREATE FUNCTION**. It must appear after the `RETURN` keyword.

A **CREATE PROCEDURE** can specify a formal parameter list. Formal parameters are specified as a comma-separated list. Enclosing parentheses are optional. The `AS` keyword between the parameter variable and its data type is optional. Optionally, you can use the `DEFAULT` keyword or `=` symbol to assign a default value to a formal parameter; if no actual parameter value is specified, this default value is used. In TSQL an input formal parameter has no keyword indicator; an output formal parameter can be specified by the `OUTPUT` keyword following the data type. Alternatively, these formal parameters can be prefaced by the optional keywords `IN`, `OUT`, or `INOUT`.

The following example shows the creation of the procedure `AvgAge` with two formal parameters:

```
CREATE PROCEDURE AvgAge @min INT, @max INT
AS
BEGIN TRY
    SELECT AVG(Age) FROM Sample.Person
    WHERE Age > @min AND Age < @max
END TRY
BEGIN CATCH
    PRINT 'error!'
END CATCH
```

The following statement executes this procedure. In this case, the specified actual parameter values limit the averaging to ages 21 through 65:

```
EXEC AvgAge 20,66
```

The following example creates a procedure that returns the results of a division operation. The `RETURNS` keyword limits the number of decimal digits in the return value:

```
CREATE PROCEDURE SQLUser.MyDivide @a INTEGER, @b INTEGER, OUT @rtn INTEGER RETURNS DECIMAL(2,3)
BEGIN
SET @rtn = @a / @b;
RETURN @rtn;
END
```

The following statement executes this procedure:

```
SELECT SQLUser.MyDivide(7,3)
```

The following example shows the creation of procedure OurReply:

```
CREATE PROCEDURE OurReply @var CHAR(16) DEFAULT 'No thanks' AS PRINT @var
```

When executed without a parameter, OurReply prints the default text (“No thanks”); when executed with a parameter OurReply prints the actual parameter value specified in the **EXEC** statement.

Note that **CREATE FUNCTION** and **CREATE PROCEDURE** cannot be issued from a stored procedure.

4.7.1.1 Importing a CREATE PROCEDURE

If imported TSQL source contains a CREATE PROC statement, then a class method containing the CREATE PROC source will be created. This class method is either placed in an existing class, or in a new class whose name is based on the schema and procedure name.

If the procedure already exists, the existing implementation is replaced. If a class matching the class name generated from the schema and procedure already exists, it is used if it was previously generated by the TSQL utility. If not, then a unique class name is generated, based on the schema and procedure name. The schema defaults to the default schema defined in the system configuration. The resulting class is compiled once the procedure has been successfully created.

If logging is requested, the source statements are logged along with the name of the containing class, class method, and the formal arguments generated. Any errors encountered by the process are also reported in the log. If errors are detected during CREATE PROC processing and a new class was generated, that class is deleted.

4.7.2 ALTER FUNCTION

Supported. The WITH EXECUTE keyword clause is supported.

4.7.3 DROP FUNCTION

Supported.

4.7.4 DROP PROCEDURE

Deletes a procedure or a comma-separated list of procedures.

```
DROP PROCEDURE [IF EXISTS] procname [,procname2 [...] ]
DROP PROC [IF EXISTS] procname [,procname2 [...] ]
```

The optional IF EXISTS clause suppresses errors if you specify a non-existent *procname*. If this clause is not specified, an SQLCODE -362 error is generated if you specify a non-existent *procname*. **DROP PROCEDURE** is an atomic operation; either all specified procedures are successfully deleted or none are deleted.

4.7.5 RETURN

Halts execution of a query or procedure. Can be argumentless or with an argument. Argumentless **RETURN** must be used when exiting a TRY or CATCH block. When returning from a procedure, **RETURN** can optionally return an integer status code. If you specify no status code, it returns the empty string ("").

4.7.6 EXECUTE

Executes a procedure, optionally passing in parameters.

```
EXECUTE procname [param1 [,param2 [,...]]]
EXEC procname [param1 [,param2 [,...]]]
```

The EXECUTE (or EXEC) statement is supported as follows:

- **EXEC <procedure>** can be used to execute a stored procedure. The EXEC keyword cannot be omitted. Named parameters are supported. The following **EXEC <procedure>** features are not currently supported: procedure variables, and procedure numbers (i.e. 'n'). WITH RECOMPILE is accepted but ignored.

```
CREATE PROCEDURE ProcTopSample
AS SELECT TOP 4 Name, Age FROM Sample.Person
GO

EXEC ProcTopSample
GO
```

The following example executes the Caché-supplied procedure Sample.PersonSets, passing it two parameters:

```
EXECUTE Sample.PersonSets 'F', 'VT'
```

If the specified procedure does not exist, an SQLCODE -428 error (Stored procedure not found) is issued.

- **EXEC (<TSQL commands>)** can be used to execute dynamic SQL. The string of TSQL commands to be executed are enclosed in single quote characters. A TSQL command string can contain line breaks and white space. Dynamic TSQL runs in the current context.

```
DECLARE @DynTopSample VARCHAR(200)
SELECT @DynTopSample='SELECT TOP 4 Name, Age FROM Sample.Person'
EXEC(@DynTopSample)
```

4.7.7 CALL

The CALL statement is functionally identical to the EXECUTE statement. It differs syntactically.

```
CALL procname ([param1 [,param2 [,...]]])
@var = CALL procname ([param1 [,param2 [,...]]])
```

The parameters are optional. The enclosing parentheses are mandatory.

4.8 Other Statements

4.8.1 CREATE USER

CREATE USER creates a new user ID and password. Executing this statement creates a Caché User with its password set to the specified user name.

User names are not case-sensitive. Caché TSQL and Caché SQL both use the same set of defined user names.

4.8.2 CREATE ROLE

CREATE ROLE creates a new 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. The following syntax is supported; the **AUTHORIZATION** clause is optional:

```
CREATE ROLE rolename AUTHORIZATION auth
```

4.8.3 GRANT and REVOKE

```
GRANT privilegelist ON (columnlist) TO granteelist
```

- *privilegelist*: a single privilege or a comma-separated list of privileges. The available privileges are ALL, EXECUTE, SELECT, INSERT, DELETE, UPDATE, and REFERENCES.
- *columnlist*: a single column name or a comma-separated list of column names, enclosed in parentheses. You can specify a parenthesized list of column names at the end of the GRANT statement. For SELECT and UPDATE privileges, you can specify a parenthesized list of column names following that privilege keyword.
- *granteelist*: a single grantee (recipient of privileges) or a comma-separated list of grantees. A grantee can be a user name, "PUBLIC" or "*".

You can use the **REVOKE** command to explicitly remove granted privileges.

```
REVOKE privilegelist ON (columnlist) FROM granteelist CASCADE
```

4.8.4 PRINT

Supported.

