InterSystems®
Caché

Using Caché Globals

Version 2017.2
2020-06-25

Using Caché Globals
Caché   Version 2017.2   2020-06-25
Copyright © 2020 InterSystems Corporation
All rights reserved.

Table of Contents

List of Figures

List of Tables

About This Book

This book describes how Caché stores its data in multidimensional sparse arrays, known as *globals*.

This book covers the following topics:

- Introduction provides an overview of the features and uses of globals.

- "Global Structure" describes how globals are stored on disk, how they are named and referenced, and their structure.

- "Using Multidimensional Storage (Globals)" describes how to work with globals programmatically.

- "SQL and Object Usage of Multidimensional Storage" describes how the object and SQL engines use globals to store data.

- "Managing Globals" describes tools for managing globals, primarily from the Management Portal.

For a detailed outline, see the Table of Contents.

For related programming guides, see:

- *Caché Programming Orientation Guide*

- *Using Caché ObjectScript*

- *Using Caché Objects*

- *Using Caché Basic*

For general information, see *Using InterSystems Documentation*.

1

Introduction

One of the central features of Caché is its multidimensional storage engine. This feature lets applications store data in compact, efficient, multidimensional sparse arrays. These arrays are referred to as *globals*.

This document describes:

- What globals are and the operations you can perform on them.

- The logical and physical structure of globals, including the use of globals in distributed database architecture.

- How you can use globals to store and retrieve data within your applications.

- How Caché uses globals within its SQL and Object engines.

1.1 Features

Globals provide an easy-to-use way to store data in persistent, multidimensional arrays.

For example, you can associate the value "Red" with the key "Color" using a global named *Settings*:

```
SET ^Settings("Color")="Red"
WRITE !,^Settings("Color")
```

You can take advantage of the multidimensional nature of globals to define a more complex structure:

```
^Settings("Auto1","Properties","Color") = "Red"
^Settings("Auto1","Properties","Model") = "SUV"
^Settings("Auto2","Owner") = "Mo"
^Settings("Auto2","Properties","Color") = "Green"
```

Globals have the following features:

- *Simple to use* — Globals are as easy to use as other programming language variables. A comprehensive set of commands in both the ObjectScript language and the Caché Basic scripting language make it extremely easy to use globals within applications.

- *Multidimensional* — You can specify the address of a node within a global using any number of subscripts. For example, in `^Settings("Auto2","Properties","Color")`, the subscript *Color* is a third-level node within the *Settings* global. Subscripts can be integer, numeric, or string values, and need not be contiguous.

- *Sparse* — The subscripts used to address global nodes are highly compacted and need not have contiguous values.

- *Efficient* — The operations on globals — inserting, updating, deleting, traversing, and retrieving — are all highly optimized for maximum performance and concurrency. There are additional commands for specialized operations

(such as bulk inserts of data). There is a special set of globals designed for temporary data structures (such as for sorting records).

- *Reliable* — The Caché database provides a number of mechanisms to ensure the reliability of data stored within globals, including both logical-level and physical-level journaling. Data stored within globals is backed up when a database backup operation is performed.

- *Distributed* — Caché provides a number of ways to control the physical location of data stored within globals. You can define a physical database used to store a global, or distribute portions of a global across several databases. Using the distributed database features of Caché, you can share globals across a network of database and application server systems. In addition, by means of data shadowing technology, data stored within globals on one system can be automatically replicated on another system.

- *Concurrent* — Globals support concurrent access among multiple processes. Setting and retrieving values within individual nodes (array elements) is always *atomic*: no locking is required to guarantee reliable concurrent access. In addition, Caché supports a powerful set of locking operations that can be used to provide concurrency for more complex cases involving multiple nodes. When using Object or SQL access, this concurrency is handled automatically.

- *Transactional* — Caché provides commands that define transaction boundaries; you can start, commit, or rollback a transaction. In the event of a rollback, all modifications made to globals within the transaction are undone; the contents of the database are restored to their pre-transaction state. By using the various Caché locking operations in conjunction with transactions, you can perform traditional ACID transactions using globals. (An ACID transaction provides Atomicity, Consistency, Isolation, and Durability.) When using object or SQL access, transactions are handled automatically.

Note: The *globals* described in this document should not be confused with another type of Caché array variable: *process-private globals*. Process-private globals are not persistent; they persist only for the duration of the process that created them. Process-private globals are also not concurrent; they can only be accessed by the process that created them. A process-private global can be easily distinguished from a global by its multi-character name prefix: either `^||` or `^|"^"|`.

1.2 Examples

A simple example can demonstrate the ease and performance of globals. The following program example creates a 10,000–node array (deleting it first if present) and stores it in the database. You can try this to get a sense of the performance of globals:

Creating a Persistent Array

```
Set start = $ZH  // get current time

Kill ^Test.Global
For i = 1:1:10000 {
    Set ^Test.Global(i) = i
}

Set elap = $ZH - start  // get elapsed time
Write "Time (seconds): ",elap
```

We can also see how long it takes to iterate over and read the values in the array (make sure to run the above example first to build the array):

Reading a Persistent Array

```
Set start = $ZH  // get current time
Set total = 0
Set count = 0

// get key and value for first node
Set i = $Order(^Test.Global(""),1,data)

While (i '= "") {
    Set count = count + 1
    Set total = total + data

  // get key and value for next node
  Set i = $Order(^Test.Global(i),1,data)
}

Set elap = $ZH - start  // get elapsed time

Write "Nodes:           ",count,!
Write "Total:           ",total,!
Write "Time (seconds): ",elap,!
```

1.3 Use in Applications

Within Caché applications, globals are used in many ways, including:

- As the underlying storage mechanism shared by the object and SQL engines.

- As the mechanism used to provide a variety of indices, including bitmap indices, for object and SQL data.

- As a work space for performing certain operations that may not fit within process memory. For example, the SQL engine uses temporary globals for sorting data when there is no preexisting index available for this purpose.

- For performing specialized operations on persistent objects or SQL tables that are difficult or inefficient to express in terms of object or SQL access. For example, you can define a method (or stored procedure or web method) to perform specialized analysis on data held in a table. By using methods, such an operation is completely encapsulated; the caller simply invokes the method.

- To implement application-specific, customized storage structures. Many applications have the need to store data that is difficult to express relationally. Using globals you can define custom structures and make them available to outside clients via object methods.

- For a variety of special purpose data structures used by the Caché system, such as configuration data, class definitions, error messages, and executable code.

Globals are not constrained by the confines of the relational model. They provide the freedom to develop customized structures optimized for specific applications. For many applications, judicious use of globals can be a secret weapon delivering performance that relational application developers can only dream about.

Whether your application makes direct use of globals or not, it is useful to understand their operation. Understanding globals and their capabilities will help you to design more efficient applications as well as provide help with determining the optimal deployment configuration for your applications.

2

Global Structure

This chapter describes the logical (programmatic) view of globals and provides an overview of how globals are physically stored on disk. Its sections are:

- Logical Structure of Globals
- Maximum Length of a Global Reference
- Physical Structure of Globals
- Referencing Globals

2.1 Logical Structure of Globals

A global is a named multidimensional array that is stored within a physical Caché database. Within an application, the mapping of globals to physical databases is based on the current namespace — a namespace provides a logical, unified view of one or more physical databases.

Topics related to the logical structure of globals are:

- Global Naming Conventions and Limits
- Introduction to Global Nodes and Subscripts
- Global Subscripts
- Global Nodes
- Collation of Globals

2.1.1 Global Naming Conventions and Limits

The name of a global specifies its purpose and use. There are two types of globals and a separate set of variables, called "process-private globals":

- A global — This is what might be called a standard global; typically, these are simply referred to as globals. It is a persistent, multidimensional array that resides in the current namespace.
- An extended global reference — This is a global located in a namespace other than the current namespace.
- A process-private global — This is an array variable that is only accessible to the process that created it.

The naming conventions for globals are:

- A global name begins with a caret character (^) prefix. This caret distinguishes a global from a local variable.

- The first character after the caret (^) prefix in a global name can be:

  - A letter or the percent character (%) — For standard globals only. If a global's name begins with "%" (but not "%Z" or "%z"), then this global is for Caché system use. % globals are typically stored within either the CACHESYS or CACHELIB databases. For more details on the % character and InterSystems naming, see "Rules and Guidelines for Identifiers" in the *Caché Programming Orientation Guide*.

  - The vertical bar (|) or the left bracket ([) — For extended global references or process-private globals. The use depends on subsequent characters. See the examples that follow this list.

- The other characters of a global name may be letters, numbers, or the period (.) character. The percent (%) character cannot be used, except as the first character of a global name. The period (.) character cannot be used as the last character of a global name.

- A global name may be up to 31 characters long (exclusive of the caret character prefix). You can specify global names that are significantly longer, but Caché treats only the first 31 characters as significant.

- Global names are case-sensitive.

- Caché imposes a limit on the total length of a global reference, and this limit, in turn, imposes limits on the length of any subscript values. See "Maximum Length of a Global Reference" for details.

In the CACHESYS database, InterSystems reserves to itself all global names except those starting with "z", "Z", "%z", and "%Z". In all other databases, InterSystems reserves all global names starting with "ISC." and "%ISC.".

2.1.1.1 Sample Global Names and Their Uses

The following are examples of the various kinds of global names and how each is used:

- `^globalname` — a standard global

- `^|"environment"|globalname` — environment syntax for an extended global reference

- `^||globalname` — a process-private global

- `^|"^"|globalname` — a process-private global

- `^[namespace]globalname` — bracket syntax for an explicit namespace in an extended global reference

- `^[directory,system]globalname` — bracket syntax for an implied namespace in an extended global reference

- `^["^"]globalname` — a process-private global

- `^["^",""]globalname` — a process-private global

Note: Global names can contain only valid identifier characters; by default, these are as specified above. However, your NLS (National Language Support) settings may define a different set of valid identifier characters. Global names cannot contain Unicode characters.

Thus, the following are all valid global names:

```
SET ^a="The quick "
SET ^A="brown fox "
SET ^A7="jumped over "
SET ^A.7="the lazy "
SET ^A1B2C3="dog's back."
WRITE ^a,^A,^A7,!,^A.7,^A1B2C3
KILL ^a,^A,^A7,^A.7,^A1B2C3 // keeps the database clean
```

2.1.2 Introduction to Global Nodes and Subscripts

A global typically has multiple *nodes*, generally identified by a *subscript* or set of subscripts. For a basic example:

```
set ^Demo(1)="Cleopatra"
```

This statement refers to the global node `^Demo(1)`, which is a node within the `^Demo` global. This node is identified by one subscript.

