Using Caché SQL

Version 2018.1
2024-04-03

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using Caché SQL
Caché Version 2018.1 2024-04-03
Copyright © 2024 InterSystems Corporation
All rights reserved.

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble® InterSystems HealthShare®, InterSystems IRIS®, and TrakCare are registered trademarks of InterSystems Corporation. HealthShare® CMS Solution Pack™ HealthShare® Health Connect Cloud™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, and InterSystems TotalView™ For Asset Management are trademarks of InterSystems Corporation. TrakCare is a registered trademark in Australia and the European Union.

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

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

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

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

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

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

List of Tables

About This Book

This book describes how to use the Caché SQL, which provides standard relational access to data stored within a Caché database.

The book addresses the following topics:

The Caché SQL language:

- "Introduction to Caché SQL" provides an overview of Caché SQL as it relates to software standards and interoperability.
- "Caché SQL Basics" describes the fundamental features of Caché SQL (such as tables and queries), especially those that are not covered by the SQL standard or are related to the Caché unified data architecture.
- "Language Elements" describes how Caché SQL handles the basic elements common to any programming language: numbers, strings, operators, NULL, and comments.
- "Identifiers" describes the conventions used for naming entities within Caché SQL.

Data Definition: creating tables and views:

- "Defining Tables" describes how to define tables in Caché SQL, by defining persistent classes or by using an SQL DDL statement.
- "Defining Views" describes how to define views in Caché SQL by using the Management Portal or by using a DDL statement.
- "Defining Foreign Keys" describes how to define foreign keys in Caché SQL.
- "Defining Triggers" describes how to define triggers in Caché SQL.
- "Defining and Using Stored Procedures" discusses stored procedures in Caché SQL.
- "Storing and Using BLOBs and CLOBs" describes stream data and how to store and use BLOBs and CLOBs in Caché SQL.
- "Users, Roles, and Privileges" addresses connections between Caché SQL and Caché security features.

Data Management: querying and modifying data:

- "Querying the Database" describes how to create and use SELECT queries.
- "Implicit Joins" describes a Caché SQL extension that provides arrow syntax for implicit joins. Caché SQL also provides standard syntax for explicit joins.
- "Modifying the Database" describes how to use INSERT, UPDATE, and DELETE to modify data, and how to use transactions to group multiple data modifications.

SQL execution interfaces:

- "Using Embedded SQL" describes how to write and execute SQL code embedded within ObjectScript code. This chapter also describes SQL cursors, which enable you to access multiple rows of data.
- "Using Dynamic SQL" describes how ObjectScript can include SQL that is executed at runtime.
- "Using the SQL Shell" describes how to write and execute SQL statements from the Terminal.
- "Using the Management Portal SQL Interface" describes how to write and execute SQL statements from the Management Portal.

- "Importing SQL Code" describes how to execute SQL statements by importing them from a text file. This interface can be used for Caché SQL code or SQL code in other vendor formats. Import SQL code can be used to define tables and to populate tables with data; it cannot be used to query data.

SQL interface:

- "Using the Caché SQL Gateway" describes how to obtain access to external databases via JDBC and ODBC, enabling you to treat external tables as if they were native Caché tables.
- "Importing and Exporting SQL Data" (an appendix) discusses tools in the Management Portal that enable you to import or export data.

For a detailed outline, see the Table of Contents.

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

- *The Caché SQL Reference* provides details on individual SQL commands and functions, as well as information on the Caché SQL data types and reserved words.
- *Caché SQL Optimization Guide* describes how to optimize a table definition by defining and building indices, how to use Tune Table to optimize table metadata based on typical data, and how to optimize query execution using cached queries, ShowPlan, frozen plans, and other optimization techniques.
- *Caché Programming Orientation Guide* is an orientation guide for programmers who are new to Caché or who are familiar with only some kinds of Caché programming.
- In *Using Caché Objects*, the chapter "Introduction to Persistent Objects" summarizes how Caché object technology interoperates with SQL. Later chapters provide additional detail.
- *Using Caché with JDBC* describes how to access Caché tables from external applications via JDBC.
- *Using Caché with ODBC* describes how to access Caché tables from external applications via ODBC.
- *Caché Advanced Configuration Settings Reference* describes the SQL configuration settings.
- *Caché Error Reference* lists the SQLCODE error messages.

For general information, see *Using InterSystems Documentation*.

1

Introduction to Caché SQL

Caché SQL provides uncompromising, standard relational access to data stored within a Caché database.

Caché SQL offers the following benefits:

- *High performance and scalability* — Caché SQL offers performance and scalability superior to other relational database products. In addition, Caché SQL runs on a wide variety of hardware and operating systems; from laptop computers to high-end, multi-CPU systems.
- *Integration with Caché objects technology* — Caché SQL is tightly integrated with Caché object technology. You can mix relational and object access to data *without* sacrificing the performance of either approach.
- *Low maintenance* — Unlike other relational databases, Caché applications do not require index rebuilding and table compression in deployed applications.
- *Support for standard SQL queries* — Caché SQL supports SQL-92 standard syntax and commands. In most cases, you can migrate existing relational applications to Caché with little difficulty and automatically take advantage of the higher performance and object capabilities of Caché.

You can use Caché SQL for many purposes including:

- *Object- and web-based applications* — You can use SQL queries within Caché Object and Caché Server Page applications to perform powerful database operations such as lookups and searches.
- *Online transaction processing* — Caché SQL offers outstanding performance for insert and update operations as well as the types of queries typically found within transaction processing applications.
- *Business intelligence and data warehousing* — The combination of the Caché multidimensional database engine and bitmap indexing technology make it an excellent choice for data warehouse-style applications.
- *Ad hoc queries and reports* — You can use the full-featured ODBC and JDBC drivers included with Caché SQL to connect to popular reporting and query tools.
- *Enterprise application integration* — The Caché SQL Gateway gives you seamless SQL access to data stored in external relational databases that are ODBC- or JDBC-compliant. This makes it easy to integrate data from a variety of sources within Caché applications.

1.1 Architecture

The core of Caché SQL consists of the following components:

- *The Unified Data Dictionary* — a repository of all meta-information stored as a series of class definitions. Caché automatically creates relational access (tables) for every persistent class stored within the Unified Dictionary.
- *The SQL Processor and Optimizer* — a set of programs that parse and analyze SQL queries, determine the best search strategy for a given query (using a sophisticated cost-based optimizer), and generate code that executes the query.
- *The Caché SQL Server* — a set of Caché server processes that are responsible for all communications with the Caché ODBC and JDBC drivers. It also manages a cache of frequently used queries; when the same query is executed multiple times, its execution plan can be retrieved from the query cache instead of having to be processed by the Optimizer again.

1.2 Features

Caché SQL includes a full set of standard, relational features. These include:

- The ability to define tables and views (DDL or Data Definition Language).
- The ability to execute queries against tables and views (DML or Data Manipulation Language).
- The ability to execute transactions, including INSERT, UPDATE, and DELETE operations. When performing concurrent operations, Caché SQL uses row-level locks.
- The ability to define and use indices for more efficient queries.
- The ability to use a wide variety of data types, including user-defined types.
- The ability to define users and roles and assign privileges to them.
- The ability to define foreign keys and other integrity constraints.
- The ability to define INSERT, UPDATE, and DELETE triggers.
- The ability to define and execute stored procedures.
- The ability to return data in different formats: ODBC mode for client access; Display mode for use within server-based applications (such as CSP pages).

Note: We continue to add support for additional features within Caché SQL. If you require a feature that is not supported within this release, please feel free to check with the InterSystems InterSystems Worldwide Response Center (WRC) to see if it will be included in a newer release.

1.2.1 SQL-92 Compliance

The SQL-92 standard is imprecise with regard to arithmetical operator precedence; assumptions on this matter differ amongst SQL implementations. Caché SQL supports two system-wide alternatives for SQL arithmetic operator precedence:

- By default, Caché SQL parses arithmetic expressions in strict left-to-right order, with no operator precedence. This is the same convention used in ObjectScript. Thus, `3+3*5=30`. You can use parentheses to enforce the desired precedence. Thus, `3+(3*5)=18`.
- You can configure Caché SQL to parse arithmetic expressions using ANSI precedence, which gives higher precedence to multiplication and division operators than addition, subtraction, and concatenation operators. Thus, `3+3*5=18`. You can use parentheses to override this precedence, where desired. Thus, `(3+3)*5=30`.

Caché SQL supports the complete entry-level SQL-92 standard with the following exceptions:

- There is no support for adding additional CHECK constraints to a table definition.

- The SERIALIZABLE isolation level is not supported.
- Delimited identifiers are not case-sensitive; the standard says that they should be case-sensitive.
- Within a subquery contained in a HAVING clause, one is supposed to be able to refer to aggregates which are "available" in that HAVING clause. This is not supported.

1.2.2 Extensions

Caché SQL supports a number of useful extensions. Many of these are related to the fact that Caché offers simultaneous object and relational access to data.

Some of these extensions include:

- Support for user-definable data type and functions.
- Special syntax for following object references.
- Support for subclassing and inheritance.
- Support for queries against external tables stored within other databases.
- A number of mechanisms for controlling the storage structures used for tables to achieve maximum performance.

1.3 Interoperability

Caché SQL supports a number of ways to interoperate *relationally* with other applications and software tools.

1.3.1 JDBC

Caché includes a standards-compliant, level 4 (all pure Java code) JDBC client.

The Caché JDBC driver offers the following features:

- High-performance
- A pure Java implementation
- Unicode support
- Thread-safety

You can use Caché JDBC with any tool, application, or development environment that supports JDBC. If you encounter problems or have questions about compatibility, contact the InterSystems InterSystems Worldwide Response Center (WRC).

1.3.2 ODBC

The C-language call level interface for Caché SQL is ODBC. Unlike other database products, the Caché ODBC driver is a *native* driver — it is not built on top of any other proprietary interface.

The Caché ODBC driver offers the following features:

- High-performance
- Portability
- Native Unicode support

- Thread-safety

You can use Caché ODBC with any tool, application, or development environment that supports ODBC. If you encounter problems or have questions about compatibility, contact the InterSystems InterSystems Worldwide Response Center (WRC).

1.3.3 Embedded SQL

Within ObjectScript, Caché SQL supports Embedded SQL: the ability to place an SQL statement within the body of a method (or other code). Using Embedded SQL, you can query a single record, or define a cursor and use that to query multiple records. Embedded SQL is compiled; it is either compiled at the same time as the ObjectScript routine (the default), or you can defer Embedded SQL compilation until runtime.

Embedded SQL is quite powerful when used in conjunction with the object access capability of Caché. For example, the following method finds the Object ID of the Product with a given SKU code and uses it to create an in-memory object instance:

Class Member

```
ClassMethod FindBySKU(sku As %String)
{
    &sql(SELECT %ID INTO :id FROM Product WHERE SKU = :sku)
      IF SQLCODE<0 {SET baderr="SQLCODE ERROR:"_SQLCODE_" "_%msg
                    RETURN baderr }
      ELSEIF SQLCODE=100 {SET nodata="Query returns no data"
                    RETURN nodata }
      ELSE {
        // ask the product to display details about itself
        SET product = ##class(Product).%OpenId(id)
        DO product.DisplayDetails()
    }
}
```

For more details, see the chapter "Using Embedded SQL."

1.3.4 Dynamic SQL

As part of its standard library, Caché provides an %SQL.Statement class that you can use to execute dynamic (that is, defined at runtime) SQL statements. You can use Dynamic SQL within ObjectScript and Caché Basic methods. For example, the following method queries for a specified number of people born in the 21st century. The query selects all people born after December 31, 1999, orders the selected records by date of birth, then selects the top *x* records:

Class Member

```
ClassMethod Born21stC(x) [ language = cache ]
{
  SET myquery=2
  SET myquery(1) = "SELECT TOP ? Name,%EXTERNAL(DOB) FROM Sample.Person "
  SET myquery(2) = "WHERE DOB > 58073 ORDER BY DOB"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(.myquery)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute(x)
  DO rset.%Display()
  WRITE !,"End of data"
 }
```

When you prepare a query, an optimized version of that query is stored as a cached query. This cached query is executed for subsequent invocations of the query, avoiding the overhead of re-optimizing a query each time it is executed.

For more details, see the chapter "Using Dynamic SQL."

1.4 Limitations

Note the following limitations of Caché SQL:

- NLS can be used to specify the behavior of **$ORDER** for a particular national locale behavior for individual globals, as well as for local variables in the currently running process. Caché SQL can be used and works well within any National Language locale. However, a current limitation of Caché SQL is that for any particular process, all the relevant globals it references have to be using the same national locale as the current process locale. See "SQL Collation and NLS Collations" in the "Collation" chapter of this guide.

2

Caché SQL Basics

This chapter provides an overview of the features of Caché SQL, especially those that are not covered by the SQL standard or are related to the Caché Unified Data Architecture. It assumes prior knowledge of SQL and is not designed to serve as an introduction to SQL concepts or syntax.

This chapter discusses the following topics:

- Tables
- Queries
- Privileges
- Data Display Options
- Data Collation Types
- Executing SQL

2.1 Tables

Within Caché SQL, data is presented within *tables*. Each table is defined to contain a number of *columns*. A table may contain zero or more rows of data values. The following terms are roughly equivalent:

Data Terms	Relational Database Terms	Caché Terms
database	schema	package
	table	persistent class
field	column	property
record	row	

For further details, refer to "Introduction to the Default SQL Projection" in the "Introduction to Persistent Objects" chapter of *Using Caché Objects*.

There are two basic types of tables: *base tables* (which contain data and are usually referred to simply as tables) and *views* (which present a logical view based on one or more tables).

To find out more on how to define tables, see the chapter "Defining Tables."

To find out more on how to define views, see the chapter "Defining Views."

In order to make queries against tables more efficient, you can define indices on tables. See the chapter "Defining and Building Indices." in the *Caché SQL Optimization Guide*.

In order to enforce referential integrity you can define foreign keys and triggers on tables. See the chapters "Defining Foreign Keys" and "Defining Triggers."

2.1.1 Schemas

SQL schemas provides a means of grouping sets of related tables, views, stored procedures, and cached queries. The use of schemas helps prevent naming collisions at the table level, because a table, view, or stored procedure name must only be unique within its schema. An application can specify tables in multiple schemas.

SQL schemas correspond to persistent class packages. Commonly a schema has the same name as its corresponding package, but these names may differ because of different schema naming conventions or because different names have been deliberately specified. Schema-to-package mapping is further described in SQL to Class Name Transformations.

Schemas are defined within a specific namespace. A schema name must be unique within its namespace. A schema (and its corresponding package) is automatically created when the first item is assigned to it and automatically deleted when the last item is deleted from it.

You can specify an SQL name as qualified or unqualified. A qualified name specifies the schema: `schema.name`. An unqualified name does not specify the schema: `name`. If you do not specify the schema, Caché supplies the schema as follows:

- For DDL operations, Caché uses the system-wide default schema name. This default is configurable. It applies to all namespaces.
- For DML operations, Caché can use either a user-supplied schema search path or the system-wide default schema name. Different techniques are used to supply a schema search path in Dynamic SQL, Embedded SQL, and the SQL Shell.

To view all the existing schemas within a namespace:

1. From the Management Portal select **System Explorer**, then **SQL**. Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces. Select a namespace.
2. Select the **Schema** drop-down list on the left side of the screen. This displays a list of the schemas in the current namespace. Select a schema from this list; the selected name appears in the **Schema** box.
3. The **applies to** drop-down list allows you to select Tables, Views, Procedures, or Cached Queries, or All of these that belong to the schema. After setting this option, click the triangles to view a list of the items. If there are no items, clicking a triangle has no effect.

2.2 Queries

Within Caché SQL, you view and modify data within tables by means of *queries*. Roughly speaking, queries come in two flavors: those that retrieve data (SELECT statements), and those that modify data (INSERT, UPDATE, and DELETE statements).

You can use SQL queries in a number of ways:

- Using Embedded SQL within ObjectScript.
- Using Dynamic SQL within ObjectScript or Caché Basic.
- Calling a stored procedure created using CREATE PROCEDURE or CREATE QUERY.
- Using a class query. For further details, refer to "Defining and Using Class Queries" in *Using Caché Objects*.

- Using the ODBC or JDBC interfaces from a variety of other environments.

SELECT queries are described in the Querying the Database chapter of this guide.

Queries are part of Caché objects or ObjectScript routines.

2.3 Privileges

Caché SQL provides a way to limit access to tables, views, and so on via *privileges*. You can define a set of *users* and *roles* and grant various privileges (read, write, and so on) to them. See the chapter "Users, Roles, and Privileges."

2.4 Data Display Options

Caché SQL uses a SelectMode option to specify how data is to be displayed or stored. The available options are Logical, Display, and ODBC. Data is stored internally in Logical mode, and can be displayed in any of these modes. Every data type class can define transformations between internal Logical format and Display format or ODBC format by using the **LogicalToDisplay()**, **LogicalToODBC()**, **DisplayToLogical()**, and **ODBCToLogical()** methods. When SQL SelectMode is Display, the LogicalToDisplay transformation is applied, and returned values are formatted for display. The default SQL SelectMode is Logical; thus by default returned values are displayed in their storage format.

SelectMode affects the format that in which query result set data is displayed, SelectMode also affects the format in which data values should be supplied, for example in the WHERE clause. Caché applies the appropriate transformation method based on the storage mode and the specified SelectMode. A mismatch between a supplied data value and the SelectMode can result in an error or in erroneous results. For example, if DOB is a date stored in $HOROLOG Logical format, and a WHERE clause specifies `WHERE DOB > 2000-01-01` (ODBC format), SelectMode = ODBC returns the intended results. SelectMode = Display generates SQLCODE -146 `Unable to convert date input to a valid logical date value`. SelectMode = Logical attempts to parse `2000-01-01` as a Logical date value, and returns zero rows.

For most data types, the three SelectMode modes return the same results. The following data types are affected by the SelectMode option:

- Date, Time, and Timestamp data types. InterSystems SQL supports numerous Date, Time, and Timestamp data types (%Library.Date, %Library.Time, %Library.TimeStamp, %Library.FilemanDate, %Library.FilemanTimeStamp, and %MV.Date). With the exception of %Library.TimeStamp, these data types use different representations for Logical, Display, and ODBC modes. In several of these data types Caché stores dates in $HOROLOG format. This Logical mode internal representation consists of an integer count of the number of days from an arbitrary starting date (December 31st, 1840), a comma separator, and an integer count of the number of seconds since midnight of the current day. In Display mode, dates and times commonly appear in the format specified by the data type's FORMAT parameter or the date and time format defaults for the current locale in %SYS.NLS.Format. The default for the American locale is `DD/MM/YYYY hh:mm:ss`. In ODBC mode, dates and times are always represented as `YYYY-MM-DD hh:mm:ss.fff`. The %Library.TimeStamp data type also uses this ODBC format for Logical and Display modes.
- %List data type. Caché Logical mode stores lists using two non-printing characters that appear before the first item in the list, and appear as a separator between list items. In ODBC SelectMode, list items are displayed with a comma separator between list items. In Display SelectMode, list items are displayed with a blank space separator between list items.
- Data types that specify VALUELIST and DISPLAYLIST. If you are in display mode and you insert a value into a table where the field has a DISPLAYLIST, the display value you enter must exactly match one of the items in the DISPLAYLIST.

- Empty strings, and empty BLOBs (stream fields). In Logical mode empty strings and BLOBs are represented by the non-display character $CHAR(0). In Display mode they are represented by an empty string ("").

The SQL SelectMode may be specified as follows:

- For the current process, using **$SYSTEM.SQL.SetSelectMode()**.
- For a Caché SQL Shell session, using the SET SELECTMODE command.
- For a query result set from the Management Portal "Execute Query" user interface (**[System] > [SQL]**), using the "Display Mode" drop-down list.
- For a Dynamic SQL %SQL.Statement instance, using the %SelectMode property.
- For a Dynamic SQL %ResultSet instance, using the *%Library.ResultSet.RuntimeMode* property.
- For Embedded SQL, using the ObjectScript #sqlcompile select preprocessor directive setting. This directive allows for a fourth value, Runtime, which sets the select mode to whatever the *RuntimeMode* property setting is: Logical, Display, or ODBC. The RuntimeMode default is Logical.
- For the SQL commands CREATE QUERY, CREATE METHOD, CREATE PROCEDURE, and CREATE FUNCTION using the SELECTMODE keyword.
- For an individual column within an SQL query by using the %EXTERNAL, %INTERNAL, and %ODBCOUT functions.
- For a Zen Report using the runtimeMode attribute of a <report> element.

2.5 Data Collation

Collation specifies how values are ordered and compared, and is part of both Caché SQL and Caché objects.

You can specify a collation type as part of field/property definition. Unless otherwise specified, a string field/property defaults to the namespace default collation. By default, the namespace default collation for strings is SQLUPPER. SQLUPPER collation transforms strings into uppercase for the purposes of sorting and comparing. Thus, unless otherwise specified, string ordering and comparison is not case-sensitive.

You can specify a collation type as part of index definition, or use the collation type of the indexed field.

An SQL query can override the defined field/property collation type by applying a collation function to a field name. The ORDER BY clause specifies the result set sequence for a query; if a specified string field is defined as SQLUPPER, query results order is not case-sensitive.

For further details refer to the "Collation" chapter of *Using Caché SQL*.

2.6 Executing SQL

Caché supports numerous ways to write and execute SQL code. These include:

- Embedded SQL: SQL code embedded within ObjectScript code.
- Dynamic SQL: SQL code executed from within ObjectScript or Caché Basic, using the %SQL.Statement class.
- Execute() method: execute SQL code using the **Execute()** method of the %SYSTEM.SQL class.
- Stored Procedure containing SQL code, created using CREATE PROCEDURE or CREATE QUERY.
- The SQL Shell: SQL statements executed from the Terminal interface.

- Execute Query Interface: SQL statements executed from the Management Portal.

You can use Caché objects (classes and methods) to:

- Define a persistent class (an SQL table).
- Define an index.
- Define and Use a Class Query.

3

Language Elements

Caché SQL supports the following language elements:

- Commands and keywords
- Functions: intrinsic and extrinsic
- String and numeric literals
- NULL and the empty string
- Arithmetic operators and functions
- Relational operators
- Logical operators
- Comments

3.1 Commands and Keywords

A Caché SQL command (also known as an SQL statement) begins with a keyword followed by one or more arguments. Some of these arguments may be clauses or functions, identified by their own keywords.

- Caché SQL commands do not have a command terminator, except in specific cases such as SQL procedure code or trigger code, in which case SQL commands are terminated by a single semicolon (;). Otherwise, Caché SQL commands do not require or accept a semicolon command terminator. Specifying a semicolon command terminator in Caché SQL results in an SQLCODE -25 error. Caché implementation of TSQL (Transact-SQL) accepts, but does not require, a semicolon command terminator. When importing SQL code to Caché SQL, semicolon command terminators are stripped out.
- Caché SQL commands have no whitespace restrictions. If command items are separated by a space, at least one space is required. If command items are separated by a comma, no space is required. No space is required before or after arithmetic operators. You may insert line breaks or multiple spaces between space-separated items, between items in a comma-separated list of arguments, or before or after arithmetic operators.

Caché SQL keywords include command names, function names, predicate condition names, data type names, field constraints, optimization options, and special variables. They also include the AND, OR, and NOT logical operators, the NULL column value indicator, and ODBC function constructs such as {d dateval} and {fn CONCAT(str1,str2)}.

- Keywords are not case-sensitive. By convention, keywords are represented by capital letters in this documentation, but Caché SQL has no letter case restriction.

- Many, but not all, keywords are SQL Reserved Words. Caché SQL only reserves those keywords that cannot be unambiguously parsed. SQL reserved words can be used as delimited identifiers.

3.2 Functions: Intrinsic and Extrinsic

A function performs an operation and returns a value. Commonly in Caché SQL a function is specified in a **SELECT** statement as a *select-item* or in a WHERE clause, performing an operation either on a table field value or on a literal value.

- Intrinsic: Caché SQL supports a large number of intrinsic (system-supplied) functions. These include numeric functions, string functions, and date and time functions. These functions are described in the *Caché SQL Reference*. The arithmetic and trigonometric functions are also listed in this chapter.

  Aggregate functions are SQL intrinsic functions that evaluate all of the values of a column and return a single aggregate value. Aggregate functions are described separately in the *Caché SQL Reference*.

- Extrinsic: Caché SQL can also support user-supplied ObjectScript function calls (extrinsic functions), as shown in the following example:

  ObjectScript

  ```
  MySQL
    &sql(SELECT Name,$$MyFunc() INTO :n,:f FROM Sample.Person)
    WRITE "name is: ",n,!
    WRITE "function value is: ",f,!
    QUIT
  MyFunc()
    SET x="my text"
    QUIT x
  ```

  An SQL statement can only invoke user-supplied (extrinsic) functions if their use is configured as a system-wide option. The default is "No"; by default, attempting to invoke user-supplied functions issues an SQLCODE -372 error. You can configure SQL use of extrinsic functions system-wide using the either the **SetAllowExtrinsicFunctions()** method of the %SYSTEM.SQL class, or the Management Portal SQL configuration interface. From the Management Portal, select **System Administration**, **Configuration**, then **SQL and Object Settings**, then **General SQL Settings**. View and edit the current setting of **Allow Extrinsic Functions in SQL Statements**. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays the `Allow extrinsic functions in SQL statements` option.

  You cannot use a user-supplied function to call a % routine (a routine with a name that begins with the % character). Attempting to do so issues an SQLCODE -373 error.

3.3 Literals

Caché SQL literals have the following syntax:

```
literal ::=
     number | string-literal

number ::=
     {digit}[.]digit{digit}[E[+|-]digit{digit}]

digit ::=
     0..9

string-literal ::=
     std-string-literal | ObjectScript-empty-string

std-string-literal ::=
     ' {std-character-representation} '

std-character-representation ::=
     nonquote-character | quote-symbol

quote-symbol ::=
     ''

ObjectScript-empty-string ::=
     ""
```

A literal is a series of characters that represents an actual (literal) value. It can be either a number or a character string.

- A number does not require any delimiter character. It can consist of the digits 0 through 9, the decimal point character, the exponent symbol and the plus and minus signs. Only one decimal point character can be used in a number. This decimal point can only be used in the base portion of a number, not in the exponent portion. The decimal point does not need to be followed by a digit. Leading and trailing zeros are permitted. The exponent (scientific notation) symbol is the letter E; both uppercase and lowercase E are accepted, but uppercase E is the preferred usage. A plus or minus sign can prefix a base number or an exponent. Multiple plus and minus signs can prefix a base number; SQL treats these signs as operators. Only a single plus or minus sign can prefix an exponent; SQL treats this sign as part of the literal. No commas or blanks are permitted in a number.
- A character string literal consists of a pair of delimiter characters enclosing a string of characters of any type. The preferred delimiter character is the single-quote character (see below). To specify a delimiter character as a literal within a character string, double the character; for example: 'Mary''s office'.

The empty string is a literal string; it is represented by two single-quote characters (''). NULL is *not* a literal value; it represents the absence of any value. For further details, see the NULL and the Empty String section of this chapter.

Note: In Embedded SQL, a few character sequences that begin with ## are not permitted within a string literal, as described in Literal Values in the "Using Embedded SQL" chapter. This restriction does not apply to other invocations of SQL, such as Dynamic SQL.

3.3.1 String Delimiters

You can use either single quote (') characters or the double quote (") characters as string delimiters. The single-quote (') character is the preferred delimiter. The use of the double-quote character (") is supported for SQL compatibility, but this use is discouraged because of potential conflict with the delimited identifier standard.

To specify the character used as the delimiter as a literal character within the string, specify a pair of these characters.

3.3.2 Concatenation

The double vertical bar (||) is the preferred SQL concatenation operator. It can be used to concatenate two numbers, two character strings, or a number and a character string.