4.8.5 RAISERROR

```
RAISERROR err_num 'message'  
RAISERROR(error,severity,state,arg) WITH LOG
```

Both syntactic forms (with and without parentheses) are supported. Both spellings, RAISERROR and RAISEERROR, are supported and synonymous. **RAISERROR** sets the value of @@ERROR to the specified error number and error message and invokes the **%SYSTEM.Error.FromXSQL()** method.

The Sybase-compatible syntax (without parentheses) requires an *err_num* error number, the other arguments are optional.

```
RAISERROR 123 'this is a big error'  
PRINT @@ERROR
```

A **RAISERROR** command raises an error condition; it is left to the user code to detect this error. However, if **RAISERROR** appears in the body of a TRY block, it transfers control to the paired CATCH block. If **RAISERROR** appears in a CATCH block it transfers control either to an outer CATCH block (if it exists) or to the procedure exit. **RAISERROR** does not trigger an exception outside of the procedure. It is up to the caller to check for the error.

When an AFTER statement level trigger executes a RAISEERROR, the returned `%msg` value contains the `err_num` and `message` values as message string components separated by a comma: `%msg="err_num,message"`.

The Microsoft-compatible syntax (with parentheses) requires an *error* (either an error number or a quoted error message). If you do not specify an error number, it defaults to 50000. The optional *severity* and *state* arguments take integer values.

```
RAISERROR('this is a big error',4,1) WITH LOG
PRINT @@ERROR
```

4.8.6 UPDATE STATISTICS

Supported. Provides query performance optimization. **UPDATE STATISTICS** generates a call to the Caché `$$SYSTEM.SQL.TuneTable()` method, passing the table name. For further details, see [Tune Table](#) in *Caché SQL Optimization Guide*.

4.8.7 USE database

Supported, also an extension: `USE NONE` to select no database. Effective at generation-time, persists as long as the transform object exists (e.g. in the shell or loading a batch).

4.9 InterSystems Extensions

TSQL supports a number of InterSystems extensions to Transact-SQL. To allow for the inclusion of these InterSystems-only statements in portable code, Caché TSQL also supports a special form of the single-line comment: two hyphens followed by a vertical bar. This operator is parsed as a comment by Transact-SQL implementations, but is parsed as an executable statement in Caché TSQL. For further details, refer to the Comments section of the [TSQL Constructs](#) chapter of this document.

TSQL includes the following InterSystems extensions:

4.9.1 CACHE

This extension allows you to include ObjectScript code in the compiled output. It takes one or more lines of ObjectScript inside curly brackets:

```
CACHE {WRITE "SQLCODE=",SQLCODE,!}
```

Note that in the above example the **WRITE** command specifies a new line (!); this is necessary because the **CACHE** extension does not issue a new line following execution.

The **CACHE** extension can also be used to embed Caché SQL into TSQL code:

```
CACHE {&sql(SET TRANSACTION %COMMITMODE EXPLICIT)}
```

For further details on embedded Caché SQL, refer to [Embedded SQL](#) in *Using Caché SQL*.

4.9.2 IMPORTASQUERY

This extension forces a stored procedure to be imported as a query rather than as a class method. This is useful for stored procedures that contain only an EXEC statement, because Caché cannot otherwise determine at import whether such a stored procedure is a query or not.

5

TSQL Settings

Settings are used to tailor the behavior of the compiler and colorizer. The TSQL configuration options are part of the standard Caché configuration.

Caché supports the following TSQL settings:

- DIALECT (default is Sybase)
- ANSI_NULLS
- CASEINSCOMPARE (String comparison is not case-sensitive.)
- QUOTED_IDENTIFIER
- TRACE

These values are used to set the corresponding `^%SYS("tsql","SET",...)` global array values.

For further details, see [TSQL Compatibility Settings](#) in the *Caché Additional Configuration Settings Reference*.

You can view and modify these settings using the Caché Management Portal or `%SYSTEM.TSQL` class methods.

- Go into the Caché Management Portal. Go to **System Administration, Configuration, SQL and Object Settings, TSQL Compatibility**. Here you can specify the **DIALECT** (Sybase or MSSQL, default is Sybase), and turn on or off the **ANSI_NULLS**, **CASEINSCOMPARE**, and **QUOTED_IDENTIFIER** settings.

If you change one or more configuration options, the **TSQL Settings** heading will be followed by an asterisk, indicating that changes have been made but not yet saved. You must press the **Save** button for configuration changes to take effect.

- Invoke the `$$SYSTEM.TSQL.CurrentSettings()` method to display the settings:

```
DO ##class(%SYSTEM.TSQL).CurrentSettings()
```

You can use `%SYSTEM.TSQL` class methods to get or set the `ANSI_NULLS`, `CaseInsCompare`, and `Quoted_Identifier` settings. While these methods take a dialect string, there are not separate settings for the TSQL dialects. For example, changing `CaseInsCompare` changes this configuration setting for both Sybase and MSSQL.

5.1 DIALECT

The `DIALECT` configuration option allows you to select the Transact-SQL dialect. The available options are Sybase and MSSQL. The default is Sybase. This option is set using the Caché Management Portal.

If DIALECT=MSSQL: a DECLARE statement binds host variable values.

If DIALECT=Sybase: host variable values are refreshed for each cursor OPEN.

5.2 ANSI_NULLS

The ANSI_NULLS configuration option allows you to specify whether comparisons to a null value return true or false. The default is OFF.

- **ON:** All comparisons to a null value evaluate to Unknown. For example, Age = Null returns false, even when Age is null. Null is unknown, so it is false/unknown to specify null=null.
- **OFF:** Comparisons of a non-Unicode value to a null value evaluates to True if both values are null. For example: Age = Null returns true for null values for Age.

You can determine the current ANSI_NULLS setting using %SYSTEM.TSQL class methods, or from the *TSQLAnsiNulls* property, as follows:

```
SET context=##class(%SYSTEM.Context.SQL).%New()  
WRITE "ANSI_NULLS is = ",context.TSQLAnsiNulls
```

You can activate (ON) or deactivate (OFF) ANSI_NULLS system-wide using either of the following ObjectScript commands:

```
WRITE ##class(%SYSTEM.TSQL).SetAnsiNulls("Sybase","OFF")  
  
SET ^%SYS("tsql","SET","ANSI_NULLS")="OFF"
```

5.3 CASEINSCOMPARE

The CASEINSCOMPARE setting specifies non-case-sensitive equality comparisons, such as 'A'='a'. The default is OFF. If this option is set to ON, the comparison operators = and <> operate without regard to case in most contexts. However, there are a few contexts where such insensitivity does not apply:

- Where a comparison is the ON condition for a JOIN.
- Where either operand is a subquery.

These exceptions exist because Caché SQL does not accept the %SQLUPPER operator in these contexts.