For another example:

```
set ^Demo("subscript1","subscript2","subscript3")=12
```

This statement refers to the global node `^Demo("subscript1","subscript2","subscript3")`, which is another node within the same global. This node is identified by three subscripts.

For yet another example:

```
set ^Demo="hello world"
```

This statement refers to the global node `^Demo`, which does not use any subscripts.

The nodes of a global form a hierarchical structure. ObjectScript provides commands that take advantage of this structure. You can, for example, remove a node or remove a node and all its children. For a full discussion, see the next chapter.

The following sections provide details on the rules for subscripts and for global nodes.

2.1.3 Global Subscripts

Subscripts have the following rules:

- Subscript values are case-sensitive.

- A subscript value can be any ObjectScript expression, provided that the expression does not evaluate to the null string ("").

   The value can include characters of all types, including blank spaces, non-printing characters, and (if your installation is a Unicode installation) Unicode characters. (Note that non-printing characters are less practical in subscript values.)

- Before resolving a global reference, Caché evaluates each subscript in the same way it evaluates any other expression. In the following example, we set one node of the `^Demo` global, and then we refer to that node in several equivalent ways:

```
SAMPLES>s ^Demo(1+2+3)="a value"

SAMPLES>w ^Demo(3+3)
a value

SAMPLES>w ^Demo(03+03)
a value

SAMPLES>w ^Demo(03.0+03.0)
a value

SAMPLES>set x=6

SAMPLES>w ^Demo(x)
a value
```

- Caché imposes a limit on the total length of a global reference, and this limit, in turn, imposes limits on the length of any subscript values. See "Maximum Length of a Global Reference" for details.

CAUTION: The preceding rules apply for all Caché supported collations. For older collations still in use for compatibility reasons, such as "pre-ISM-6.1", the rules for subscripts are more restrictive. For example, character subscripts cannot have a control character as their initial character; and there are limitations on the number of digits that can be used in integer subscripts.

Because of restrictions such as these, there is no guarantee that subscripts used for supported collations will be valid in pre-Caché collations.

2.1.4 Global Nodes

Unless long strings are enabled for your installation, each global node can contain approximately 32K characters of text. If long strings are enabled, the limit is much larger. (See "General System Limits" in the *Caché Programming Orientation Guide*.)

Within applications, nodes typically contain the following types of structure:

1. String or numeric data. With a Unicode version of Caché, string data may contain native Unicode characters.

2. A string with multiple fields delimited by a special character:

   ```
   ^Data(10) = "Smith^John^Boston"
   ```

   You can use the ObjectScript **$PIECE** function to pull such data apart.

3. Multiple fields contained within a Caché **$LIST** structure. The **$LIST** structure is a string containing multiple length-encoded values. It requires no special delimiter characters.

4. A null string (""). In cases where the subscripts are themselves used as the data, no data is stored in the actual node.

5. A bitstring. In cases where a global is used to store part of a bitmap index, the value stored within a node is a bitstring. A bitstring is a string containing a logical, compressed set of 1 and 0 values. You can construct a bitstring using the **$BIT** functions.

6. Part of a larger set of data. For example, the object and SQL engines store streams (BLOBs) as a sequential series of 32K nodes within a global. By means of the stream interface, users of streams are unaware that streams are stored in this fashion.

2.1.5 Collation of Globals

Within a global, nodes are stored in a collated (sorted) order.

Applications typically control the order in which nodes are sorted by applying a conversion to values used as subscripts. For example, the SQL engine, when creating an index on string values, converts all string values to uppercase letters and prepends a space character to make sure that the index is both not case-sensitive and collates as text (even if numeric values are stored as strings).

2.2 Maximum Length of a Global Reference

The total length of a global reference — that is, the reference to a specific global node or subtree — is limited to 511 encoded characters (which may be fewer than 511 typed characters).

For a conservative determination of the size of a given global reference, use the following guidelines:

1. For the global name: add 1 for each character.

2. For a purely numeric subscript: add 1 for each digit, sign, or decimal point.

3. For a subscript that includes nonnumeric characters: add 3 for each character.

   If a subscript is not purely numeric, the actual length of the subscript varies depending on the character set used to encode the string. A multibyte character can take up to 3 bytes.

   Note that an ASCII character can take up 1 or 2 bytes. If the collation does case folding, an ASCII character can take 1 byte for the character and 1 byte for the disambiguation byte. If the collation does not perform case folding, an ASCII character takes 1 byte.

4. For each subscript, add 1.

If the sum of these numbers is greater than 511, the reference may be too long.

Because of the way that the limitation is determined, if you must have long subscript or global names, it is helpful to avoid a large number of subscript levels. Conversely, if you are using multiple subscript levels, avoid long global names and long subscripts. Because you may not be able to control the character set(s) you are using, it is useful to keep global names and subscripts shorter.

When there are doubts about particular references, it is useful to create test versions of global references that are of equivalent length to the longest expected global reference (or even a little longer). Data from these tests provides guidance on possible revisions to your naming conventions prior to building your application.

2.3 Physical Structure of Globals

Globals are stored within physical files using a highly optimized structure. The code that manages this data structure is also highly optimized for every platform that Caché runs on. These optimizations ensure that operations on globals have high throughput (number of operations per unit of time), high concurrency (total number of concurrent users), efficient use of cache memory, and require no ongoing performance-related maintenance (such as frequent rebuilding, re-indexing, or compaction).

The physical structure used to store globals is completely encapsulated; applications do not worry about physical data structure in any way.

Globals are stored on disk within a series of data blocks; the size of each block (typically 8KB) is determined when the physical database is created. To provide efficient access to data, Caché maintains a sophisticated B-tree-like structure that uses a set of pointer blocks to link together related data blocks. Caché maintains a buffer pool — an in-memory cache of frequently referenced blocks — to reduce the cost of fetching blocks from disk.

While many database technologies use B-tree-like structures for data storage, Caché is unique in many ways:

- The storage mechanism is exposed via a safe, easy-to-use interface.

- Subscripts and data are compressed to save disk space as well as valuable in-memory cache space.

- The storage engine is optimized for transaction processing operations: inserts, updates, and deletes are all fast. Unlike relational systems, Caché never requires rebuilding indices or data in order to restore performance.

- The storage engine is optimized for maximum concurrent access.

- Data is automatically clustered for efficient retrieval.

2.3.1 How Globals Are Stored

Within data blocks, globals are stored sequentially. Both subscripts and data are stored together. There is a special case for large node values (long strings) which are stored within separate blocks. A pointer to this separate block is stored along with the node subscript.

For example, suppose you have a global with the following contents:

```
^Data(1999) = 100
^Data(1999,1) = "January"
^Data(1999,2) = "February"
^Data(2000) = 300
^Data(2000,1) = "January"
^Data(2000,2) = "February"
```

Most likely, this data would be stored within a single data block with a contiguous structure similar to (the real representation is a series of bytes):

```
Data(1999):100|1:January|2:February|2000:300|1:January|2:February|...
```

An operation on ^*Data* can retrieve its entire contents with a minimum number of disk operations.

There are a number of additional techniques used to ensure that inserts, updates, and deletes are performed efficiently.

2.4 Referencing Globals

A global resides within a particular Caché database. Portions of a global can reside in different databases if appropriate mappings are used. A database can be physically located on the current system, or on a remote system accessed through Caché networking. The term *dataset* refers to the system and the directory that contain a Caché database. For further details on networking, see the "Distributed Data Management Guide".

A *namespace* is a logical definition of the datasets and global mappings that together form a set of related information.

A simple global reference applies to the currently selected namespace. The namespace definition can cause this to physically access a database on the local system or a remote system. Different globals can be mapped to different locations or datasets (where a dataset refers to the system and the directory that contain a Caché database).

For example, to create a simple reference to the global ORDER in the namespace to which it currently has been mapped, use the following syntax:

```
^ORDER
```

This section describes two topics:

- Setting Global Mappings
- Extended Global References

2.4.1 Setting Global Mappings

You can map globals and routines from one database to another on the same or different systems. This allows simple references to data which can exist anywhere and is the primary feature of a namespace. You can map whole globals or pieces of globals; mapping a piece of a global (or a subscript) is known as *subscript-level mapping (SLM)*. Because you can map global subscripts, data can easily span disks.

To establish this type of mapping, see the "Add Global, Routine, and Package Mapping to a Namespace" section of the "Configuring Caché" chapter of the *Caché System Administration Guide*.

Global mapping is applied hierarchically. For example, if the NSX namespace has an associated DBX database, but maps the ^x global to the DBY database and ^x(1) to the DBZ database, then any subscripted form of the ^x global — except those that are part of the ^x(1) hierarchy — is mapped to DBY; those globals that are part of the ^x(1) hierarchy are mapped to DBZ. The following diagram illustrates this hierarchy:

Figure 2–1: Simple Subscript-level Mapping
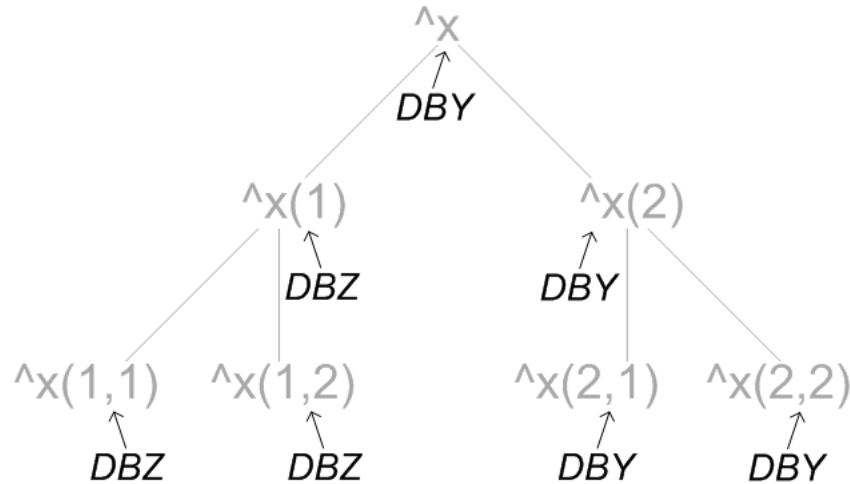
In this diagram, the globals and their hierarchy appear in gray, and the databases to which they are mapped appear in black.