The underscore (_) is provided as an SQL concatenation operator for ObjectScript compatibility. This concatenation operator can only be used to concatenate two character strings.

If the two operands are both character strings, and both strings have the same collation type, the resulting concatenated string has that collation type. In all other cases, the result of concatenation is of collation type EXACT.

3.4 NULL and the Empty String

Use the NULL keyword to indicate that a value is not specified. NULL is always the preferred way in SQL to indicate that a data value is unspecified or nonexistent for any reason.

The SQL zero-length string (empty string) is specified by two single quote characters. The empty string ('') is *not* the same thing as NULL. An empty string is a defined value, a string that contains no characters, a string of length 0. A zero-length string is represented internally by the non-display character $CHAR(0).

Note: The SQL zero-length string is *not recommended* for use as a field input value or a field default value. Use NULL to represent the absence of a data value.

The SQL zero-length string should be avoided in SQL coding. However, because many SQL operations delete trailing blank spaces, a data value that contains only whitespace characters (spaces and tabs) may result in an SQL zero-length string.

Note that different SQL length functions return different values: LENGTH, CHAR_LENGTH, and DATALENGTH return SQL lengths. $LENGTH returns ObjectScript representation length. See "The Length of NULL" below. **LENGTH** does not count trailing blank spaces; all other length functions count trailing blank spaces.

The SQL zero-length string, like all SQL strings, can also be represented using double quote characters (""); this usage should be avoided because of potential conflict with SQL delimited identifiers.

3.4.1 NULL Processing

The NOT NULL data constraint requires that a field must receive a data value; specifying NULL rather than a value is not permitted. This constraint does not prevent the use of an empty string value. For further details, refer to the CREATE TABLE command.

The IS NULL predicate in the WHERE or HAVING clause of a **SELECT** statement selects NULL values; it does not select empty string values.

The IFNULL function evaluates a field value and returns the value specified in its second argument if the field evaluates to NULL. It does not treat an empty string value as a non-NULL value.

The COALESCE function selects the first non-NULL value from supplied data. It treats empty string values as non-NULL.

When the CONCAT function or the concatenate operator (||) concatenate a string and a NULL, the result is NULL. This is shown in the following example:

```
SELECT {fn CONCAT('fred',NULL)} AS FuncCatNull,
       'fred'||NULL AS OpCatNull
```

The AVG, COUNT, MAX, MIN, and SUM aggregate functions ignore NULL values when performing their operations. (COUNT * counts all rows, because there cannot be a record with NULL values for all fields.) The DISTINCT keyword of the **SELECT** statement includes NULL in its operation; if there are NULL values for the specified field, DISTINCT returns one NULL row.

The AVG, COUNT, and MIN, aggregate functions are affected by empty string values. The **MIN** function considers an empty string to be the minimum value, even when there are rows that have a value of zero. The MAX and SUM aggregate functions are not affected by empty string values.

3.4.2 NULL in Expressions

Supplying NULL as an operand to most SQL functions returns NULL.

Any SQL arithmetic operation that has NULL as an operand returns a value of NULL. Thus, 7+NULL=NULL. This includes the binary operations addition (+), subtraction (-), multiplication (*), division (/), integer division (\), and modulo (#), and the unary sign operators plus (+) and minus (-).

An empty string specified in an arithmetic operation is treated as a value of 0 (zero). Division (/), integer division (\), or modulo (#) by empty string (6/'') results in a <DIVIDE> error.

3.4.3 The Length of NULL

Within SQL, the length of a NULL is undefined (it returns <null>). The length of an empty string, however, is defined as length zero. This is shown in the following example:

```
SELECT LENGTH(NULL) AS NullLen,
       LENGTH('') AS EmpStrLen
```

As shown in this example, the SQL LENGTH function returns the SQL lengths.

You can convert an SQL zero-length string to a NULL by using the ASCII function, as shown in the following example:

```
SELECT LENGTH('') AS EmpStrLen,
       LENGTH({fn ASCII('')}) As NullLen
```

However, certain Caché extensions to standard SQL treat the length of NULL and the empty string differently. The $LENGTH function returns the Caché internal representation of these values: NULL is represented as a defined value with length 0, the SQL empty string is represented as a string of length 0. This functionality is compatible with ObjectScript.

```
SELECT $LENGTH(NULL) AS NullLenZero,
$LENGTH('') AS EmpStrLenZero,
$LENGTH('a') AS StrLenOne,
$LENGTH(CHAR(0)) AS CharLenZero
```

Another place where the internal representation of these values is significant is in the **%STRING**, **%SQLSTRING** and **%SQLUPPER** functions, which append a blank space to a value. Since a NULL truly has no value, appending a blank to it creates a string of length 1. But an empty string does have a character value, so appending a blank to it creates a string of length 2. This is shown in the following example:

```
SELECT CHAR_LENGTH(%STRING(NULL)) AS BlankLenOne,
CHAR_LENGTH(%STRING('')) AS EmpStrAndBlankLenTwo
```

Note that this example uses **CHAR_LENGTH**, not **LENGTH**. Because the **LENGTH** function removes trailing blanks, `LENGTH(%STRING(NULL))` returns a length of 0; `LENGTH(%STRING(''))` returns a length of 2, because **%STRING** appends a leading blank, not a trailing blank.

3.4.4 ObjectScript and SQL

When an SQL NULL is output to ObjectScript, it is represented by an ObjectScript empty string (""), a string of length zero.

When an SQL zero-length string data is output to ObjectScript, it is represented by a string containing $CHAR(0), a string of length 1.

ObjectScript

```
 &sql(SELECT NULL,''
      INTO :a,:b)
WRITE !,"NULL length: ",$LENGTH(a)         // returns 0
WRITE !,"empty string length: ",$LENGTH(b) // returns 1
```

In ObjectScript, the absence of a value is usually indicated by an empty string (""). When this value is passed into embedded SQL, it is treated as a NULL value, as shown in the following example:

ObjectScript

```
SET x=""
SET myquery="SELECT NULL As NoVal,:x As EmpStr"
SET tStatement=##class(%SQL.Statement).%New()
SET qStatus=tStatement.%Prepare(myquery)
IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset=tStatement.%Execute()
WHILE rset.%Next() {
  WRITE "NoVal:",rset.%Get("NoVal")," length ",$LENGTH(rset.%Get("NoVal")),!      // length 0
  WRITE "EmpStr:",rset.%Get("EmpStr")," length ",$LENGTH(rset.%Get("EmpStr")),!   // length 0
}
WRITE "End of data"
```

If you specify an input host variable that is not defined, embedded SQL treats its value as NULL.

When passing a NULL or empty string value out from embedded SQL to ObjectScript, the NULL is translated to a string of length 0, and the empty string is translated to a string of length 1. This is shown in the following example:

ObjectScript

```
 &sql(SELECT
      NULL,
      ''
      INTO :a,:b)
WRITE !,"The length of NULL is: ",$LENGTH(a)         // length 0
WRITE !,"The length of empty string is: ",$LENGTH(b) // length 1
```

In the following example, the SQL empty string with an appended blank is passed out as string of length 2:

ObjectScript

```
 &sql(SELECT %SQLUPPER('')
      INTO :y)
WRITE !,"SQL empty string length: ",$LENGTH(y)
```

3.5 Arithmetic Operators and Functions

Caché SQL supports the following arithmetic operators:

+	Addition operator. For example, `17+7` equals 24.
–	Subtraction operator. For example, `17-7` equals 10. Note that a pair of these characters is the Caché SQL comment indicator. Therefore, to specify two or more subtraction operators or negative signs you must use either spaces or parentheses. For example, `17- -7` or `17-(-7)` equals 24.
*	Multiplication operator. For example, `17*7` equals 119.
/	Division operator. For example, `17/7` equals 2.428571428571428571.
\	Integer division operator. For example, `17\7` equals 2.
#	Modulo operator. For example, `17 # 7` equals 3. Note that because the # character is also a valid identifier character, to use it as a modulo operator you should specify it separated from its operands by spaces before and after.
E	Exponentiation (scientific notation) operator. Can be uppercase or lowercase. For example, `7E3` equals 7000. A too-large exponent results in an SQLCODE -7 “Exponent out of range” error. For example, `1E309` or `7E308`.
()	Grouping operators. Used to nest arithmetic operations. Unless parentheses are used, the execution sequence of arithmetic operations in Caché SQL is strict left-to-right order. For example, `17+7*2` equals 48, but `17+(7*2)` equals 31.
||	Concatenate operator. For example, `17||7` equals 177.

Arithmetic operations are performed on numbers in their canonical form.

For numbers of any data type, the *data type* for the result of an addition (+), subtraction (-), multiplication (*), or division (/, \, or #) operation is NUMERIC, unless one or both of the arguments is of data type DOUBLE; in that case, the data type of the result is DOUBLE. For example, adding two fields of data type INTEGER results in a value of data type NUMERIC. Concatenating two numbers of any data type results in a VARCHAR string.

In Dynamic SQL you can use SQL column metadata to determine the data type of a result set field. For further details on numeric data types refer to SQL Data Types.

3.5.1 Operator Precedence

The SQL-92 standard is imprecise with regard to operator precedence; assumptions on this matter differ amongst SQL implementations.

- Caché SQL, by default, does not provide precedence of arithmetic operators. By default, Caché SQL executes arithmetic expressions in strict left-to-right order, with no operator precedence. This is the same convention used in ObjectScript. Thus, `3+3*5` equals 30. You can use parentheses to enforce the desired precedence. Thus, `3+(3*5)` equals 18. Careful developers should use parentheses to explicitly state their intentions.
- Caché SQL can be configured to support ANSI precedence of arithmetic operators. This is a system-wide configuration setting. When ANSI precedence is configured, the "*", "\", "/", and "#" operators have a higher precedence than the "+", "-", and "||" operators. Operators with a higher precedence are executed before operators with a lower precedence. Thus, `3+3*5` equals 18. You can use parentheses to override precedence when desired. Thus, `(3+3)*5` equals 30.

  You can configure precedence using the Management Portal. In the General SQL Settings, you select the **Apply ANSI Operator Precedence** check box, then press the **Save** button. Changing this SQL option takes effect immediately system-wide. Changing this option causes all cached queries to be purged system-wide.

  You can also configure precedence using the **$SYSTEM.SQL.SetANSIPrecedence()** method.

  Changing SQL precedence has no effect on ObjectScript. ObjectScript always follows strict left-to-right execution of arithmetic operators.

3.5.2 Precision and Scale

The *precision* (maximum number of digits present in the number) for a NUMERIC result for:

- addition or subtraction is determined using the following algorithm: resultprecision=max(scale1, scale2) + max(precision1–scale1, precision2–scale2)+1. If the calculated resultprecision is greater than 36, the precision value is set to 36.
- multiplication is determined using the following algorithm: resultprecision=min(36, precision1+precision2+1).
- division (value1 / value2) is determined using the following algorithm: resultprecision=min(36, precision1–scale1+scale2+max(6, scale1+precision2+1)).

The *scale* (maximum number of fractional digits) for a NUMERIC result for:

- addition or subtraction is determined using the following algorithm: resultscale=max(scale1, scale2).
- multiplication is determined using the following algorithm: resultscale=min(17, scale1+scale2).
- division (value1 / value2) is determined using the following algorithm: resultscale=min(17, max(6, scale1+precision2+1)).

For further details on data types, precision, and scale, refer to SQL Data Types.

3.5.3 Arithmetic and Trigonometric Functions

Caché SQL supports the following arithmetic functions:

ABS	Returns the absolute value of a numeric expression.
CEILING	Returns the smallest integer greater than or equal to a numeric expression.
EXP	Returns the log exponential (base e) value of a numeric expression.
FLOOR	Returns the largest integer less than or equal to a numeric expression.
GREATEST	Returns the largest number from a comma-separated list of numbers.
ISNUMERIC	Returns a boolean code specifying whether an expression is a valid number.
LEAST	Returns the smallest number from a comma-separated list of numbers.
LOG	Returns the natural log (base e) value of a numeric expression.
LOG10	Returns the base–10 log value of a numeric expression.
MOD	Returns the modulus value (remainder) of a division operation. Same as the # operator.
PI	Returns the numeric constant pi.
POWER	Returns the value of a numeric expression raised to a specified power.
ROUND	Returns a numeric expression rounded (or truncated) to a specified number of digits.
SIGN	Returns a numeric code specifying whether a numeric expression evaluates to positive, zero, or negative.
SQRT	Returns the square root of a numeric expression.
SQUARE	Returns the square of a numeric expression.
TRUNCATE	Returns a numeric expression truncated to a specified number of digits.

Caché SQL supports the following trigonometric functions.

ACOS	Returns the arc-cosine of a numeric expression.
ASIN	Returns the arc-sine of a numeric expression.
ATAN	Returns the arc-tangent of a numeric expression.
COS	Returns the cosine of a numeric expression.
COT	Returns the cotangent of a numeric expression.
SIN	Returns the sine of a numeric expression.
TAN	Returns the tangent of a numeric expression.

DEGREES	Converts radians to degrees.
RADIANS	Converts degrees to radians.

3.6 Relational Operators

A conditional expression evaluates to a boolean value. A conditional expression can use the following relational operators:

=	Equals operator.
!= <>	Does not equal operator. The two syntactical forms are functionally identical.
<	Less than operator.
>	Greater than operator.
<=	Less than or equal to operator.
>=	Greater than or equal to operator.

When comparing a table field value, these equality operators use the field's default collation. The Caché default is not case-sensitive. When comparing two literals, the comparison is case-sensitive.

Equality operators (equals, does not equal) should be avoided when comparing floating point numbers. Floating point numbers (data types classes %Library.Decimal and %Library.Double) are stored as binary values, not as fixed-precision numbers. During conversion, rounding operations may result in two floating point numbers that are intended to represent the same number not being precisely equal. Use less-than / greater-than tests to determine if two floating point numbers are "the same" to the desired degree of precision.

3.6.1 Contains and Follows Operators

Caché SQL also supports the Contains and Follows comparison operators:

[	Contains operator. Returns all values that contain the operand, including values equal to the operand. This operator uses EXACT (case-sensitive) collation. The inverse is `NOT[`.

- The Contains operator determines if a value contains a specified character or string of characters. It is case-sensitive.

- The %STARTWITH predicate condition determines if a value starts with a specified character or string of characters. It is not case-sensitive.

]	Follows operator. Returns all values that follow the operand in collation sequence. Excludes the operand value itself. This operator uses the field's default collation. The Caché default is not case-sensitive. The inverse is NOT].

For example, `SELECT Age FROM MyTable WHERE Age ] 88` returns 89 and greater, but also returns 9 because 9 is after 88 in the collation sequence. `SELECT Age FROM MyTable WHERE Age > 88` returns 89 and greater; it does not return 9. A string operand such as 'ABC' collates before any string that contains additional characters, such as 'ABCA'; therefore, to exclude the operand string from a] operator or a > operator you must specify the entire string. `Name ] 'Smith,John'` excludes 'Smith,John' but not 'Smith,John P.'

3.7 Logical Operators

SQL logical operators are used in condition expressions that are evaluated as being True or False. These conditional expressions are used in the **SELECT** statement WHERE and HAVING clauses, in the **CASE** statement WHEN clauses, in the **JOIN** statement ON clause, and the **CREATE TRIGGER** statement WHEN clause.

3.7.1 NOT Unary Operator

You can use the NOT unary logical operator to specify the logical inverse of a condition, as shown in the following examples:

```
SELECT Name,Age FROM Sample.Person
WHERE NOT Age>21
ORDER BY Age

SELECT Name,Age FROM Sample.Person
WHERE NOT Name %STARTSWITH('A')
ORDER BY Name
```

You can place the NOT operator before the condition (as shown above). Or you can place NOT immediately before a single-character operator; for example, `NOT<`, `NOT[`, and so forth. Note that there must be no space between NOT and the single-character operator it inverts.

3.7.2 AND and OR Operators

You can use the AND and OR logical operators between two operands in a series of two or more conditions. These logical operators can be specified by keyword or symbol:

AND	&
OR	!

Spaces are not required (though recommended for readability) between a symbol operator and its operand. Spaces are required before and after a keyword operator.

These logical operators can be used with the NOT unary logical operator, such as the following: `WHERE Age<65 & NOT Age=21`.

The following two examples use logical operators to schedule an assessment based on age. People between the ages of 20 and 40 are assessed every three years, people from 40 to 64 are assessed every two years, and those 65 and over are assessed every year. The examples give identical results; the first example uses keywords, the second uses symbols:

```
SELECT Name,Age FROM Sample.Person
WHERE Age>20
      AND Age<40 AND (Age # 3)=0
      OR Age>=40 AND (Age # 2)=0
      OR Age>=65
ORDER BY Age

SELECT Name,Age FROM Sample.Person
WHERE Age>20
      & Age<40 & (Age # 3)=0
      ! Age>=40 & (Age # 2)=0
      ! Age>=65
ORDER BY Age
```

Logical operators can be grouped using parentheses. This establishes a grouping level; evaluation proceeds from the lowest grouping level to the highest. In the first of the following examples, the AND condition is applied only to the second OR condition. It returns persons of any age from MA, and persons with age less than 25 from NY:

```
SELECT Name,Age,Home_State FROM Sample.Person
WHERE Home_State='MA' OR Home_State='NY' AND Age < 25
ORDER BY Age
```

Using parentheses to group conditions gives a different result. The following example returns persons from MA or NY whose age is less than 25:

```
SELECT Name,Age,Home_State FROM Sample.Person
WHERE (Home_State='MA' OR Home_State='NY') AND Age < 25
ORDER BY Age
```

- SQL execution uses short-circuit logic. If a condition fails, the remaining AND conditions will not be tested. If a condition succeeds, the remaining OR conditions will not be tested.
- However, because SQL optimizes WHERE clause execution, the order of execution of multiple conditions (at the same grouping level) cannot be predicted and should not be relied upon.

3.8 Comments

Caché SQL supports both single-line comments and multi-line comments. Comment text can contain any characters or strings, except, of course, the character(s) that indicate the end of the comment.

Note: Using Embedded SQL marker syntax (`&sql<marker>(...)<reversemarker>`) imposes a restriction on the contents of SQL comments. If you are using marker syntax, the comments within the SQL code may not contain the character sequence "`)<reversemarker>`". For further details, refer to The &sql Directive in the "Using Embedded SQL" chapter of this manual.

You can use the **preparse()** method to return an SQL DML statement stripped of comments. The **preparse()** method also replaces each query argument with a ? character and returns a %List structure of these arguments. The **preparse()** method in the following example returns a parsed version of the query, stripped of single-line and multi-line comments and whitespace:

ObjectScript

```
SET myq=4
SET myq(1)="SELECT TOP ? Name /* first name */, Age "
SET myq(2)="   FROM Sample.MyTable -- this is the FROM clause"
SET myq(3)="   WHERE  /* various conditions "
SET myq(4)="apply */ Name='Fred' AND Age > 21 -- end of query"
DO ##class(%SQL.Statement).preparse(.myq,.stripped,.args)
WRITE stripped,!
WRITE $LISTTOSTRING(args)
```

3.8.1 Single Line Comments

A single-line comment is specified by a two-hyphen prefix. A comment can be on a separate line, or can appear on the same line as SQL code. When a comment follows SQL code on the same line, at least one blank space must separate the code from the double-hyphen comment operator. A comment can contain any characters, including hyphens, asterisks, and slashes. The comment continues to the end of the line.

The following example contains multiple single-line comments:

SQL

```
-- This is a simple SQL query
-- containing -- (double hyphen) comments
SELECT TOP 10 Name,Age, -- Two columns selected
Home_State -- A third column
FROM Sample.Person -- Table name
-- Other clauses follow
WHERE Age > 20 AND -- Comment within a clause
Age < 40
ORDER BY Age,  -- Comment within a clause
Home_State
-- End of query
```

3.8.2 Multiple Line Comments

A multiple-line comment is specified by a /* opening delimiter and a */ closing delimiter. A comment can appear on one or more separate lines, or can begin or end on the same line as SQL code. A comment delimiter should be separated from SQL code by at least one blank space. A comment can contain any characters, including hyphens, asterisks and slashes, with the obvious exception of the */ character pair.

The following example contains several multiple-line comments:

```
/* This is
   a simple
   SQL query. */
SELECT TOP 10 Name,Age /* Two fields selected */
FROM Sample.Person  /* Other clauses
could appear here */ ORDER BY Age
/* End of query */
```

When commenting out Embedded SQL code, always begin the comment before the &sql directive or within the parentheses. The following example correctly comments out two the Embedded SQL code blocks:

ObjectScript

```
   SET a="default name",b="default age"
   WRITE "(not) Invoking Embedded SQL",!
   /*&sql(SELECT Name INTO :a FROM Sample.Person) */
   WRITE "The name is ",a,!
   WRITE "Invoking Embedded SQL (as a no-op)",!
   &sql(/* SELECT Age INTO :b FROM Sample.Person */)
   WRITE "The age is ",b
```

3.8.3 SQL Code Retained as Comments

Embedded SQL statements can be retained as comments in the .INT code version of routines. This is done by setting a configuration option as follows:

- Invoke the **$SYSTEM.SQL.SetRetainSQL()** method. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**, which displays the `Retain SQL as Comments` setting.
- Go to the Management Portal. From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **General SQL Settings** (**[System] > [Configuration] > [General SQL Settings]**). On this screen you can view the current setting of **Retains SQL Statements as Comments in .INT Code**. The default is "Yes".

Set this option to "Yes" to retain SQL statements as comments in the .INT code version of a routine. Setting this option to "Yes" also lists all non-% variables used by the SQL statements in the comment text. These listed variables should also be listed in the ObjectScript procedure's PUBLIC variable list and re-initialized using the NEW command. For further details, refer to Host Variables in the "Embedded SQL" chapter of this manual.

4

Identifiers

An identifier is the name of an SQL entity, such as a table, a view, a column (field), a schema, a table alias, a column alias, an index, a stored procedure, a trigger, or some other SQL entity. An identifier name must be unique within its context; for example, two tables in the same schema, or two fields within the same table cannot have the same name. However, two tables in different schemas, or two fields in different tables can have the same name. In most cases, the same identifier name can be used for SQL entities of different types; for example, a schema, a table in that schema, and a field in that table can all have the same name without conflict. However, a table and a view in the same schema cannot have the same name.

Identifiers follow a set of naming conventions, which may be further restricted according to the use of the identifier. Identifiers are not case-sensitive.

An identifier may be either a simple identifier or a delimited identifier. The Caché SQL default is to support both simple identifiers and delimited identifiers.

4.1 Simple Identifiers

A simple identifier has the following syntax:

```
simple-identifier ::= identifier-start { identifier-part }
   identifier-start ::= letter | % | _
   identifier-part ::=  letter | number | _ | @ | # | $
```

4.1.1 Naming Conventions

The *identifier-start* is the first character of an SQL identifier. It must be one of the following:

- An uppercase or lowercase letter. A letter is defined as any character that passes validation by the ObjectScript $ZNAME function; by default these are the uppercase letters A through Z (ASCII 65–90), the lowercase letters a through z (ASCII 97–122), and the letters with accent marks (ASCII 192–255, exclusive of ASCII 215 and 247). If you have installed the Unicode version of Caché, you can use any valid Unicode (16-bit) letter character within an SQL identifier. Simple identifiers are not case-sensitive (however, see below). By convention they are represented with initial capital letters.

  The Japanese locale does not support accented Latin letter characters in identifiers. Japanese identifiers may contain (in addition to Japanese characters) the Latin letter characters A-Z and a-z (65–90 and 97–122), and the Greek capital letter characters (913–929 and 931–937).

- An underscore (_).

- A percent sign (%). Caché names beginning with a % character (except those beginning with %Z or %z) are reserved as system elements and should not be used as identifiers. For further details, refer to the chapter "Rules and Guidelines for Identifiers" in the *Caché Programming Orientation Guide*.

The *identifier-part* is any of the subsequent characters of an SQL identifier. These remaining characters may consist of zero or more:

- Letters (including Unicode characters).
- Numbers. A number is defined as the digits 0 through 9.
- Underscores (_).
- At signs (@).
- Pound signs (#).
- Dollar signs ($).

Some symbol characters are also used as operators. In SQL, the # sign is used as the modulo operator. In SQL, the underscore character can be used to concatenate two strings; this usage is provided for compatibility with ObjectScript, the preferred SQL concatenation operator is ||. The interpretation of a symbol as an identifier character always take precedence over its interpretation as an operator. Any ambiguity concerning the correct parsing of a symbol character as an operator can be resolved by adding spaces before and after the operator.

A simple identifier cannot contain blank spaces or non-alphanumeric characters (other than those symbol characters specified above). The InterSystems SQL import tool removes blank spaces from imported table names.

Note: SQL cursor names do not follow identifier naming conventions. For details on cursor naming conventions, refer to the DECLARE statement.

Caché SQL includes reserved words that cannot be used as simple identifiers. For a list of these reserved words, see the "Reserved Words" section in the *Caché SQL Reference*; to test if a word is a reserved word use the **$SYSTEM.SQL.IsReservedWord()** method. However, a delimited identifier can be the same as an SQL reserved word.

Any identifier that does not follow these naming conventions must be represented as a delimited identifier within an SQL statement.

4.1.2 Case of Letters

Caché SQL identifiers by default are not case-sensitive. Caché SQL implements this by comparing identifiers after converting them to all uppercase letters. This has no effect on the actual case of the names being used. (Note that other implementations of SQL may handle case sensitivity of identifiers differently. For this reason, it is recommended that you avoid case-based identifiers.)

Note that cursor names and passwords in Caché SQL are case-sensitive.

4.1.3 Testing Valid Identifiers

Caché provides the **IsValidRegularIdentifier()** method of the %SYSTEM.SQL class, which tests whether a string is a valid identifier. It tests both for character usage and for reserved words. It also performs a maximum length test of 200 characters (this is an arbitrary length used to avoid erroneous input; it is not an identifier validation). The following ObjectScript example shows the use of this method:

ObjectScript

```
WRITE !,$SYSTEM.SQL.IsValidRegularIdentifier("Fred")
WRITE !,$SYSTEM.SQL.IsValidRegularIdentifier("%Fred#123")
WRITE !,$SYSTEM.SQL.IsValidRegularIdentifier("%#$@_Fred")
WRITE !,$SYSTEM.SQL.IsValidRegularIdentifier("_1Fred")
WRITE !,$SYSTEM.SQL.IsValidRegularIdentifier("%#$")

WRITE !,$SYSTEM.SQL.IsValidRegularIdentifier("1Fred")
WRITE !,$SYSTEM.SQL.IsValidRegularIdentifier("Fr ed")
WRITE !,$SYSTEM.SQL.IsValidRegularIdentifier("%alphaup")
```

The first three method calls return 1, indicating a valid identifier. The fourth and fifth method calls also return 1; these are valid identifiers, although they are not valid for use as table or field names. The last three method calls return 0, indicating an invalid identifier. Two of these are invalid because they violate the character rules — in these cases by beginning with a number or containing a blank. The final method call returns 0 because the specified string is a reserved word. Note that these rule tests are a minimum requirement; they do not certify an identifier as valid for all SQL uses.

This method can also be called as a stored procedure from ODBC or JDBC: %SYSTEM.SQL_IsValidRegularIdentifier("nnnn").

4.1.4 Namespace Names

A namespace name (also referred to as a database name) follows identifier naming conventions, with additional restrictions on punctuation characters and maximum length. For further details, refer to the CREATE DATABASE command.

A namespace name can be delimited identifier and can be the same as an SQL reserved word. However, the same namespace name punctuation restrictions apply to both simple identifiers and delimited identifiers.

4.1.5 Identifiers and Class Entity Names

SQL table names, view names, field names, index names, trigger names, and procedure names are used to generate corresponding persistent class entities by stripping out non-alphanumeric characters. The generated names of class entities and globals follow these translation rules.

Note: Namespace names and SQL schema names and corresponding package names *do not* follow these translation rules.