You can determine the current CASEINSCOMPARE setting using %SYSTEM.TSQL class methods, or from the *TSQLCaseInsCompare* property, as follows:

```
SET context=##class(%SYSTEM.Context.SQL).%New()  
WRITE "ANSI_NULLS is = ",context.TSQLCaseInsCompare
```

You can activate (ON) or deactivate (OFF) CASEINSCOMPARE system-wide using either of the following ObjectScript commands:

```
WRITE ##class(%SYSTEM.TSQL).SetCaseInsCompare("Sybase","OFF")  
  
SET ^%SYS("tsql","SET","CASEINSCOMPARE")="OFF"
```

5.4 QUOTED_IDENTIFIER

The QUOTED_IDENTIFIER configuration option allows you to select whether quoted identifiers are supported. The default is OFF (not supported). This option is set using the Caché Management Portal. When QUOTED_IDENTIFIER is on, double quotes are parsed as delimiting an identifier. When QUOTED_IDENTIFIER is off, double quotes are parsed as alternative delimiters for string literals. The preferable delimiters for string literals are single quotes.

You can determine the current QUOTED_IDENTIFIER setting using %SYSTEM.TSQL class methods, or from the *TSQLQuotedIdentifier* property, as follows:

```
SET context=##class(%SYSTEM.Context.SQL).%New()
WRITE "ANSI_NULLS is = ",context.TSQLQuotedIdentifier
```

You can activate (ON) or deactivate (OFF) QUOTED_IDENTIFIER system-wide using either of the following ObjectScript commands:

```
WRITE ##class(%SYSTEM.TSQL).SetQuotedIdentifier("Sybase","OFF")

SET ^%SYS("tsql","SET","QUOTED_IDENTIFIER")="OFF"
```

5.5 TRACE

The TRACE configuration option is not available from the Management Portal. It controls the behavior of the TSQL compiler. When a method is compiled with TRACE active, running this method will log traced messages to the active log file, by default. The active log file is located in the same namespace as CACHE.DAT, and is named using the current process number.

You can activate (ON) or deactivate (OFF) TRACE system-wide using the following ObjectScript command:

```
SET ^%SYS("tsql","SET","TRACE")="ON"
```

The default value is ON (TRACE active).

6

TSQL Functions

6.1 Supported Functions

The following TSQL functions are implemented.

6.1.1 ABS

```
ABS ( num )
```

Returns the absolute value of *num*. Thus both 123.99 and -123.99 return 123.99.

6.1.2 ACOS

```
ACOS ( float )
```

Arc cosine: returns the angle in radians whose cosine is *float*. Thus 1 returns 0.

6.1.3 ASCII

```
ASCII ( char )
```

Returns the integer value corresponding to the first character in string *char*. Thus, ASCII ('A') returns 65.

ASCII is functionally identical to **UNICODE**. The reverse of this function is **CHAR**.

6.1.4 ASIN

```
ASIN ( float )
```

Arc sine: returns the angle in radians whose sine is *float*. Thus 1 returns 1.570796326...

6.1.5 ATAN

```
ATAN ( float )
```

Arc tangent: returns the angle in radians whose tangent is *float*. Thus 1 returns .785398163...

6.1.6 AVG

```
AVG(numfield)
AVG(DISTINCT numfield)
```

Aggregate function: used in a query to return the average of the values in the *numfield* column. For example, `SELECT AVG(Age) FROM Sample.Person`. `AVG(DISTINCT numfield)` averages the number of unique values in the *field* column. Fields with NULL are ignored.

6.1.7 CAST

```
CAST(expression AS datatype)
```

Returns the *expression* converted to the specified *datatype*. **CAST** can be used with any supported data type. For further details, refer to [Data Types](#) in the *Caché SQL Reference*.

When *expression* is a date value string, such as '2004-11-23' and *datatype* is `TIMESTAMP` or `DATETIME`, a time value of '00:00:00' is supplied.

When *expression* is a time value string, such as '1:35PM' and *datatype* is `TIMESTAMP` or `DATETIME`, the time is converted to a 24-hour clock, the AM or PM suffix is removed, a missing seconds interval is filled in with zeros, and the default date value of '1900-01-01' is supplied. Thus '1:35PM' is converted to '1900-01-01 13:35:00'.

When *expression* is a date value string, such as '2004-11-23' and *datatype* is `DATE`, the date is returned in Caché [\\$HOROLOG](#) date format, such as 60703 (March 14, 2007).

Caché TSQL does not support data type XML. However, instead of generating an error during compilation, `CAST(x AS XML)` in SQL mode generates `CAST(x AS VARCHAR(32767))`. In procedure mode, `CAST(x AS XML)` does not generate any conversion.

See **CONVERT**.

6.1.8 CEILING

```
CEILING(num)
```

Returns the closest integer greater than or equal to *num*. Thus 123.99 returns 124, -123.99 returns -123.

6.1.9 CHAR

```
CHAR(num)
```

Returns the character corresponding to the integer value *num*. Thus `CHAR(65)` returns A.

CHAR is functionally identical to **NCHAR**. The reverse of this function is **ASCII**.

6.1.10 CHAR_LENGTH / CHARACTER_LENGTH

```
CHAR_LENGTH(string)
CHARACTER_LENGTH(string)
```

Returns the number of characters in *string*.

6.1.11 CHARINDEX

```
CHARINDEX(seekstring,target[,startpoint])
```

Returns the position in *target* (counting from 1) corresponding to first character of the first occurrence of *seekstring*. You can use the optional *startpoint* integer to specify where to begin the search. The return value counts from the beginning of *target*, regardless of the *startpoint*. If *startpoint* is not specified, specified as 0, 1, or as a negative number, *target* is searched from the beginning. **CHARINDEX** returns 0 if *seekstring* is not found.

6.1.12 COALESCE

```
COALESCE(expression1,expression2,...)
```

Returns the first non-null expression from the specified list of expressions.

6.1.13 COL_NAME

```
COL_NAME(object_id,column_id)
```

Returns the name of the column. Can be used in procedure code or trigger code.

TSQL supports the two-argument form of this function. It does not support a third argument.

The following example returns the column name of the 4th column of Sample.Person:

```
SET sql=2
SET sql(1)="SELECT 'column name'=COL_NAME(id,4) FROM Sample.Person"
SET sql(2)="WHERE id=OBJECT_ID('Sample.Person')"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()
```

6.1.14 CONVERT

```
CONVERT(datatype,expression[,format-code])
```

Returns the *expression* converted to the specified *datatype*.

When *datatype* is BIT and *expression* is a boolean value: if the input value is a non-zero number, the result is 1. if the input value is 0, the result is 0. If the input value is the string 'TRUE' (case insensitive), the result is 1. If the input value is the string 'FALSE' (case insensitive), the result is 0. If the input value is NULL, the result is NULL. Any other input value generates an SQLCODE -141 error.

When *datatype* is datetime or timestamp and *expression* is a date value string, such as '2004-11-23', a time value of '00:00:00' is supplied. When *expression* is a time value string, such as '1:35PM' and *datatype* is datetime or timestamp, the time is converted to a 24-hour clock, the AM or PM suffix is removed, a missing seconds interval is filled in with zeros, and the default date value of '1900-01-01' is supplied. Thus '1:35PM' is converted to '1900-01-01 13:35:00'.