It is also possible to map part of a mapped, subscripted global to another database, or even back to the database to which the initial global is mapped. Suppose that the previous example had the additional mapping of the ^x(1,2) global back to the DBY database. This would appear as follows:

Figure 2–2: More Complex Subscript-level Mapping
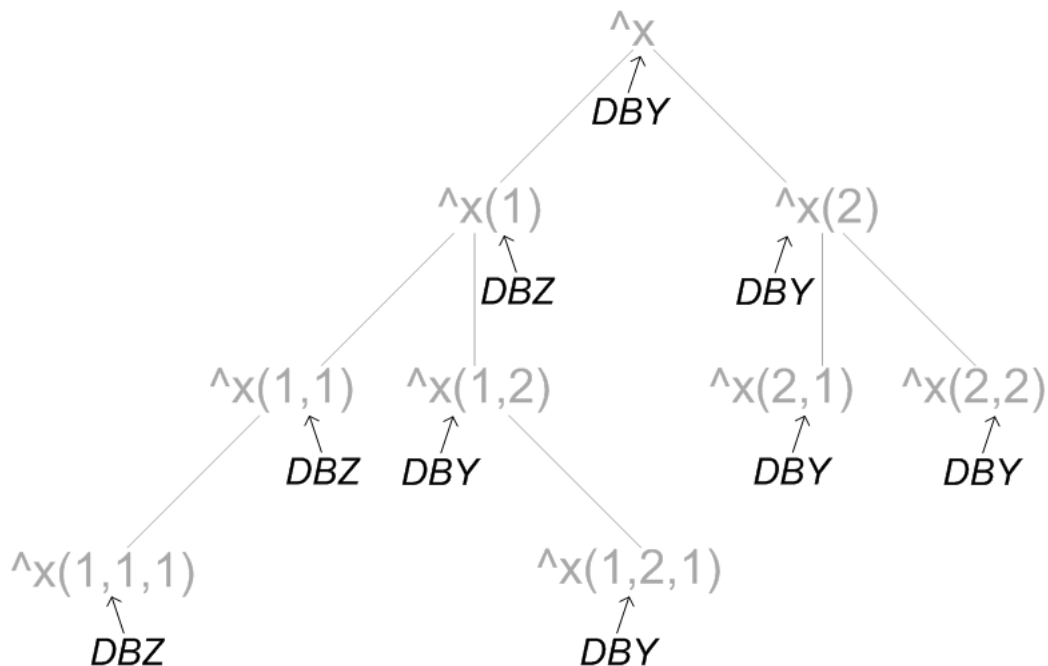
Again, the globals and their hierarchy appear in gray, and the databases to which they are mapped appear in black.

Once you have mapped a global from one namespace to another, you can reference the mapped global as if it were in the current namespace — with a simple reference, such as *^ORDER* or *^X(1)*.

Important: When establishing subscript-level mapping ranges, the behavior of string subscripts differs from that of integer subscripts. For strings, the first character determines the range, while the range for integers uses numeric values. For example, a subscript range of ("A"):("C") contains not only AA but also AC and ABCDEF; by contrast, a subscript range of (1):(2) does not contain 11.

2.4.1.1 Using Distinct Ranges of Globals and Subscripts

Each of a namespace's mappings must refer to distinct ranges of globals or subscripts. Mapping validation prevents the establishment of any kind of overlap. For example, if you attempt to use the Management Portal to create a new mapping that overlaps with an existing mapping, the Portal prevents this from occurring and displays an error message.

2.4.1.2 Logging Changes

Successful changes to the mappings through the Portal are also logged in cconsole.log; unsuccessful changes are not logged. Any failed attempts to establish mappings by hand-editing the Caché parameter (CPF) file are logged in cconsole.log; for details on editing the CPF file, see the section "Editing the Active CPF File" in the "Introduction to the Caché Parameter File" chapter of the *Caché Parameter File Reference*.

2.4.2 Extended Global References

You can refer to a global located in a namespace other than the current namespace. This is known as an *extended global reference* or simply an *extended reference*.

There are two forms of extended references:

- Explicit namespace reference — You specify the name of the namespace where the global is located as part of the syntax of the global reference.

- Implied namespace reference — You specify the directory and, optionally, the system name as part of the syntax of the global reference. In this case, no global mappings apply, since the physical dataset (directory and system) is given as part of the global reference.

The use of explicit namespaces is preferred, because this allows for redefinition of logical mappings externally, as requirements change, without altering your application code.

Caché supports two forms of extended references:

- Bracket syntax, which encloses the extended reference with square brackets ([ ]).

- Environment syntax, which encloses the extended reference with vertical bars (| |).

Note: The examples of extended globals references use the Windows directory structure. In practice, the form of such references is operating-system dependent.

2.4.2.1 Bracket Syntax

You can use bracket syntax to specify an extended global reference with either an explicit namespace or an implied namespace:

Explicit namespace:

```
^[nspace]glob
```

Implied namespace:

```
^[dir,sys]glob
```

In an explicit namespace reference, *nspace* is a defined namespace that the global *glob* has not currently been mapped or replicated to. In an implied namespace reference, *dir* is a directory (the name of which includes a trailing backslash: "\"), *sys* is a system, and *glob* is a global within that directory. If *nspace* or *dir* is specified as a carat ("^"), the reference is to a process-private global.

You must include quotation marks around the directory and system names or the namespace name unless you specify them as variables. The directory and system together comprise an implied namespace. An implied namespace can reference either:

- The specified directory on the specified system.

- The specified directory on your local system, if you do not specify a system name in the reference. If you omit the system name from an implied namespace reference, you must supply a double caret (^^) within the directory reference to indicate the omitted system name.

To specify an implied namespace on a remote system:

```
["dir","sys"]
```

To specify an implied namespace on the local system:

```
["^^dir"]
```

For example, to access the global ORDER in the C:\BUSINESS\ directory on a machine called SALES:

```
SET x = ^["C:\BUSINESS\","SALES"]ORDER
```

To access the global ORDER in the C:\BUSINESS\ directory on your local machine:

```
SET x = ^["^^C:\BUSINESS\"]ORDER
```

To access the global ORDER in the defined namespace MARKETING:

```
SET x = ^["MARKETING"]ORDER
```

To access the process-private global ORDER:

```
SET x = ^["^"]ORDER
```

Note: When creating an implied namespace extended reference involving a mirrored database, you can use its mirrored database path, in the format :mirror:*mirror_name*:*mirror_DB_name*. For example, when referring to the database with the mirror database name mirdb1 in the mirror CORPMIR, you could form an implied reference as follows:

```
["^^:mirror:CORPMIR:mirdb1"]
```

The mirrored database path can be used for both local and remote databases.

2.4.2.2 Environment Syntax

The environment syntax is defined as:

```
^|"env"|global
```

"env" can have one of five formats:

- The null string ("") — The current namespace on the local system.

- `"namespace"` — A defined namespace that *global* is not currently mapped to. Namespace names are not case-sensitive. If *namespace* has the special value of `"^"`, it is a process-private global.

- `"^^dir"` — An implied namespace whose default directory is the specified directory on your local system, where *dir* includes a trailing backslash ("\").

- `"^system^dir"` — An implied namespace whose default directory is the specified directory on the specified remote system, where *dir* includes a trailing backslash ("\").

- omitted — If there is no *"env"* at all, it is a process-private global.

To access the global ORDER in your current namespace on your current system, when no mapping has been defined for ORDER, use the following syntax:

```
SET x = ^|""|ORDER
```

This is the same as the simple global reference:

```
SET x = ^ORDER
```

To access the global ORDER mapped to the defined namespace MARKETING:

```
SET x = ^|"MARKETING"|ORDER
```

You can use an implied namespace to access the global ORDER in the directory C:\BUSINESS\ on your local system:

```
SET x = ^|"^^C:\BUSINESS\"|ORDER
```

You can use an implied namespace to access the global ORDER in the directory C:\BUSINESS on a remote system named SALES:

```
SET x = ^|"^SALES^C:\BUSINESS\"|ORDER
```

To access the process-private global ORDER:

```
SET x = ^||ORDER
SET x=^|"^"|ORDER
```

3

Using Multidimensional Storage (Globals)

This chapter describes the various operations you can perform using multidimensional storage (global variables). It includes the following topics:

3.1 Storing Data in Globals

Storing data in global nodes is simple: you treat a global as you would any other variable. The difference is that operations on globals are automatically written to the database.

3.1.1 Creating Globals

There is no setup work required to create a new global; simply setting data into a global implicitly creates a new global structure. You can create a global (or a global subscript) and place data in it with a single operation, or you can create a global (or subscript) and leave it empty by setting it to the null string. In ObjectScript, these operations are done using the **SET** command.

The following examples define a global named *Color* (if one does not already exist) and associate the value "Red" with it. If a global already exists with the name *Color*, then these examples modify it to contain the new information.

In Caché Basic:

```
^Color = "Red"
```

In ObjectScript:

```
 SET ^Color = "Red"
```

Note: When using direct global access within applications, develop and adhere to a naming convention to keep different parts of an application from "walking over" one another; this is similar to developing naming convention for classes, method, and other variables. Also, avoid certain global names that Caché uses; for a list of these, see the section "Global Variable Names to Avoid" in the "Rules and Guidelines for Identifiers" appendix of the *Caché Programming Orientation Guide*.

3.1.2 Storing Data in Global Nodes

To store a value within a global subscript node, simply set the value of the global node as you would any other variable array. If the specified node did not previously exist, it is created. If it did exist, its contents are replaced with the new value.

You specify a node within a global by means of an expression (referred to as a global reference). A global reference consists of the caret character (^), the global name, and (if needed) one or more subscript values. Subscripts (if present) are enclosed within parentheses "( )" and are separated by commas. Each subscript value is itself an expression: a literal value, a variable, a logical expression, or even a global reference.

Setting the value of a global node is an *atomic* operation: It is guaranteed to succeed and you do not need to use any locks to ensure concurrency.

The following are all valid global references:

In Caché Basic:

```
^Data = 2
^Data("Color") = "Red"
^Data(1,1) = 100
^Data(^Data) = 10      ' The value of ^Data is the subscript
^Data(a,b) = 50        ' The values of local variables a and b are subscripts
^Data(a + 10) = 50
```

In ObjectScript:

```
    SET ^Data = 2
    SET ^Data("Color")="Red"
    SET ^Data(1,1)=100        /* The 2nd-level subscript (1,1) is set
                                 to the value 100. No value is stored at
                                 the 1st-level subscript (^Data(1)). */
    SET ^Data(^Data)=10       /* The value of global variable ^Data
                                 is the name of the subscript. */
    SET ^Data(a,b)=50         /* The values of local variables a and b
                                 are the names of the subscripts. */
    SET ^Data(a+10)=50
```

If you are using ObjectScript, you can construct global references at runtime using indirection.