- Identifiers that differ only in their inclusion of punctuation characters are valid. Because class object names cannot include punctuation characters, Caché generates corresponding unique object names by stripping out all punctuation characters. If stripping out the punctuation characters of an identifier results in a non-unique class object name, Caché creates a unique name by replacing the last alphanumeric character with an incremented character suffix.

  For tables, views, fields, triggers, and procedure classmethod names, this is an integer suffix, beginning with 0. For example, `myname` and `my_name` generate myname and mynam0, adding `my#name` generates mynam1. If the number of generated unique names is larger than 10 (mynam9), additional names are generated by substituting a capital letter suffix, starting with A (mynamA). Because tables and views share the same name space, the same suffix counter is incremented for either a table or a view.

  For index names, this suffix is a capital letter, beginning with A. For example, `myindex` and `my_index` generate myindex and myindeA.

  If you have defined a name that ends in a suffix character (for example `my_name0` or `my_indexA`, Caché handles unique name generation by incrementing to the next unused suffix.

- Identifiers that have a punctuation character as the first character and a number as the second character are not valid for table names, view names, or procedure names. They are valid for field names and index names. If the first character of an SQL field name or index name is a punctuation character (% or _) and the second character is a number, Caché appends a lowercase “n” as the first character of the corresponding property name.

- Identifiers that consist *entirely* of punctuation characters, or begin with two underscore characters (__name), or contains two pound signs together (nn##nn) are generally invalid as SQL entity names and should be avoided in all contexts.

You can configure translation of specific characters in SQL identifiers to other characters in corresponding object identifiers. This facilitates the use of identifiers across environments where the rules for permitted identifier characters differ. Use the **SetDDLIdentifierTranslations()** method of the %SYSTEM.SQL class. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**.

When converting an SQL identifier to an Objects identifier at DDL runtime, the characters in the "From" string are converted to the characters in the "To" string.

4.1.5.1 Specifying SQL Names in a Class Definition

When you define a persistent class that projects SQL entities, the name of each SQL entity are the same as the name of its corresponding persistent class definition element. To make an SQL table, field, index, or procedure classmethod name different, use the SqlTableName, SqlFieldName, or SqlName (for an index) keyword to specify the SQL name within your class definition. For example:

Class Member

```
Property LName As %String [SqlFieldName = "Family#Name"];
```

Class Member

```
Index NameIdx As %String [SqlName = "FullNameIndex"];
```

4.1.6 Identifier Length Considerations

The maximum length for SQL identifiers is 128 characters. When Caché maps an SQL identifier to the corresponding object entity, it creates the corresponding property, method, query, or index name with a maximum of 96 characters. If two SQL identifiers are identical for the first 96 characters, Caché replaces the 96th character of the corresponding object name with an integer (beginning with 0) to create a unique name.

The maximum length for schema and table names is subject to additional considerations and restrictions. Refer to Table Names and Schema Names in the "Defining Tables" chapter of this manual.

4.2 Delimited Identifiers

A delimited identifier has the following syntax:

```
delimited-identifier ::= " delimited-identifier-part { delimited-identifier-part }
"
   delimited-identifier-part ::= non-double-quote-character | double-quote-symbol
   double-quote-symbol ::= ""
```

A delimited identifier is a unique identifier enclosed by delimiter characters. Caché SQL supports double quote characters (") as delimiter characters. Delimited identifiers are generally used to avoid the naming restrictions of simple identifiers.

Note that Caché SQL uses single quote characters (') to delimit literals. For this reason, delimited identifiers must be specified using double quote characters ("), and literals must be specified using single quote characters ('). For example, '7' is the numeric literal 7, but "7" is a delimited identifier.

4.2.1 Delimited Identifier Valid Names

A delimited identifier must be a unique name. Delimited identifiers are not case-sensitive; by convention, identifiers are represented with initial capital letters.

A delimited identifier can be the same as an SQL reserved word. Delimited identifiers are commonly used to avoid concerns about naming conflicts with SQL reserved words.

A delimited identifier may contain almost any printable character, including blank spaces. Most delimited identifier names cannot contain the following characters: comma (,), period (.), caret (^), and the two-character arrow sequence (->); however delimited identifier role names and user names may contain these characters. A delimited identifier classname may contain periods (.). No delimited identifier may begin with an asterisk (*). The following term cannot be used as a delimited identifier: `%vid`. Violating these naming conventions results in an SQLCODE -1 error.

A delimited identifier used as a table, schema, column, or index name must be able to be converted to a valid class entity name. Therefore, it must contain at least one alphanumeric character. A delimited identifier that begins with a number (or punctuation followed by a number) generates a corresponding class entity name with the letter "n" prefix.

The following example shows a query that makes use of delimited identifiers for both column and table names:

SQL

```
SELECT "My Field" FROM "My Table" WHERE "My Field" LIKE 'A%'
```

Note that the delimited identifiers are delimited with double quotes, and the string literal `A%` is delimited with single quotes.

When specifying a delimited identifier for a table name, you must separately delimit the table name and the schema name. Thus, `"schema"."tablename"` or `schema."tablename"` are valid identifiers, but `"schema.tablename"` is not a valid identifier.

4.2.2 Disabling Delimited Identifier Support

By default, support is enabled for delimited identifiers.

When delimited identifier support is disabled, characters within double quotes are treated as string literals.

You can set delimited identifier support system-wide using the following:

- The **SetDelimitedIdentifiers()** method of the %SYSTEM.SQL class. This method changes both the current system-wide value and the configuration file setting.
- The Management Portal. From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **General SQL Settings**. On this screen you can view and edit the current setting of **Support Delimited Identifiers**.
- The SET OPTION command with the SUPPORT_DELIMITED_IDENTIFIERS keyword.

To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**.

4.3 SQL Reserved Words

SQL includes a long list of reserved words that cannot be used as simple identifiers, but can be used as delimited identifiers. For a list of these reserved words, see the "Reserved Words" section in the *Caché SQL Reference*.

5

Defining Tables

This chapter describes how you create tables in Caché SQL. It discusses the following topics:

- Table names and schema names
- RowID (ID) field
- Primary Key
- RowVersion and Serial counter fields
- Defining a table by creating a persistent class definition
- Defining a table by using SQL DDL commands
- Defining tables by importing and executing a DDL script
- Defining a Table by Querying an Existing Table
- Defining an external table by using the Caché SQL Gateway
- Listing tables and their properties
- Listing a table's column names and column numbers

5.1 Table Names and Schema Names

You can create a table either by defining the table (using **CREATE TABLE**) or by defining a persistent class that is projected to a table:

- DDL: Caché uses the table name specified in **CREATE TABLE** to generate a corresponding persistent class name, and uses the specified schema name to generate a corresponding package name.
- Class Definition: Caché uses the persistent class name to generate a corresponding table name, and uses the package name to generate a corresponding schema name.

The correspondence between these two names may not be identical for the following reasons:

- Persistent classes and SQL tables follow different naming conventions. Different valid character and length requirements apply. Schema and table names are not case-sensitive; package and class names are case-sensitive. The system automatically converts a valid supplied name to a valid corresponding name, insuring that the generated name is unique.
- The match between a persistent class name and the corresponding SQL table name is a default. You can use the SqlTableName class keyword to supply a different SQL table name.

- The default schema name may not match the default package name. If you specify an unqualified SQL table name or persistent class name, the system supplies a default schema name or package name. The initial default schema name is SQLUser; the initial default package name is User.

5.1.1 Schema Name

A table, view, or stored procedure name is either qualified (`schema.name`) or unqualified (`name`).

- If you specify a schema name (qualified name), when creating a table, view, or stored procedure, the specified item is assigned to that schema. If the schema does not exist, Caché SQL creates the schema and assigns the table, view, or stored procedure to it.
- If you do not specify a schema name (unqualified name) when creating or referencing a table, view, or stored procedure, Caché SQL assigns a schema using either the default schema name or a schema search path, as described below.

5.1.2 Default Schema Name

- When performing a DDL operation, such as creating or deleting a table, view, trigger, or stored procedure, an unqualified name is supplied the default schema name. Schema search path values are ignored.
- When performing a DML operation, such as a SELECT, CALL, INSERT, UPDATE, or DELETE to access an existing table, view, or stored procedure, an unqualified name is supplied the schema name from the schema search path (if provided). If there is no schema search path, or the named item is not located using the schema search path, the default schema name is supplied.

The initial setting is to use the same default schema name for all namespaces (system-wide). You can set the same default schema name for all namespace, or set a default schema name for the current namespace.

If you create a table or other item with an unqualified name, Caché assigns it the default schema name, and the corresponding persistent class package name. If a named or default schema does not exist, Caché creates the schema (and package) and assigns the created item to the schema. If you delete the last item in a schema, Caché deletes the schema (and package). The following description of schema name resolution applies to table names, view names, and stored procedure names.

The initial system-wide default schema name is `SQLUser`. The corresponding persistent class package name is `User`. Therefore, either the unqualified table name Employee or the qualified table name SQLUser.Employee would generate the class `User.Employee`.

Because USER is a reserved word, attempting to specify a qualified name with the schema name of `User` (or any SQL Reserved Word) results in an SQLCODE -1 error.

To return the current default schema name, invoke the **$SYSTEM.SQL.DefaultSchema()** method:

ObjectScript

```
WRITE $SYSTEM.SQL.DefaultSchema()
```

Or use the following pre-processor macro:

ObjectScript

```
#include %occConstant
  WRITE $$$DefSchema
```

You can change the default schema name using either of the following:

- Go to the Management Portal. From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **SQL**. On this screen you can view and edit the current system-wide setting of **Default Schema**. This option sets the

default schema name system-wide. This system-wide setting can be overridden by a **SetDefaultSchema()** method value for the current namespace.

- The **$SYSTEM.SQL.SetDefaultSchema()** method. By default, this method sets the default schema name system-wide. However, by setting the Boolean 3rd argument = 1, you can set the default schema for just the current namespace. When different namespaces have different default schema names, the **DefaultSchema()** method returns the default schema name for the current namespace.

CAUTION: When you change the default SQL schema name, the system automatically purges all cached queries in all namespaces on the system. By changing the default schema name, you change the meaning of all queries that contain unqualified table, view, or stored procedure names. It is strongly recommended that the default SQL schema name be established at Caché installation and not subsequently modified.

The schema name is used to generate the corresponding class package name. Because these names have different naming conventions, they may not be identical.

You can create a schema with the same name as an SQL reserved word by setting this as the system-wide **Default Schema**, though this is not recommended. A default schema named `User` generates the corresponding class package name `Use0`, following the class naming uniqueness convention.

5.1.2.1 _CURRENT_USER Keyword

- As Default Schema Name: If you specify _CURRENT_USER as the default schema name, Caché assigns the user name of the currently logged-in process as the default schema name. The _CURRENT_USER value is the first part of the $USERNAME ObjectScript special variable value. If **$USERNAME** consists of a name and a system address (Deborah@TestSys), _CURRENT_USER contains only the name piece; this means that _CURRENT_USER can assign the same default schema name to more than one user. If the process has not logged in, _CURRENT_USER specifies SQLUser as the default schema name.

  If you specify _CURRENT_USER/*name* as the default schema name, where *name* is any string of your choice, then Caché assigns the user name of the currently logged-in process as the default schema name. If the process has not logged in, *name* is used as the default schema name. For example, _CURRENT_USER/HMO uses HMO as the default schema name if the process has not logged in.

  In **$SYSTEM.SQL.SetDefaultSchema()**, specify `"_CURRENT_USER"` as a quoted string.

- As Schema Name in DDL Command: If you specify _CURRENT_USER as the explicit schema name in a DDL statement, Caché replaces it with the current default schema name. For example, if the system-wide default schema is SQLUser, the command `DROP TABLE _CURRENT_USER.OldTable` drops SQLUser.OldTable. This is a convenient way to qualify a name to explicitly indicate that the system-wide default schema should be used. It is functionally identical to specifying an unqualified name. This keyword cannot be used in DML statements.

5.1.3 Schema Search Path

When accessing an existing table (or view, or stored procedure) for a DML operation, an unqualified name is supplied the schema name from the schema search path. Schemas are searched in the order specified and the first match is returned. If no match is found in the schemas specified in the search path, or no search path exists, the default schema name is used. (Note that the #import macro directive uses a different search strategy and does not “fall through” to the default schema name.)

- In Embedded SQL you can use the #sqlcompile path macro directive or the #import macro directive to supply a schema search path that Caché uses to resolve unqualified names. **#sqlcompile path** resolves an unqualified name with the first match encountered. **#import** resolves an unqualified name if there is exactly one match for all the schemas listed in the search path.

- The following example provides a search path containing two schema names:

  ObjectScript

  ```
  #sqlcompile path=Customers,Employees
  ```

  For further details, refer to "ObjectScript Macros and the Macro Preprocessor" in *Using Caché ObjectScript*.

- In Dynamic SQL you can use the %SchemaPath property to supply a schema search path that Caché uses to resolve unqualified table names. You can specify the %SchemaPath property directly or specify it as the second parameter of the %SQL.Statement **%New()** method. The following example provides a search path containing two schema names:

  ObjectScript

  ```
  SET tStatement = ##class(%SQL.Statement).%New(0,"Customers,Employees")
  ```

  For further details, refer to "Using Dynamic SQL" in *Using Caché SQL*.

- In SQL Shell you can set the PATH SQL Shell configuration parameter to supply a schema search path that Caché uses to resolve unqualified names.

If the unqualified name does not match any of the schemas specified in the schema search path or the default schema name, an SQLCODE -30 error is issued, such as the following: `SQLCODE: -30 Message: Table 'PEOPLE' not found within schemas: CUSTOMERS,EMPLOYEES,SQLUSER`.

5.1.4 Schema Naming Considerations

Schema names follow identifier conventions, with significant considerations concerning the use of non-alphanumeric characters. A schema name should not be specified as a delimited identifier. Attempting to specify "USER" or any other SQL reserved word as a schema name results in an SQLCODE -1 or -312 error. The INFORMATION_SCHEMA schema name and the corresponding INFORMATION.SCHEMA package name are reserved in all namespaces. Users should not create tables/classes within this schema/package.

When you issue a create operation, such as **CREATE TABLE**, that specifies a schema that does not yet exist, Caché creates the new schema. Caché uses the schema name to generate a corresponding package name. Because the naming conventions for schemas and their corresponding packages differ, the user should be aware of name conversion considerations for non-alphanumeric characters. These name conversion considerations are not the same as for tables:

- Initial character:
  - % (percent): Specify % as the first character of a schema name denotes the corresponding package as a system package, and all of its classes as system classes. This usage requires appropriate privileges; otherwise, this usage issues an SQLCODE -400 error with the %msg indicating a <PROTECT> error.
  - _ (underscore): If the first character of a schema name is the underscore character, this character is replaced by a lowercase "u" in the corresponding package name. For example, the schema name _MySchema generates the package name uMySchema.
- Subsequent characters:
  - _ (underscore): If any character other than the first character of a schema name is the underscore character, this character is replaced by a period (.) in the corresponding package name. Because a period is the class delimiter, an underscore divides a schema into a package and a sub-package. Thus My_Schema generates the package My containing the package Schema (My.Schema).
  - @, #, $ characters: If a schema name contains any of these characters, these characters are stripped from the corresponding package name. If stripping these characters would produce a duplicate package name, the stripped package name is further modified: the final character of the stripped schema name is replaced by a sequential

integer (beginning with 0) to produce a unique package name. Thus My@#$Schema generates package MySchema, and subsequently creating My#$Schema generates package MySchem0. The same rules apply to table name corresponding class names.

5.1.5 Platform-Specific Schema Names

When creating an ODBC-based query to run from Microsoft Excel via Microsoft Query on the Mac, if you choose a table from the list of those available, the generated query does not include the table's schema (equivalent to the package for a class). For example, if you choose to return all the rows of the Person table from the Sample schema (in the Samples namespace), the generated query is:

SQL

```
SELECT * FROM Person
```

Because Caché interprets an unqualified table name as being in the SQLUser schema, this statement either fails or returns data from the wrong table. To correct this, edit the query (on the SQL View tab) to explicitly refer to the desired schema. The query should then be:

```
SELECT * FROM Sample.Person
```

5.1.6 Listing Schemas

The INFORMATION.SCHEMA.SCHEMATA persistent class lists all schemas in the current namespace.

The following example returns all non-system schema names in the current namespace:

SQL

```
SELECT SCHEMA_NAME
FROM INFORMATION_SCHEMA.SCHEMATA WHERE NOT SCHEMA_NAME %STARTSWITH '%'
```

The left side of the Management Portal SQL interface allows you to view the contents of a schema (or multiple schemas that match a filter pattern). See Filtering Schema Contents for further details.

5.1.7 Table Naming Considerations

Every table has a unique name within its schema. A table has both an SQL table name and a corresponding persistent class name; these names differ in permitted characters, case-sensitivity, and maximum length. If defined using the SQL **CREATE TABLE** command, you specify an SQL table name that follows identifier conventions; the system generates a corresponding persistent class name. If defined as a persistent class definition, you must specify a name that contains only alphanumeric characters; this name is used as both the case-sensitive persistent class name and (by default) the corresponding non-case-sensitive SQL table name. The optional SqlTableName class keyword allows the user to specify a different SQL table name.

When you use the **CREATE TABLE** command to create a table, Caché uses the table name to generate a corresponding persistent class name. Because the naming conventions for tables and their corresponding classes differ, the user should be aware of name conversion considerations for non-alphanumeric characters:

- Initial character:
  - % (percent): % as the first character of a table name is reserved and should be avoided (see Identifiers). If specified, the % character is stripped from the corresponding persistent class name.

  - _ (underscore): If the first character of a table name is the underscore character, this character is stripped from the corresponding persistent class name. For example, the table name _MyTable generates the class name MyTable.
  - Numbers: The first character of a table name cannot be a number. If the first character of the table name is a punctuation character, the second character cannot be a number. This results in an SQLCODE -400 error, with a %msg value of "ERROR #5053: Class name 'schema.name' is invalid" (without the punctuation character). For example, specifying the table name _7A generates the %msg "ERROR #5053: Class name 'User.7A' is invalid".

- Subsequent characters:
  - Letters: A table name must include at least one letter. Either the first character of the table name or the first character after initial punctuation characters must be a letter. A character is a valid letter if it passes the $ZNAME test; **$ZNAME** letter validation differs for different locales. (Note that **$ZNAME** cannot be used validate SQL identifiers because an identifier can contain punctuation characters.)
  - _ (underscore), @, #, $ characters: If a table name contains any of these characters, these characters are stripped from the corresponding class name and a unique persistent class name is generated. Because generated class names do not include punctuation characters, it is not advisable to create table names that differ only in their punctuation characters.

- A table name must be unique within its schema. Attempting to create a table with a name that differs only in letter case from an existing table generates an SQLCODE -201 error.

  A view and a table in the same schema cannot have the same name. Attempting to do so results in an SQLCODE -201 error.

  You can determine if a table name already exists using the **$SYSTEM.SQL.TableExists()** method. You can determine if a view name already exists using the **$SYSTEM.SQL.ViewExists()** method. These methods also return the class name corresponding to the table or view name. The Management Portal SQL interface Catalog Details **Table Info** option displays the Class Name corresponding to the selected SQL table name.

Attempting to specify "USER" or any other SQL reserved word as a table name or schema name results in an SQLCODE -312 error. To specify an SQL reserved word as a table name or schema name, you can specify the name as a delimited identifier. If you use a delimited identifier to specify a table or schema name that contains non-alphanumeric characters, Caché strips out these non-alphanumeric characters when generating the corresponding class or package name.

The following table name length limits apply:

- Uniqueness: Caché performs uniqueness checking on the first 59 alphanumeric characters of the persistent class name. The corresponding SQL table name may be more than 59 characters long, but, when stripped of non-alphanumeric characters, it must be unique within this 59 character limit. Caché performs uniqueness checking on the first 189 characters of a package name.
- Recommended maximum length: as a general rule, a table name should not exceed 128 characters. A table name may be much longer than 96 characters, but table names that differ in their first 96 alphanumeric characters are much easier to work with.
- Combined maximum length: a package name and its persistent class name (when added together) cannot exceed 220 characters. This includes the default schema (package) name (if no schema name was specified) and the dot character separating the package name and class name. A combined schema and table name can be longer than 220 characters when the characters in excess of 220 are stripped out when the table name is converted to the corresponding persistent class name.

For further details on table names, refer to the **CREATE TABLE** command in the *Caché SQL Reference*. For further details, on classes refer to "Caché Classes" in the *Using Caché Objects* manual.

5.2 RowID Field

In SQL, every record is identified by a unique integer value, known as the RowID. In InterSystems SQL you do not need to specify a RowID field. When you create a table and specify the desired data fields, a RowID field is automatically created. This RowID is used internally, but is not mapped to a class property. By default, its existence is only visible when a persistent class is projected to an SQL table. In this projected table, an additional RowID field appears. By default, this field is named "ID" and is assigned to column 1.

By default, when a table is populated with data, Caché assigns sequential positive integers to this field, starting with 1. The RowID data type is INTEGER (%Library.Integer). The values generated for the RowID have the following constraints: Each value is unique. The NULL value is not permitted. Collation is EXACT. By default, values are not modifiable.

By default, Caché names this field "ID". However this field name is not reserved. The RowID field name is re-established each time the table is compiled. If the user defines a field named "ID", when the table is compiled Caché names the RowID as "ID1". If, for example, the user then uses **ALTER TABLE** to define a field named "ID1", the table compile renames the RowID as "ID2", and so forth. In a persistent class definition you can use the SqlRowIdName class keyword to directly specify the RowID field name for the table to which this class is projected. For these reasons, referencing the RowID field by name should be avoided.

Caché SQL provides the **%ID** pseudo-column name (alias) which always returns the RowID value, regardless of the field name assigned to the RowID. (Caché TSQL provides the $IDENTITY pseudo-column name, which does the same thing.)

ALTER TABLE cannot modify or delete the RowID field definition.

When records are inserted into the table, Caché assigns each record an integer ID value. RowID values always increment. They are not reused. Therefore, if records have been inserted and deleted, the RowID values will be in ascending numeric sequence, but may not be numerically contiguous.

- By default, a table defined using **CREATE TABLE** performs ID assignment using $SEQUENCE, allowing for the rapid simultaneous populating of the table by multiple processes. When **$SEQUENCE** is used to populate the table, a sequence of RowID values is allocated to a process, which then assigns them sequentially. Because concurrent processes are assigning RowIDs using their own allocated sequences, records inserted by more than one process cannot be assumed to be in the order of insert.

  You can configure Caché to perform ID assignment using $INCREMENT by setting the **SetDDLUseSequence()** method; to determine the current setting, call the **$SYSTEM.SQL.CurrentSettings()** method.

- By default, a table defined by creating a persistent class performs ID assignment using $INCREMENT. In a persistent class definition, the IdFunction storage keyword can be set to either `sequence` or `increment`; for example, `<IdFunction>sequence</IdFunction>`.

In a persistent class definition, the IdLocation storage keyword global (for example, for persistent class Sample.Person: `<IdLocation>^Sample.PersonD</IdLocation>`) contains the highest assigned value of the RowID counter. (This is the highest integer assigned to a record, not the highest allocated to a process.) Note that this RowID counter value may no longer correspond to an existing record. To determine if record with a specific RowID value exists, invoke the table's **%ExistsId()** method.

The RowID counter is reset by the TRUNCATE TABLE command. It is not reset by a **DELETE** command, even when the **DELETE** command deletes all rows in the table. If no data has been inserted into the table, or **TRUNCATE TABLE** has been used to delete all table data, the `^Sample.PersonD` global is undefined.

By default, RowID values are not user-modifiable. Modifying RowID values can have serious consequences and should only be done in very specific cases and with extreme caution. The *Config.SQL.AllowRowIDUpdate* property allows RowID values to be user-modifiable.

5.2.1 RowID Based on Fields

By defining a persistent class that projects a table, you can define the RowID to have values from a field or a combination of fields. To do this, specify an index with the IdKey index keyword. For example, a table can have a RowID whose values are the same as the values of the PatientName field by specifying the index definition `IdxId On PatientName [IdKey];`, or the combined values of the PatientName and SSN fields by specifying the index definition `IdxId On (PatientName,SSN) [IdKey];`.

- A RowID based on fields is less efficient than a RowId that takes system-assigns sequential positive integers.
- On INSERT: The values specified for the field or combination of fields that make up the RowId must be unique. Specifying a non-unique value generates an SQLCODE -119 “UNIQUE or PRIMARY KEY constraint failed uniqueness check upon INSERT”.
- On UPDATE: By default, the values of each of the fields that makes up the RowId are non-modifiable. Attempting to modify the value of one of these fields generates an SQLCODE -107 “Cannot UPDATE RowID or RowID based on fields”.

When a RowID is based on multiple fields, the RowID value is the values of each of its component fields joined by the || operator. For example, `Ross,Betsy||123-45-6789`. The maximum length of a RowID that is based on multiple fields defaults to 254.

For further details, refer to Primary Key.

5.2.2 RowID Hidden?

- When using **CREATE TABLE** to create a table, the RowID is hidden by default. A hidden field is not displayed by `SELECT *` and is PRIVATE. When you create a table you can specify the %PUBLICROWID keyword to make the RowID not hidden and public. This optional %PUBLICROWID keyword can be specified anywhere in the **CREATE TABLE** comma-separated list of table elements. It cannot be specified in **ALTER TABLE**. For further details, refer to The RowID Field and %PUBLICROWID in the **CREATE TABLE** reference page.
- When creating a persistent class that projects as a table, the RowID is not hidden by default. It is displayed by `SELECT *` and is PUBLIC. You can define a persistent class with a RowID that is hidden and PRIVATE by specifying the class keyword SqlRowIdPrivate.

A RowID used as a foreign key reference must be public.

By default, a table with a public RowID cannot be used as either source or destination table to copy data into a duplicate table using `INSERT INTO Sample.DupTable SELECT * FROM Sample.SrcTable`. Refer to INSERT for further details.

You can display whether the RowID is hidden using the Management Portal SQL interface **Catalog Details** Fields listing **Hidden** column.