CONVERT supports the DATETIME2 data type. Caché maps DATETIME2 to system-defined DDL mapping %Library.TimeStamp. This mapping is supplied with new installs; if you are using an upgrade install, you may need to create this mapping.

The optional *format-code* argument is used to specify a date/time format when converting a datetime or timestamp value to a string. By specifying various format codes you can return a dates and times in a variety of different formats. The

available format codes are 0 through 14 (100 through 114), 20 & 21 (120 & 121), 126, 130, and 131. The default *format-code* is 120:

```
yyyy-mm-dd hh:mm:ss
```

The 20 & 21 (120 & 121) format codes also support the format:

```
yyyy-mm-ddThh:mm:ss.fff
```

For further details, refer to the functionally identical Caché SQL [CONVERT](#) function in the *Caché SQL Reference*.

See **CAST**.

6.1.15 COS

```
COS(float)
```

Cosine: returns the cosine of the angle specified in *float*. Thus 1 returns .540302305...

6.1.16 COT

```
COT(float)
```

Cotangent: returns the cotangent of the angle specified in *float*. Thus 1 returns .64209261593...

6.1.17 COUNT

```
COUNT(field)
COUNT(DISTINCT field)
COUNT(*)
COUNT(1)
```

Aggregate function: used in a query to return the count of the values in the *field* column. Fields with NULL are not counted. For example, `SELECT COUNT(Name) FROM Sample.Person`. `COUNT(*)` and `COUNT(1)` are synonyms, they count all rows. `COUNT(DISTINCT field)` counts the number of unique values in the *field* column. Fields with NULL are not counted.

6.1.18 CURRENT_DATE

```
CURRENT_DATE
CURRENT DATE
```

Returns the current local date in the following format:

```
yyyy-mm-dd
```

The two syntax forms, with and without an underscore, are identical. Note that no parentheses are used with this function.

This function is provided for compatibility with SQL Anywhere; it is supported by both the Sybase and MSSQL dialects.

6.1.19 CURRENT_TIME

```
CURRENT_TIME
CURRENT TIME
```

Returns the current local time in the following format:

```
hh:mm:ss
```

Time is specified using a 24-hour clock, Fractional seconds are not returned.

The two syntax forms, with and without an underscore, are identical. Note that no parentheses are used with this function.

This function is provided for compatibility with SQL Anywhere; it is supported by both the Sybase and MSSQL dialects.

6.1.20 CURRENT_TIMESTAMP

```
CURRENT_TIMESTAMP
CURRENT TIME
```

Returns the current local date and time in the following format:

```
yyyy-mm-dd hh:mm:ss
```

Time is specified using a 24-hour clock, Fractional seconds are not returned.

The two syntax forms, with and without an underscore, are identical. Note that no parentheses are used with this function.

6.1.21 CURRENT_USER

```
CURRENT_USER
```

Returns the name of the current user.

Note that no parentheses are used with this function.

6.1.22 DATALENGTH

```
DATALENGTH(expression)
```

Returns an integer specifying the number of bytes used to represent *expression*. Thus 'fred' returns 4, and +007.500 returns 3.

6.1.23 DATEADD

```
DATEADD(code, num, date)
```

Returns the value of *date* modified by adding the interval specified in *code* the *num* number of times. The *date* can be a date, time, or date/time string in a variety of formats. You can specify any of the following *code* values, either the abbreviation (left column) or the name (right column):

yy	Year
qq	Quarter
mm	Month
dy	DayofYear
dd	Day
dw, w	Weekday
wk	Week
hh	Hour
mi	Minute
ss	Second
ms	Millisecond

Code values are not case-sensitive. Day, DayofYear, and Weekday all return the same value.

The value returned by DATEADD always includes both date and time in the format:

yyyy-mm-dd hh:mm:ss.n

Fractional seconds are only returned if the source contained fractional seconds.

If a date is not specified (that is, if *date* contains only a time value), it defaults to 1/1/1900.

If a time is not specified in *date*, it defaults to 00:00:00. Hours are always returned based on a 24-hour clock.

6.1.24 DATEDIFF

DATEDIFF(*code*, *startdate*, *enddate*)

Returns the number of *code* intervals between *startdate* and *enddate*. The two dates can be a date, a time, or a date/time string. in the following format:

yyyy-mm-dd hh:mm:ss.n

You can specify any of the following *code* values, either the abbreviation (left column) or the name (right column):

yy	Year
mm	Month
dd	Day
dw, w	Weekday
wk	Week
hh	Hour
mi	Minute
ss	Second
ms	Millisecond

Code values are not case-sensitive. Day, DayofYear, and Weekday all return the same value.

If a date is not specified (that is, if *startdate* or *enddate* contains only a time value), it defaults to 1/1/1900.

If a time is not specified in *startdate* or *enddate*, it defaults to 00:00:00.

6.1.25 DATENAME

DATENAME(*code*, *date*)

Returns the value of the part of the date specified by *code* as a string. The *date* can be a date, time, or date/time string in a variety of formats. *date* must be specified as a quoted string; *code* permits, but does not require enclosing quotes. Available *code* values are:

yyyy, yy year	Year. Returns a four-digit year. If a two-digit year is specified, DATENAME supplies '19' as first two digits.
qq, q quarter	Quarter. Returns an integer 1 through 4.
mm, m month	Month. Returns the full name of the month. For example, 'December'.
dy, y dayofyear	Day of Year. Returns an integer count of days 1 through 366.
dd, d day	Day of Month. Returns an integer count 1 through 31.
wk, ww week	Week of Year. Returns an integer count 1 through 53.
dw, w weekday	Day of Week. Returns the number of the day of the week, counting from Sunday. For example, 3 is Tuesday.
hh hour	Hour. Returns the hour of the day (24-hour clock), an integer 0 through 23.
mi, n minute	Minute. Returns an integer 0 through 59.
ss, s second	Second. Returns a decimal number 0 through 59 which may have a fractional part representing milliseconds.
ms millisecond	Millisecond. Returns the fractional part of a second as an integer.

Code values are not case-sensitive.

If a date is not specified, it defaults to 1/1/1900. Two-digit years default to 19xx.

If a time is not specified, it defaults to 00:00:00. Hours are always returned based on a 24-hour clock. Seconds are always returned with fractional seconds, if fractional seconds are defined. Milliseconds are returned as an integer, not a decimal fraction.

6.1.26 DATEPART

```
DATEPART (code, date)
```

Returns the value of the part of the date specified in *code* as an integer. The *date* can be a date, time, or date/time string in a variety of formats. Available *code* values are listed in **DATENAME**.

6.1.27 DAY

```
DAY(date)
```

Returns the day portion of the specified date or date/time string. The *date* can be specified in ODBC timestamp format:

```
yyyy-mm-dd hh:mm:ss.n
```

The *date* must contain a date component. The date separator must be a hyphen (-).