3.1.3 Storing Structured Data in Global Nodes

Each global node can contain a single string of up to 32K characters.

Data is typically stored within nodes in one of the following ways:

- As a single string of up to 32K characters (specifically, 32K minus 1).

- As a character-delimited string containing multiple pieces of data.

  To store a set of fields within a node using a character delimiter, simply concatenate the values together using the concatenate operator (_). The following ObjectScript examples use the # character as a delimiter:

  ```
  SET ^Data(id)=field(1)_"#"_field(2)_"#"_field(3)
  ```

  When the data is retrieved, you can pull the fields apart using the **$PIECE** function:

  ```
  SET data = $GET(^Data(id))
  FOR i=1:1:3 {
    SET field(i) = $PIECE(data,"#",i)
    }
  QUIT
  ```

- As a **$LIST**-encoded string containing multiple pieces of data.

  The **$LIST** functions use a special length-encoding scheme that does not require you to reserve a delimiter character. (This is the default structure used by Caché objects and SQL.)

  To store a set of fields within a node use the $LISTBUILD function to construct a list:

  ```
  SET ^Data(id)=$LISTBUILD(field(1),field(2),field(3))
  ```

  When the data is retrieved, you can pull the fields apart using the $LIST or $LISTGET functions:

  ```
  SET data = $GET(^Data(id))
  FOR i = 1:1:3 {
      SET field(i)=$LIST(data,i)
      }
  QUIT
  ```

- As one part of a larger set of data (such as a stream or "BLOB").

  As individual nodes are limited to just under 32K of data, larger structures, such as streams, are implemented by storing data in a set of successive nodes:

  ```
  SET ^Data("Stream1",1) = "First part of stream...."
  SET ^Data("Stream1",2) = "Second part of stream...."
  SET ^Data("Stream1",3) = "Third part of stream...."
  ```

  Code that fetches the stream (such as that provided by the %GlobalCharacterStream class) loops over the successive nodes in such a structure providing the data as a continuous string.

- As a bitstring.

  If you are implementing a bitmap index (an index where a bit in a bitstring corresponds to a row in a table), you would set the node values of an index global to bit strings. Note that Caché uses a compression algorithm for encoding bit strings; therefore, bit strings can only be handled using the Caché **$BIT** functions. Refer to the Bit String Functions Overview for more details on bit strings.

- As an empty node.

  If the data you are interested in is provided by the nodes themselves, then it is typical to set the actual subscript to a null string (""). For example, an index that associates a name with an ID value typically looks like this:

  ```
  SET ^Data("APPLE",1) = ""
  SET ^Data("ORANGE",2) = ""
  SET ^Data("BANANA",3) = ""
  ```

3.2 Deleting Global Nodes

To remove a global node, a group of subnodes, or an entire global from the database, use the ObjectScript **KILL** or **ZKILL** commands, or the Caché Basic **Erase** command.

The **KILL** command deletes all nodes (data as well as its corresponding entry in the array) at a specific global reference, including any descendant subnodes. That is, all nodes starting with the specified subscript are deleted.

For example, the ObjectScript statement:

```
KILL ^Data
```

deletes the entire ^Data global. A subsequent reference to this global would return an <UNDEFINED> error.

The ObjectScript statement:

```
KILL ^Data(100)
```

deletes contents of node 100 within the ^Data global. If there are descendant subnodes, such as ^Data(100,1), ^Data(100,2), and ^Data(100,1,2,3), these are deleted as well.

The ObjectScript **ZKILL** command deletes a specified global or global subscript node. It does not delete descendant subnodes.

Note: Following the kill of a large global, the space once occupied by that global may not have been completely freed, since the blocks are marked free in the background by the Garbage Collector daemon. Thus, a call to the **ReturnUnusedSpace** method of the SYS.Database class immediately after killing a large global may not return as much space as expected, since blocks occupied by that global may not have been released as yet.

You cannot use the **NEW** command on global variables.

3.3 Testing the Existence of a Global Node

To test if a specific global (or its descendants) contains data, use the ObjectScript **$DATA** function.

$DATA returns a value indicating whether or not the specified global reference exists. The possible return values are:

Status Value	Meaning
0	The global variable is undefined.
1	The global variable exists and contains data, but has no descendants. Note that the null string ("") qualifies as data.
10	The global variable has descendants (contains a downward pointer to a subnode) but does not itself contain data. Any direct reference to such a variable will result in an <UNDEFINED> error. For example, if $DATA(^y) returns 10, SET x=^y will produce an <UNDEFINED> error.
11	The global variable both contains data and has descendants (contains a downward pointer to a subnode).

3.4 Retrieving the Value of a Global Node

To get the value stored within a specific global node, simply use the global reference as an expression.

Using Caché Basic:

```
color = ^Data("Color")   ' assign to a local variable
Print ^Data("Color")     ' use as an argument to a command
MyMethod(^Data("Color")) ' use as a function argument
```

Using Caché ObjectScript:

```
SET color = ^Data("Color")    ; assign to a local variable
WRITE ^Data("Color")          ; use as a command argument
SET x=$LENGTH(^Data("Color")) ; use as a function parameter
```

3.4.1 The $GET Function

Within ObjectScript, you can also get the value of a global node using the **$GET** function:

```
SET mydata = $GET(^Data("Color"))
```

This retrieves the value of the specified node (if it exists) or returns the null string ("") if the node has no value. You can use the optional second argument of **$GET** to return a specified default value if the node has no value.

3.4.2 The WRITE, ZWRITE, and ZZDUMP Commands

You can display the contents of a global or a global subnode by using the various ObjectScript display commands. The **WRITE** command returns the value of the specified global or subnode as a string. The **ZWRITE** command returns the name of the global variable and its value, and each of its descendant nodes and their values. The **ZZDUMP** command returns the value of the specified global or subnode in hexadecimal dump format.

3.5 Traversing Data within a Global

There are a number of ways to traverse (iterate over) data stored within a global.

3.5.1 The $ORDER (Next / Previous) Function

The ObjectScript **$ORDER** function (and its Caché Basic equivalent: **Traverse**) allows you to sequentially visit each node within a global.

The **$ORDER** function returns the value of the next subscript at a given level (subscript number). For example, suppose you have defined the following global:

```
Set ^Data(1) = ""
Set ^Data(1,1) = ""
Set ^Data(1,2) = ""
Set ^Data(2) = ""
Set ^Data(2,1) = ""
Set ^Data(2,2) = ""
Set ^Data(5,1,2) = ""
```

To find the first, first-level subscript, we can use:

```
SET key = $ORDER(^Data(""))
```

This returns the first, first-level subscript following the null string (""). (The null string is used to represent the subscript value *before* the first entry; as a return value it is used to indicate that there are no following subscript values.) In this example, *key* will now contain the value 1.

We can find the next, first-level subscript by using 1 or *key* in the **$ORDER** expression:

```
SET key = $ORDER(^Data(key))
```

If *key* has an initial value of 1, then this statement will set it to 2 (as ^Data(2) is the next first-level subscript). Executing this statement again will set *key* to 5 as that is the next first-level subscript. Note that 5 is returned even though there is no data stored directly at ^Data(5). Executing this statement one more time will set *key* to the null string (""), indicating that there are no more first level subscripts.

By using additional subscripts with the **$ORDER** function, you can iterate over different subscript levels. **$ORDER** returns the next value of the last subscript in its argument list. Using the data above, the statement:

```
SET key = $ORDER(^Data(1,""))
```

will set *key* to 1 as ^Data(1,1) is the next second-level subscript. Executing this statement again will set *key* to 2 as that is the next second-level subscript. Executing this statement one more time will set *key* to " " indicating that there are no more second-level subscripts under node ^Data(1).

3.5.1.1 Looping with $ORDER

The following ObjectScript code defines a simple global and then loops over all of its first-level subscripts:

```
// clear ^Data in case it has data
Kill ^Data

// fill in ^Data with sample data
For i = 1:1:100 {
    // Set each node to a random person's name
    Set ^Data(i) = ##class(%PopulateUtils).Name()
}

// loop over every node
// Find first node
Set key = $Order(^Data(""))

While (key '= "") {
    // Write out contents
    Write "#", key, " ", ^Data(key),!

    // Find next node
    Set key = $Order(^Data(key))
}
```

3.5.1.2 Additional $ORDER Arguments

The ObjectScript **$ORDER** function takes optional second and third arguments. The second argument is a direction flag indicating in which direction you wish to traverse a global. The default, 1, specifies forward traversal, while –1 specifies backward traversal.

The third argument, if present, contains a local variable name. If the node found by **$ORDER** contains data, the data found is written into this local variable. When you are looping over a global and you are interested in node values as well as subscript values, this operates more efficiently.

3.5.2 Looping Over a Global

If you know that a given global is organized using contiguous numeric subscripts, you can use a simple For loop to iterate over its values. For example, in Caché Basic:

```
For i = 1 To 100
    Print ^Data(i)
Next
```

or the equivalent in ObjectScript:

```
For i = 1:1:100 {
    Write ^Data(i),!
}
```

Generally, it is better to use the **$ORDER** function described above: it is more efficient and you do not have to worry about gaps in the data (such as a deleted node).

3.5.3 The $QUERY Function

If you need to visit every node and subnode within a global, moving up and down over subnodes, use the ObjectScript $QUERY function. (Alternatively you can use nested **$ORDER** loops).

The **$QUERY** function takes a global reference and returns a string containing the global reference of the next node in the global (or "" if there are no following nodes). To use the value returned by **$QUERY**, you must use the ObjectScript indirection operator (@).

For example, suppose you define the following global:

```
Set ^Data(1) = ""
Set ^Data(1,1) = ""
Set ^Data(1,2) = ""
Set ^Data(2) = ""
Set ^Data(2,1) = ""
Set ^Data(2,2) = ""
Set ^Data(5,1,2) = ""
```

The following call to **$QUERY**:

```
SET node = $QUERY(^Data(""))
```

sets *node* to the string "^Data(1)", the address of the first node within the global. Then, to get the next node in the global, call **$QUERY** again and use the indirection operator on *node*:

```
SET node = $QUERY(@node)
```

At this point, *node* contains the string "^Data(1,1)".