You can use the following program to return whether a specified field (in this example, ID) is hidden:

ObjectScript

```
SET myquery = "SELECT FIELD_NAME,HIDDEN FROM %Library.SQLCatalog_SQLFields(?) WHERE FIELD_NAME='ID'"

SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

To list the field names (hidden and non-hidden) in a table, refer to “Column Names and Numbers”.

5.3 Primary Key

Caché provides two ways to uniquely identify a row in a table: the RowID and the Primary Key.

The optional primary key is a meaningful value an application can use to uniquely identify a row in the table (for example in joins). A primary key can be user-specified data field or can be a combination of more than one data fields. Primary key values must be unique, but are not required to be integer values. The RowID is an integer value used internally to identify a row in the table. Often the primary key is a value generated by the application, while the RowID is a unique integer value generated by Caché.

The system automatically creates a Master Map to access rows of data using the RowID field. If you define a primary key field, the system automatically creates and maintains a primary key index.

Obviously, the duality of having two different fields and indexes to identify rows might not necessarily be a good thing. You can resolve to a single row identifier and index in either of two ways:

- Use the application-generated primary key value as the IDKEY. You can do this by identifying the primary key index in the class definition with both keywords PrimaryKey and IdKey (you can also do that from DDL if you set the PKey is IDKey flag for this purpose). This makes the primary key index the table's Master Map. Thus, the primary key will be used as the main internal address for the rows. This can be less efficient if the primary key consists of more than one field, or if the primary key values are not integers.
- Do not use an application-generated primary key value, but instead use the system-generated RowID integer within the application as the application-used primary key (for example in joins). The advantage of doing this is that the integer RowID lends itself to more efficient processing, including use of bitmap indices.

Depending on the nature of the application, you may wish to resolve to a single row identifier and index or to have separate indexes for the application-generated primary key and the system-generated RowID.

5.4 RowVersion and Serial Counter Fields

InterSystems SQL supports two special-purpose data types for automatically-incrementing counter values:

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

5.4.1 RowVersion Field

The RowVersion field is an optional user-defined field that provides row-level version control, allowing you to determine the order in which changes were made to the data in each row namespace-wide. Caché maintains a namespace-wide counter, and assigns a unique incremental positive integer to this field each time the row data is modified (insert, update, or %Save). Because this counter is namespace-wide, an operation on one table with a ROWVERSION field sets the increment point for the ROWVERSION counter that is used for all other tables with a ROWVERSION field in the same namespace.

You create a RowVersion field by specifying a field of data type ROWVERSION. You can only specify one ROWVERSION data type field per table. Attempting to create a table with more than one ROWVERSION field results in a 5320 compilation error.

This field can have any name and can appear in any column position. The ROWVERSION (%Library.RowVersion) data type maps to BIGINT (%Library.BigInt).

This field receives a positive integer from an automatically incremented counter, starting with 1. This counter increments whenever data in any ROWVERSION-enabled table is modified by an insert, update, or %Save operation. The incremented value is recorded in the ROWVERSION field of the row that has been inserted or updated.

A namespace can contain tables with a RowVersion field and tables without this field. Only data changes to tables that have a RowVersion field increment the namespace-wide counter.

When a table is populated with data, Caché assigns sequential integers to this field for each inserted row. If you use **ALTER TABLE** to add a ROWVERSION field to a table that already contains data, this field is created as NULL for pre-existing fields. Any subsequent insert or update to the table assigns a sequential integer to the RowVersion field for that row. This field is read-only; attempting to modify a RowVersion value generates an SQLCODE -138 error: `Cannot INSERT/UPDATE a value for a read only field`. Therefore, a RowVersion field is defined as unique and non-modifible, but not required or non-null.

RowVersion values always increment. They are not reused. Therefore, inserts and updates assign unique RowVersion values in temporal sequence. Delete operations remove numbers from this sequence. Therefore, RowVersion values may not be numerically contiguous.

This counter is never reset. Deleting all table data does not reset the RowVersion counter. Even dropping all tables in the namespace that contain a ROWVERSION field does not reset this counter.

The RowVersion field should not be included in a unique key or primary key. The RowVersion field cannot be part of an IDKey index.

The RowVersion field is not hidden (it is displayed by `SELECT *`).

This is shown in the following example of three tables in the same namespace.

1. Create Table1 and Table3, each of which has a ROWVERSION field, and Table2 that does not have a ROWVERSION field.
2. Insert ten rows into Table1. The ROWVERSION values of these rows are the next ten counter increments. Since the counter has not previously been used, they are 1 through 10.
3. Insert ten rows into Table2. Because Table2 does not have a ROWVERSION field, the counter is not incremented.
4. Update a row of Table1. The ROWVERSION values for this row is changed to the next counter increment (11 in this case).
5. Insert ten rows into Table3. The ROWVERSION values of these rows are the next ten counter increments (12 through 21).
6. Update a row of Table1. The ROWVERSION values for this row is changed to the next counter increment (22 in this case).
7. Delete a row of Table1. The ROWVERSION counter is unchanged.
8. Update a row of Table3. The ROWVERSION values for this row is changed to the next counter increment (23 in this case).

5.4.2 Serial Field

You can use the SERIAL data type (%Library.Counter in a persistent class table definition) to specify one or more optional integer counter fields to record the order of inserts of records into a table. By default, this field receives a positive integer from an automatically incremented table counter whenever a row is inserted into the table. However, a user can specify an integer value for this field during an insert, overriding the table counter default.

- If an INSERT does not specify a value for the counter field, it automatically receives a positive integer counter value. Counting starts from 1. Each successive value is an increment of 1 from the highest allocated counter value for this field.
- If an INSERT specifies an integer value for the counter field, the field receives that value. It can be a positive or negative integer value, can be lower or higher than the current counter value, and can be an integer already assigned to this field. If this value is higher than any assigned counter value, it sets the increment starting point for the next automatically assigned counter to that value.

Attempting to UPDATE a counter field value results in an SQLCODE -105 error.

This counter is reset to 1 by the TRUNCATE TABLE command. It is not reset by a **DELETE** command, even when the **DELETE** command deletes all rows in the table.

5.5 Defining a Table by Creating a Persistent Class

The primary way to define tables within Caché is to use Studio to create persistent class definitions. When these classes are saved and compiled within the Caché database, they automatically projects to a relational table that corresponds to the class definition: each class represents a table; each property represents a column, and so on. The maximum number of properties (columns) definable for a class (table) is 1000.

For example, the following defines the persistent class MyApp.Person:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String(MAXLEN=50) [Required];
Property SSN As %String(MAXLEN=15) [InitialExpression = "Unknown"];
Property DateOfBirth As %Date;
Property Sex As %String(MAXLEN=1);
}
```

When compiled, this creates the MyApp.Person persistent class and the corresponding SQL table, `Person` within the `MyApp` schema. For details on how to perform these operations, refer to "Defining and Compiling Classes" in the *Defining and Using Classes* manual.

In this example, the package name MyApp is specified. When defining a persistent class, an unspecified package name defaults to User; this corresponds to the default SQL schema name SQLUser. For example, defining a table named Students as a persistent class creates the class User.Students, and the corresponding SQL schema.table name SQLUser.Students.

In this example, the persistent class name Person is the default SQL table name. You can use the SqlTableName class keyword to supply a different SQL table name.

The same MyApp.Person table could have been defined using the DDL **CREATE TABLE** statement, specifying the SQL schema.table name. Successful execution of this SQL statement generates a corresponding persistent class with package name MyApp and class name Person:

SQL

```
CREATE TABLE MyApp.Person (
    Name VARCHAR(50) NOT NULL,
    SSN VARCHAR(15) DEFAULT 'Unknown',
    DateOfBirth DATE,
    Sex VARCHAR(1)
)
```

CREATE TABLE does not specify an explicit StorageStrategy in the corresponding class definition. It instead takes the defined default storage strategy.

By default, **CREATE TABLE** specifies the Final class keyword in the corresponding class definition, indicating that it cannot have subclasses.

For an introduction to how the object view of the database corresponds to the relational view, see "Introduction to the Default SQL Projection" in the chapter "Introduction to Persistent Objects" of *Using Caché Objects*.

Note that a persistent class definition such as the one shown above creates the corresponding table when it is compiled, but this table definition cannot be modified or deleted using SQL DDL commands (or by using the Management Portal Drop action), which give you the message "`DDL not enabled for class 'schema.name'...`"). You must specify [DdlAllowed] in the table class definition to permit these operations:

```
Class MyApp.Person Extends %Persistent [DdlAllowed]
```

You can specify %Populate in the class definition to enable automatic populating of the table with test data.

```
Class MyApp.Person Extends (%Persistent,%Populate) [DdlAllowed]
```

This provides a **Populate()** method for the class. Running this method populates the table with ten rows of test data.

5.5.1 Defining Data Value Parameters

Every property (field) definition must specify a data type class, which specifies the class that the property is based on. A specified data type limits a field's allowed data values to that data type. When defining a persistent class that projects to a table you must specify this data type using a class in the %Library package. This class can be specified as either %Library.Datatype or as %Datatype.

Many data type classes provide parameters that allow you to further define allowed data values. These parameters are specific to individual data types. The following are some of the more common data definition parameters:

- Data Value Physical Limits
- Permitted Data Values: Enumerated or Pattern Match
- Unique Data Values by defining a Unique index
- Computed Data Values by defining SqlComputeCode

5.5.1.1 Data Value Limits

For numeric data types, you can specify MAXVAL and MINVAL parameters to limit the range of allowed values. By definition, a numeric data type has a maximum supported values (positive and negative). You can use MAXVAL and MINVAL to further limit the allowed range.

For string data types, you can specify a MAXLEN and MINLEN parameters to limit the allowed length (in characters). By definition, a string data type has a maximum supported length. You can use MAXLEN and MINLEN to further limit the allowed range. By default, a data value that exceeds MAXLEN generates a field validation error: SQLCODE -104 for **INSERT** or SQLCODE -105 for **UPDATE**. You can specify TRUNCATE=1 to permit string data values that exceed MAXLEN; the specified string is truncated to the MAXLEN length.

5.5.1.2 Permitted Data Values

You can limit the actual data values in two ways:

- A list of allowed values (Enumerated Values with VALUELIST and DISPLAYLIST).
- A match pattern for allowed values (PATTERN).

Enumerated Values

Defining a table as a persistent class allows you to define properties (fields) that can only contain certain specified values. This is done by specifying the VALUELIST parameter. VALUELIST (which specifies a list of logical storage values) is commonly used with DISPLAYLIST (which specifies a list of corresponding display values). Both lists begin with the list delimiter character. Several data types can specify VALUELIST and DISPLAYLIST. The following example defines two properties with enumerated values:

Class Definition

```
Class Sample.Students Extends %Persistent
{
Property Name As %String(MAXLEN=50) [Required];
Property DateOfBirth As %Date;
Property ChoiceStr As %String(VALUELIST=",0,1,2",DISPLAYLIST=",NO,YES,MAYBE");
Property ChoiceODBCStr As %EnumString(VALUELIST=",0,1,2",DISPLAYLIST=",NO,YES,MAYBE");
}
```

If VALUELIST is specified, an **INSERT** or **UPDATE** can only specify one of the values listed in VALUELIST, or be provided with no value (NULL). VALUELIST valid values are case-sensitive. Specifying a data value that doesn't match the VALUELIST values results in a field value failed validation error: SQLCODE -104 for **INSERT** or SQLCODE -105 for **UPDATE**.

The %String and the %EnumString data types behave differently when displayed in ODBC mode. Using the example above, when displayed in Logical mode, both ChoiceStr and ChoiceODBCStr display their VALUELIST values. When displayed in Display mode, both ChoiceStr and ChoiceODBCStr display their DISPLAYLIST values. When displayed in ODBC mode, ChoiceStr displays VALUELIST values; ChoiceODBCStr displays DISPLAYLIST values.

Pattern Match for Values

Several data types can specify a PATTERN parameter. PATTERN restricts allowed data values to those that match the specified ObjectScript pattern, specified as a quoted string with the leading question mark omitted. The following example defines a property with a pattern:

Class Definition

```
Class Sample.Students Extends %Persistent
{
Property Name As %String(MAXLEN=50) [Required];
Property DateOfBirth As %Date;
Property Telephone As %String(PATTERN = "3N1""-""3N1""-""4N");
}
```

Because a pattern is specified as a quoted string, literals specified in the pattern need to have their enclosing quotes doubled. Note that pattern matching is applied before MAXLEN and TRUNCATE. Therefore, if you are specifying a pattern for a string that may exceed MAXLEN and be truncated, you may wish to end the pattern with "`.E`" (an unlimited number of trailing characters of any type).

A data value that does not match PATTERN generates a field validation error: SQLCODE -104 for **INSERT** or SQLCODE -105 for **UPDATE**.

5.5.1.3 Unique Values

CREATE TABLE allows you to define a field as UNIQUE. This means that every field value is a unique (non-duplicate) value.

Defining a table as a persistent class does not support a corresponding uniqueness property keyword. Instead, you must define both the property and a unique index on that property. The following example provides for a unique Num value for each record:

Class Definition

```
Class Sample.CaveDwellers Extends %Persistent [ DdlAllowed ]
{
Property Num As %Integer;
Property Troglodyte As %String(MAXLEN=50);
Index UniqueNumIdx On Num [ Type=index,Unique ];
}
```

The index name follows the naming conventions for properties. The optional Type keyword specifies the index type. The Unique keyword defines the property (field) as unique.

Having a unique value field is necessary for using the INSERT OR UPDATE statement.

5.5.1.4 Computed Values

The following class definition example defines a table that includes a field (Birthday) that uses SqlComputed to compute its value when you initially set the DateOfBirth field value and SqlComputeOnChange to recompute its value when you update the DateOfBirth field value. The Birthday field value includes the current timestamp to record when this field value was computed/recomputed:

Class Definition

```
Class Sample.MyStudents Extends %Persistent [DdlAllowed]
{
  Property Name As %String(MAXLEN=50) [Required];
  Property DateOfBirth As %Date;
  Property Birthday As %String
          [ SqlComputeCode = {SET {Birthday}=$PIECE($ZDATE({DateOfBirth},9),",")_
                              " changed: "_$ZTIMESTAMP},
                              SqlComputed, SqlComputeOnChange = DateOfBirth ];
}
```

Note that an **UPDATE** to DateOfBirth that specifies the existing DateOfBirth value does not recompute the Birthday field value. For the corresponding SQL code, refer to the COMPUTECODE section of the **CREATE TABLE** reference page.

For reference material on class property keywords, refer to the "Property Keywords" chapter of *Class Definition Reference*.

5.5.2 Embedded Object (%SerialObject)

You can simplify the structure of a persistent table by referencing an embedded serial object class that defines properties. For example, you want the MyData.Person to contain address information, consisting of street, city, state, and postal code. Rather than specifying these properties in MyData.Person, you can define a serial object (%SerialObject) class that defines these properties, and then in MyData.Person specify a single *Home* property that references that embedded object. This is shown in the following class definitions:

```
Class MyData.Person Extends (%Persistent) [ DdlAllowed ]
{  Property Name As %String(MAXLEN=50);
   Property Home As MyData.Address;
   Property Age As %Integer;
}


Class MyData.Address Extends (%SerialObject)
{  Property Street As %String;
   Property City As %String;
   Property State As %String;
   Property PostalCode As %String;
 }
```

You cannot access the data in a serial object property directly, you must access them through a persistent class/table that references it:

- To refer to an individual serial object property from the persistent table, use an underscore. For example, `SELECT Name, Home_State FROM MyData.Person` returns the State serial object property value as a string. Serial object property values are returned in the order specified in the query.
- To refer to all of the serial object properties from the persistent table, specify the referencing field. For example, `SELECT Home FROM MyData.Person` returns values of all of the MyData.Address properties as a %List structure. Serial object property values are returned in the order specified in the serial object: Home_Street, Home_City, Home_State, Home_PostalCode. In the Management Portal SQL interface Catalog Details, this referencing field is referred to as a **Container** field. It is a **Hidden** field, and therefore not returned by `SELECT *` syntax.
- A `SELECT *` for a persistent class returns all of the serial object properties individually, including nested serial objects. For example, `SELECT * FROM MyData.Person` returns Age, Name, Home_City, Home_PostalCode, Home_State, and Home_Street values (in that order); it does not return the Home %List structure value. Serial object property values are returned in collation sequence. `SELECT *` first lists all of the fields in the persistent class in collation sequence (commonly alphabetical order), followed by the nested serial object properties in collation sequence.

Note that an embedded serial object does not have to be in the same package as the persistent table that references it.

Defining embedded objects can simplify persistent table definitions:

- A persistent table can contain multiple properties that reference different records in the same embedded object. For example, the MyData.Person table can contain a *Home* and an *Office* property, both of which reference the MyData.Address serial object class.
- Multiple persistent tables can reference instances of the same embedded object. For example, the MyData.Person table *Home* property and the MyData.Employee *WorkPlace* property can both reference the MyData.Address serial object class.
- An embedded object can reference another embedded object. For example, the MyData.Address embedded object contains the *Phone* property that references the MyData.Telephone embedded object, containing CountryCode, AreaCode, and PhoneNum properties. From the persistent class you use multiple underscores to refer to a nested serial object property, for example Home_Phone_AreaCode.

For further details, refer to Introduction to Serial Objects in *Defining and Using Classes*.

For information on creating an index for a serial object property, refer to Indexing an Embedded Object (%SerialObject) Property.

5.5.3 Class Methods

You can specify class methods as part of a table definition, as shown in the following example:

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String(MAXLEN=50) [Required];
Property SSN As %String(MAXLEN=15) [InitialExpression = "Unknown"];
Property DateOfBirth As %Date;
Property Sex As %String(MAXLEN=1);
ClassMethod Numbers() As %Integer [ SqlName = Numbers, SqlProc ]
  {
   QUIT 123
  }
}
```

In a **SELECT** query you can invoke this method as follows:

SQL

```
SELECT Name,SSN,MyApp.Numbers() FROM MyApp.Person
```

5.6 Defining a Table by Using DDL

You can define tables in Caché SQL using standard DDL commands:

Table 5–1: Available DDL Commands in Caché SQL

ALTER Commands	CREATE Commands	DROP Commands
ALTER TABLE	CREATE TABLE	DROP TABLE
ALTER VIEW	CREATE VIEW	DROP VIEW
	CREATE INDEX	DROP INDEX
	CREATE TRIGGER	DROP TRIGGER

These are described in the *Caché SQL Reference*.

You can execute DDL commands in a variety of ways, including:

- Using Dynamic SQL.
- Using Embedded SQL.
- Using a DDL script file.
- Using ODBC calls.
- Using JDBC calls.

5.6.1 Using DDL in Embedded SQL

Within an ObjectScript method or routine, you can use embedded SQL to invoke DDL commands.

For example, the following method creates a TEST.EMPLOYEE table:

Class Member

```
ClassMethod CreateTable() As %String
{
 &sql(CREATE TABLE TEST.EMPLOYEE (
    EMPNUM              INT NOT NULL,
    NAMELAST            CHAR (30) NOT NULL,
    NAMEFIRST           CHAR (30) NOT NULL,
    STARTDATE           TIMESTAMP,
    SALARY              MONEY,
    ACCRUEDVACATION     INT,
    ACCRUEDSICKLEAVE    INT,
    CONSTRAINT EMPLOYEEPK PRIMARY KEY (EMPNUM)))

  IF SQLCODE=0 {WRITE "Table created"  RETURN "Success"}
  ELSEIF SQLCODE=-201 {WRITE "Table already exists"  RETURN SQLCODE}
  ELSE {WRITE "Serious SQL Error, returning SQLCODE"  RETURN SQLCODE_" "_%msg}
 }
```

When this method is invoked it attempts to create a TEST.EMPLOYEE table (as well as the corresponding TEST.EMPLOYEE class). If successful, the SQLCODE variable is set to 0. If unsuccessful, SQLCODE contains an SQL Error Code indicating the reason for the failure.

The most common reasons that a DDL command such as this one will fail are:

- SQLCODE -99 (Privilege Violation): This error indicates that you do not have permission to execute the desired DDL command. Typically this is because an application has not established who the current user is. You can do this programmatically using the **$SYSTEM.Security.Login()** method:

  ObjectScript

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

- SQLCODE -201 (Table or view name not unique): This error indicates that you are attempting to create a new table using the name of a table that already exists.

5.6.2 Using a Class Method to Execute DDL

Within ObjectScript or Caché Basic, you can use the Dynamic SQL %SQL.Statement object to prepare and execute DDL commands using Dynamic SQL.

The following example defines a class method to create a table using Dynamic SQL:

Class Definition

```
Class Sample.NewT
{
ClassMethod DefTable(user As %String,pwd As %String) As %Status [Language=cache]
  {
  DO ##class(%SYSTEM.Security).Login(user,pwd)
  SET myddl=2
  SET myddl(1)="CREATE TABLE Sample.MyTest "
  SET myddl(2)="(NAME VARCHAR(30) NOT NULL,SSN VARCHAR(15) NOT NULL)"
  SET tStatement=##class(%SQL.Statement).%New()
  SET tStatus=tStatement.%Prepare(.myddl)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset=tStatement.%Execute()
  IF rset.%SQLCODE=0 {WRITE "Created a table"}
  ELSEIF rset.%SQLCODE=-201 {WRITE "table already exists"}
  ELSE {WRITE "Unexpected error SQLCODE=",rset.%SQLCODE}
  }
}
```

This method is invoked as follows:

ObjectScript

```
DO ##class(Sample.NewT).DefTable("myname","mycachepassword")
```

As with the embedded SQL example, this method will fail if there is no current user logged in.

5.6.3 Defining Tables by Importing and Executing a DDL Script

You can import Caché SQL DDL script files using either the **Cache()** method interactively from a Terminal session, or the **DDLImport("CACHE")** method as a background job. This method can import and execute multiple SQL commands, enabling you to use a txt script file to define tables and views and populate them with data. For further details, refer to the Importing SQL Code chapter of this guide.

If you are migrating tables from another vendor's relational database to Caché, you may have one or more DDL scripts within text files. Caché provides several %SYSTEM.SQL methods to help load such DDL scripts into Caché. You can use the general-purpose **DDLImport()** method, or the %SYSTEM.SQL method for the specific vendor. The vendor-specific SQL is converted to Caché SQL and executed. Errors and unsupported features are recorded in log files. For further details, refer to the Code Migration: Importing non-Caché SQL in the "Importing SQL Code" chapter of this guide.

For example, to load an Oracle DDL file from the Caché command line:

1. Start a Terminal session using the **Terminal** command in the "Caché Cube" menu.

2. Switch to the namespace in which you wish to load the table definitions:

   ObjectScript

   ```
   ZN "MYNAMESPACE"
   ```

3. Invoke the desired DDL import method:

   ObjectScript

   ```
   DO $SYSTEM.SQL.Oracle()
   ```

   and follow the directions displayed at the terminal.

5.7 Defining a Table by Querying an Existing Table

You can use the **$SYSTEM.SQL.QueryToTable()** method to define and populate a new table based on an existing table. You specify a query and a new table name. The existing table name and/or the new table name can be qualified or unqualified. The query can contain JOIN syntax. The query can supply column name aliases that become the column names in the new table.

1. **QueryToTable()** copies the DDL definition of a existing table and assigns it the specified new table name. It copies the definitions of the fields specified in the query, including the data type, maxlength, and minval/maxval. It does not copy field data constraints, such as default value, required value, or unique value.

   If the query specifies SELECT * or SELECT %ID, the RowID field of the original table is copied as a non-required, non-unique data field of data type integer. **QueryToTable()** generates a unique RowID field for the new table. If the copied RowID is named ID, the generated RowID is named ID1.

   QueryToTable() creates a corresponding persistent class for this new table. The persistent class is defined as DdlAllowed. The owner of the new table is the current user.

   The new table is defined with %Cache storage = YES and Supports Bitmap Indices = YES, regardless of these settings in the source table.

   The only index created for the new table is the IDKEY index. No bitmap extent index is generated. Index definitions for the copied fields are not copied into the new table.

   References from a field to another table are not copied.

2. **QueryToTable()** then populates the new table with data from the fields selected by the query. It sets the table's Extent Size to 100,000. It estimates the IDKEY Block Count. Run Tune Table to set the actual Extent Size and Block Count, and the Selectivity and Average Field Size values for each field.

QueryToTable() both creates a table definition and populates the new table with data. If you wish to only create a table definition, specify a condition in the query WHERE clause that selects for no data rows. For example, `WHERE Age < 20 AND Age > 20`.

The following example copies the Name, and Age, fields from Sample.Person and creates an AVG(Age) field. These field definitions are used to create a new table named Sample.Youth. The method then Populates Sample.Youth with the Sample.Person data for those records where Age < 21. The AvgInit field contains the aggregate value for the selected records at the time that the table was created.

ObjectScript

```
  DO $SYSTEM.SQL.QueryToTable("SELECT Name,Age,AVG(Age) AS AvgInit FROM Sample.Person WHERE Age <
21","Sample.Youth",1,.errors)
```

5.8 External Tables

In Caché SQL, you can also have "external tables," tables that are defined within the Caché dictionary but are stored within an external relational database. External tables act as if they were native Caché tables: you can issue queries against them and perform INSERT, UPDATE, and DELETE operations. The access to external database is provided by the Caché SQL Gateway, which offers transparent connectivity using ODBC or JDBC. See "Using the Caché SQL Gateway" for more details.

5.9 Listing Tables

The INFORMATION.SCHEMA.TABLES persistent class displays information about all tables (and views) in the current namespace. It provides a number of properties including the schema and table names, the owner of the table, and whether you can insert new records. The TABLETYPE property indicates whether it is a base table or a view.

The following example returns the table type, schema name, table name, and owner for all tables and views in the current namespace:

SQL

```
SELECT Table_Type,Table_Schema,Table_Name,Owner FROM INFORMATION_SCHEMA.TABLES
```

The INFORMATION.SCHEMA.CONSTRAINTTABLEUSAGE persistent class displays one row for each Primary Key (explicit or implicit), Foreign Key, or Unique constraint defined for each table in the current namespace. INFORMATION.SCHEMA.KEYCOLUMNUSAGE displays one row for each field defined as part of one of these constraints for each table in the current namespace.

You can display much of the same information for a single table using the Catalog Details tab in the Management Portal SQL Interface.

5.10 Listing Column Names and Numbers

You can list all of the column names (field names) for a specified table in four ways:

- The **GetColumns()** method. This lists all column names and column numbers, including hidden columns. The ID (RowID) field may or may not be hidden. The x__classname column is always hidden; it is automatically defined unless the persistent class is defined with the Final class keyword.
- The Management Portal SQL interface (**System Explorer**, **SQL**) schema contents Catalog Details tab. This lists all column names and column numbers (including hidden columns) and other information, including data types and a flag indicating if a column is hidden.
- `SELECT TOP 0 * FROM` *tablename*. This lists all non-hidden column names in column number order. Note that because hidden columns can appear anywhere in the column number order, you cannot determine the column number by counting these non-hidden column names. For further details on Asterisk Syntax, refer to the SELECT command.

- The INFORMATION.SCHEMA.COLUMNS persistent class lists a row for each non-hidden column in each table or view in the current namespace. INFORMATION.SCHEMA.COLUMNS provides a large number of properties for listing characteristics of table and view columns. Note that ORDINALPOSITION is not the same as column number, because hidden fields are not counted. The **GetColumns()** method counts both hidden and non-hidden fields.

  The following example uses INFORMATION.SCHEMA.COLUMNS to list some of the column properties:

  SQL

```
SELECT TABLE_NAME,COLUMN_NAME,ORDINAL_POSITION,DATA_TYPE,CHARACTER_MAXIMUM_LENGTH,
       COLUMN_DEFAULT,IS_NULLABLE,UNIQUE_COLUMN,PRIMARY_KEY
       FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_SCHEMA='Sample'
```

5.10.1 The GetColumns() Method

To list the names of the columns in a table in column number order, you can use the **GetColumns()** method, as follows:

ObjectScript

```
SET stat=##class(%SYSTEM.SQL).GetColumns("Sample.Person",.byname,.bynum)
IF stat=1 {
  SET i=1
  WHILE $DATA(bynum(i)) { WRITE "name is ",bynum(i),"   col num is ",i,!
                          SET i=i+1 }
}
ELSE { WRITE "GetColumns() cannot locate specified table" }
```

GetColumns() lists all defined columns, including hidden columns. If a table references an embedded %SerialObject class, **GetColumns()** first lists all of the columns in the persistent class, including the property that references the %SerialObject, then lists all of the %SerialObject properties. This is shown in the following **GetColumns()** results:

```
name is ID   col num is 1
name is Age   col num is 2
name is Home   col num is 3
name is Name   col num is 4
name is x__classname   col num is 5
name is Home_City   col num is 6
name is Home_Phone   col num is 7
name is Home_Phone_AreaCode   col num is 8
name is Home_Phone_Country   col num is 9
name is Home_Phone_TNum   col num is 10
name is Home_PostalCode   col num is 11
name is Home_State   col num is 12
name is Home_Street   col num is 13
```

You can also use this method to determine the column number for a specified column name, as follows:

ObjectScript