The *date* can also be specified in Caché **\$HOROLOG** date format, such as 60703 (March 14, 2007).

6.1.28 DB_NAME

```
DB_NAME ( )
```

Returns the current namespace name. No argument is permitted.

6.1.29 DEGREES

```
DEGREES(float)
```

Converts an angle measurement in radians to the corresponding measurement in degrees.

6.1.30 ERROR_MESSAGE

When invoked from within a CATCH block, returns the current error message. Otherwise, returns NULL.

6.1.31 ERROR_NUMBER

When invoked from within a CATCH block, returns the current SQLCODE error. Otherwise, returns NULL.

6.1.32 EXEC

```
EXEC (@var)
```

Executes dynamic SQL at runtime, as shown in the following example:

```
DECLARE @dyncode VARCHAR(200)
SELECT @dyncode='SELECT TOP 4 Name,Age FROM Sample.Person'
EXEC(@dyncode)
```

Compare this dynamic execution with the [EXECUTE](#) command that executes a stored procedure.

6.1.33 EXP

```
EXP(num)
```

Returns the exponential of *num*. This is the e constant (2.71828182) raised to the power of *num*. Thus EXP(2) returns 7.3890560989.

6.1.34 FLOOR

```
FLOOR(num)
```

Returns the closest integer less than or equal to *num*. Thus 123.99 returns 123, -123.99 returns -124.

6.1.35 GETDATE

```
GETDATE()
```

Returns the current local date and time in the following format:

```
yyyy-mm-dd hh:mm:ss.n
```

Time is specified using a 24-hour clock, Fractional seconds are returned.

6.1.36 GETUTCDATE

```
GETUTCDATE()
```

Returns the current UTC (Greenwich Mean Time) date and time in the following format:

```
yyyy-mm-dd hh:mm:ss.n
```

Time is specified using a 24-hour clock, Fractional seconds are returned.

6.1.37 HOST_NAME

```
HOST_NAME()
```

Returns the system name of the current host system.

6.1.38 INDEX_COL

```
INDEX_COL(table_name,index_id,key,[,user_id])
```

Returns the name of the indexed column in the specified table. *table_name* can be fully qualified. *index_id* is the number of the table's index. *key* is a key in the index, a value between 1 and sysindexes.keycnt (for a clustered index) or sysindexes.keycnt+1 (for a non-clustered index). *user_id* is parsed but ignored.

6.1.39 ISNULL

```
ISNULL(expr, default)
```

If *expr* is NULL, returns *default*. If *expr* is not NULL, returns *expr*.

6.1.40 ISNUMERIC

```
ISNUMERIC(expression)
```

A boolean function that returns 1 if *expression* is a valid numeric value; otherwise, returns 0.

6.1.41 LEFT

```
LEFT(string, int)
```

Returns *int* number of characters from *string*, counting from the left. If *int* is larger than *string*, the full string is returned. See RIGHT.

6.1.42 LEN

```
LEN(string)
```

Returns the number of characters in *string*.

6.1.43 LOG

```
LOG(num)
```

Returns the natural logarithm of *num*. Thus LOG(2) returns .69314718055.

6.1.44 LOG10

```
LOG10(num)
```

Returns the base-10 logarithm of *num*. Thus LOG10(2) returns .301029995663.

6.1.45 LOWER

```
LOWER(string)
```

Returns *string* with all uppercase letters converted to lowercase. See UPPER.

6.1.46 LTRIM

```
LTRIM(string)
```

Removes leading blanks from *string*. See RTRIM.

6.1.47 MAX

```
MAX(numfield)
```

Aggregate function: used in a query to return the largest (maximum) of the values in the *numfield* column. For example:

```
SELECT MAX(Age) FROM Sample.Person
```

Fields with NULL are ignored.

6.1.48 MIN

```
MIN(numfield)
```

Aggregate function: used in a query to return the smallest (minimum) of the values in the *numfield* column. For example:

```
SELECT MIN(Age) FROM Sample.Person
```

Fields with NULL are ignored.

6.1.49 MONTH

```
MONTH(date)
```

Returns the month portion of the specified date or date/time string. The *date* can be specified in ODBC timestamp format:

```
yyyy-mm-dd hh:mm:ss.n
```

The date separator must be a hyphen (-). Dates in any other format return 0.

The *date* can also be specified in Caché [\\$HOROLOG](#) date format, such as 60703 (March 14, 2007).

6.1.50 NCHAR

```
NCHAR(num)
```

Returns the character corresponding to the integer value *num*. Thus `NCHAR(65)` returns A.

NCHAR is functionally identical to **CHAR**. The reverse of this function is **ASCII**.

6.1.51 NEWID

```
NEWID()
```

Returns a unique value of a type compatible with the SQL Server UNIQUEIDENTIFIER data type. UNIQUEIDENTIFIER is a system-generated 16-byte binary string, also known as 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 numbers separated into five groups by hyphens. Caché TSQL does not support UNIQUEIDENTIFIER; it instead uses VARCHAR(36) as the data type for a Globally Unique ID.

The **NEWID** function takes no arguments. Note that the argument parentheses are required.

NEWID() can be used to specify the DEFAULT value when defining a field.

The corresponding Caché SQL function is [\\$TSQL_NEWID](#):

```
SELECT $TSQL_NEWID()
```

6.1.52 NOW

```
NOW( * )
```

Returns the current local date and time in the following format:

```
yyyy-mm-dd hh:mm:ss
```

Time is specified using a 24-hour clock, Fractional seconds are not returned.

Note that the asterisk within the parentheses is required.

6.1.53 NULLIF

```
NULLIF( expr1 , expr2 )
```

Returns NULL if *expr1* is equivalent to *expr2*. Otherwise, returns *expr1*.

6.1.54 OBJECT_ID

```
OBJECT_ID( objname , objtype )
```

Takes the object name as a quoted string, and optionally the object type, and returns the corresponding object ID of the specified object as an integer. The available *objtype* values are as follows: RI = FOREIGN KEY constraint; K = PRIMARY KEY or UNIQUE constraint; P = Stored procedure; S = System table; TR = Trigger; U = User table; V = View.

```
CREATE PROCEDURE GetName  
AS SELECT OBJECT_ID( 'Person' , 'U' )  
GO
```

Returns the NULL if *objname* does not exist, or if the optional *objtype* is specified and does not match the *objname*. Can be used within procedure code or trigger code. The inverse of OBJECT_NAME.