The following example defines a set of global nodes and then walks over them using **$QUERY**, writing the address of each node as it does:

```
Kill ^Data // make sure ^Data is empty

// place some data into ^Data
Set ^Data(1) = ""
Set ^Data(1,1) = ""
Set ^Data(1,2) = ""
Set ^Data(2) = ""
Set ^Data(2,1) = ""
Set ^Data(2,2) = ""
Set ^Data(5,1,2) = ""

// now walk over ^Data
// find first node
Set node = $Query(^Data(""))
While (node '= "") {
    Write node,!
    // get next node
    Set node = $Query(@node)
}
```

3.6 Copying Data within Globals

To copy the contents of a global (entire or partial) into another global (or a local array), use the ObjectScript **MERGE** command.

The following example demonstrates the use of the **MERGE** command to copy the entire contents of the *OldData* global into the *NewData* global:

```
Merge ^NewData = ^OldData
```

If the source argument of the **MERGE** command has subscripts then all data in that node and its descendants are copied. If the destination argument has subscripts, then the data is copied using the destination address as the top level node. For example, the following code:

```
Merge ^NewData(1,2) = ^OldData(5,6,7)
```

copies all the data at and beneath *^OldData(5,6,7)* into *^NewData(1,2)*.

3.7 Maintaining Shared Counters within Globals

A major concurrency bottleneck of large-scale transaction processing applications can be the creation of unique identifier values. For example, consider an order processing application in which each new invoice must be given a unique identifying number. The traditional approach is to maintain some sort of counter table. Every process creating a new invoice waits to acquire a lock on this counter, increments its value, and unlocks it. This can lead to heavy resource contention over this single record.

To deal with this issue, Caché provides the ObjectScript **$INCREMENT** function. **$INCREMENT** atomically increments the value of a global node (if the node has no value, it is set to 1). The atomic nature of **$INCREMENT** means that no locks are required; the function is guaranteed to return a new incremented value with no interference from any other process.

You can use **$INCREMENT** as follows. First, you must decide upon a global node in which to hold the counter. Next, whenever you need a new counter value, simply invoke **$INCREMENT**:

```
SET counter = $INCREMENT(^MyCounter)
```

The default storage structure used by Caché objects and SQL uses **$INCREMENT** to assign unique object (row) identifier values.

3.8 Using Temporary Globals

For certain operations, you may need the power of globals without requiring persistence. For example, you may want to use a global to sort some data which you do not need to store to disk. For these operations, Caché provides temporary globals.

Temporary globals have the following characteristics:

- Temporary globals are stored within the CACHETEMP database, which is always defined to be a local (that is, a non-network) database. All globals mapped to the CACHETEMP database are treated as temporary globals.

- Changes to temporary globals are not written to disk. Instead the changes are maintained within the in-memory buffer pool. A large temporary global may be written to disk if there is not sufficient space for it within the buffer pool.

- For maximum efficiency, changes to temporary globals are not logged to a journal file.

- Temporary globals are automatically deleted whenever the Caché system is restarted. (Note: it can be a very long time before a live system is restarted; so you should not count on this for cleaning up temporary globals.)

By default, Caché defines any global whose name starts with "CacheTemp" as being a temporary global. To avoid conflict with any temporary globals that Caché itself may use, you should start your temporary global names with "CacheTempUser".

Caché SQL uses temporary globals as scratch space for optimizing complex queries. It may also uses temporary globals as temporary indices during the execution of certain queries (for sorting, grouping, calculating aggregates, etc.)

3.9 Sorting Data within Globals

Data stored within globals is automatically sorted according to the value of the subscripts. For example, the following ObjectScript code defines a set of globals (in random order) and then iterates over them to demonstrate that the global nodes are automatically sorted by subscript:

```
// Erase any existing data
Kill ^Data

// Define a set of global nodes
Set ^Data("Cambridge") = ""
Set ^Data("New York") = ""
Set ^Data("Boston") = ""
Set ^Data("London") = ""
Set ^Data("Athens") = ""

// Now iterate and display (in order)
Set key = $Order(^Data(""))
While (key '= "") {
    Write key,!
    Set key = $Order(^Data(key)) // next subscript
}
```

Applications can take advantage of the automatic sorting provided by globals to perform sort operations or to maintain ordered, cross-referenced indices on certain values. Caché SQL and ObjectScript use globals to perform such tasks automatically.

3.9.1 Collation of Global Nodes

The order in which the nodes of a global are sorted (referred to as collation) is controlled at two levels: within the global itself and by the application using the global.

At the application level, you can control how global nodes are collated by performing data transformations on the values used as subscripts (Caché SQL and objects do this via user-specified collation functions). For example, if you wish to create a list of names that is sorted alphabetically but ignores case, then typically you use the uppercase version of the name as a subscript:

```
// Erase any existing data
Kill ^Data

// Define a set of global nodes for sorting
For name = "Cobra","jackal","zebra","AARDVark" {
    // use UPPERCASE name as subscript
    Set ^Data($ZCONVERT(name,"U")) = name
}

// Now iterate and display (in order)
Set key = $Order(^Data(""))
While (key '= "") {
    Write ^Data(key),!  // write untransformed name
    Set key = $Order(^Data(key)) // next subscript
}
```

This example converts each name to uppercase (using the $ZCONVERT function) so that the subscripts are sorted without regard to case. Each node contains the untransformed value so that the original value can be displayed.

3.9.2 Numeric and String-Valued Subscripts

Numeric values are collated *before* string values; that is a value of 1 comes before a value of "a". You need to be aware of this fact if you use both numeric and string values for a given subscript. If you are using a global for an index (that is, to sort data based on values), it is most common to either sort values as numbers (such as salaries) or strings (such as postal codes).

For numerically collated nodes, the typical solution is to coerce subscript values to numeric values using the unary + operator. For example, if you are building an index that sort *id* values by *age*, you can coerce *age* to always be numeric:

```
Set ^Data(+age,id) = ""
```

If you wish to sort values as strings (such as "0022", "0342", "1584") then you can coerce the subscript values to always be strings by prepending a space (" ") character. For example, if you are building an index that sort *id* values by *zipcode*, you can coerce *zipcode* to always be a string:

```
Set ^Data(" "_zipcode,id) = ""
```

This ensures that values with leading zeroes, such as "0022" are always treated as strings.

3.9.3 The $SORTBEGIN and $SORTEND Functions

Typically you do not have to worry about sorting data within Caché. Whether you use SQL or direct global access, sorting is handled automatically.

There are, however, certain cases where sorting can be done more efficiently. Specifically, in cases where (1) you need to set a large number of global nodes that are in random (that is, unsorted) order and (2) the total size of the resulting global approaches a significant portion of the Caché buffer pool, then performance can be adversely affected — since many of the **SET** operations involve disk operations (as data does not fit in the cache). This scenario usually arises in cases involving the creation of index globals such as bulk data loads, index population, or sorting of unindexed values in temporary globals.

To handle these cases efficiently, ObjectScript provides the $SORTBEGIN and $SORTEND functions. The **$SORTBEGIN** function initiates a special mode for a global (or part thereof) in which data set into the global is written to a special scratch buffer and sorted in memory (or temporary disk storage). When the **$SORTEND** function is called at the end of the operation, the data is written to actual global storage sequentially. The overall operation is much more efficient as the actual writing is done in an order requiring far fewer disk operations.

The **$SORTBEGIN** function is quite easy to use; simply invoke it with the name of the global you wish to sort before beginning the sort operation and call **$SORTEND** when the operation is complete:

```
// Erase any existing data
Kill ^Data

// Initiate sort mode for ^Data global
Set ret = $SortBegin(^Data)

// Write random data into ^Data
For i = 1:1:10000 {
    Set ^Data($Random(1000000)) = ""
}

Set ret = $SortEnd(^Data)

// ^Data is now set and sorted

// Now iterate and display (in order)
Set key = $Order(^Data(""))
While (key '= "") {
    Write key,!
    Set key = $Order(^Data(key)) // next subscript
}
```

The **$SORTBEGIN** function is designed for the special case of global creation and must be used with some care. Specifically, you must not *read* from the global to which you are writing while in **$SORTBEGIN** mode; as the data is not written, reads will be incorrect.

Caché SQL automatically uses these functions for creation of temporary index globals (such as for sorting on unindexed fields).

3.10 Using Indirection with Globals

By means of indirection, ObjectScript provides a way to create global references at runtime. This can be useful in applications where you do not know global structure or names at program compilation time.

Indirection is supported via the indirection operator, @, which de-references a string containing an expression. There are several types of indirection, based on how the @ operator is used.

The following code provides an example of name indirection in which the @ operator is used to de-reference a string containing a global reference:

```
// Erase any existing data
Kill ^Data

// Set var to an global reference expression
Set var = "^Data(100)"

// Now use indirection to set ^Data(100)
Set @var = "This data was set indirectly."

// Now display the value directly:
Write "Value: ",^Data(100)
```

You can also use subscript indirection to mix expressions (variables or literal values) within indirect statements:

```
// Erase any existing data
Kill ^Data

// Set var to a subscript value
Set glvn = "^Data"

// Now use indirection to set ^Data(1) to ^Data(10)
For i = 1:1:10 {
    Set @glvn@(i) = "This data was set indirectly."
}

// Now display the values directly:
Set key = $Order(^Data(""))
While (key '= "") {
    Write "Value ",key, ": ", ^Data(key),!
    Set key = $Order(^Data(key))
}
```

Indirection is a fundamental feature of ObjectScript; it is not limited to global references. For more information, refer to the "Indirection" section in the "Operators" chapter of *Using Caché ObjectScript*. Indirection is less efficient than direct access, so you should use it judiciously.

3.11 Managing Transactions

Caché provides the primitive operations needed to implement full transaction processing using globals. Caché objects and SQL make use of these features automatically. If you are directly writing transactional data into globals, you can make use of these operations.

The transaction commands are **TSTART**, which defines the start of a transaction; **TCOMMIT**, which commits the current transaction; and **TROLLBACK**, which aborts the current transaction and undoes any changes made to globals since the start of the transaction.