```
SET stat=##class(%SYSTEM.SQL).GetColumns("Sample.Person",.byname)
IF stat=1 {
       WRITE "Home_State is column number ",byname("Home_State"),!  }
ELSE { WRITE "GetColumns() cannot locate specified table" }
```

6

Defining and Using Views

A view is a virtual table consisting of data retrieved from one or more physical tables by means of a **SELECT** statement or a **UNION** of several **SELECT** statements. Thus a view is a defined way of viewing existing table data.

Caché SQL supports the ability to define and execute queries on views. All views are either updateable or read-only, as described later in this chapter.

Note: You cannot create a view on data stored in a database that is mounted read-only.

You cannot create a view on data stored in an Informix table linked through an ODBC or JDBC gateway connection. This is because Caché query conversion uses subqueries in the FROM clause for this type of query; Informix does not support FROM clause subqueries. See the ISQL Migration Guide for InterSystems support for Informix SQL.

6.1 Creating a View

You can define views in several ways:

- Using the SQL CREATE VIEW command (either in a DDL script or via JDBC or ODBC).
- Using the Management Portal Create View interface.

A view name may be unqualified or qualified. An unqualified view name is a simple identifier: `MyView`. A qualified view name consists of two simple identifiers, a schema name and a view name, separated by a period: `MySchema.MyView`. View names and table names follow the same naming conventions and perform the same schema name resolution for unqualified names. A view and a table in the same schema cannot have the same name.

You can determine if a view name already exists using the **$SYSTEM.SQL.ViewExists()** method. This method also returns the class name that projected the view. You can determine if a table name already exists using the **$SYSTEM.SQL.TableExists()** method.

A view can be used to create a restricted subset of a table. For example, the following view restricts both the rows and columns of the original table that can be accessed thorough the view:

ObjectScript

```
   &sql(CREATE VIEW VSrStaff
        AS SELECT Name AS Vname,Age AS Vage
        FROM Sample.Person WHERE Age>75)
   IF SQLCODE=0 {WRITE "Created a view",!}
   ELSEIF SQLCODE=-201 {WRITE "View already exists",!}
   ELSE {WRITE "We have a problem: ",SQLCODE,! }

SELECT * FROM VSrStaff ORDER BY Vage
```

The following example creates a view based on all of the rows of the SalesPeople table, creating a new calculated value column TotalPay:

SQL

```
CREATE VIEW VSalesPay AS
    SELECT Name,(Salary + Commission) AS TotalPay
    FROM Sample.SalesPeople
```

6.1.1 Management Portal Create View Interface

You can create a view from the Management Portal. Go to the Caché Management Portal. From **System Explorer**, select **SQL** (**[System] > [SQL]**). Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces. Once you have selected a namespace, click the **Actions** drop-down list and select **Create View**.

This displays the Create a View window with the following fields:

- Schema: You can decide to include the view within an existing schema, or create a new schema. If you opt to select an existing schema, a drop-down list of existing schemas is provided. If you opt to create a new schema, you enter a schema name. In either case, if you omit the schema, Caché uses the system-wide default schema name.
- View Name: a valid view name. You cannot use the same name for a table and a view in the same schema.
- With Check Option: the options are READONLY, LOCAL, CASCADED.
- Grant all privilege on the view to _PUBLIC: if selected, this option gives all users execution privileges for this view. The default is to not give all users access to the view.
- View Text: you can specify the View Text in any of the following three ways:
  - Type a **SELECT** statement into the View Text area.
  - Use the Query Builder to create a **SELECT** statement, then press **OK** to supply this query to the View Text area.
  - If you select a Cached Query name (for example `%sqlcq.SAMPLES.cls4`) on the left side of the Management Portal SQL interface, then invoke **Create View**, this cached query is provided to the View Text area. Note that in the View Text area you must replace host variable references (question marks) with actual values before saving the view text.

6.2 Updateable Views

An updateable view is one on which you can perform INSERT, UPDATE, and DELETE operations. A view is considered updateable only if the following conditions are true:

- The FROM clause of the view's query contains only *one* table reference. This table reference must identify either an updateable base table or an updateable view.
- The value expressions within the SELECT list of the view's query must all be column references.
- The view's query must not specify GROUP BY, HAVING, or SELECT DISTINCT.
- The view is not a class query projected as a view.
- The view's class does not contain the class parameter *READONLY*=1 (true if the view definition contains a WITH READ ONLY clause).

6.2.1 The WITH CHECK Option

In order to prevent an INSERT or UPDATE operation on a view which would result in a row in the underlying base table which is not part of the derived view table, Caché SQL supports the WITH CHECK OPTION clause within a View definition. This clause can only be used with updateable views.

The WITH CHECK OPTION clause specifies that any INSERT or UPDATE operations on an updateable view must validate the resulting row against the WHERE clause of the view definition to make sure the inserted or modified row will be part of the derived view table.

For example, the following DDL statement defines an updateable GoodStudent view containing all Students with a high GPA (grade point average):

SQL

```
CREATE VIEW GoodStudent AS
    SELECT Name, GPA
      FROM Student
        WHERE GPA > 3.0
    WITH CHECK OPTION
```

Because the view contains a WITH CHECK OPTION, any attempt to INSERT or UPDATE a row in the GoodStudent view with a GPA value of 3.0 or less will fail (such a row would not represent a "good student").

There are two flavors of WITH CHECK OPTION:

- WITH LOCAL CHECK OPTION means that only the WHERE clause of the view specified in the INSERT or UPDATE statement is checked.
- WITH CASCADED CHECK OPTION (and WITH CASCADE CHECK OPTION) means that the WHERE clause of the view specified in the INSERT or UPDATE statement as well as ALL views on which that view is based are checked, regardless of the appearance or absence of other WITH LOCAL CHECK OPTION clauses in those view definitions.

The default is CASCADED if just WITH CHECK OPTION is specified.

During an UPDATE or INSERT, the WITH CHECK OPTION conditions are checked after all default values and triggered computed fields have been calculated for the underlying table's fields and before the regular table's validation (required fields, data type validation, constraints, and so on).

After the WITH CHECK OPTION validation passes, the INSERT or UPDATE operation continues as if the INSERT or UPDATE was performed on the base table itself. All constraints are checked, triggers pulled, and so on.

If the %NOCHECK option is specified on the INSERT or UPDATE statement, the WITH CHECK OPTION validation is not checked.

There are two SQLCODE values related to the WITH CHECK OPTION validation (the INSERT/UPDATE would have resulted in a row not existing in the derived view table):

- SQLCODE -136—View's WITH CHECK OPTION validation failed in INSERT.
- SQLCODE -137—View's WITH CHECK OPTION validation failed in UPDATE.

6.3 Read-only Views

A read-only view is one on which you cannot perform INSERT, UPDATE, and DELETE operations. Any view that does not meet the criteria for updateable views is a read-only view.

A view definition may specify a WITH READ ONLY clause to force it to be a read-only view.

If you attempt to compile/prepare an INSERT, UPDATE, or DELETE statement against a read-only view an SQLCODE -35 error is generated.

6.4 View ID: %VID

Caché assigns an integer view ID (%VID) to each row returned by a view or by a FROM clause subquery. Like table row ID numbers, these view row ID numbers are system-assigned, unique, non-null, non-zero, and non-modifiable. This %VID is commonly invisible to the user, and is only returned when explicitly specified. It is returned as data type INTEGER. Because %VID values are sequential integers, they are far more meaningful if the view returns ordered data; a view can only use an ORDER BY clause when it is paired with a TOP clause. The following Embedded SQL example creates a view named VSrStaff:

ObjectScript

```
&sql(CREATE VIEW VSrStaff
     AS SELECT TOP ALL Name AS Vname,Age AS Vage
     FROM Sample.Person WHERE Age>75
     ORDER BY Name)
IF SQLCODE=0 {WRITE "Created a view",!}
ELSEIF SQLCODE=-201 {WRITE "View already exists",!}
ELSE {WRITE "We have a problem: ",SQLCODE,! }
```

The following example returns all of the data defined by the VSrStaff view (using SELECT *) and also specifies that the view ID for each row should be returned. Unlike the table row ID, the view row ID is not displayed when using asterisk syntax; it is only displayed when explicitly specified in the SELECT:

```
SELECT *,%VID AS ViewID FROM VSrStaff
```

The %VID can be used to further restrict the number of rows returned by a SELECT from a view, as shown in the following example:

```
SELECT *,%VID AS ViewID FROM VSrStaff WHERE %VID BETWEEN 5 AND 10
```

Thus %VID can be used instead of TOP (or in addition to TOP) to restrict the number of rows returned by a query. Generally, a TOP clause is used to return a small subset of the data records; %VID is used to return most or all of the data records, returning records in small subsets. This feature may be useful, especially for porting Oracle queries (%VID maps easily to Oracle ROWNUM). However, the user should be aware of some performance limitations in using %VID, as compared to TOP:

- %VID does not perform time-to-first-row optimization. TOP optimizes to return the first row of data as quickly as possible. %VID optimizes to return the full data set as quickly as possible.
- %VID does not perform a limited sort (which is a special optimization performed by TOP) if the query specifies sorted results. The query first sorts the full data set, then restricts the return data set using %VID. TOP is applied before sorting, so the SELECT performs a limited sort involving only a restricted subset of rows.

To preserve time to first row optimization and limited sort optimization, you can use a FROM clause subquery with a combination of TOP and %VID. Specify the upper bound (in this case, 10) in the FROM subquery as the value of TOP,

rather than using TOP ALL. Specify the lower bound (in this case, >4) in the WHERE clause with %VID. The following example uses this strategy to return the same results as the previous view query:

```
SELECT *,%VID AS SubQueryID
   FROM (SELECT TOP 10 Name,Age
         FROM Sample.Person
         WHERE Age > 75
         ORDER BY Name)
   WHERE %VID > 4
```

6.5 Listing View Properties

The INFORMATION.SCHEMA.VIEWS persistent class displays information about all views in the current namespace. It provides a number of properties including the view definition, the owner of the view, and the timestamps when the view was created and last modified. These properties also include whether the view is updateable and if so, whether it was defined with a check option.

When specified in Embedded SQL, INFORMATION.SCHEMA.VIEWS requires the **#include %occInclude** macro preprocessor directive. This directive is not required for Dynamic SQL.

The *VIEWDEFINITION* property (SqlFieldName = VIEW_DEFINITION) returns as a string the view field names and the view's query expression for all views in the current namespace. For example,

SQL

```
SELECT View_Definition FROM INFORMATION_SCHEMA.VIEWS
```

returns strings such as: `"(vName,vAge) SELECT Name,Age FROM Sample.Person WHERE Age > 21"`. When issued from the Management Portal SQL Execute Query interface, display of this string is limited to the first 100 characters with whitespace and line breaks removed and (if necessary) an appended ellipsis (...) indicating truncated content. Otherwise, issuing this query returns a string of up to 1048576 characters for each view, with a line break between the view fields list and the query text, with the whitespace specified in the view's query expression preserved, and (if necessary) an appended ellipsis (...) indicating truncated content.

The following example returns the view name (Table_Name field) and owner name for all views in the current namespace:

SQL

```
SELECT Table_Name,Owner FROM INFORMATION_SCHEMA.VIEWS
```

The following example returns all information for all non-system views in the current namespace:

SQL

```
SELECT * FROM INFORMATION_SCHEMA.VIEWS WHERE Owner != '_SYSTEM'
```

The INFORMATION.SCHEMA.VIEWCOLUMNUSAGE persistent class displays the names of the source table fields for each of the views in the current namespace:

SQL

```
SELECT * FROM INFORMATION_SCHEMA.VIEW_COLUMN_USAGE WHERE View_Name='MyView'
```

You can display much of the same information as INFORMATION.SCHEMA.VIEWS for a single view using the Catalog Details tab in the Management Portal SQL Interface. The Catalog Details for a view include the definition of each view field (data type, max length, minval/maxval, etc.), details that are not provided by the INFORMATION.SCHEMA view classes. The Catalog Details View Info display also provides an option to edit the view definition.

6.6 Listing View Dependencies

The INFORMATION.SCHEMA.VIEWTABLEUSAGE persistent class displays all views in the current namespace and the tables they depend on. This is shown in the following example:

SQL

```
SELECT View_Schema,View_Name,Table_Schema,Table_Name FROM INFORMATION_SCHEMA.VIEW_TABLE_USAGE
```

You can invoke the %Library.SQLCatalog.SQLViewDependsOn class query to list the tables that a specified view depends upon. You specify *schema.viewname* to this class query. If you specify only *viewname*, it uses the system-wide default schema name. The caller must have privileges for the specified view to execute this class query. This is shown in the following example:

ObjectScript

```
SET statemt=##class(%SQL.Statement).%New()
SET cqStatus=statemt.%PrepareClassQuery("%Library.SQLCatalog","SQLViewDependsOn")
  IF cqStatus'=1 {WRITE "%PrepareClassQuery failed:" DO $System.Status.DisplayError(cqStatus) QUIT}
SET rset=statemt.%Execute("vschema.vname")
DO rset.%Display()
```

This SQLViewDependsOn query lists the tables that the view depends upon, listing the table schema followed by the table name. If the caller does not have privileges for a table that the view depends upon, that table and its schema are listed as <NOT PRIVILEGED>. This allows a caller without table privileges to determine how many tables the view depends upon, but not the names of the tables.

7

Relationships Between Tables

To enforce referential integrity between tables you can define foreign keys. When a table containing a foreign key constraint is modified, the foreign key constraints are checked.

7.1 Defining a Foreign Key

There are several ways to define foreign keys in Caché SQL:

- You can define a relationship between two classes. Defining a relationship automatically projects a foreign key constraint to SQL. For more information on relationships, see *Using Caché Objects*.
- You can add an explicit foreign key definition to a class definition (for cases not covered by relationships). For information, see "Foreign Key Definitions" in the *Caché Class Definition Reference*.
- You can add a foreign key using the CREATE TABLE or ALTER TABLE command. You can remove a foreign key using the ALTER TABLE command. These commands are described in the *Caché SQL Reference*.

A RowID field used as a foreign key reference must be public. Refer to RowID Hidden? for how to define a table with a public (or private) RowID field.

The maximum number of foreign keys for a table (class) is 400.

7.2 Foreign Key Referential Integrity Checking

By default, Caché performs foreign key referential integrity checking on INSERT, UPDATE and DELETE operations. If the operation would violate referential integrity, it is not performed; the operation issues an SQLCODE -121, -122, -123, or -124 error. A failed referential integrity check generates an error such as the following:

```
ERROR #5540: SQLCODE: -124 Message: At least 1 Row exists in table 'HealthLanguage.FKey2'
which references key NewIndex1 - Foreign Key Constraint 'NewForeignKey1' (Field 'Pointer1')
failed on referential action of NO ACTION [Execute+5^CacheSql16:USER]
```

This checking can be suppressed systemwide using either of the following:

- Go to the Management Portal. From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **General SQL Settings** (**[System] > [Configuration] > [General SQL Settings]**). On this screen you can view the current setting of **Perform Referential Integrity Checks on Foreign Keys for INSERT, UPDATE, and DELETE**. The default is "Yes".
- Invoke the **$SYSTEM.SQL.SetFilerRefIntegrity()** method.

When using a persistent class definition to define a table, you can define a foreign key with the NoCheck keyword to suppress future checking of that foreign key. **CREATE TABLE** does not provide this keyword option.

By default, when a row with a foreign key is deleted, Caché acquires a long term (until the transaction ends) shared lock on the corresponding referenced table's row. This prevents an update or delete of the referenced row until the **DELETE** transaction on the referencing row completes. This prevents the situation where the referenced row is deleted and then the delete of the referencing row is rolled back. If that happened, the foreign key would reference a non-existent row. This lock is not acquired if the **DELETE** of the referencing row is specified with %NOCHECK or %NOLOCK.

You can suppress checking for a specific operation by using the %NOCHECK keyword option.

By default, Caché also performs foreign key referential integrity checking on the following operations. If the specified action violates referential integrity, the command is not executed:

- ALTER TABLE DROP COLUMN.
- ALTER TABLE DROP CONSTRAINT. Issues SQLCODE -317. Foreign Key integrity checking can be suppressed using SET OPTION COMPILEMODE=NOCHECK.
- DROP TABLE. Issues SQLCODE -320. Foreign Key integrity checking can be suppressed using SET OPTION COMPILEMODE=NOCHECK.
- TRUNCATE TABLE (same considerations as DELETE).
- Trigger events, including BEFORE events. For example, a BEFORE DELETE trigger is not executed if the DELETE operation would not be performed because it violates foreign key referential integrity.

7.3 Parent and Child Tables

This section provides a brief overview on defining and working with parent/child relationships. For further details, refer to the "Defining and Using Relationships" chapter of *Defining and Using Classes*.

7.3.1 Defining Parent and Child Tables

When defining persistent classes that project to tables you can specify a parent/child relationship between two tables using the Relationship property.

The following example defines the parent table:

Class Definition

```
Class Sample.Invoice Extends %Persistent
{
Property Buyer As %String(MAXLEN=50) [Required];
Property InvoiceDate As %TimeStamp;
Relationship Pchildren AS Sample.LineItem [ Cardinality = children, Inverse = Cparent ];
}
```

The following example defines a child table:

Class Definition

```
Class Sample.LineItem Extends %Persistent
{
Property ProductSKU As %String;
Property UnitPrice As %Numeric;
Relationship Cparent AS Sample.Invoice [ Cardinality = parent, Inverse = Pchildren ];
}
```

In the Management Portal SQL interface Catalog Details tab, the **Table Info** provides the name of the Child Table(s) and/or the Parent Table. If a child table, it provides references to the parent table, such as `Cparent->Sample.Invoice`.

A child table can itself be the parent of a child table. (This child of a child is known as a "grandchild" table.) In this case, the **Table Info** provides the names of both the Parent Table and the Child Table.

7.3.2 Inserting Data into Parent and Child Tables

You must insert each record into the parent table before inserting the corresponding records in the child table. For example:

```
INSERT INTO Sample.Invoice (Buyer,InvoiceDate) VALUES ('Fred',CURRENT_TIMESTAMP)

INSERT INTO Sample.LineItem (Cparent,ProductSKU,UnitPrice) VALUES (1,'45-A7',99.95)
INSERT INTO Sample.LineItem (Cparent,ProductSKU,UnitPrice) VALUES (1,'22-A1',0.75)
```

Attempting to insert a child record for which no corresponding parent record ID exists generates an SQLCODE -104 error with a %msg `Child table 'Sample.LineItem' references non-existent row in parent table`.

During an **INSERT** operation on a child table, a shared lock is acquired on the corresponding row in the parent table. This row is locked while inserting the child table row. The lock is then released (it is not held until the end of the transaction). This ensures that the referenced parent row is not changed during the insert operation.

7.3.3 Identifying Parent and Child Tables

In Embedded SQL, you can use a host variable array to identify parent and child tables. In a child table, Subscript 0 of the host variable array is set to the parent reference (Cparent), with the format `parentref`, Subscript 1 is set to the child record ID with the format `parentref||childref`. In a parent table, Subscript 0 is undefined. This is shown in the following examples:

```
   KILL tflds,SQLCODE,C1
   &sql(DECLARE C1 CURSOR FOR
        SELECT *,%TABLENAME INTO :tflds(),:tname
        FROM Sample.Invoice)
   &sql(OPEN C1)
        QUIT:(SQLCODE<0)
   &sql(FETCH C1)
       IF SQLCODE=100 {WRITE "The ",tname," table contains no data",! QUIT}
       WHILE $DATA(tflds(0)) {
                               WRITE tname," is a child table",!,"parent ref: ",tflds(0)," %ID:
",tflds(1),!
                               &sql(FETCH C1)
                               IF SQLCODE=100 {QUIT}
                              }
      IF $DATA(tflds(0))=0 {WRITE tname," is a parent table",!}
    &sql(CLOSE C1)

   KILL tflds,SQLCODE,C1
   &sql(DECLARE C1 CURSOR FOR
        SELECT *,%TABLENAME INTO :tflds(),:tname
        FROM Sample.LineItem)
   &sql(OPEN C1)
        QUIT:(SQLCODE<0)
   &sql(FETCH C1)
       IF SQLCODE=100 {WRITE "The ",tname," table contains no data",! QUIT}
       WHILE $DATA(tflds(0)) {
                               WRITE tname," is a child table",!,"parent ref: ",tflds(0)," %ID:
",tflds(1),!
                               &sql(FETCH C1)
                               IF SQLCODE=100 {QUIT}
                              }
      IF $DATA(tflds(0))=0 {WRITE tname," is a parent table",!}
    &sql(CLOSE C1)
```

For a child table, tflds(0) and tflds(1) return values such as the following:

```
parent ref: 1 %ID: 1||1
parent ref: 1 %ID: 1||2
parent ref: 1 %ID: 1||3
parent ref: 1 %ID: 1||9
parent ref: 2 %ID: 2||4
parent ref: 2 %ID: 2||5
parent ref: 2 %ID: 2||6
parent ref: 2 %ID: 2||7
parent ref: 2 %ID: 2||8
```

For a “grandchild” table (a table that is the child of a child table), tflds(0) and tflds(1) return values such as the following:

```
parent ref: 1||1 %ID: 1||1||1
parent ref: 1||1 %ID: 1||1||7
parent ref: 1||1 %ID: 1||1||8
parent ref: 1||2 %ID: 1||2||2
parent ref: 1||2 %ID: 1||2||3
parent ref: 1||2 %ID: 1||2||4
parent ref: 1||2 %ID: 1||2||5
parent ref: 1||2 %ID: 1||2||6
```

8

Modifying the Database

You can use either SQL statements against an existing table or ObjectScript operations on the corresponding persistent class to modify the contents of a Caché database. You cannot modify a persistent class (table) that is defined as *READONLY*.

Using SQL commands provides automatic support for maintaining the integrity of the data. An SQL command is an atomic (all or nothing) operation. If there are indices defined on the table, SQL will automatically update them to reflect the changes. If there are any data or referential integrity constraints defined, SQL will automatically enforce them. If there are any defined triggers, performing these actions will pull the corresponding trigger.

This chapter discusses the following topics:

- How to insert data records (rows)
- How to use the UPDATE statement
- Computing a field value on INSERT or UPDATE
- How to use the DELETE statement
- How perform transaction processing

8.1 Inserting Data

You can insert data into a table either by using the SQL statements or by setting and saving persistent class properties.

8.1.1 Insert Data Using SQL

The INSERT statement inserts a new record into an SQL table. You can insert a single record or multiple records.

The following example inserts a single record. It is one of several available syntax forms to insert a single record:

SQL

```
INSERT INTO MyApp.Person
   (Name,HairColor)
   VALUES ('Fred Rogers','Black')
```

The following example inserts multiple records by querying data from an existing table:

SQL

```
INSERT INTO MyApp.Person
   (Name,HairColor)
    SELECT Name,Haircolor FROM Sample.Person WHERE Haircolor IS NOT NULL
```

You can also issue an INSERT OR UPDATE statement. This statement inserts a new record into an SQL table if the record does not already exist. If the record exists, this statement updates the record data with the supplied field values.

8.1.2 Insert Data Using Object Properties

You can use ObjectScript to insert one or more records of data. Create an instance of an existing persistent class, set one or more property values, then use %Save() to insert the data record:

The following example inserts a single record:

ObjectScript

```
SET oref=##class(MyApp.Person).%New()
SET oref.Name="Fred Rogers"
SET oref.HairColor="Black"
DO oref.%Save()
```

The following example inserts multiple records:

ObjectScript

```
SET nom=$LISTBUILD("Fred Rogers","Fred Astare","Fred Flintstone")
SET hair=$LISTBUILD("Black","Light Brown","Dark Brown")
FOR i=1:1:$LISTLENGTH(nom) {
    SET oref=##class(MyApp.Person).%New()
    SET oref.Name=$LIST(nom,i)
    SET oref.HairColor=$LIST(hair,i)
    SET status = oref.%Save() }
```

8.2 UPDATE Statements

The UPDATE statement modifies values in one or more existing records within an SQL table:

SQL

```
UPDATE MyApp.Person
    SET HairColor = 'Red'
    WHERE Name %STARTSWITH 'Fred'
```

8.3 Computed Field Values on INSERT or UPDATE

When you define a computed field, you can specify ObjectScript code to compute a data value for that field. This data value can be computed when the row is inserted, updated, both inserted and updated, or when queried. The following table shows the keywords required for each type of compute operation and a field/property definition example:

Compute on INSERT only	Compute on UPDATE only	Compute on both INSERT and UPDATE	Compute on Query
SQL DDL COMPUTECODE keyword Birthday VARCHAR(50) COMPUTECODE {SET {Birthday}=$PIECE($ZDATE({DOB},9),"")_" changed: "_$ZTIMESTAMP }	*SQL DDL* DEFAULT, COMPUTECODE, and COMPUTEONCHANGE keywords Birthday VARCHAR(50) DEFAULT ' ' COMPUTECODE {SET {Birthday}=$PIECE($ZDATE({DOB},9),"")_" changed: "_$ZTIMESTAMP } COMPUTEONCHANGE (DOB)	*SQL DDL* COMPUTECODE and COMPUTEONCHANGE keywords Birthday VARCHAR(50) COMPUTECODE {SET {Birthday}=$PIECE($ZDATE({DOB},9),"")_" changed: "_$ZTIMESTAMP } COMPUTEONCHANGE (DOB)	*SQL DDL* COMPUTECODE and CALCULATED or TRANSIENT keywords Birthday VARCHAR(50) COMPUTECODE {SET {Birthday}=$PIECE($ZDATE({DOB},9),"")_" changed: "_$ZTIMESTAMP } CALCULATED
Persistent Class Definition SqlComputeCode and SqlComputed property keywords Property Birthday As %String(MAXLEN = 50) [SqlComputeCode = {SET {Birthday}=$PIECE($ZDATE({DOB},9),"")_" changed: "_$ZTIMESTAMP}, SqlComputed];		*Persistent Class Definition* SqlComputeCode, SqlComputed, and SqlComputeOnChange property keywords Property Birthday As %String(MAXLEN = 50) [SqlComputeCode = {SET {Birthday}=$PIECE($ZDATE({DOB},9),"")_" changed: "_$ZTIMESTAMP}, SqlComputed, SqlComputeOnChange = DOB];	*Persistent Class Definition* SqlComputeCode, SqlComputed, and Calculated or Transient property keywords Property Birthday As %String(MAXLEN = 50) [SqlComputeCode = {SET {Birthday}=$PIECE($ZDATE({DOB},9),"")_" changed: "_$ZTIMESTAMP}, SqlComputed, Calculated];

The DDL DEFAULT keyword takes precedence over computing a data value upon insert. DEFAULT must take a data value, such as an empty string; it cannot be NULL. In a persistent class definition, the InitialExpression property keyword does not override an SqlComputed data value upon insert.

The DDL COMPUTEONCHANGE keyword can take a single field name or a comma-separated list of field names. These field names specify the fields which when updated will trigger the compute of this field; the listed field names must exist in the table, but they do not have to appear in the compute code. You must specify actual field names; you cannot specify asterisk syntax.

The DDL CALCULATED or TRANSIENT keyword computes a data value each time the field is accessed by a query. The field does not need to be specified in the select list. For example, `SELECT Name FROM MyTable WHERE LENGTH(Birthday)=36` computes the Birthday field before evaluating the condition expression. The Management Portal Open Table option performs a query, and therefore computes CALCULATED and TRANSIENT data values.

Computed field restrictions:

- UPDATE that doesn't update: An UPDATE that supplies the same values to the fields in a record as their prior values does not actually update the record. COMPUTEONCHANGE is not invoked if no real update is performed for a record. If you wish to always recalculate a computed field upon update regardless of whether the record was actually updated, use an update trigger.

- User-specified explicit value for a computed field:
  - INSERT: On INSERT you can always supply an explicit value to a COMPUTECODE or DEFAULT field. Caché SQL always takes the explicit value rather than the generated value.
  - UPDATE COMPUTEONCHANGE: An UPDATE operation can supply an explicit value to a COMPUTEONCHANGE field. Caché SQL always takes the explicit value rather than the computed value.
  - CALCULATED or TRANSIENT: An INSERT or UPDATE operation cannot supply an explicit value to a CALCULATED or TRANSIENT field, because a CALCULATED or TRANSIENT field does not store data. However, Caché SQL does perform field validation on the explicit value and can, for example, generate an SQLCODE -104 error if the supplied value is longer than the maximum data size.

8.4 DELETE Statements

The DELETE statement removes one or more existing records from an SQL table:

SQL