6.1.55 OBJECT_NAME

```
OBJECT_NAME( id )
```

Takes the object ID integer and returns the corresponding object name of the specified object. Returns the empty string if *id* does not exist. Can be used within procedure code or trigger code. The inverse of OBJECT_ID.

```
CREATE PROCEDURE GetID  
AS SELECT OBJECT_NAME(22)  
GO
```

6.1.56 PATINDEX

```
PATINDEX( pattern , string )
```

Returns an integer specifying the beginning position of the first occurrence of *pattern* in *string*, counting from 1. If *pattern* is not found in *string*, 0 is returned. Specify *pattern* as a quoted string. Comparisons are case-sensitive. The *pattern* can contain the following wildcard characters:

%	Zero or more characters. For example, '%a%' returns the position of the first occurrence of 'a' in <i>string</i> , including 'a' as the first character in string.
_	Any single character. For example, '_l%' returns 1 if <i>string</i> begins with a substring such as 'Al', 'el', and 'il'.
[xyz]	Any single character from the specified list of characters. For example, '[ai]l%' returns 1 if <i>string</i> begins with the substring 'al' or 'il', but not 'el' or 'Al'.
[a-z]	Any single character from the specified range of characters. For example, '%s[a-z]t%' matches 'sat', 'set', and 'sit'. A range must be specified in ascending ASCII sequence.

The caret (^) character is not a wildcard character; if included within square brackets it is treated as a literal. A *pattern* commonly consists of a search string enclosed in percent (%) characters '%Chicago%' indicating that the entire *string* should be searched.

6.1.57 PI

```
PI ( )
```

Returns the constant pi. The parentheses are required; no argument is permitted. Thus PI() returns 3.141592653589793238.

6.1.58 POWER

```
POWER ( num , exponent )
```

Returns the value *num* raised to *exponent*.

6.1.59 QUOTENAME

```
QUOTENAME ( value )
```

Returns *value* as a [delimited identifier](#). TSQL supports double quotes ("value") as delimiter characters. For example:

```
PRINT 123
// returns 123
PRINT QUOTENAME(123)
// returns "123"
```

6.1.60 RADIANS

```
RADIANS ( float )
```

Converts an angle measurement in degrees to the corresponding measurement in radians.

6.1.61 RAND

```
RAND([seed])
```

Returns a random number as a fractional number less than 1. The optional *seed* integer argument is ignored; it is provided for compatibility. If **RAND** is used more than once in a query it returns different random values.

6.1.62 REPLACE

```
REPLACE(target, search, replace)
```

Finds every instance of the *search* string in the *target* string and replaces it with the *replace* string, and returns the resulting string. To remove the *search* string from the *target* string, specify *replace* as an empty string.

6.1.63 REPLICATE

```
REPLICATE(expression, repeat-count)
```

REPLICATE returns a string of *repeat-count* instances of *expression*, concatenated together.

If *expression* is NULL, **REPLICATE** returns NULL. If *expression* is the empty string, **REPLICATE** 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.

Repeats *string* for the number of times specified by the integer *num*.

REPLICATE returns NULL if either argument is NULL. If *string* is the empty string ("), **REPLICATE** returns the empty string. If *num* is a negative number, **REPLICATE** returns NULL. If *num* is zero, **REPLICATE** returns the empty string.

6.1.64 REVERSE

```
REVERSE(string)
```

Reverses the order of the characters in *string*.

6.1.65 RIGHT

```
RIGHT(string, int)
```

Returns *int* number of characters from *string*, counting from the right. If *int* is larger than *string*, the full string is returned. See LEFT.

6.1.66 ROUND

```
ROUND(num, length)
```

Returns *num* rounded to the number of decimal digits specified by the integer *length*. If *length* is greater than the number of decimal digits, no rounding is performed. If *length* is 0, *num* is rounded to an integer. If the *length* argument is omitted, it defaults to 0. If *length* is a negative integer, *num* is rounded to the left of the decimal point. A third argument is not accepted by **ROUND**.

6.1.67 RTRIM

```
RTRIM(string)
```

Removes trailing blanks from *string*.

6.1.68 SCOPE_IDENTITY

Returns the last identity value inserted into an **IDENTITY** column in the same scope. However, the last **IDENTITY** is not limited to the scope of the current procedure. Therefore, you should only use **SCOPE_IDENTITY** when you know that a statement within the current procedure has generated an **IDENTITY** value. For example, **SCOPE_IDENTITY** should be used after an **INSERT** command in the same procedure.

The following Dynamic SQL example returns the **IDENTITY** value from the second **INSERT**:

```
SET sql=6
SET sql(1)="CREATE TABLE #mytest (MyId INT IDENTITY(1,1),"
SET sql(2)="Name VARCHAR(20))"
SET sql(3)="INSERT INTO #mytest(Name) VALUES ('John Smith')"
SET sql(4)="INSERT INTO #mytest(Name) VALUES ('Walter Jones')"
SET sql(5)="PRINT SCOPE_IDENTITY()"
SET sql(6)="DROP TABLE #mytest"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()
```

6.1.69 SIGN

```
SIGN(num)
```

Returns a value indicating the sign of *num*. If *num* is negative (for example, -32), it returns -1. If *num* is positive (for example, 32 or +32), it returns 1. If *num* is zero (for example, 0 or -0), it returns 0.

6.1.70 SIN

```
SIN(float)
```

Sine: returns the sine of the angle specified in *float*. Thus 1 returns .841470984807...

6.1.71 SPACE

```
SPACE(num)
```

Returns a string of blank spaces of length *num*.

6.1.72 SQRT

```
SQRT(num)
```

Returns the square root of *num*. Thus **SQRT(9)** returns 3.

6.1.73 SQUARE

```
SQUARE(num)
```

Returns the square of *num*. Thus `SQUARE(9)` returns 81.

6.1.74 STR

```
STR(num, [length[, precision]])
```

Returns a string of *length* characters. If the integer *length* is equal to or greater than the number of characters in the numeric *num* (including decimal point and sign characters), **STR** returns *num* converted to a string and padded with leading blanks to make the resulting string of *length* characters.

If the optional integer *precision* is specified, *num* is truncated to the specified number of decimal digits before string conversion. If *precision* is omitted, *num* is truncated to its integer portion. If *precision* is larger than the number of decimal digits, *num* is padded with trailing zeros before string conversion.

If *length* is omitted, it defaults to 10. If *length* is less than the number of characters in *num* (after adjustment by *precision*) a dummy string consisting of all asterisks of *length* number of characters is returned.

6.1.75 STUFF

```
STUFF(string, start, length, replace)
```

Returns *string* with *length* number of characters removed and the *replace* string inserted. The point of removal and insertion is specified by the *start* integer, counting from the beginning of *string*. If *length* is 0, no characters are removed. If *replace* is the empty string, no characters are inserted.

If *start* is greater than the number of characters in *string*, no value is returned. If *start* is 1, *length* number of characters are removed from the beginning of *string* and the *replace* string inserted. If *start* is 0, *length* minus 1 number of characters are removed from the beginning of *string* and the *replace* string inserted.

If *length* is greater than or equal to the number of characters in *string*, the *replace* string is returned. The *replace* string length is not limited by the length of *string* or *length*.

6.1.76 SUBSTRING

```
SUBSTRING(string, start, length)
```

Returns a substring of *string* beginning at the location *start* for the *length* number of characters. If *start* is greater than the length of *string*, or if *length* is 0, no string is returned.