For example, the following ObjectScript code defines the start of a transaction, sets a number of global nodes, and then commits or rolls back the transaction depending on the value of *ok*:

```
TSTART

Set ^Data(1) = "Apple"
Set ^Data(2) = "Berry"

If (ok) {
    TCOMMIT
}
Else {
    TROLLBACK
}
```

The **TSTART** writes a transaction start marker in the Caché journal file. This defines the starting boundary of the transaction. If the variable *ok* is true (nonzero) in the above example, then the **TCOMMIT** command marks the successful end of the transaction and a transaction completion marker is written to the journal file. If *ok* is false (0), then the **TROLLBACK** command will undo every set or kill operation made since the start of the transaction. In this case, *^Data(1)* and *^Data(2)* are restored to their previous values.

Note that no data is written at the successful completion of a transaction. This is because all modifications to the database during a transaction are carried out as normal during the course of a transaction. Only in the case of a rollback is the data in the database affected. This implies that the transaction in this example has limited *isolation*; that is, other processes can see the modified global values before the transaction is committed. This is typically referred to as an uncommitted read. Whether this is good or bad depends on application requirements; in many cases this is perfectly reasonable behavior. If an application requires a higher degree of isolation, then this is accomplished by using locks. This is described in the following section.

3.11.1 Locks and Transactions

To create isolated transactions—that is, to prevent other processes from seeing modified data before a transaction is committed—requires the use of locks. Within ObjectScript, you can directly acquire and release locks by means of the **LOCK** command. Locks work by convention; for a given data structure (such as used for a persistent object) all code that requires locks uses the same logical lock reference (that is, the same address is used by the **LOCK** command).

Within a transaction, locks have a special behavior; any locks acquired during the course of a transaction are not released until the end of the transaction. To see why this is, consider the actions carried out by typical transaction:

1. Start the transaction using **TSTART**.

2. Acquire a lock (or locks) on the node (or nodes) you wish to modify. This is usually referred to as a "write" lock.

3. Modify the node (or nodes).

4. Release the lock (or locks). Because we are in a transaction, these locks are not actually released at this time.

5. Commit the transaction using **TCOMMIT**. At this point, all the locks released in the previous step are actually released.

If another process wants to look at the nodes involved in this transaction and does not want to see uncommitted modifications, then it simply tests for a lock (referred to a "read" lock) before reading the data from the nodes. Because the write locks are held until the end of the transaction, the reading process does not see the data until the transaction is complete (committed or rolled back).

Most database management systems use a similar mechanism to provide transaction isolation. Caché is unique in that it makes this mechanism available to developers. This makes it possible to create custom database structure for new application

types while still supporting transactions. Of course, you can simply use Caché objects or SQL to manage your data and let your transactions be managed automatically.

3.11.2 Nested Calls to TSTART

Caché maintains a special system variable, $TLEVEL, that tracks how many times the **TSTART** command has been called. *$TLEVEL* starts with a value of 0; each call to **TSTART** increments the value of *$TLEVEL* by 1, while each call to **TCOMMIT** decrements its value by 1. If a call to **TCOMMIT** results in setting *$TLEVEL* back to 0, the transaction ends (with a commit).

A call to the **TROLLBACK** command always terminates the current transaction and sets *$TLEVEL* back to 0, regardless of the value of *$TLEVEL*.

This behavior gives applications the ability to wrap transactions around code (such as object methods) that itself contains a transaction. For example, the **%Save** method, provided by persistent objects, always performs its operation as a transaction. By explicitly calling **TSTART** and **TCOMMIT** you can create a larger transaction that encompasses several object save operations:

```
TSTART
Set sc = object1.%Save()
If ($$$ISOK(sc)) {
    // first save worked, do the second
    Set sc = object2.%Save()
}

If ($$$ISERR(sc)) {
    // one of the saves failed, rollback
    TROLLBACK
}
Else {
    // everything is ok, commit
 TCOMMIT
}
```

3.12 Managing Concurrency

The operation of setting or retrieving a single global node is atomic; it is guaranteed to always succeed with consistent results. For operations on multiple nodes or for controlling transaction isolation (see the section on Lock and Transactions), Caché provides the ability to acquire and release locks.

Locks are managed by the Caché Lock Manager. Within ObjectScript, you can directly acquire and release locks by means of the **LOCK** command. (Caché objects and SQL automatically acquire and release locks as needed).

For details on the **LOCK** command, refer to the **LOCK** command reference page.

3.13 Checking the Most Recent Global Reference

The most recent global reference is recorded in the ObjectScript **$ZREFERENCE** special variable. **$ZREFERENCE** contains the most recent global reference, including subscripts and extended global reference, if specified. Note that **$ZREFERENCE** indicates neither whether the global reference succeeded, nor if the specified global exists. Caché simply records the most recently specified global reference.

3.13.1 Naked Global Reference

Following a subscripted global reference, Caché sets a *naked indicator* to that global name and subscript level. You can then make subsequent references to the same global and subscript level using a *naked global reference*, omitting the global name and higher level subscripts. This streamlines repeated references to the same global at the same (or lower) subscript level.

Specifying a lower subscript level in a naked reference resets the naked indicator to that subscript level. Therefore, when using naked global references, you are always working at the subscript level established by the most recent global reference.

The naked indicator value is recorded in the **$ZREFERENCE** special variable. The naked indicator is initialized to the null string. Attempting a naked global reference when the naked indicator is not set results in a <NAKED> error. Changing namespaces reinitializes the naked indicator. You can reinitialize the naked indicator by setting **$ZREFERENCE** to the null string ("").

In the following example, the subscripted global ^Produce("fruit",1) is specified in the first reference. Caché saves this global name and subscript in the naked indicator, so that the subsequent naked global references can omit the global name "Produce" and the higher subscript level "fruit". When the ^(3,1) naked reference goes to a lower subscript level, this new subscript level becomes the assumption for any subsequent naked global references.

```
SET ^Produce("fruit",1)="Apples"   /* Full global reference  */
SET ^(2)="Oranges"                 /* Naked global references */
SET ^(3)="Pears"                   /* assume subscript level 2 */
SET ^(3,1)="Bartlett pears"        /* Go to subscript level 3  */
SET ^(2)="Anjou pears"             /* Assume subscript level 3 */
WRITE "latest global reference is: ",$ZREFERENCE,!
ZWRITE ^Produce
KILL ^Produce
```

This example sets the following global variables: ^Produce("fruit",1), ^Produce("fruit",2), ^Produce("fruit",3), ^Produce("fruit",3,1), and ^Produce("fruit",3,2).

With few exceptions, every global reference (full or naked) sets the naked indicator. The **$ZREFERENCE** special variable contains the full global name and subscripts of the most recent global reference, even if this was a naked global reference. The **ZWRITE** command also displays the full global name and subscripts of each global, whether or not it was set using a naked reference.

Naked global references should be used with caution, because Caché sets the naked indicator in situations that are not always obvious, including the following:

- A full global reference initially sets the naked indicator, and subsequent full global references or naked global references change the naked indicator, even when the global reference is not successful. For example attempting to **WRITE** the value of a nonexistent global sets the naked indicator.

- A command postconditional that references a subscripted global sets the naked indicator, regardless of how Caché evaluates the postconditional.

- An optional function argument that references a subscripted global may or may not set the naked indicator, depending on whether Caché evaluates all arguments. For example the second argument of **$GET** always sets the naked indicator, even when the default value it contains is not used. Caché evaluates arguments in left-to-right sequence, so the last argument may reset the naked indicator set by the first argument.

- The **TROLLBACK** command, which rolls back a transaction, does not roll back the naked indicator to its value at the beginning of the transaction.

If a full global reference contains an extended global reference, subsequent naked global references assume the same extended global reference; you do not have to specify the extended reference as part of a naked global reference.

4

SQL and Object Use of Multidimensional Storage

This chapter describes how the Caché object and SQL engines make use of multidimensional storage (globals) for storing persistent objects, relational tables, and indices.

Though the Caché object and SQL engines automatically provide and manage data storage structures, it can be useful to understand the details of how this works.

The storage structures used by the object and relational view of data are identical. For simplicity, this document only describes storage from the object perspective.

4.1 Data

Every persistent class that uses the %CacheStorage storage class (the default) can store instances of itself within the Caché database using one or more nodes of multidimensional storage (globals).

Every persistent class has a storage definition that defines how its properties are stored within global nodes. This storage definition (referred to as "default structure") is managed automatically by the class compiler. (You can modify this storage definition or even provide alternate versions of it if you like. This is not discussed in this document.)

4.1.1 Default Structure

The default structure used for storing persistent objects is quite simple:

- Data is stored in a global whose name starts with the complete class name (including package name). A "D" is appended to form the name of the data global, while an "I" is appended for the index global.

- Data for each instance is stored within a single node of the data global with all non-transient properties placed within a **$List** structure.

- Each node in the data global is subscripted by object ID value. By default, object ID values are integers provided by invoking the **$Increment** function on a counter node stored at the root (with no subscript) of the data global.

For example, suppose we define a simple persistent class, MyApp.Person, with two literal properties:

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
}
```

If we create and save two instances of this class, the resulting global will be similar to:

```
^MyApp.PersonD = 2  // counter node
^MyApp.PersonD(1) = $LB("",530,"Abraham")
^MyApp.PersonD(2) = $LB("",680,"Philip")
```

Note that the first piece of the **$List** structure stored in each node is empty; this is reserved for a class name. If we define any subclasses of this Person class, this slot contains the subclass name. The **%OpenId** method (provided by the %Persistent class) uses this information to polymorphically open the correct type of object when multiple objects are stored within the same extent. This slot shows up in the class storage definition as a property named "%%CLASSNAME".

For more details, refer to the section on subclasses below.

4.1.2 IDKEY

The IDKEY mechanism allows you to explicitly define the value used as an object ID. To do this, you simply add an IDKEY index definition to your class and specify the property or properties that will provide the ID value. Note that once you save an object, its object ID value cannot change. This means that after you save an object that uses the IDKEY mechanism, you can no longer modify any of the properties on which the object ID is based.

For example, we can modify the Person class used in the previous example to use an IDKEY index:

```
Class MyApp.Person Extends %Persistent
{
Index IDKEY On Name [ Idkey ];

Property Name As %String;
Property Age As %Integer;
}
```

If we create and save two instances of the Person class, the resulting global is now similar to:

```
^MyApp.PersonD("Abraham") = $LB("",530,"Abraham")
^MyApp.PersonD("Philip") = $LB("",680,"Philip")
```

Note that there is no longer any counter node defined. Also note that by basing the object ID on the Name property, we have implied that the value of Name must be unique for each object.

If the IDKEY index is based on multiple properties, then the main data nodes has multiple subscripts. For example:

```
Class MyApp.Person Extends %Persistent
{
Index IDKEY On (Name,Age) [ Idkey ];

Property Name As %String;
Property Age As %Integer;
}
```

In this case, the resulting global will now be similar to:

```
^MyApp.PersonD("Abraham",530) = $LB("",530,"Abraham")
^MyApp.PersonD("Philip",680) = $LB("",680,"Philip")
```

Important: There must not be a sequential pair of vertical bars (||) within the values of any property used by an IDKEY index, unless that property is a valid reference to an instance of a persistent class. This restriction is imposed by the way in which the Caché SQL mechanism works. The use of || in IDKey properties can result in unpredictable behavior.

4.1.3 Subclasses

By default, any fields introduced by a subclass of a persistent object are stored in an additional node. The name of the subclass is used as an additional subscript value.

For example, suppose we define a simple persistent MyApp.Person class with two literal properties:

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
}
```

Now we define a persistent subclass, MyApp.Student, that introduces two additional literal properties:

```
Class MyApp.Student Extends Person
{
Property Major As %String;
Property GPA As %Float;
}
```

If we create and save two instances of this MyApp.Student class, the resulting global will be similar to:

```
^MyApp.PersonD = 2  // counter node
^MyApp.PersonD(1) = $LB("Student",19,"Jack")
^MyApp.PersonD(1,"Student") = $LB(3.2,"Physics")

^MyApp.PersonD(2) = $LB("Student",20,"Jill")
^MyApp.PersonD(2,"Student") = $LB(3.8,"Chemistry")
```

The properties inherited from the Person class are stored in the main node, and those introduced by the Student class are stored in an additional subnode. This structure ensures that the Student data can be used interchangeably as Person data. For example, an SQL query listing names of all Person objects correctly picks up both Person and Student data. This structure also makes it easier for the Class Compiler to maintain data compatibility as properties are added to either the super- or subclasses.

Note that the first piece of the main node contains the string "Student" — this identifies nodes containing Student data.

4.1.4 Parent-Child Relationships

Within parent-child relationships, instances of child objects are stored as subnodes of the parent object to which they belong. This structure ensures that child instance data is physically clustered along with parent data.

For example, here is the definition for two related classes, Invoice:

```
/// An Invoice class
Class MyApp.Invoice Extends %Persistent
{
Property CustomerName As %String;

/// an Invoice has CHILDREN that are LineItems
Relationship Items As LineItem  [inverse = TheInvoice, cardinality = CHILDREN];
}
```

and LineItem:

```
/// A LineItem class
Class MyApp.LineItem Extends %Persistent
{
Property Product As %String;
Property Quantity As %Integer;

/// a LineItem has a PARENT that is an Invoice
Relationship TheInvoice As Invoice [inverse = Items, cardinality = PARENT];
}
```

If we store several instances of Invoice object, each with associated LineItem objects, the resulting global will be similar to:

```
^MyApp.InvoiceD = 2  // invoice counter node
^MyApp.InvoiceD(1) = $LB("","Wiley Coyote")
^MyApp.InvoiceD(1,"Items",1) = $LB("","Rocket Roller Skates",2)
^MyApp.InvoiceD(1,"Items",2) = $LB("","Acme Magnet",1)

^MyApp.InvoiceD(2) = $LB("","Road Runner")
^MyApp.InvoiceD(2,"Items",1) = $LB("","Birdseed",30)
```

For more information on relationships, refer to the "Relationships" chapter in the *Using Caché Objects.*

4.1.5 Embedded Objects

Embedded objects are stored by first converting them to a serialized state (by default a **$List** structure containing the object's properties) and then storing this serial state in the same way as any other property.

For example, suppose we define a simple serial (embeddable) class with two literal properties:

```
Class MyApp.MyAddress Extends %SerialObject
{
Property City As %String;
Property State As %String;
}
```

We now modify our earlier example to add an embedded Home address property:

```
Class MyApp.MyClass Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
Property Home As MyAddress;
}
```

If we create and save two instances of this class, the resulting global is equivalent to:

```
 ^MyApp.MyClassD = 2   // counter node
 ^MyApp.MyClassD(1) = $LB(530,"Abraham",$LB("UR","Mesopotamia"))
 ^MyApp.MyClassD(2) = $LB(680,"Philip",$LB("Bethsaida","Israel"))
```

4.1.6 Streams

Global streams are stored within globals by splitting their data into a series of chunks, each smaller than 32K bytes, and writing the chunks into a series of sequential nodes. File streams are stored in external files.

4.2 Indices

Persistent classes can define one or more indices; additional data structures are used to make operations (such as sorting or conditional searches) more efficient. Caché SQL makes use of such indices when executing queries. Caché Object and SQL automatically maintain the correct values within indices as insert, update, and delete operations are carried out.

4.2.1 Storage Structure of Standard Indices

A standard index associates an ordered set of one or more property values with the object ID values of the object containing the properties.

For example, suppose we define a simple persistent MyApp.Person class with two literal properties and an index on its Name property:

```
Class MyApp.Person Extends %Persistent
{
Index NameIdx On Name;

Property Name As %String;
Property Age As %Integer;
}
```

If we create and save several instances of this Person class, the resulting data and index globals is similar to:

```
// data global
^MyApp.PersonD = 3   // counter node
^MyApp.PersonD(1) = $LB("",34,"Jones")
^MyApp.PersonD(2) = $LB("",22,"Smith")
^MyApp.PersonD(3) = $LB("",45,"Jones")


// index global
^MyApp.PersonI("NameIdx"," JONES",1) = ""
^MyApp.PersonI("NameIdx"," JONES",3) = ""
^MyApp.PersonI("NameIdx"," SMITH",2) = ""
```

Note the following things about the index global:

1. By default, it is placed in a global whose name is the class name with an "I" (for Index) appended to it.

2. By default, the first subscript is the index name; this allows multiple indices to be stored in the same global without conflict.

3. The second subscript contains the *collated* data value. In this case, the data is collated using the default SQLUPPER collation function. This converts all characters to uppercase (to sort without regard to case) and prepends a space character (to force all data to collate as strings).

4. The third subscript contains the Object ID value of the object that contains the indexed data value.

5. The nodes themselves are empty; all the needed data is held within the subscripts. Note that if an index definition specifies that data should be stored along with the index, it is placed in the nodes of the index global.

This index contains enough information to satisfy a number of queries, such as listing all Person class order by Name.

4.3 Bitmap Indices

A bitmap index is similar to a standard index except that it uses a series of bitstrings to store the set of object ID values that correspond to the indexed value.

4.3.1 Logical Operation of Bitmap Indices

A bitstring is a string containing a set of bits (0 and 1 values) in a special compressed format. Caché includes a set of functions to efficiently create and work with bitstrings. These are listed in the following table:

Table 4–1: BitString Operations

Function	Description
$Bit	Set or get a bit within a bitstring.
$BitCount	Count the number of bits within a bitstring.
$BitFind	Find the next occurrence of a bit within a bitstring.
$BitLogic	Perform logical (AND, OR) operations on two or more bitstrings.

Within a bitmap index, ordinal positions within a bitstring correspond to rows (Object ID number) within the indexed table. For a given value, a bitmap index maintains a bitstring that contains 1 for each row in which the given value is present, and contains 0 for every row in which it is absent. Note that bitmap indices only work for objects that use the default storage structure with system-assigned, numeric Object ID values.

For example, suppose we have a table similar to the following:

ID	State	Product
1	MA	Hat
2	NY	Hat
3	NY	Chair
4	MA	Chair
5	MA	Hat

If the State and Product columns have bitmap indices, then they contain the following values:

A bitmap index on the State column contains the following bitstring values:

MA	1	0	0	1	1
NY	0	1	1	0	0

Note that for the value, "MA", there is a 1 in the positions (1, 4, and 5) that correspond to the table rows with State equal to "MA".

Similarly, a bitmap index on the Product column contains the following bitstring values (note that the values are collated to uppercase within the index):

CHAIR	0	0	1	1	0
HAT	1	1	0	0	1

The Caché SQL Engine can execute a number of operations by iterating over, counting the bits within, or performing logical combinations (AND, OR) on the bitstrings maintained by these indices. For example, to find all rows that have State equal to "MA" and Product equal to "HAT", the SQL Engine can simply combine the appropriate bitstrings together with logical AND.

In addition to these indices, the system maintains an additional index, called an "extent index," that contains a 1 for every row that exists and a 0 for rows that do not (such as deleted rows). This is used for certain operations, such as negation.

4.3.2 Storage Structure of Bitmap Indices

A bitmap index associates an ordered set of one or more property values with one or more bitstrings containing the Object ID values corresponding to the property values.

For example, suppose we define a simple persistent MyApp.Person class with two literal properties and a bitmap index on its Age property:

```
Class MyApp.Person Extends %Persistent
{
Index AgeIdx On Age [Type = bitmap];

Property Name As %String;
Property Age As %Integer;
}
```

If we create and save several instances of this Person class, the resulting data and index globals is similar to:

```
 // data global
^MyApp.PersonD = 3   // counter node
^MyApp.PersonD(1) = $LB("",34,"Jones")
^MyApp.PersonD(2) = $LB("",34,"Smith")
^MyApp.PersonD(3) = $LB("",45,"Jones")

 // index global
^MyApp.PersonI("AgeIdx",34,1) = 110...
^MyApp.PersonI("AgeIdx",45,1) = 001...

 // extent index global
^MyApp.PersonI("$Person",1) = 111...
^MyApp.PersonI("$Person",2) = 111...
```

Note the following things about the index global:

1. By default, it is placed in a global whose name is the class name with an "I" (for Index) appended to it.

2. By default, the first subscript is the index name; this allows multiple indices to be stored in the same global without conflict.

3. The second subscript contains the *collated* data value. In this case, a collation function is not applied as this is an index on numeric data.

4. The third subscript contains a *chunk* number; for efficiency, bitmap indices are divided into a series of bitstrings each containing information for about 64000 rows from the table. Each of these bitstrings are referred to as a chunk.

5. The nodes contain the bitstrings.

Also note: because this table has a bitmap index, an extent index is automatically maintained. This extent index is stored within the index global and uses the class name, with a "$" character prepended to it, as its first subscript.