```
DELETE FROM MyApp.Person
    WHERE HairColor = 'Aqua'
```

You can issue a TRUNCATE TABLE command to delete all records in a table. You can also delete all records in a table using **DELETE**. **DELETE** (by default) pulls delete triggers; **TRUNCATE TABLE** does not pull delete triggers. Using **DELETE** to delete all records does not reset table counters; **TRUNCATE TABLE** resets these counters.

8.5 Transaction Processing

A transaction is a series of **INSERT**, **UPDATE**, **DELETE**, **INSERT OR UPDATE**, and **TRUNCATE TABLE** data modification statements that comprise a single unit of work.

The SET TRANSACTION command can be used to set the transaction parameters for the current process. The same parameters can also be set using the **START TRANSACTION** command. These transaction parameters continue in effect across multiple transactions until explicitly changed.

A START TRANSACTION command explicitly starts a transaction. This command is generally optional; if transaction %COMMITMODE is either IMPLICIT or EXPLICIT, a transaction begins automatically with the first database modification operation. If transaction %COMMITMODE is NONE, you must explicitly specify **START TRANSACTION** to initiate transaction processing.

If a transaction succeeds, committing its changes can be implicit (automatic) or explicit; the %COMMITMODE value determines whether you need to explicitly use the COMMIT statement to permanently add the data modifications to the database and release resources.

If a transaction fails, you can use the ROLLBACK statement to undo its data modifications so that these do not go into the database.

Note: SQL transaction statements are *not* supported when running SQL through the Management Portal **Execute SQL Query** interface. This interface is intended as a test environment for developing SQL code, not for modifying actual data.

8.5.1 Transactions and Savepoints

In Caché SQL, you can perform two kinds of transaction processing: full transaction processing and transaction processing using savepoints. With full transaction processing, a transaction begins with **START TRANSACTION** statement (explicit or implicit) and continues until either a **COMMIT** statement (explicit or implicit) concludes the transaction and commits all work, or a **ROLLBACK** statement reverses all work done during the transaction.

With savepoints, Caché SQL supports levels within a transaction. You begin a transaction with a **START TRANSACTION** statement (explicit or implicit). Then during the transaction you use SAVEPOINT to specify one or more named savepoints within the program. You can specify a maximum of 255 named savepoints in a transaction. Adding a savepoint increments the $TLEVEL transaction level counter.

- A **COMMIT** commits all work performed during the transaction. Savepoints are ignored.
- A **ROLLBACK** rolls back all work performed during the transaction. Savepoints are ignored.
- A **ROLLBACK TO SAVEPOINT pointname** rolls back all work performed since the **SAVEPOINT** specified by *pointname* and decrements an internal transaction level counter by the appropriate number of savepoint levels. For example, if you established two savepoints, svpt1 and svpt2, and then rolled back to svpt1, the **ROLLBACK TO SAVEPOINT svpt1** reverse the work done since svpt1 and, in this case, decrements the transaction level counter by 2.

8.5.2 Non-transaction Operations

While a transaction is in effect, the following operations are not included in the transaction and therefore cannot be rolled back:

- The IDKey counter increment is not a transaction operation. The IDKey is automatically generated by $INCREMENT (or $SEQUENCE), which maintains a count independent of the SQL transaction. For example, if you insert records with IDKeys of 17, 18, and 19, then rollback this insert, the next record to be inserted will have an IdKey of 20.
- Cached query creation, modification, and purging are not transaction operations. Therefore, if a cached query is purged during a transaction, and that transaction is then rolled back, the cached query will remain purged (will not be restored) following the rollback operation.
- A DDL operation or a Tune Table operation that occur within a transaction may create and run a temporary routine. This temporary routine is treated the same as a cached query. That is, the creation, compilation, and deletion of a temporary routine are not treated as part of the transaction. The execution of the temporary routine is considered part of the transaction.

For non-SQL items rolled back or not rolled back, refer to the ObjectScript TROLLBACK command.

8.5.3 Transaction Locks

A transaction uses locks to safeguard unique data values. For example, if a process deletes a unique data value, this value is locked for the duration of the transaction. Therefore, another process could not insert a record using this same unique data value until the first transaction completed. This prevents a rollback resulting in a duplicate value for a field with a uniqueness constraint. These locks are automatically applied by the **INSERT**, **UPDATE**, **INSERT OR UPDATE**, and **DELETE** statements, unless the statement includes a %NOLOCK restriction argument.

8.5.4 Transaction Size Limitations

There is no limitation on the number of operations you can specify in a transaction, other than space availability for journal files. The size of the lock table does not normally impose a limit, because Caché provides automatic lock escalation.

There is a default lock threshold of 1000 locks per table. A table can have 1000 unique data value locks for the current transaction. The 1001st lock operation escalates the locking for that table to a table lock for the duration of the transaction.

This lock threshold value is configurable using either of the following:

- Invoke the **$SYSTEM.SQL.SetLockThreshold()** method. This method changes both the current system-wide value and the configuration file setting. To determine the current lock escalation threshold, use the **$SYSTEM.SQL.GetLockThreshold()** method.
- Go to the Management Portal. From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **General SQL Settings**. On this screen you can view and edit the current setting of **Lock Threshold**.

There is no limit on the number of subnodes (child tables) that can be killed. All subnode kills are journaled, and thus can be rolled back.

8.5.5 Reading Uncommitted Data

You can specify the read isolation level by setting SET TRANSACTION or START TRANSACTION for the process issuing the query.

- ISOLATION LEVEL READ UNCOMMITTED: Uncommitted inserts, updates, and deletes to data are visible for query (read only) access by other users. This is the default if no transaction is specified.
- ISOLATION LEVEL READ VERIFIED: Uncommitted inserts, updates, and deletes to data are visible for query (read only) access by other users. Provides re-checking of data used by query conditions and displayed by the query.
- ISOLATION LEVEL READ COMMITTED: Changes made to the data by uncommitted inserts and updates are not shown in the query result set. The query result set only contains inserts and updates that have been committed. However, changes made to the data by uncommitted deletes are shown in the query result set.

The following SELECT command clauses always return uncommitted data, regardless of the current isolation level: an aggregate function, a DISTINCT clause, a GROUP BY clause, or a **SELECT** with the %NOLOCK keyword. For further details, refer to Isolation Level.

8.5.6 ObjectScript Transaction Commands

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

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

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

9

Querying the Database

This chapter discusses how to query the database. It includes information on the following topics:

- Types of Queries
- Using a SELECT Statement
- Defining and Executing Named Queries
- Queries Invoking User-defined Functions
- Querying Serial Object Properties
- Querying Collection Properties
- Queries Invoking Free-text Search
- Pseudo-Field Variables: %ID, %TABLENAME, %CLASSNAME
- Query Metadata
- Queries and Enterprise Cache Protocol (ECP)

9.1 Types of Queries

A query is a statement which performs data retrieval and generates a result set. A query can consist of any of the following:

- A simple SELECT statement that accesses the data in a specified table or view.
- A SELECT statement with **JOIN** syntax that accesses the data from several tables or views.
- A UNION statement that combines the results of multiple **SELECT** statements.
- A subquery that uses a SELECT statement to supply a single data item to an enclosing **SELECT** query.
- In Embedded SQL, a SELECT statement that uses an SQL cursor to access multiple rows of data using a FETCH statement.

9.2 Using a SELECT Statement

A SELECT statement selects one or more rows of data from one or more tables or views. A simple **SELECT** is shown in the following example:

```
SELECT Name,DOB FROM Sample.Person WHERE Name %STARTSWITH 'A' ORDER BY DOB
```

In this example, Name and DOB are columns (data fields) in the Sample.Person table.

The order that clauses must be specified in a **SELECT** statement is: SELECT DISTINCT TOP ... *select-items* INTO ... FROM ... WHERE ... GROUP BY ... HAVING ... ORDER BY. This is the command syntax order. All of these clauses are optional, except SELECT *select-items*. (The optional FROM clause is required to perform any operations on stored data, and therefore is almost always required in a query.) Refer to the SELECT statement syntax for details on the required order for specifying **SELECT** clauses.

9.2.1 SELECT Clause Order of Execution

The operation of a **SELECT** statement can be understood by noting its semantic processing order (which is *not* the same as the **SELECT** syntax order). The clauses of a **SELECT** are processed in the following order:

1. FROM clause — specifies a table, a view, multiple tables or views using JOIN syntax, or a subquery.
2. WHERE clause — restricts what data is selected using various criteria.
3. GROUP BY clause — organizes the selected data into subsets with matching values; only one record is returned for each value.
4. HAVING clause — restricts what data is selected from groups using various criteria.
5. select-item — selects a data fields from the specified table or view. A *select-item* can also be an expression which may or may not reference a specific data field.
6. DISTINCT clause — applied to the **SELECT** result set, it limits the rows returned to those that contain a distinct (non-duplicate) value.
7. ORDER BY clause — applied to the **SELECT** result set, it sorts the rows returned in collation order by the specified field(s).

This semantic order shows that a table alias (which is defined in the **FROM** clause) can be recognized by all clauses, but a column alias (which is defined in the **SELECT** *select-items*) can only be recognized by the **ORDER BY** clause.

To use a column alias in other **SELECT** clauses you can use a subquery, as shown in the following example:

```
SELECT Interns FROM
      (SELECT Name AS Interns FROM Sample.Employee WHERE Age<21)
WHERE Interns %STARTSWITH 'A'
```

In this example, Name and Age are columns (data fields) in the Sample.Person table, and Interns is a column alias for Name.

9.2.2 Selecting Fields

When you issue a SELECT, Caché SQL attempts to match each specified *select-item* field name to a property defined in the class corresponding to the specified table. Each class property has both a property name and a SqlFieldName. If you defined the table using SQL, the field name specified in the CREATE TABLE command is the SqlFieldName, and Caché generated the property name from the SqlFieldName.

Field names, class property names, and SqlFieldName names have different naming conventions:

- Field names in a **SELECT** statement are not case-sensitive. SqlFieldName names and property names are case-sensitive.
- Field names in a **SELECT** statement and SqlFieldName names can contain certain non-alphanumeric characters following identifier naming conventions. Property names can only contain alphanumeric characters. When generating a property name, Caché strips out non-alphanumeric characters. Caché may have to append a character to create a unique property name.

The translation between these three names for a field determine several aspects of query behavior. You can specify a *select-item* field name using any combination of letter case and Caché SQL will identify the appropriate corresponding property. The data column header name in the result set display is the SqlFieldName, not the field name specified in the *select-item*. This is why the letter case of the data column header may differ from the *select-item* field name.

You can specify a column alias for a *select-item* field. A column alias can be in any mix of letter case, and can contain non-alphanumeric characters, following identifier naming conventions. A column alias can be referenced using any combination of letter case (for example, in the ORDER BY clause) and Caché SQL resolves to the letter case specified in the *select-item* field. Caché always attempts to match to the list of column aliases before attempting to match to the list of properties corresponding to defined fields. If you have defined a column alias, the data column header name in the result set display is the column alias in the specified letter case, not the SqlFieldName.

When a **SELECT** query completes successfully, Caché SQL generates a result set class for that query. The result set class contains a property corresponding to each selected field. If a **SELECT** query contains duplicate field names, the system generates unique property names for each instance of the field in the query by appending a character. For this reason, you cannot include more than 36 instances of the same field in a query.

The generated result set class for a query also contains properties for column aliases. To avoid the performance cost of letter case resolution, you should use the same letter case when referencing a column alias as the letter case used when specifying the column alias in the **SELECT** statement.

In addition to user-specified column aliases, Caché SQL also automatically generates up to three aliases for each field name, aliases which correspond to common letter case variants of the field name. These generated aliases are invisible to the user. They are provided for performance reasons, because accessing a property through an alias is faster than resolving letter case through letter case translation. For example, if **SELECT** specifies FAMILYNAME and the corresponding property is familyname, Caché SQL resolves letter case using a generated alias (FAMILYNAME AS familyname). However, if **SELECT** specifies fAmILyNaMe and the corresponding property is familyname, Caché SQL must resolves letter case using the slower letter case translation process.

A *select-item* item can also be an expression, an aggregate function, a subquery, a user-defined function, as asterisk, or some other value. For further details on *select-item* items other than field names, refer to The select-item section of the **SELECT** command reference page.

9.2.3 The JOIN Operation

A JOIN provides a way to link data in one table with data in another table and are frequently used in defining reports and queries. Within SQL, a **JOIN** is an operation that combines data from two tables to produce a third, subject to a restrictive condition. Every row of the resulting table must satisfy the restrictive condition.

Caché SQL supports five types of joins (some with multiple syntactic forms): CROSS JOIN, INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN. Outer joins support the ON clause with a full range of conditional expression predicates and logical operators. There is partial support for NATURAL outer joins and outer joins with a USING clause. For definitions of these join types and further details, refer to the JOIN page in the *Caché SQL Reference*.

If a query contains a join, all of the field references within that query must have an appended table alias. Because Caché does not include the table alias in the data column header name, you may wish to provide column aliases for *select-item* fields to clarify which table is the source of the data.

The following example uses a join operation to match the “fake” (randomly-assigned) zip codes in Sample.Person with the real zip codes and city names in Sample.USZipCode. A WHERE clause is provided because USZipCode does not include all possible 5-digit zip codes:

```
SELECT P.Home_City,P.Home_Zip AS FakeZip,Z.ZipCode,Z.City AS ZipCity,Z.State
FROM Sample.Person AS P LEFT OUTER JOIN Sample.USZipCode AS Z
ON P.Home_Zip=Z.ZipCode
WHERE Z.ZipCode IS NOT NULL
ORDER BY P.Home_City
```

9.2.4 Queries Selecting Large Numbers of Fields

A query cannot select more than 1,000 *select-item* fields.

A query selecting more than 150 *select-item* fields may have the following performance consideration. Caché automatically generates result set column aliases. These generated aliases are provided for field names without user-defined aliases to enable rapid resolution of letter case variations. Letter case resolution using an alias is significantly faster than letter case resolution by letter case translation. However, the number of generated result set column aliases is limited to 500. Because commonly Caché generates three of these aliases (for the three most common letter case variations) for each field, the system generates aliases for roughly the first 150 specified fields in the query. Therefore, a query referencing less than 150 fields commonly has better result set performance than a query referencing significantly more fields. This performance issue can be avoided by specifying an exact column alias for each field *select-item* in a very large query (for example, `SELECT FamilyName AS FamilyName`) and then making sure that you use the same letter case when referencing the result set item by column alias.

9.3 Defining and Executing Named Queries

You can define and execute a named query as follows:

- Define the query using CREATE QUERY. This query is defined as a stored procedure, and can be executed using CALL.
- Define a class query (a query defined in a class definition). A class query is projected as a stored procedure.
  - Execute the class query using CALL.
  - Prepare the class query using the %SQL.Statement %PrepareClassQuery() method, and then executed using the **%Execute()** method. See “Using Dynamic SQL”.
  - Execute the class query using the **Execute()** method of %Library.ResultSet. See “Dynamic SQL Using Older Result Set Classes”.

9.3.1 CREATE QUERY and CALL

You can define a query using CREATE QUERY, and then execute it by name using CALL. In the following example, the first is an SQL program that defines the query AgeQuery, the second is Dynamic SQL that executes the query:

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

ObjectScript

```
  ZNSPACE "Samples"
  SET mycall = "CALL Sample.AgeQuery(11,65)"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(mycall)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  DO rset.%Display()

DROP QUERY Sample.AgeQuery
```

9.3.2 Class Queries

You can define a query in a class. The class may be a %Persistent class, but does not have to be. This class query can reference data defined in the same class, or in another class in the same namespace. The tables, fields, and other data entities referred to in a class query must exist when the class that contains the query is compiled.

The following class definition example defines a class query:

```
Class Sample.QClass Extends %Persistent [DdlAllowed]
  {
  Query MyQ(Myval As %String) As %SQLQuery (CONTAINID=1,ROWSPEC="Name,Home_State") [SqlProc]
     {
     SELECT Name,Home_State FROM Sample.Person
     WHERE Home_State = :Myval  ORDER BY Name
     }

  }
```

The following example executes the MyQ query defined in the Sample.QClass in the previous example:

ObjectScript

```
  SET Myval="NY"
  SET stmt=##class(%SQL.Statement).%New()
  SET status = stmt.%PrepareClassQuery("Sample.QClass","MyQ")
    IF status'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(status) QUIT}
  SET rset = stmt.%Execute(Myval)
  DO rset.%Display()
  WRITE !,"End of data"
```

The following Dynamic SQL example uses %SQL.Statement to execute the ByName query defined in the Sample.Person class, passing a string to limit the names returned to those that start with that string value:

ObjectScript

```
  SET statemt=##class(%SQL.Statement).%New()
  SET cqStatus=statemt.%PrepareClassQuery("Sample.Person","ByName")
    IF cqStatus'=1 {WRITE "%PrepareClassQuery failed:" DO $System.Status.DisplayError(cqStatus) QUIT}
  SET rs=statemt.%Execute("L")
  DO rs.%Display()
```

For further details, refer to "Defining and Using Class Queries" in *Using Caché Objects*.

For information on query names automatically assigned to executed queries, refer to the Cached Queries chapter of *Caché SQL Optimization Guide*.

9.4 Queries Invoking User-defined Functions

Caché SQL allows you to invoke class methods within SQL queries. This provides a powerful mechanism for extending the syntax of SQL.

To create a user-defined function, define a class method within a persistent Caché class. The method must have a literal (non-object) return value. This has to be a *class* method because there will not be an object instance within an SQL query on which to invoke an instance method. It also has to be defined as being an SQL stored procedure.

For example, we can define a **Cube()** method within the class MyApp.Person:

Class Definition

```
Class MyApp.Person Extends %Persistent [DdlAllowed,language = basic]
{
/// Find the Cube of a number
ClassMethod Cube(val As %Integer) As %Integer [SqlProc]
{
    Return val * val * val
}
}
```

You can create SQL functions with the CREATE FUNCTION, CREATE METHOD or CREATE PROCEDURE statements.

To call an SQL function, specify the name of the SQL procedure. A SQL function may be invoked in SQL code anywhere where a scalar expression may be specified. The function name may be qualified with its schema name, or unqualified. Unqualified function names take either a user-supplied schema search path or the system-wide default schema name. A function name may be a delimited identifier.

An SQL function must have a parameter list, enclosed in parentheses. The parameter list may be empty, but the parentheses are mandatory. All specified parameters act as input parameters. Output parameters are not supported.

An SQL function must return a value.

For example, the following SQL query invokes a user-defined SQL function as a method, just as if it was a built-in SQL function:

SQL

```
SELECT %ID, Age, MyApp.Person_Cube(Age) FROM MyApp.Person
```

For each value of Age, this query will invoke the **Cube()** method and place its return value within the results.

SQL functions may be nested.

If the specified function is not found, Caché issues an SQLCODE -359 error. If the specified function name is ambiguous, Caché issues an SQLCODE -358 error.

9.5 Querying Serial Object Properties

A serial object property that is projected as a child table to SQL from a class using default storage (%Storage.Persistent) is also projected as a single column in the table projected by the class. The value of this column is the serialized value of the serial object properties. This single column property is projected as an SQL %List field.

For example, the column Home in Sample.Person is defined as `Property Home As Sample.Address;`. It is projected to `Class Sample.Address Extends (%SerialObject)`, which contains the properties Street, City, State, and PostalCode. See Embedded Object (%SerialObject) in the "Defining Tables" chapter for details on defining a serial object.

The following example returns values from individual serial object columns:

```
SELECT TOP 4 Name,Home_Street,Home_City,Home_State,Home_PostalCode
FROM Sample.Person
```

The following example returns the values for all of the serial object columns (in order) as a single %List format string, with the value for each column as an element of the %List:

```
SELECT TOP 4 Name,$LISTTOSTRING(Home,'^')
FROM Sample.Person
```

By default, this Home column is hidden and is not projected as a column of Sample.Person.

9.6 Querying Collections

Collections may be referenced from the SQL WHERE clause, as follows:

```
 WHERE FOR SOME %ELEMENT(collectionRef) [AS label] (predicate)
```

The FOR SOME %ELEMENT clause can be used for list collections and arrays that specify STORAGEDEFAULT="list". The *predicate* may contain one reference to the pseudo-columns %KEY, %VALUE, or both. A few examples should help to clarify how the FOR SOME %ELEMENT clause may be used. The following returns the name and the list of FavoriteColors for each person whose FavoriteColors include 'Red'.

```
SELECT Name,FavoriteColors FROM Sample.Person
    WHERE FOR SOME %ELEMENT(FavoriteColors) (%Value = 'Red')
```

Any SQL predicate may appear after the %Value (or %Key), so for example the following is also legal syntax:

```
SELECT Name,FavoriteColors FROM Sample.Person
    WHERE FOR SOME %ELEMENT(Sample.Person.FavoriteColors)
        (%Value IN ('Red', 'Blue', 'Green'))
```

A list collection is considered a special case of an array collection that has sequential numeric keys 1, 2, and so on. Array collections may have arbitrary non-null keys:

```
 FOR SOME (children) (%Key = 'betty' AND %Value > 5)
```

In addition to the built-in list and array collection types, generalized collections may be created by providing a **BuildValueArray()** class method for any property. The **BuildValueArray()** class method transforms the value of a property into a local array, where each subscript of the array is a %KEY and the value is the corresponding %VALUE.

In addition to simple selections on the %KEY or %VALUE, it is also possible to logically connect two collections, as in the following example:

```
   FOR SOME %ELEMENT(flavors) AS f
      (f.%VALUE IN ('Chocolate', 'Vanilla') AND
       FOR SOME %ELEMENT(toppings) AS t
           (t.%VALUE = 'Butterscotch' AND
            f.%KEY = t.%KEY))
```

This example has two collections: flavors and toppings, that are positionally related through their key. The query qualifies a row that has chocolate or vanilla specified as an element of flavors, and that also has butterscotch listed as the corresponding topping, where the correspondence is established through the %KEY.

You can use the $SYSTEM.SQL configuration methods **GetCollectionProjection()** and **SetCollectionProjection()** to determine whether to project a collection as a column if the collection is projected as a child table. Changes made to this system-wide setting takes effect for each class when that class is compiled or recompiled.

For information on indexing a collection, refer to Indexing Collections in the "Defining and Building Indices" chapter of the *Caché SQL Optimization Guide*.

9.6.1 Collection Indexing and Querying Collections through SQL

Collections may be referenced from the SQL WHERE clause with a FOR clause. For example:

```
FOR SOME %ELEMENT(collectionRef) [AS label] (predicate)
```

The FOR SOME %ELEMENT clause can be used for list collections and arrays that specify STORAGEDEFAULT="list". The *predicate* may contain one reference to the pseudo-columns %KEY, %VALUE, or both. A few examples should help to clarify how the FOR SOME %ELEMENT clause may be used. The following returns the name and the list of FavoriteColors for each person whose FavoriteColors include 'Red'.

```
SELECT Name,FavoriteColors FROM Sample.Person
    WHERE FOR SOME %ELEMENT(FavoriteColors) (%Value = 'Red')
```

Any SQL predicate may appear after the %Value (or %Key), so for example the following is also legal syntax:

```
SELECT Name,FavoriteColors FROM Sample.Person
    WHERE FOR SOME %ELEMENT(Sample.Person.FavoriteColors)
        (%Value IN ('Red', 'Blue', 'Green'))
```

A list collection is considered a special case of an array collection that has sequential numeric keys 1, 2, and so on. Array collections may have arbitrary non-null keys:

```
FOR SOME (children) (%Key = 'betty' AND %Value > 5)
```

In addition to the built-in list and array collection types, generalized collections may be created by providing a **BuildValueArray()** class method for any property. The **BuildValueArray()** class method transforms the value of a property into a local array, where each subscript of the array is a %KEY and the value is the corresponding %VALUE.

In addition to simple selections on the %KEY or %VALUE, it is also possible to logically connect two collections, as in the following example:

```
FOR SOME %ELEMENT(flavors) AS f
   (f.%VALUE IN ('Chocolate', 'Vanilla') AND
    FOR SOME %ELEMENT(toppings) AS t
        (t.%VALUE = 'Butterscotch' AND
         f.%KEY = t.%KEY))
```

This example has two collections: flavors and toppings, that are positionally related through their key. The query qualifies a row that has chocolate or vanilla specified as an element of flavors, and that also has butterscotch listed as the corresponding topping, where the correspondence is established through the %KEY.

9.6.2 Usage Notes and Restrictions

- FOR SOME %ELEMENT may only appear in the WHERE clause.
- %CONTAINS may only appear in the WHERE clause.
- %KEY and/or %VALUE may only appear in a FOR predicate.
- Any particular %KEY or %VALUE may be referenced only once.
- %KEY and %VALUE may not appear in an outer join.
- %KEY and %VALUE may not appear in a value expression (only in a predicate).
- Streams longer than the maximum length of a string require the use of the %CONTAINSTERM predicate rather than the %CONTAINS predicate. For information on the maximum length of a string, see the section "Support for Long String Operations" in the chapter "Server Configuration Options" in the *Caché Programming Orientation Guide*.

- Streams longer than the maximum length of a string only support the use of %SIMILARITY on indexed fields. For information on the maximum length of a string, see the section "Support for Long String Operations" in the chapter "Server Configuration Options" in the *Caché Programming Orientation Guide*.

9.7 Queries Invoking Free-text Search

Caché supports what is called "free-text search," which includes support for:

- Stemming
- Multiple-word searches (also called *n-grams*)
- Automatic classification
- Dictionary management

This feature enables SQL to support full text indexing, and also enables SQL to index and reference individual elements of a collection without projecting the collection property as a child table. While the underlying mechanisms that support collection indexing and full text indexing are closely related, text retrieval has many special properties, and therefore special classes and SQL features have been provided for text retrieval.

For details on the underlying classes that support these features, see the %Text.Text class.

9.7.1 Full Text Indexing and Text Retrieval through SQL

The %Library.Text class and the %Text package has been provided to index text and to search textual data with SQL. To use the feature on an existing %String property, change %String to %Text and set the *LANGUAGECLASS* parameter. For English text, the declaration would be as follows:

```
Property myNotes As %Text ( LANGUAGECLASS="%Text.English", MAXLEN=1000 );
   Index myFullTextIndex On myNotes(KEYS);
```

LANGUAGECLASS specifies the name of a helper class that provides the necessary interface to SQL and to the indexer so that efficient text indexing and text search may be carried out. Specifying a *MAXLEN* value (in bytes) is required for %Text properties.

The available text-aware predicates are %CONTAINS, %CONTAINSTERM, and %SIMILARITY.

Once the %Text property has been declared and optionally indexed, you can issue full text queries using the SQL %CONTAINS predicate, as follows:

```
SELECT plaintiff, legalBrief FROM transcript
    WHERE plaintiff = 'John Doe' AND
          legalBrief %CONTAINS ('neighbor', 'tree', 'roof')
```

The query above returns all transcripts where the plaintiff is 'John Doe' and where the terms 'neighbor' AND 'tree' AND 'roof' are in its legalBrief.

Caché includes language-specific parsers in the %Text package for English, Spanish, French, Italian, German, Japanese, Portuguese. While easy to use, the language specific subclasses can be configured to perform many sophisticated operations, such as word *stemming* to map various forms of the same word into a common root (block, blocks, blocking, and so on), or to support multi-word phrases (n-grams), or to filter out noise words, or to perform automatic text classification (for example, for junk-mail filtering), as well as other features.

Multi-word strings may be specified to **%CONTAINS**, even if the type class is not configured to support n-grams of the full length of the query. For example, the following predicate may be specified even if the %Text class is configured to store only individual words:

```
SELECT author FROM famousQuotations WHERE
  quotetext %CONTAINS('eggs in one basket')
```

The query above would return all authors where the documents contain exactly the specified phrases, even if the text class represents the document only with single words (NGRAMLEN=1). When the pattern is longer than the n-gram length as in the case above, SQL filters the rows with the "[" (contains) operator. Because the "[" operator is case-sensitive, the **%CONTAINS** predicate is also case-sensitive on patterns longer than NGRAMLEN. An implication is that NGRAMLEN can also affect which rows get returned, as in the following case:

```
 mission %CONTAINS('Fortune 5')
```

If NGRAMLEN >= 2, then only the rows containing 'Fortune 5' are returned.

If NGRAMLEN < 2, then rows containing 'Fortune 5', 'Fortune 50', 'Fortune 500', and so on may be returned, since all of these patterns satisfy the "[" test.

See the class documentation of the %Text.Text class and its language-specific subclasses (such as %Text.English) for more information about the capabilities of the Text interface.

9.8 Pseudo-Field Variables

Caché SQL queries support the following pseudo-field values:

- %ID — returns the RowId field value, regardless of the actual name of the RowId field.
- %TABLENAME — returns the qualified name of an existing table that is specified in the FROM clause. The qualified table name is returned in the letter case used when defining the table, not the letter case specified in the FROM clause. If the FROM clause specifies an unqualified table name, %TABLENAME returns the qualified table name (schema.table), with the schema name supplied from either a user-supplied schema search path or the system-wide default schema name. For example, if the FROM clause specified `mytable`, the %TABLENAME variable might return `SQLUser.MyTable`.
- %CLASSNAME — returns the qualified class name (package.class) corresponding to an existing table specified in the FROM clause. For example, if the FROM clause specified `SQLUser.mytable`, the %CLASSNAME variable might return `User.MyTable`.

  Note: The %CLASSNAME pseudo-field value should not be confused with the **%ClassName()** instance method. They return different values.

Pseudo-field variables can only be returned for a table that contains data.

If multiple tables are specified in the FROM clause you must use table aliases, as shown in the following example:

ObjectScript

```
&sql(SELECT P.Name,P.%ID,P.%TABLENAME,E.%TABLENAME
     INTO :name,:rid,:ptname,:etname
   FROM Sample.Person AS P,Sample.Employee AS E)
   WRITE ptname," Name is: ",name,!
   WRITE ptname," RowId is: ",rid,!
   WRITE "1st TableName is: ",ptname,!
   WRITE "2nd TableName is: ",etname,!
```

The %TABLENAME and %CLASSNAME columns are assigned the default column name `Literal_n`, where *n* is the *select-item* position of the pseudo-field variable in the SELECT statement.

9.9 Query Metadata

You can use Dynamic SQL to return metadata about the query, such as the number of columns specified in the query, the name (or alias) of a column specified in the query, and the data type of a column specified in the query.

The following ObjectScript Dynamic SQL example returns the column name and an integer code for the column's ODBC data type for all of the columns in Sample.Person:

ObjectScript

```
SET myquery="SELECT * FROM Sample.Person"
SET rset = ##class(%SQL.Statement).%New()
SET qStatus = rset.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET x=rset.%Metadata.columns.Count()
WHILE x>0 {
  SET column=rset.%Metadata.columns.GetAt(x)
WRITE !,x," ",column.colName," ",column.ODBCType
SET x=x-1 }
WRITE !,"end of columns"
```

In this example, columns are listed in reverse column order. Note that the FavoriteColors column, which contains list structured data, returns a data type of 12 (VARCHAR) because ODBC represents a Caché list data type value as a string of comma-separated values.

For further details, refer to the Dynamic SQL chapter of this manual, and the %SQL.Statement class in the *InterSystems Class Reference*.

9.10 Queries and ECP

Caché implementations that use Enterprise Cache Protocol (ECP) can synchronize query results. ECP is a distributed data caching architecture that manages the distribution of data and locks among a heterogeneous network of server systems.

If ECP synchronization is active, each time a **SELECT** statement is executed Caché forces all pending ECP requests to the database server. On completion this guarantees that the client cache is in sync. This synchronization occurs in the Open logic of the query. This is in the OPEN cursor execution if this is a cursor query.

You can activate ECP synchronization using the Management Portal. Go to the Management Portal. From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **General SQL Settings**. On this screen you can view and edit the current setting of **SQL SELECT Synchronizes ECP Cache** The default is "No" (ECP synchronization is not performed). This is a system-wide setting; changing this setting immediately affects all Caché processes on the system.

You can also activate ECP synchronization using the **SetECPSync()** method of the %SYSTEM.SQL class.

To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**.

For further details, refer to the Caché Distributed Data Management Guide

10

Collation

Collation specifies how values are ordered and compared, and is part of both Caché SQL and Caché Objects. There are two fundamental collations: numeric and string.

- Numeric collation orders numbers based on the complete number in the following order: null, then negative numbers from largest to smallest, zero, then positive numbers from smallest to largest. This creates a sequence such as the following: –210, –185, –54, –34, -.02, 0, 1, 2, 10, 17, 100, 120.
- String collation orders strings by collating on each sequential character. This creates an order such as the following: null, A, AA, AAA, AAB, AB, B. For numbers, this creates an order such as the following: –.02, –185, –210, –34, –54, 0, 1, 10, 100, 120, 17, 2.

The default string collation is SQLUPPER; this default is set for each namespace. SQLUPPER collation converts all letters to uppercase (for the purpose of collation), and appends a space character to the beginning of the string. This conversion is for the purposes of collation only; in Caché SQL strings are usually displayed in uppercase and lowercase letters, regardless of the collation applied, and the length of a string does not include the appended space character.

A timestamp is a string, and therefore follows the current string collation. However, because a timestamp is in ODBC format, the string collation is the same as chronological sequence, if leading zeros are specified.

- A string expression (such as those using the scalar string functions **LEFT** or **SUBSTR**) makes its result collation EXACT.
- Any comparison of two literals uses EXACT collation.

You can use the ObjectScript Sorts After operator to determine the relative collation sequence order of two values.

You can specify collation as follows:

- Namespace default
- Table field/property definition
- Index definition
- Query SELECT item
- Query DISTINCT and GROUP BY clause

Also see “SQL Collation and NLS Collations,” later in this chapter.

10.1 Collation Types

Collation can be specified as a keyword in the definition of a field/property or the definition of an index.

Collation can be specified by applying a collation function to a field name in a query clause. The % prefix is required when specifying a collation function.

Collation is in ascending ASCII/Unicode sequence, with the following transformations:

- EXACT — Enforces case sensitivity for string data. Not recommended for use if your string data contains values in *canonical numeric* format (for example `123` or `-.57`).
- MVR — (For compatibility with MultiValue database systems.) For a string containing both numeric and non-numeric characters. MVR collation divides the string into substrings, each substring containing either all numeric or all non-numeric characters. The numeric substrings are sorted in signed numeric order. The non-numeric substrings are sorted in case-sensitive ASCII collation sequence. (Note that this collation does not appear in the Studio New Index Wizard.)
- SQLSTRING — Strips trailing whitespace (spaces, tabs, and so on), and adds one leading blank space to the beginning of the string. It collates any value containing only whitespace (spaces, tabs, and so on) as the SQL empty string. SQLSTRING supports an optional *maxlen* integer value.
- SQLUPPER — Converts all alphabetic characters to uppercase, strips trailing whitespace (spaces, tabs, and so on), and then adds one leading space character to the beginning of the string. The reason this space character is appended is to force numeric values to be collated as strings (because the space character is not a valid numeric character). This transformation also causes SQL to collate the SQL empty string ('') value and any value containing only whitespace (spaces, tabs, and so on) as a single space character. SQLUPPER supports an optional *maxlen* integer value. Note that the SQLUPPER transform is *not* the same as the result of the SQL function UPPER.
- TRUNCATE — Enforces case sensitivity for string data and (unlike EXACT) allows you to specify a length at which to truncate the value. This is useful when indexing exact data that is longer than what is supported for use in a subscript. It takes a positive integer argument, in the form `%TRUNCATE(`*string*`,`*n*`)`, to truncate the string to the first *n* characters, which improves indexing and sorting on long strings. If you do not specify a length for TRUNCATE, it behaves identically to EXACT; while this behavior is supported. your definitions and code may be easier to maintain if you use TRUNCATE only when you have a length defined and EXACT when you do not.
- PLUS — Makes the value numeric. A non-numeric string value is returned as 0.
- MINUS — Makes the value numeric and changes its sign. A non-numeric string value is returned as 0.

Note: There are also various legacy collation types, the use of which is not recommended.

In an SQL query, you can specify a collation function without parentheses `%SQLUPPER Name` or with parentheses `%SQLUPPER(Name)`. If the collation function specifies truncation, the parentheses are required `%SQLUPPER(Name,10)`.

Three collation types: SQLSTRING, SQLUPPER, and TRUNCATE support an optional *maxlen* integer value. If specified, *maxlen* truncates parsing of the string to the first *n* characters. This can be used to improve performance when indexing and sorting long strings. You can use *maxlen* in a query to sort on, group by, or return a truncated string value.

You can also perform collation type conversions using the **%SYSTEM.Util.Collation()** method.

10.2 Namespace-wide Default Collation

Each namespace has a current string collation setting. This string collation is defined for the data type in %Library.String. The default is SQLUPPER. This default can be changed.

You can define the collation default on a per-namespace basis. By default, namespaces have no assigned collation, which means they use SQLUPPER collation. You can assign a different default collation to a namespace. This namespace default collation applies to all processes, and persists across Caché restarts until explicitly reset.

ObjectScript

```
   SET stat=$$GetEnvironment^%apiOBJ("collation","%Library.String",.collval)
   WRITE "initial collation for ",$NAMESPACE,!
     ZWRITE collval
SetNamespaceCollation
  DO SetEnvironment^%apiOBJ("collation","%Library.String","SQLstring")
  SET stat=$$GetEnvironment^%apiOBJ("collation","%Library.String",.collnew)
    WRITE "user-assigned collation for ",$NAMESPACE,!
     ZWRITE collnew
ResetCollationDefault
  DO SetEnvironment^%apiOBJ("collation","%Library.String",.collval)
  SET stat=$$GetEnvironment^%apiOBJ("collation","%Library.String",.collreset)
     WRITE "restored collation default for ",$NAMESPACE,!
     ZWRITE collreset
```

Note that if you have never set the namespace collation default, $$GetEnvironment returns an undefined collation variable, such as *.collval* in this example. This undefined collation defaults to SQLUPPER.

Note: If your data contains German text, uppercase collation may not be a desirable default. This is because the German *eszett* character ($CHAR(223)) has only a lowercase form. The uppercase equivalent is the two letters “SS”. SQL collations that convert to uppercase do not convert *eszett*, which remains unchanged as a single lowercase letter.

10.3 Table Field/Property Definition Collation

Within SQL, collation can be assigned as part of field/property definition. The data type used by a field determines its default collation. The default collation for string data types is SQLUPPER. Non-string data types do not support collation assignment.

You can specify collation for a field in **CREATE TABLE** and **ALTER TABLE**:

SQL

```
CREATE TABLE Sample.MyNames (
    LastName CHAR(30),
    FirstName CHAR(30) COLLATE SQLstring)
```

Note: When specifying collation for a field using **CREATE TABLE** and **ALTER TABLE**, the % prefix is optional: COLLATE SQLstring or COLLATE %SQLstring.

You can specify collation for a property when defining a table using a persistent class definition:

Class Definition

```
Class Sample.MyNames Extends %Persistent [DdlAllowed]
{
Property LastName As %String;
Property FirstName As %String(COLLATION = "SQLstring");
}
```

Note: When specifying collation for class definitions and class methods do not use the % prefix for collation type names.

In these examples, the LastName field takes default collation (SQLUPPER, which is not case-sensitive), the FirstName field is defined with SQLSTRING collation, which is case-sensitive.

If you change the collation for a class property and you already have stored data for that class, any indices on the property become invalid. You must rebuild all indices based on this property.

10.4 Index Definition Collation

The **CREATE INDEX** command cannot specify an index collation type. The index uses the same collation as the field being indexed.

An index defined as part of class definition can specify a collation type. By default, an index on a given property (or properties) uses the collation type of the property data. For example, suppose you have defined a property Name of type %String:

Class Definition

```
Class MyApp.Person Extends %Persistent [DdlAllowed]
{
Property Name As %String;
Index NameIDX On Name;
}
```

The collation for Name is SQLUPPER (the default for %String). Suppose that the Person table contains the following data:

ID	Name
1	Jones
2	JOHNSON
3	Smith
4	jones
5	SMITH

Then an index on Name will contain the following entries:

Name	ID(s)
JOHNSON	2
JONES	1, 4
SMITH	3, 5

The SQL Engine can use this index directly for ORDER BY or comparison operations using the Name field.

You can override the default collation used for an index by adding an As clause to the index definition:

Class Definition

```
Class MyApp.Person Extends %Persistent [DdlAllowed]
{
Property Name As %String;
Index NameIDX On Name As SQLstring;
}
```

In this case the NameIDX index will now store values in SQLSTRING (case-sensitive) form. Using the data from the above example:

Name	ID(s)
JOHNSON	2
Jones	1
jones	4
SMITH	5
Smith	3

In this case, the SQL Engine can take advantage of this index for any queries requiring case-sensitive collation.

In general, you should not have to change the collations of indices. If you want to use a different collation, it is better to define it at the property level and let any indices on the property pick up the correct collation.

If you are performing a property comparison using an indexed property, the property as specified in the comparison should have the same collation type as the corresponding index. For example, the Name property in the WHERE clause of a SELECT or in the ON clause of a JOIN should have the same collation as the index defined for the Name property. If there is a mismatch between the property collation and the index collation, the index may be less effective or may not be used at all. For further details, refer to Index Collation in the "Defining and Building Indices" chapter of the *Caché SQL Optimization Guide*.

If your index is defined to use multiple properties, you can specify the collation of each individually:

Class Member

```
Index MyIDX On (Name As SQLstring, Code As Exact);
```

10.5 Query Collation

Caché SQL provides collation functions that can be used to change the collation or display of a field.

10.5.1 select-item Collation

Applying a collation function to a query select-item changes the display of that item.

- Letter Case: By default, a query displays strings with uppercase and lowercase letters. The exceptions to this are the DISTINCT or GROUP BY operations on a field of collation type SQLUPPER. These operations display that field in all uppercase letters. You can use the %EXACT collation function to reverse this letter case transformation and display the field in uppercase and lowercase letters. You should not use an %SQLUPPER collation function in the *select-item* list to display a field in all uppercase letters. This is because %SQLUPPER adds a space character to the length of the string. Use the UPPER function instead:

  ```
  SELECT TOP 5 Name,$LENGTH(Name) AS NLen,
               %SQLUPPER(Name) AS UpColln,$LENGTH(%SQLUPPER(Name)) AS UpCollLen,
               UPPER(Name) AS UpN,$LENGTH(UPPER(Name)) AS UpLen
  FROM Sample.Person
  ```

- String Truncation: You can use the %TRUNCATE collation function to limit the length of the string data you wish to display. %TRUNCATE is preferable to %SQLUPPER, which adds a space character to the length of the string.

  ```
  SELECT TOP 5 Name,$LENGTH(Name) AS NLen,
               %TRUNCATE(Name,8) AS TruncN,$LENGTH(%TRUNCATE(Name,8)) AS TruncLen
  FROM Sample.Person
  ```

Note that you cannot nest collation functions or case-transformation functions.

- WHERE clause comparisons: Most WHERE clause predicate condition comparisons use the collation type of the field/property. Because string fields default to SQLUPPER, these comparisons are commonly not case-sensitive. You can use the %EXACT collation function to make them case-sensitive:

  The following example returns Home_City string matches regardless of letter case:

  ```
  SELECT Home_City FROM Sample.Person WHERE Home_City = 'albany'
  ```

  The following example returns Home_City string matches that are case-sensitive:

  ```
  SELECT Home_City FROM Sample.Person WHERE %EXACT(Home_City) = 'albany'
  ```

  The SQL Follows operator ( ]) uses the field/property collation type.

  However, the SQL Contains operator ([ ) uses EXACT collation, regardless of the collation type of the field/property:

  ```
  SELECT Home_City FROM Sample.Person WHERE Home_City [ 'c'
  ORDER BY Home_City
  ```

  The %MATCHES and %PATTERN predicate conditions use EXACT collation, regardless of the collation type of the field/property. The **%PATTERN** predicate provides both case-sensitive wildcards and a wildcard ('A') which is not case-sensitive.

- ORDER BY clause: The ORDER BY clause uses the namespace default collation to order string values. Therefore, ORDER BY does not order based on lettercase. You can use %EXACT collation to order strings based on lettercase.

10.5.2 DISTINCT and GROUP BY Collation

By default, these operation use the current namespace collation. The default namespace collation is SQLUPPER.

- DISTINCT: The DISTINCT keyword uses the namespace default collation to eliminate duplicate values. Therefore, DISTINCT Name returns values in all uppercase letters. You can use EXACT collation to return values in mixed uppercase and lowercase. DISTINCT eliminates duplicates that differ only in letter case. To preserve duplicates that differ in case, but eliminate exact duplicates, use EXACT collation. The following example eliminates exact duplicates (but not lettercase variants) and returns all values in mixed uppercase and lowercase:

  ```
  SELECT DISTINCT %EXACT(Name) FROM Sample.Person
  ```

  A UNION involves an implicit DISTINCT operation.

- GROUP BY: The GROUP BY clause uses the namespace default collation to eliminate duplicate values. Therefore, GROUP BY Name returns values in all uppercase letters. You can use EXACT collation to return values in mixed uppercase and lowercase. GROUP BY eliminates duplicates that differ only in letter case. To preserve duplicates that differ in case, but eliminate exact duplicates, you must specify the %EXACT collation function on the GROUP BY clause, not the select-item.

  The following example returns values in mixed uppercase and lowercase; the GROUP BY eliminates duplicates, including those that differ in lettercase:

  ```
  SELECT %EXACT(Name) FROM Sample.Person GROUP BY Name
  ```

  The following example returns values in mixed uppercase and lowercase; the GROUP BY eliminates exact duplicates (but not lettercase variants):

  ```
  SELECT Name FROM Sample.Person GROUP BY %EXACT(Name)
  ```

10.6 Legacy Collation Types

Caché SQL supports several legacy collation types. These are deprecated and not recommended for use with new code, as their purpose is to provide continued support for legacy systems. They are:

- ALPHAUP — Removes all punctuation characters except question marks (“?”) and commas (“,”), and translates all the lowercase letters to uppercase. Used mostly for mapping legacy globals. Replaced by SQLUPPER.
- STRING — Converts a logical value to uppercase, strips all punctuation and white space (except for commas), and adds one leading blank space to the beginning of the string. It collates any value containing only whitespace (spaces, tabs, and so on) as the SQL empty string. Replaced by SQLUPPER.
- UPPER — Translates all lowercase letters into uppercase letters. Used mostly for mapping legacy globals. Replaced by SQLUPPER.
- SPACE — SPACE collation appends a single leading space to a value, forcing it to be evaluated as a string. To establish SPACE collation, CREATE TABLE provides a SPACE collation keyword, and ObjectScript provides a SPACE option in the **Collation()** method of the %SYSTEM.Util class. There is no corresponding SQL collation function.

Note: If a string data type field is defined with EXACT, UPPER, or ALPHAUP collation, and a query applies a %STARTSWITH condition on this field, inconsistent behavior may result. If the *substring* you specify to **%STARTSWITH** is a canonical number (especially a negative and/or fractional number), **%STARTSWITH** may give different results depending on whether the field is indexed. The **%STARTSWITH** should perform as expected if the column is not indexed. If the column is indexed, unexpected results may occur.

10.7 SQL and NLS Collations

The SQL collations described above should not be confused with the Caché NLS collation feature, which provides subscript-level encoding that adhere to particular national language collation requirements. These are two separate systems of providing collations, and they work at different levels of the product.

Caché NLS collations can have a *process-level collation* for the current process, and different collations for specific globals.

To ensure proper functioning when using Caché SQL, it is a requirement that the process-level NLS collation matches exactly the NLS collation of all globals involved, including globals used by the tables and globals used for temporary files such as process private globals and for CACHETEMP globals; otherwise, different processing plans devised by the Query Processor might give different results. In situations where sorting occurs, such as an ORDER BY clause or a range condition, the Query Processor selects the most efficient sorting strategy. It may use an index, use a temporary file in a process-private global, sort within a local array, or use a "]]" (Sorts After) comparison. All these are subscript-type comparisons that adhere to the Caché NLS collation that is in effect, which is why it is necessary that all these types of globals use the exact same NLS collation.

The system creates a global with the data base default collation. You can use the **Create()** method of the %Library.GlobalEdit class to create a global with a different collation. The only requirement is that the specified collation be either built-in (such as the Caché standard) or one of the national collations available in the current locale. See “Using %Library.GlobalEdit to Set Collation For A Global” in *Caché Specialized System Tools and Utilities*.

11

Implicit Joins (Arrow Syntax)

Caché SQL provides a special –> operator as a shorthand for getting values from a related table without the complexity of specifying explicit JOINs in certain common cases. This arrow syntax can be used instead of explicit join syntax, or in combination with explicit join syntax. Arrow syntax performs a left outer join.

Note: Certain combinations of implicit join syntax (arrow syntax) with explicit join syntax are not permitted. You cannot use arrow syntax (–>) in an ON clause. You cannot combine arrow syntax (–>) with symbolic join syntax (*= and =*) as follows: you cannot use arrow syntax on the right side of a left outer join (=*) or on the left side of a right outer join (*=). For further information, refer to the JOIN page of the *Caché SQL Reference*.

Arrow syntax can be used for a reference of a property of a class, or a relationship property of a parent table. Other types of relationships and foreign keys do not support arrow syntax.

11.1 Property Reference

You can use the –> operator as a shorthand for getting values from a "referenced table."

For example, suppose you define two classes: Company:

Class Definition

```
Class Sample.Company Extends %Persistent [DdlAllowed]
{
/// The Company name
Property Name As %String;
}
```

and Employee:

Class Definition

```
Class Sample.Employee Extends %Persistent [DdlAllowed]
{
/// The Employee name
Property Name As %String;

/// The Company this Employee works for
Property Company As Company;
}
```

The Employee class contains a property that is a *reference* to a Company object. Within an object-based application, you can follow this reference using dot syntax. For example, to find the name of a company that an employee works for:

```
name = employee.Company.Name
```

You can perform the same task using an SQL statement that uses an OUTER JOIN to join the Employee and Company tables:

```
SELECT Sample.Employee.Name, Sample.Company.Name AS CompName
FROM Sample.Employee LEFT OUTER JOIN Sample.Company
ON Sample.Employee.Company = Sample.Company.ID
```

Using the –> operator, you can perform the same OUTER JOIN operation more succinctly:

```
SELECT Name, Company->Name AS CompName
FROM Sample.Employee
```

You can use the –> operator any time you have a *reference column* within a table; that is, a column whose value is the ID of a referenced table (essentially a special case of foreign key). In this case, the Company field of Sample.Employee contains IDs of records in the Sample.Company table. You can use the –> operator anywhere you can use a column expression within a query. For example, in a WHERE clause:

```
SELECT Name,Company AS CompID,Company->Name AS CompName
FROM Sample.Employee
WHERE Company->Name %STARTSWITH 'G'
```

This is equivalent to:

```
SELECT E.Name,E.Company AS CompID,C.Name AS CompName
FROM Sample.Employee AS E, Sample.Company AS C
WHERE E.Company = C.ID AND C.Name %STARTSWITH 'G'
```

Note that in this case, this equivalent query uses an INNER JOIN.

The following example uses arrow syntax to access the Spouse field in Sample.Person. As the example shows, the Spouse field in Sample.Employee contains the ID of a record in Sample.Person. This example returns those records where the employee has the same Home_State or Office_State as the Home_State of their spouse:

```
SELECT Name,Spouse,Home_State,Office_State,Spouse->Home_State AS SpouseState
FROM Sample.Employee
WHERE Home_State=Spouse->Home_State OR Office_State=Spouse->Home_State
```

You can use the –> operator in a GROUP BY clause:

```
SELECT Name,Company->Name AS CompName
FROM Sample.Employee
GROUP BY Company->Name
```

You can use the –> operator in an ORDER BY clause:

```
SELECT Name,Company->Name AS CompName
FROM Sample.Employee
ORDER BY Company->Name
```

or refer to a column alias for a –> operator column in an ORDER BY clause:

```
SELECT Name,Company->Name AS CompName
FROM Sample.Employee
ORDER BY CompName
```

Compound arrow syntax is supported, as shown in the following example. In this example, the Cinema.Review table includes the Film field, which contains Row IDs for the Cinema.Film table. The Cinema.Film table includes the Category field, which contains Row IDs for the Cinema.Category table. Thus `Film->Category->CategoryName` accesses these three tables to return the CategoryName of each film that has a ReviewScore:

```
SELECT ReviewScore,Film,Film->Title,Film->Category,Film->Category->CategoryName
FROM Cinema.Review
ORDER BY ReviewScore
```

11.2 Child Table Reference

You can use –> operator to reference a child table. For example, if LineItems is a child table of the Orders table, you can specify:

SQL

```
SELECT LineItems->amount
FROM Orders
```

Note that there is no property called LineItems in Orders; LineItems is the name of a child table that contains the `amount` field. This query produces multiple rows in the result set for each Order row. It is equivalent to:

SQL

```
SELECT L.amount
FROM Orders O LEFT JOIN LineItems L ON O.id=L.custorder
```

Where `custorder` is the parent reference field of the LineItems table.

11.3 Arrow Syntax Privileges

When using arrow syntax, you must have SELECT privileges on the referenced data in both tables. Either you must have a table-level SELECT privilege or a column-level SELECT privilege on the referenced column. With column-level privileges, you need SELECT privilege on the ID of the referenced table, as well as the referenced column.

The following example demonstrates the required column-level privileges:

```
SELECT Name,Company->Name AS CompanyName
FROM Sample.Employee
GROUP BY Company->Name
ORDER BY Company->Name
```

In the above example, you must have column-level SELECT privilege for Sample.Employee.Name, Sample.Company.Name, and Sample.Company.ID:

ObjectScript