6.1.77 SUM

```
SUM(numfield)
SUM(DISTINCT numfield)
```

Aggregate function: used in a query to return the sum of the values in the *numfield* column. For example:

```
SELECT SUM(Age) FROM Sample.Person
```

`SUM(DISTINCT numfield)` sums the unique values in the *field* column. Fields with NULL are ignored.

6.1.78 SUSER_NAME

```
SUSER_NAME ( )
```

Returns the name of the current OS user. Parentheses are required, no argument is permitted. Equivalent to TSQL `USER_NAME()`, the Caché SQL `USER` function, and the ObjectScript `$USERNAME` special variable.

6.1.79 SUSER_SNAME

```
SUSER_SNAME ( )
```

Returns the name of the current OS user. Parentheses are required, no argument is permitted. Equivalent to TSQL `USER_NAME()`, the Caché SQL `USER` function, and the ObjectScript `$USERNAME` special variable.

6.1.80 TAN

```
TAN(float)
```

Tangent: returns the tangent of the angle specified in *float*. Thus 1 returns 1.55740772465...

6.1.81 TEXTPTR

```
TEXTPTR(field)
```

Returns an internal pointer to the image or text column data specified in *field*. The data type of this pointer is `VARBINARY(16)`.

6.1.82 TEXTVALID

```
TEXTVALID('table.field',textpointer)
```

Takes an internal pointer to an image or text column from `TEXTPTR`, and compares it to a specified in *table.field*. Returns 1 if the pointer points to the specified *table.field*. Otherwise, returns 0.

6.1.83 UNICODE

```
UNICODE(char)
```

Returns the Unicode integer value corresponding to the first character in the string *char*. Thus, `UNICODE('A')` returns 65.

`UNICODE` is functionally identical to `ASCII`. The reverse of this function is `CHAR`.

6.1.84 UPPER

```
UPPER(string)
```

Returns *string* with all lowercase letters converted to uppercase. See `LOWER`.

6.1.85 USER

```
USER
```

Returns the name of the current user.

Note that no parentheses are used with this function.

6.1.86 USER_NAME

```
USER_NAME([userid])
```

Returns the name of the user specified by user ID. If the optional *userid* is omitted, returns the name of the current user. The argument is optional; the parentheses are mandatory.

6.1.87 YEAR

```
YEAR(date)
```

Returns the year portion of the specified date or date/time string. The *date* can be specified in ODBC timestamp format:

```
yyyy-mm-dd hh:mm:ss.n
```

The date separator can be either a hyphen (-) or a slash (/).

The *date* can also be specified in Caché [\\$HOROLOG](#) date format, such as 60703 (March 14, 2007).

6.2 Unsupported Functions

The following Microsoft Transact-SQL functions are not supported by TSQL at this time: APP_NAME, ATN2, BINARY_CHECKSUM, CHECKSUM, COL_LENGTH, COLLATIONPROPERTY, COLUMNPROPERTY, CURSOR_STATUS, DATABASEPROPERTY, DATABASEPROPERTYEX, DB_ID, DIFFERENCE, FILE_ID, FILE_NAME, FILEGROUP_ID, FILEGROUP_NAME, FILEGROUPPROPERTY, FILEPROPERTY, FORMATMESSAGE, FULLTEXTCATALOGPROPERTY, FULLTEXTSERVICEPROPERTY, GETANSINULL, HOST_ID, IDENT_CURRENT, IDENT_INCR, IDENT_SEED, IDENTITY, INDEXKEY_PROPERTY, INDEXPROPERTY, ISDATE, IS_MEMBER, IS_SRVROLEMEMBER, OBJECTPROPERTY, PARSENAME, PERMISSIONS, ROWCOUNT_BIG, SERVERPROPERTY, SESSIONPROPERTY, SESSION_USER, SOUNDEX, SQL_VARIANT_PROPERTY, STATS_DATE, STDEV, STDEVP, SYSTEM_USER, TYPEPROPERTY.

7

TSQL Variables

7.1 Local Variables

By default, TSQL local variables are specified using an At Sign (@) prefix. For example, @myvar. You can override this default to also allow PLAINLOCALS, TSQL local variables specified without an At Sign (@) prefix. For example, myvar.

7.1.1 Declaring a Local Variable

A local variable must be declared (using `DECLARE` or as a formal parameter) before use. A variable name must be a valid [identifier](#). Local variable names are not case-sensitive. The declaration must specify a data type, though strict data typing is not enforced in Caché TSQL. For a list of supported data types, refer to the [TSQL Constructs](#) chapter of this document.

The **DECLARE** command has the following syntax:

```
DECLARE @var [AS] datatype [ = initval ]
```

If declaring variables is inconvenient, you can switch this check off using the NDC setting. However, cursors must be declared, even if NDC is used.

Stored procedure arguments are automatically declared as local variables.

7.1.2 Setting a Local Variable

A local variable can be set using either the **SET** command or the **SELECT** command. A local variable can be displayed using either the **PRINT** command or the **SELECT** command. The following Dynamic SQL examples show two local variables being declared, set, and displayed:

```
SET myquery = 3
SET myquery(1) = "DECLARE @a CHAR(20),@b CHAR(20) "
SET myquery(2) = "SET @a='hello ' SET @b='world!' "
SET myquery(3) = "PRINT @a,@b"
SET tStatement = ##class(%SQL.Statement).%New(,,"MSSQL")
SET qStatus = tStatement.%Prepare(.myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

```

SET myquery = 3
SET myquery(1) = "DECLARE @a CHAR(20),@b CHAR(20) "
SET myquery(2) = "SELECT @a='hello ', @b='world!'"
SET myquery(3) = "SELECT @a,@b"
SET tStatement = ##class(%SQL.Statement).%New(,,"MSSQL")
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()

```

7.1.3 Initial and Default Values

By default, **DECLARE** initializes local variables to " " (SQL NULL). Optionally, you can specify an initial value (*initval*) for a local variable in the **DECLARE** command.

If a declared variable is set to the results of a scalar subquery, and the subquery returns no rows, Caché TSQL sets the variable to " " (SQL NULL). This default is compatible with MS SQLServer; it is not compatible with Sybase.

7.1.4 Plain Local Variables

By default, local variables require an @ prefix. However, you can specify plain locals, local variables that do not require an @ prefix. The following command activates plain local variables:

```
SET PLAINLOCALS ON
```

You must activate plain local variables before declaring these variables. With plain local variables activated you can declare both local variables with an @ prefix and local variables without an @ prefix. However, you cannot declare two variables that only differ by the @ prefix. For example, @myvar and myvar are considered the same variable. When declaring, selecting, or printing a plain local variable, you can specify the same variable with or without the @ prefix.

Plain local variables follow all of the other TSQL variable conventions.

7.2 @@ Special Variables

TSQL special variables are identified by an @@ prefix. @@ variables are system-defined; they cannot be created or modified by user processes. @@ variables are global in scope (available to all processes). They are thus sometimes referred to elsewhere in the Transact-SQL literature as “global variables.” Because the term “global variable” is used widely in Caché and differs significantly in meaning, these TSQL @@ variables are referred to here as “special variables” to avoid confusion.

The following special variables are implemented. Invoking an unimplemented special variable generates a #5001 '@@nnn' unresolved symbol error or a #5002 <UNDEFINED> error. The corresponding ObjectScript (COS) and Caché SQL generated code for each special variable is provided:

7.2.1 @@ERROR

Contains the error number of the most recent TSQL error. 0 indicates that no error has occurred. A 0 value is returned when either SQLCODE=0 (successful completion) or SQLCODE=100 (no data, or no more data). To differentiate these two results, use @@SQLSTATUS.