4.3.3 Direct Access of Bitmap Indices

The following example uses a class extent index to compute the total number of stored object instances (rows). Note that it uses **$Order** to iterate over the chunks of the extent index (each chunk contains information for about 64000 rows):

```
/// Return the number of objects for this class.<BR>
/// Equivalent to SELECT COUNT(*) FROM Person
ClassMethod Count() As %Integer
{
    New total,chunk,data
    Set total = 0

    Set chunk = $Order(^MyApp.PersonI("$Person",""),1,data)
    While (chunk '= "") {
        Set total = total + $bitcount(data,1)
        Set chunk = $Order(^MyApp.PersonI("$Person",chunk),1,data)
    }

    Quit total
}
```

5

Managing Globals

The Management Portal provides tools for managing globals, and the system classes provide methods to perform some of the same tasks. This chapter describes how to use these tools. It includes the following topics:

- General Advice

- Introduction to the Globals Page

- Viewing Globals

- Editing Globals

- Finding Values in Globals

- Exporting Globals

- Importing Globals

- Deleting Globals

- APIs for Management Tasks

For information on defining global mappings, see the chapter "Configuring Caché" in the *Caché System Administration Guide*.

5.1 General Advice

As with the ObjectScript commands **SET**, **MERGE**, **KILL**, and others, the tools described here provide direct access to manipulate globals. If you delete or modify via global access, you bypass all object and SQL integrity checking and there is no undo option. It is therefore important to be very careful when doing these tasks. (Viewing and exporting do not affect the database and are safe activities.)

When using the tools described in this chapter, make sure of the following:

- Be sure that you know which globals Caché uses. Not all of these are treated as "system" globals — that is, some of them are visible even when you do not select the **System** check box. Some of these globals store code, including your code.

  See the section "Global Variable Names to Avoid" in the "Rules and Guidelines for Identifiers" appendix of the *Caché Programming Orientation Guide*.

- Be sure that you know which globals your application uses.

Even if your application never performs any direct global access, your application uses globals. Remember that if you create persistent classes, their data and any indices are stored in globals, whose names are based on the class names (by default). See "Data," in the previous chapter.

5.2 Introduction to the Globals Page

The Management Portal includes the **Globals** page, which allows you to manage globals. On this page, you can:

- Select **View** in the row for that global to examine it.

- Select **Edit** in the row for that global to modify it.

- Select **Export** to export globals.

- Select **Import** to import globals.

- Select **Delete** to delete globals.

- Select **Find** to find values in globals.

This page also includes options for viewing routines and classes; these options are not discussed here.

To access this page from the Management Portal home page:

1. Select **System Explorer** > **Globals**.

2. Select the namespace or database of interest:

   - Select either **Namespaces** or **Databases** from the **Lookin** list.

   - Select the desired namespace or database from the displayed list.

   Selecting a namespace or database updates the page to display its globals.

3. If you are looking for a particular global and do not initially see its name:

   - Optionally specify a search mask. To do so, enter a value into the **Globals** field. If you end the string with an asterisk "*", the asterisk is treated as a wildcard, and the page displays each global whose name begins with the string before the asterisk.

     After entering a value, press **Enter**.

   - Optionally select **System items** to include all system globals in the search.

   - Optionally select a value from **Page size**, which controls the number of globals to list on any page.

5.3 Viewing Global Data

The **View Global Data** page lists nodes of the given global. In the table, the first column displays the row number, the next column lists the nodes, and the right column shows the values. This page initially shows the first hundred nodes in the global.

To access this page, display the Globals page and select the **View** link next to the name of a global. Or click the **View** button.

On this page, you can do the following:

- Specify a search mask. To do so, edit the value in **Global Search Mask** as follows:

  - To display a single node, use a complete global reference. For example: `^Sample.PersonD(9)`

  - To display a subtree, use a partial global reference without the right parenthesis. For example: `^%SYS("JOURNAL"`

  - To display all nodes that match a given subscript, include the desired subscript and leave other subscript fields empty. For example: `^CacheMsg(,"en")`

  - To display all subtrees that match a given subscript, use a value as in the previous option but also omit the right parenthesis. For example: `^CacheMsg(,"en"`

  - To display nodes that match a range of subscripts, use *subscriptvalue1*:*subscriptvalue2* in the place of a subscript. For example: `^Sample.PersonD(50:60)`

    As with the previous option, if you omit the right parenthesis, the system displays the subtrees.

  Then click **Display** or press **Enter**.

- Specify a different number of nodes to display. To do so, enter an integer into **Maximum Rows**.

- Repeat a previous search. To do so, select the search mask in the **Search History** drop-down.

- Select **Allow Edit** to make the data editable; see the next topic.

To close this page, click **Cancel**.

5.4 Editing Globals

CAUTION: Before making any edits, be sure that you know which globals Caché uses and which globals your application uses; see "General Advice." There is no undo option. A modified global cannot be restored.

The **Edit Global Data** page enables you to edit globals. In the table, the first column displays the row number, the next column lists the nodes, and the right column shows the values (with a blue underline to indicate that the value can be edited). This page initially shows the first hundred nodes in the global.

To access and use this page:

1. Display the Globals page.

2. Select the **Edit** link next to the name of a global.

3. Optionally use the **Global Search Mask** field to refine what is displayed. See "Viewing Global Data."

4. Optionally specify a different number of nodes to display. To do so, enter an integer into **Maximum Rows**.

5. If necessary, navigate to the value you want to edit by selecting the subscripts that correspond to it.

6. Select the value that you want to edit.

   The page then displays two editable fields:

   - The top field contains the full global reference for the node you are editing. For example:
     `^Sample.PersonD("18")`

     You can edit this to refer to a different global node. If you do so, your action affects the newly specified global node.

   - The bottom field contains the current value of this node. For example:

```
$lb("",43144,$lb("White","Orange"),$lb("8262 Elm Avenue","Islip","RI",57581),"Rogers,Emilio
L.",
$lb("7430 Washington Street","Albany","GA",66833),"650-37-4263","")
```

Edit the values as needed.

7. If you make edits, click **Save** to save your changes, or click **Cancel**.

Or, to delete a node:

1. Optionally select **Delete global subnodes during deletion**

2. Click **Delete**.

3. Click **OK** to confirm this action.

Also see "Performing Wholesale Replacements," later in this chapter.

5.5 Finding Values in Globals

The **Find Global String** page enables you to find a given string in the subscripts or in the values of selected globals.

To access and use this page:

1. Display the Globals page.

2. Select the globals to work with. To do so, see steps 2 and 3 in the section "Introduction to the Globals Page."

3. Click the **Find** button.

4. For **Find What**, enter the string to search for.

5. Optionally clear **Match Case**. By default, the search is case-sensitive.

6. Click either **Find First** or **Find All**.

    The page then displays either the first node or all nodes whose subscripts or values contain the given string, within the selected globals. The table shows the node subscripts on the left and the corresponding values on the right.

7. If you used **Find First**, click **Find Next** to see the next node, as needed.

8. When you are done, click **Close Window**.

5.5.1 Performing Wholesale Replacements

CAUTION: Before making any edits, be sure that you know which globals Caché uses and which globals your application uses; see "General Advice." This option changes the data permanently. It is not recommended for use in production systems.

For development purposes, the **Find Global String** page also provides an option to make wholesale changes to values in global nodes. To use this option:

1. Display the Globals page.

2. Select the globals to work with. To do so, see steps 2 and 3 in the section "Introduction to the Globals Page."

3. Click the **Replace** button.

4. Use this page to find values as described in the previous section.

5. Specify a value for **Replace With**.

6. Click **Replace All**.

7. Click **OK** to confirm this action.

    The page then displays a preview of the change.

8. If the results are acceptable, click **Save**.

9. Click **OK** to confirm this action.

5.6 Exporting Globals

CAUTION: Because of how easy it is to import globals (which is an irreversible change), it is good practice to export only the globals you need to import. Note that if you export all globals, the export includes all the globals that contain code. Be sure that you know which globals Caché uses and which globals your application uses; see "General Advice."

The **Export Globals** page enables you to export globals.

To access and use this page:

1. Display the Globals page.

2. Specify the globals to work with. To do so, see steps 2 and 3 in the section "Introduction to the Globals Page."

3. Click the **Export** button.

4. Specify the file into which you wish to export the globals. To do this, either enter a file name (including its absolute or relative pathname) in the **Enter the path and name of the export on server <hostname>** field or click **Browse** and navigate to the file.

5. Select the export file's character set with the **Character set** list.

6. In the page's central box:

    • Choose an **Output format**

    • Choose a **Record format**

7. Select or clear **Check here to run export in the background...**

8. Click **Export**.

9. If the file already exists, click **OK** to overwrite it with a new version.

The export creates a .gof file.

5.7 Importing Globals

CAUTION: Before importing any globals, be sure that you know which globals Caché uses and which globals your application uses; see "General Advice." There is no undo option. After you import a global into an existing global (thus merging the data), there is no way to restore the global to its previous state.

The **Import Globals** page enables you to import globals. To access and use this page:

1. Display the Globals page.

2. Click the **Import** button.

3. Specify the import file. To do this, either enter a file (including its absolute or relative pathname) in the **Enter the path and name of the import file** field or click **Browse** and navigate to the file.

4. Select the import file's character set with the **Character set** list.

5. Select **Next**.

6. Choose the globals to import using the check boxes in the table.

7. Optionally select **Run import in the background**. If you select this, the task is run in the background.

8. Click **Import**.

5.8 Deleting Globals

CAUTION: Before deleting any globals, be sure that you know which globals Caché uses and which globals your application uses; see "General Advice." There is no undo option. A deleted global cannot be restored.

The **Delete Globals** page enables you to delete globals. To access and use this page:

1. Display the Globals page.

2. Select the globals to work with. To do so, see steps 2 and 3 in the section "Introduction to the Globals Page."

3. Click the **Delete** button.

4. Click **OK** to confirm this action.

5.9 APIs for Management Tasks

Caché also provides the following APIs to perform some of the tasks described in this chapter:

- The class %SYSTEM.OBJ provides the following methods:

  - **Export()** enables you to export globals to an XML file.

  - **Load()** and **LoadDir()** enable you to import globals contained in XML files.

  These are both available via the **$SYSTEM** variable, for example: **$SYSTEM.OBJ.Export**

- The class %Library.Global provides the following methods:

  - **Export()** enables you to export globals to .gof and other file formats (not including XML).

  - **Import()** enables you to import globals to .gof and other file formats (not including XML).

  %Library.Global also provides the **Get()** class query, which you can use to find globals, given search criteria.

For pointers to additional APIs, see "Globals" in the *InterSystems Programming Tools Index*.