```
COS SQLCODE
```

```
SQL : SQLCODE
```

7.2.2 @@FETCH_STATUS

Contains an integer specifying the status of the last FETCH cursor statement. The available options are: 0=row successfully fetched; -1=no data could be fetched; -2 row fetched is missing or some other error occurred. A value of -1 can indicate that there is no data to FETCH, or that the fetch has reached the end of the data.

```
SET myquery = "SELECT @@FETCH_STATUS AS FetchStat"
SET tStatement = ##class(%SQL.Statement).%New(,,"MSSQL")
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The corresponding Caché SQL function is:

```
SELECT $TSQL_FETCH_STATUS()
```

```
COS $Case($Get(SQLCODE,0),0:0,100:-1,:-2)
```

```
SQL CASE :SQLCODE WHEN 0 THEN 0 WHEN 100 THEN -1 ELSE -2 END
```

7.2.3 @@IDENTITY

Contains the IDENTITY field value of the most recently inserted, updated, or deleted row.

```
COS %ROWID
```

```
SQL :%ROWID
```

7.2.4 @@LOCK_TIMEOUT

Contains an integer specifying the timeout value for locks, in seconds. Lock timeout is used when a resource needs to be exclusively locked for inserts, updates, deletes, and selects. The default is 10.

```
SET myquery = "SELECT @@LOCK_TIMEOUT AS LockTime"
SET tStatement = ##class(%SQL.Statement).%New(,,"MSSQL")
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The corresponding Caché SQL function is:

```
SELECT $TSQL_LOCK_TIMEOUT()
```

```
COS LOCK command
```

```
SQL SET OPTION LOCK_TIMEOUT
```

7.2.5 @@NESTLEVEL

Contains an integer specifying the nesting level of the current process.

```
SET myquery = "PRINT @@NESTLEVEL"
SET tStatement = ##class(%SQL.Statement).%New(,,"MSSQL")
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The corresponding Caché SQL function is:

```
SELECT $TSQL_NESTLEVEL()
```

COS \$STACK

7.2.6 @@ROWCOUNT

Contains the number of rows affected by the most recent **SELECT**, **INSERT**, **UPDATE**, or **DELETE** command. A single-row **SELECT** always returns a @@ROWCOUNT value of either 0 (no row selected) or 1.

When invoking an AFTER statement level trigger, the @@ROWCOUNT value upon entering the trigger is the @@ROWCOUNT immediately prior to the trigger. Rows affected within the scope of the trigger code are reflected in the @@ROWCOUNT value. Upon completion of the trigger code, @@ROWCOUNT reverts to the value immediately prior to the trigger invocation.

COS %ROWCOUNT

SQL :%ROWCOUNT

7.2.7 @@SERVERNAME

Contains the Caché instance name.

```
SET myquery = "SELECT @@SERVERNAME AS CacheInstance"
SET tStatement = ##class(%SQL.Statement).%New(, "MSSQL")
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The corresponding Caché SQL function is:

```
SELECT $TSQL_SERVERNAME()
```

COS \$PIECE(\$system, ":", 2)

7.2.8 @@SPID

Contains the server process ID of the current process.

```
SET myquery = "SELECT @@SPID AS ProcessID"
SET tStatement = ##class(%SQL.Statement).%New(, "MSSQL")
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The corresponding Caché SQL function is:

```
SELECT $TSQL_SPID()
```

COS \$JOB

7.2.9 @@SQLSTATUS

Contains an integer specifying the completion status of the most recent SQL statement. Available values are: 0=successful completion; 1=failure; 2=no (more) data available.

```
SET myquery = "SELECT @@SQLSTATUS AS SQLStatus"
SET tStatement = ##class(%SQL.Statement).%New(, "MSSQL")
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The corresponding Caché SQL function is:

```
SELECT $TSQL_SQLSTATUS()

COS $Case($Get(SQLCODE,0),0:0,100:2,:1)

SQL CASE :SQLCODE WHEN 0 THEN 0 WHEN 100 THEN 2 ELSE 1 END
```

7.2.10 @@TRANCOUNT

Contains the number of currently active transactions.

```
SET myquery = "SELECT @@TRANCOUNT AS ActiveTransactions"
SET tStatement = ##class(%SQL.Statement).%New(,,"MSSQL")
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The corresponding Caché SQL function is:

```
SELECT $TSQL_TRANCOUNT()

COS $TLEVEL
```

7.2.11 @@VERSION

Contains the Caché version number and date and time of its installation.

```
SET myquery = "SELECT @@VERSION AS CacheVersion"
SET tStatement = ##class(%SQL.Statement).%New(,,"MSSQL")
SET qStatus = tStatement.%Prepare(myquery)
IF qStatus'=1 { WRITE "%Prepare failed",$System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
```

The corresponding Caché SQL function is:

```
SELECT $TSQL_VERSION()

COS $ZVERSION
```


8

Using the TSQL Shell

The TSQL Shell is one way to execute Transact-SQL code from Caché.

To use the TSQL Shell, invoke the **TSQShell()** method from the Terminal as follows: `DO $SYSTEM.SQL.TSQShell()`. This invokes the [Caché SQL Shell](#) and sets its DIALECT configuration parameter to MSSQL.

You can perform the same operation by invoking the SQL Shell from the Terminal as follows: `DO $SYSTEM.SQL.Shell()`, then setting the DIALECT configuration parameter. The SQL Shell supports three SET DIALECT options: MSSQL, Sybase, and Cache.

When in multiline mode, the **GO** command is used to execute a multiline TSQL statement and exit multiline mode. This statement delimiter is configurable to the statement delimiter defined in imported TSQL code.

For further details about this shell, refer to the “[Using the SQL Shell Interface](#)” chapter of *Using Caché SQL*.

8.1 Other Ways to Execute Transact-SQL Code

You can also execute Transact-SQL code:

- as `ClassMethod methodName()` [`Language = tsql, SqlProc`],
- as [Dynamic SQL](#) program using the `%Dialect` property.
- from the Management Portal [Execute SQL Statements](#) option, by setting the **Dialect** option, which is displayed if you click the **more** option.