```
  ZNSPACE "SAMPLES"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET privchk1="%CHECKPRIV SELECT (Name,ID) ON Sample.Company"
  SET privchk2="%CHECKPRIV SELECT (Name) ON Sample.Employee"
CompanyPrivTest
  SET qStatus = tStatement.%Prepare(privchk1)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  IF rset.%SQLCODE=0 {WRITE !,"have Company privileges",! }
  ELSE {  WRITE !,"No privilege: SQLCODE=",rset.%SQLCODE,! }
EmployeePrivTest
  SET qStatus = tStatement.%Prepare(privchk2)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  IF rset.%SQLCODE=0 {WRITE !,"have Employee privilege",! }
  ELSE {  WRITE !,"No privilege: SQLCODE=",rset.%SQLCODE }
```

12

Using Embedded SQL

You can embed SQL statements within ObjectScript code. These Embedded SQL statements are converted to optimized, executable code at compilation time.

There are two kinds of Embedded SQL:

- A simple Embedded SQL query can only return values from a single row. Simple Embedded SQL can also be used for single-row insert, update, and delete, and for other SQL operations.
- A cursor-based Embedded SQL query can iterate through a query result set, returning values from multiple rows. Cursor-based Embedded SQL can also be used for multiple row update and delete SQL operations.

This chapter discusses the following topics:

- Compiling Embedded SQL and the Macro Preprocessor
- How to embed SQL within ObjectScript
- SQL coding considerations for Embedded SQL
- Using host variables to pass values between Embedded SQL and ObjectScript
- Using an SQL cursor to fetch multiple records in Embedded SQL
- Returning SQLCODE and other Embedded SQL variables
- Auditing Embedded SQL

Note: Embedded SQL cannot be input to the Terminal command line, or specified in an XECUTE statement. To execute SQL from the command line, either use the **$SYSTEM.SQL.Execute()** method or the SQL Shell interface.

Embedded SQL is not supported within Caché Basic. To execute SQL within Basic code, you can do either of the following: use Dynamic SQL, or use Embedded SQL within ObjectScript methods and then call these ObjectScript methods from Basic.

12.1 Compiling Embedded SQL and the Macro Preprocessor

You can use Embedded SQL within methods (provided that they are defined to use ObjectScript) or within ObjectScript .MAC routines. A .MAC routine (or a method using ObjectScript) is processed by the Caché Macro Preprocessor and

converted to .INT (intermediate) code which is subsequently compiled to executable code. The Macro Preprocessor replaces Embedded SQL statements with the code that actually executes the SQL statement.

If an Embedded SQL statement itself contains Caché Macro Preprocessor statements (# commands, ## functions, or $$$macro references) these statements are compiled *before* the SQL code. The may affect **CREATE PROCEDURE**, **CREATE FUNCTION**, **CREATE METHOD**, **CREATE QUERY**, or **CREATE TRIGGER** statements that contain an ObjectScript code body.

An Embedded SQL statement must be able to access all resources necessary for its compilation. If an Embedded SQL statement references a class external to its compilation unit, and that class references data items defined by an #include file, the compilation unit that contains the Embedded SQL statement must also reference the same #include file. For further details, refer to the ObjectScript Macros and the Macro Preprocessor chapter of *Using Caché ObjectScript*.

The Macro Preprocessor provides four preprocessor directives for use with Embedded SQL:

- #sqlcompile mode specifies the compilation mode for Embedded SQL statements coded after this preprocessor directive in the routine. It supports the following two options: `Embedded` (the default) — compiles ObjectScript code and Embedded SQL code at compile time, validating that tables, fields, etc. specified in the Embedded SQL exist at compile time. `Deferred` — compiles ObjectScript code, but defers compiling Embedded SQL code until runtime. This enables you to compile a routine containing SQL that references a table that does not yet exist at compile time.
  - `#sqlcompile mode=Deferred` can be used for a Simple (non-cursor) SELECT, **INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE TABLE** statement.
  - `#sqlcompile mode=Deferred` cannot be used for cursor-based SELECT commands. The compiler fails with a 5663 error when you attempt to compile these commands if they specify a table, field, etc. that does not exist at compile time.
  - `#sqlcompile mode=Deferred` cannot be used for an **UPDATE** or **DELETE** with a WHERE CURRENT OF cursor clause with an undeclared cursor. The compiler fails with an SQLCODE -52 error.

  Note: `#sqlcompile mode=Deferred` should not be confused with the similarly-name **%SYSTEM.SQL.SetCompileModeDeferred()** method and the **%SYSTEM.SQL.GetCompileMode()** method, which are used for a completely different purpose.

- #sqlcompile select specifies the format for data display when returned from a **SELECT** statement, or the required format for data input when specified to an **INSERT** or **UPDATE** statement, or a **SELECT** input host variable. It supports the following six options: `Logical` (the default), `Display`, `ODBC`, `Runtime`, `Text` (synonym for `Display`), and `FDBMS` (see below). If `#sqlcompile select=Runtime`, you can use the **$SYSTEM.SQL.SetSelectMode()** method to change how the data is displayed.

  Regardless of the `#sqlcompile select` option specified, an **INSERT** or **UPDATE** automatically converts the specified data value to its corresponding Logical format for storage.

  Regardless of the `#sqlcompile select` option specified, a **SELECT** automatically converts an input host variable value its corresponding Logical format for predicate matching.

  Using`#sqlcompile select` for query display is shown in the following examples. These examples display the DOB (date of birth) value, then change the SelectMode to ODBC format, then display the DOB again. In the first example, changing the SelectMode has no effect on the display; in the second example, because `#sqlcompile select=Runtime`, changing the SelectMode changes the display:

  ObjectScript

```
#sqlcompile select=Display
&sql(SELECT DOB INTO :a FROM Sample.Person)
WRITE "1st date of birth is ",a,!
DO $SYSTEM.SQL.SetSelectMode(1)
WRITE "changed select mode to: ",$SYSTEM.SQL.GetSelectMode(),!
&sql(SELECT DOB INTO :b FROM Sample.Person)
WRITE "2nd date of birth is ",b
```

ObjectScript

```
#sqlcompile select=Runtime
&sql(SELECT DOB INTO :a FROM Sample.Person)
WRITE "1st date of birth is ",a,!
DO $SYSTEM.SQL.SetSelectMode(1)
WRITE "changed select mode to: ",$SYSTEM.SQL.GetSelectMode(),!
&sql(SELECT DOB INTO :b FROM Sample.Person)
WRITE "2nd date of birth is ",b
```

For further details on SelectMode options, refer to "Data Display Options" in the "Caché SQL Basics" chapter of this book.

– `#sqlcompile select=FDBMS` is provided to enable Embedded SQL to format data in the same way as FDBMS. If a query has a constant value in the WHERE clause, FDBMS mode assumes it to be a Display value and converts it using DisplayToLogical conversion. If a query has a variable in the WHERE clause, FDBMS mode converts it using FDBMSToLogical conversion. The FDBMSToLogical conversion method should be designed to handle the three FDBMS variable formats: Internal, Internal_$c(1)_External, and $c(1)_External. If a query selects into a variable, it invokes the LogicalToFDBMS conversion method. This method returns Internal_$c(1)_External.

- #sqlcompile path (or #import) specifies the schema search path used to resolves unqualified table, view, and stored procedure names in data management commands such as **SELECT**, **CALL**, **INSERT**, **UPDATE**, **DELETE**, and **TRUNCATE TABLE**. If no schema search path is specified, or if the table is not found in the specified schemas, Caché uses the system-wide default schema. `#sqlcompile path` and `#import` are ignored by data definition statements such as **ALTER TABLE**, **DROP VIEW**, **CREATE INDEX**, or **CREATE TRIGGER**. Data definition statements use the system-wide default schema to resolve unqualified names.
- #sqlcompile audit is a boolean switch specifying whether or not the execution of Embedded SQL statements should be recorded in the system events audit log. For further details, refer to Auditing Embedded SQL.

For further details on these preprocessor directives, refer to the Preprocessor Directives Reference section of *Using Caché ObjectScript*.

12.1.1 Recompilation Required following Change to Dependent Class

In Embedded SQL, you must recompile a class or routine that references a persistent class if that persistent class is changed.

For example, Class A contains a method with an Embedded SQL query, and that query references persistent Class B. After Class A has been compiled, Class B gets modified (for example, a new property is added to Class B). Class A now needs to be recompiled. The same is true for Routine A that references persistent Class B.

- Class references class: if you are changing Class B using Studio, when you recompile Class B the Studio **compile dependent classes** setting should also recompile Class A. How to set the compile dependent classes option is described in *Using Studio*.
- Routine references class: if A is a routine that references Class B, you need to recompile Routine A manually.

12.2 Embedded SQL Syntax

The syntax of the Embedded SQL directive is described below.

12.2.1 The &sql Directive

Embedded SQL statements are set off from the rest of the code by the &sql() directive, as shown in the following example:

ObjectScript

```
   NEW SQLCODE,a
   WRITE "Invoking Embedded SQL",!
   &sql(SELECT Name INTO :a FROM Sample.Person)
   WRITE "The SQL error code is ",SQLCODE,!
   IF $DATA(a) {WRITE "The name is ",a}
```

Results are returned using the INTO clause specifying one or more host variables. In this case, the host variable is named :a. For further details, see the "Host Variables" section of this chapter, which includes information on interactions between SQLCODE and host variables.

The following example shows Embedded SQL within a method:

Class Member

```
Method CountStudents() As %Integer
{
    &sql(SELECT COUNT(*) INTO :count
        FROM MyApp.Student)

    Quit count
}
```

The &sql directive is not case-sensitive; you can use &sql, &SQL, &Sql, and so on. The &sql directive must be followed by an open parenthesis, with no intervening spaces, line breaks, or comments. The &sql directive can be used on the same line as a label, as shown in the following example:

ObjectScript

```
Mylabel  &sql(
       SELECT Name INTO :a
       FROM Sample.Person
       )
```

The body of an &sql directive should contain a valid SQL statement, enclosed in parentheses. You can format your SQL statements in any way you like: white space and new lines are ignored by SQL. Studio recognizes the &sql directive and uses an SQL-aware colorizer to syntax color the SQL code statements.

When the Macro Preprocessor encounters an &sql directive, it hands the enclosed SQL statement to the SQL Query Processor. The Query Processor returns the code needed (in ObjectScript INT format) to execute the query. The Macro Preprocessor then replaces the &sql directive with this code (or a call to a label containing the code). From within Studio, you can view the generated code, if you like, by looking at the INT code generated for a class or routine (using the **View Other Code** option from the **View** menu).

If an &sql directive contains an invalid SQL statement, the Macro Preprocessor generates a compilation error. An invalid SQL statement may have syntax errors, or refer to tables or columns that do not exist at compile time.

An &sql directive can contain SQL-style comments anywhere within its parentheses, can contain no SQL code, or contain only comment text. If an &sql directive contains no SQL code or only commented text, the directive is parsed as a no-op and the SQLCODE variable is not defined.

ObjectScript

```
  NEW SQLCODE
  WRITE !,"Entering Embedded SQL"
  &sql()
  WRITE !,"Leaving Embedded SQL"
```

ObjectScript

```
  NEW SQLCODE
  WRITE !,"Entering Embedded SQL"
  &sql(/* SELECT Name INTO :a FROM Sample.Person */)
  WRITE !,"Leaving Embedded SQL"
```

12.2.2 &sql Alternative Syntax

Because complex Embedded SQL programs may contain multiple &sql directives — including nested &sql directives — the following alternative syntax formats are provided:

- `##sql(...)`: this directive is functionally equivalent to &sql. It provides an alternative syntax for clarity of code. However, it cannot include marker syntax.
- `&sql<marker>(...)<reversemarker>`: this directive allows you to specify multiple &sql directives, identifying each with a user-selected marker character or string. This marker syntax is described in the following section.

12.2.3 &sql Marker Syntax

You can identify a specific &sql directive using user-defined marker syntax. This syntax consists of a character or string specified between "&sql" and the open parenthesis character. The reverse of this marker must appear immediately after the closing parenthesis at the end of the Embedded SQL. The syntax is as follows:

```
&sql<marker>( SQL statement )<reverse-marker>
```

Note that no white space (space, tab, or line return) is permitted between &sql, *marker*, and the open parenthesis, and no white space is permitted between the closing parenthesis and *reverse-marker*.

A *marker* can be a single character or a series of characters. A *marker* cannot contain the following punctuation characters:

```
( + - / \ | * )
```

A *marker* cannot contain a whitespace character (space, tab, or line return). It may contain all other printable characters and combinations of characters, including Unicode characters. The *marker* and *reverse-marker* are case-sensitive.

The corresponding *reverse-marker* must contain the same characters as *marker* in the reverse order. For example: `&sqlABC( ... )CBA`. If *marker* contains a [or { character, *reverse-marker* must contain the corresponding] or } character. The following are examples of valid &sql *marker* and *reverse-marker* pairs:

```
&sql@@( ... )@@
&sql[( ... )]
&sqltest( ... )tset
&sql[Aa{( ... )}aA]
```

When selecting a marker character or string, note the following important SQL restriction: the SQL code cannot contain the character sequence "`)<reversemarker>`" anywhere in the code, *including in literal strings and comments*. For example, if the marker is "ABC", the character string ")CBA" cannot appear anywhere in the Embedded SQL code. If this occurs, the combination of a valid marker and valid SQL code will fail compilation. Thus it is important to use care in selecting a *marker* character or string to prevent this collision.

12.2.4 Embedded SQL and Line Offsets

The presence of Embedded SQL affects ObjectScript line offsets, as follows:

- Embedded SQL adds (at least) 2 to the total number of INT code lines at that point in the routine. Therefore, a single line of Embedded SQL counts as 3 lines, two lines of Embedded SQL count as 4 lines, and so forth. Embedded SQL that invokes other code can add many more lines to the INT code.

  A dummy Embedded SQL statement, containing only a comment counts as 2 INT code lines, as in the following example: `&sql( /* for future use */)`.
- All lines within Embedded SQL count as line offsets, including comments and blank lines.

You can display INT code lines using the ^ROUTINE global.

12.3 Embedded SQL Code

Considerations for writing SQL code in Embedded SQL include the following:

- Simple (non-cursor) Embedded SQL statements
- Schema name resolution
- Literal data values
- Data formatting for %List and date/time data values
- Privilege Checking

Output host variables, which are used to export data values from Embedded SQL are described later in this chapter.

12.3.1 Simple SQL Statements

You can use a simple SQL statement (a single Embedded SQL statement) for a variety of operations including:

- INSERT, UPDATE, INSERT OR UPDATE, and DELETE statements.
- DDL statements.
- GRANT and REVOKE statements.
- SELECT statements that return only a single row (or if you are only interested in the first returned row).

Simple SQL statements are also referred to as *non-cursor–based SQL statements*. Cursor-based Embedded SQL is described later in this chapter.

For example, the following statement finds the name of the (one and only) Patient with ID of 43:

ObjectScript

```
&sql(SELECT Name INTO :name
   FROM Patient
   WHERE %ID = 43)
```

If you use a simple statement for a query that can return multiple rows, then only the first row is returned:

ObjectScript

```
&sql(SELECT Name INTO :name
   FROM Patient
   WHERE Age = 43)
```

Depending on the query, there is no guarantee which row will actually be returned first. Also, if a query includes an INTO statement and no data is returned (SQLCODE=100), executing the query may either result in an undefined host variable, or the host variable containing a prior value.

12.3.2 Schema Name Resolution

A table name, view name, or stored procedure name is either qualified (specifies a schema name) or unqualified (does not specify a schema name). If the name does not specify a schema name, Caché resolves the schema name as follows:

- Data Definition: Caché uses the system-wide default schema to resolve an unqualified name. If the default schema does not exist, Caché creates the schema and the corresponding class package. All data definition statements use the system-wide default schema; data definition statements ignore the **#import** and **#sqlcompile path** macro preprocessor directives.
- Data Management: Caché uses the schema search path specified by the #sqlcompile path and/or the #import macro preprocessor directive(s) in effect for the class or routine that contains the Embedded SQL statement. The **#import** and **#sqlcompile path** directives are mutually independent lists of possible schema names with different functionality. Either or both may be used to supply a schema name for an unqualified table, view, or stored procedure name. If no schema search path is specified, Caché uses the system-wide default schema name.

See the chapter "Packages" in *Using Caché Objects* for more details on schemas.

12.3.3 Literal Values

Embedded SQL queries may contain literal values (strings, numbers, or dates). Strings should be enclosed within single (') quotes. (In Caché SQL, double quotes indicate a delimited identifier):

ObjectScript

```
&sql(SELECT 'Employee (' || Name || ')' INTO :name
     FROM Sample.Employee)
WRITE name
```

Numeric values can be used directly. Literal numbers and timestamp values are "lightly normalized" before Caché compares these literal values to field values, as shown in the following example where +0050.000 is normalized to 50:

ObjectScript

```
&sql(SELECT Name,Age INTO :name,:age
     FROM Sample.Person
     WHERE Age = +0050.000)
WRITE name," age=",age
```

Arithmetic, function, and special variable expressions can be specified:

ObjectScript

```
&sql(DECLARE C1 CURSOR FOR
     SELECT Name,Age-65,$HOROLOG INTO :name,:retire,:today
     FROM Sample.Person
     WHERE Age > 60
     ORDER BY Age,Name)
&sql(OPEN C1)
    QUIT:(SQLCODE'=0)
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
   WRITE $ZDATE(today)," ",name," has ",retire," eligibility years",!
  &sql(FETCH C1) }
&sql(CLOSE C1)
```

You can also input a literal value using an input host variable. Input host numeric values are also "lightly normalized." For further details, see the "Host Variables" section of this chapter.

In Embedded SQL, a few character sequences that begin with ## are not permitted within a string literal and must be specified using ##lit. These character sequences are: `##;`, `##beginlit`, `##expression(`, `##function(`, `##quote(`, `##stripq(`, and `##unique(`. For example, the following example fails:

ObjectScript

```
WRITE "Embedded SQL test",!
&sql(SELECT 'the sequence ##unique( is restricted' INTO :x)
WRITE x
```

The following workaround succeeds:

ObjectScript

```
WRITE "Embedded SQL test",!
&sql(SELECT 'the sequence ##lit(##unique() is restricted' INTO :x)
WRITE x
```

12.3.4 Data Format

Within Embedded SQL, data values are in "Logical mode"; that is, values are in the native format used by the SQL Query Processor. For string, integers, and other data types that do not define a **LogicalToODBC** or **LogicalToDisplay** conversion, this has no effect. Data format affects %List data, and the %Date and %Time data types.

The %List data type displays in Logical mode as element values prefaced with non-printing list encoding characters. The **WRITE** command displays these values as concatenated elements. For example, the FavoriteColors field of Sample.Person stores data in %List data type, such as the following: `$LISTBUILD('Red','Black')`. In Embedded SQL this displays in Logical mode as `RedBlack`, with a length of 12 characters. In Display mode it displays as `Red Black`; in ODBC mode it displays as `Red,Black`. This is shown in the following example:

ObjectScript

```
&sql(DECLARE C1 CURSOR FOR
     SELECT TOP 10 FavoriteColors INTO :colors
     FROM Sample.Person WHERE FavoriteColors IS NOT NULL)
&sql(OPEN C1)
    QUIT:(SQLCODE'=0)
&sql(FETCH C1)
WHILE (SQLCODE = 0) {
   WRITE $LENGTH(colors),": ",colors,!
  &sql(FETCH C1) }
&sql(CLOSE C1)
```

The %Date and %Time data types provided by Caché use the Caché internal date representation ($HOROLOG format) as their Logical format. A %Date data type returns INTEGER data type values in Logical mode; VARCHAR data type values in Display mode, and DATE data type values in ODBC mode. The %TimeStamp data type uses ODBC date-time format (YYYY-MM-DD HH:MM:SS) for its Logical, Display, and ODBC format.

For example, consider the following class definition:

Class Definition

```
Class MyApp.Patient Extends %Persistent
{
/// Patient name
Property Name As %String(MAXLEN = 50);

/// Date of birth
Property DOB As %Date;

/// Date and time of last visit
Property LastVisit As %TimeStamp;
}
```

A simple Embedded SQL query against this table will return values in logical mode. For example, consider the following query:

ObjectScript

```
&sql(SELECT Name, DOB, LastVisit
       INTO :name, :dob, :visit
        FROM Patient
       WHERE %ID = :id)
```

This query returns logical value for the three properties into the host variables *name*, *dob*, and *visit*:

Host Variable	Value
name	"Weiss,Blanche"
dob	44051
visit	"2001-03-15 11:11:00"

Note that *dob* is in $HOROLOG format. You can convert this to a display format using the $ZDATETIME function:

ObjectScript

```
Set dob = 44051
Write $ZDT(dob,3),!
```

The same consideration as true within a WHERE clause. For example, to find a Patient with a given birthday, you must use a logical value in the WHERE clause:

ObjectScript

```
&sql(SELECT Name INTO :name
       FROM Patient
       WHERE DOB = 43023)
```

or, alternatively, using a host variable:

ObjectScript

```
Set dob = $ZDH("01/02/1999",1)

&sql(SELECT Name INTO :name
       FROM Patient
       WHERE DOB = :dob)
```

In this case, we use the $ZDATEH function to convert a display format date into its logical, $HOROLOG equivalent.

12.3.5 Privilege Checking

Embedded SQL does not perform SQL privilege checking. You can access all tables, views, and columns and perform any operation, regardless of the privileges assignments. It is assumed that applications using Embedded SQL will check for privileges before using Embedded SQL statements.

You can use the Caché SQL %CHECKPRIV statement in Embedded SQL to determine the current privileges.

For further details, refer to the Users, Roles, and Privileges chapter of this manual.

12.4 Host Variables

A host variable is a local variable that passes a literal value into or out of Embedded SQL. Most commonly, host variables are used to either pass the value of a local variable as an input value into Embedded SQL, or to pass an SQL query result value as an output host variable from an Embedded SQL query.

A host variable cannot be used to specify an SQL identifier, such as a schema name, table name, field name, or cursor name. A host variable cannot be used to specify an SQL keyword.

- Output host variables are only used in Embedded SQL. They are specified in an INTO clause, an SQL query clause that is only supported in Embedded SQL.

- Input host variables can be used in either Embedded SQL or Dynamic SQL. In Dynamic SQL, you can also input a literal to an SQL statement using the "?" input parameter. This "?" syntax cannot be used in Embedded SQL.

Note: Caché Basic does not support Embedded SQL. Either use Dynamic SQL to perform SQL operations from Caché Basic, or have Caché Basic call an ObjectScript routine that contains Embedded SQL.

Within Embedded SQL, input host variables can be used in any place that a literal value can be used. Output host variables are specified using an INTO clause of a **SELECT** or **FETCH** statement.

To use a variable or a property reference as a host variable, precede it with a colon (:). A host variable in embedded Caché SQL can be one of the following:

- One or more ObjectScript local variables, such as :myvar, specified as a comma-separated list. A local variable can be fully formed and can include subscripts. Like all local variables, it is case-sensitive and can contain Unicode letter characters.
- A single ObjectScript local variable array, such as :myvars(). A local variable array can receive only field values from a single table (not joined tables or a view). For details, refer to "Host Variable Subscripted by Column Number", below.
- An object reference, such as :oref.Prop, where Prop is a property name, with or without a leading % character. This can be a simple property or a multidimensional array property, such as :oref.Prop(1). It can be an instance variable, such as :i%Prop or :i%%Data. The property name may be delimited; for example `:Person."Home City"`. Delimited property names can be used even when the **Support Delimited Identifiers** configuration option is not set. Multidimensional properties may include :i%Prop() and :m%Prop() host variable references. An object reference host variable can include any number of dot syntax levels; for example, `:Person.Address.City`.

  When an oref.Prop is used as a host variable inside a procedure block method, the system automatically adds the oref variable (not the entire oref.Prop reference) to the PublicList and **NEW**s it.

Host variables should be listed in the ObjectScript procedure's PublicList variables list and reinitialized using the NEW command. You can configure Caché to also list all host variables used in Embedded SQL in comment text; this is described in the Comment section of *Using Caché SQL*.

Host variable values have the following behavior:

- Input host variables are never modified by the SQL statement code. They retain their original values even after Embedded SQL has run. However, input host variable values are "lightly normalized" before being supplied to the SQL statement code: Valid numeric values are stripped of leading and trailing zeros, a single leading plus sign, and a trailing decimal point. Timestamp values are stripped of trailing spaces, trailing zeros in fractional seconds, and (if there are no fractional seconds) a trailing decimal point.
- In FETCH ... INTO statements, the output host variables in the INTO clause are only modified if SQLCODE equals 0, that is, when a valid row is returned; otherwise, they are not modified.
- In SELECT ... INTO and DECLARE ... SELECT ... INTO statements, the output host variables in the INTO clause are modified if SQLCODE equals 0 (when a valid row is returned), and may have been modified even when SQLCODE is not 0, that is, when no new row was returned.
- In DECLARE ... SELECT ... INTO statements, do not modify the output host variables in the INTO clause between two FETCH calls, since that might cause unpredictable query results.

You must check the SQLCODE value before processing output host variables.

When using a comma-separated list of host variables in the INTO clause, you must specify the same number of host variables as the number of select-items (fields, aggregate functions, scalar functions, arithmetic expressions, literals). Too many or too few host variables results in an SQLCODE -76 cardinality error upon compilation.

This is often a concern when using SELECT * in Embedded SQL. For example, `SELECT * FROM Sample.Person` is only valid with a comma-separated list of 15 host variables (the exact number of non-hidden columns, which, depending on the table definition, may or may not include the system-generated ID (RowId) column). Note that this number of columns may not be a simple correspondence to the number of properties listed in the *InterSystems Class Reference*.

Because the number of columns can change, it is usually not a good idea to specify `SELECT *` with an **INTO** clause list of individual host variables. When using `SELECT *`, it is usually preferable to use a host variable subscripted array, such as the following:

ObjectScript

```
 NEW SQLCODE
&sql(SELECT * INTO :tflds() FROM Sample.Person )
 IF SQLCODE=0 {
   FOR i=0:1:25 {
     IF $DATA(tflds(i)) {
     WRITE "field ",i," = ",tflds(i),! }
   } }
 ELSE {WRITE "SQLCODE=",SQLCODE,! }
```

Note that in this example the field number subscripts are not a continuous sequence; some fields in Sample.Person are hidden and return no data in this example. Using a host variable array is described in "Host Variable Subscripted by Column Number", below.

It is good programming practice to check the SQLCODE value immediately after exiting Embedded SQL. Output host variable values should only be used when SQLCODE=0.

12.4.1 Host Variable Examples

In the following ObjectScript example, an Embedded SQL statement uses output host variables to return a name and home state address from an SQL query to ObjectScript:

ObjectScript

```
&sql(SELECT Name,Home_State
     INTO :CName,:CAddr
     FROM Sample.Person)
IF SQLCODE=0 {
   WRITE !,"Name is: ",CName
   WRITE !,"State is: ",CAddr
}
ELSE {
   WRITE !,"SQLCODE=",SQLCODE
}
```

The Embedded SQL uses an INTO clause that specifies the host variables `:CName` and `:CAddr` to return the selected customer's name in the local variable *CName*, and home state in the local variable *CAddr*.

The following example performs the same operation, using subscripted local variables:

ObjectScript

```
&sql(SELECT Name,Home_State
     INTO :CInfo(1),:CInfo(2)
     FROM Sample.Person)
IF SQLCODE=0 {
   WRITE !,"Name is: ",CInfo(1)
   WRITE !,"State is: ",CInfo(2)
}
ELSE {
   WRITE !,"SQLCODE=",SQLCODE
}
```

These host variables are simple local variables with user-supplied subscripts (`:CInfo(1)`). However, if you omit the subscript (`:CInfo()`), Caché populates the host variable subscripted array using SqlColumnNumber, as described below.

In the following ObjectScript example, an Embedded SQL statement uses both input host variables (in the WHERE clause) and output host variables (in the INTO clause):

ObjectScript

```
SET minval = 10000
SET maxval = 50000
&sql(SELECT Name,Salary INTO :outname, :outsalary
     FROM MyApp.Employee
     WHERE Salary > :minval AND Salary < :maxval)
 IF SQLCODE=0 {
    WRITE !,"Name is: ",outname
    WRITE !,"Salary is: ",outsalary
 }
 ELSE {
    WRITE !,"SQLCODE=",SQLCODE
 }
```

The following example performs “light normalization” on an input host variable. Note that Caché treats the input variable value as a string and does not normalize it, but Embedded SQL normalizes this number to 65 to perform the equality comparison in the WHERE clause:

ObjectScript

```
SET x="+065.000"
&sql(SELECT Name,Age
     INTO :a,:b
     FROM Sample.Person
     WHERE Age=:x)
WRITE !,"Input value is: ",x
IF SQLCODE = 0 {
  WRITE !,"Name value is: ",a
  WRITE !,"Age value is: ",b }
ELSE {WRITE !,"SQLCODE=",SQLCODE }
```

In the following ObjectScript example, an Embedded SQL statement uses object properties as host variables:

ObjectScript

```
 &sql(SELECT Name, Title INTO :obj.Name, :obj.Title
      FROM MyApp.Employee
      WHERE %ID = :id )
```

In this case, *obj* must be a valid reference to an object that has mutable (that is, they can be modified) properties Name and Title. Note that if a query includes an INTO statement and no data is returned (that is, that SQLCODE is 100), then executing the query may result in the value of the host variable being modified.

12.4.2 Host Variable Subscripted by Column Number

If the FROM clause contains a single table, you can specify a subscripted host variable for fields selected from that table; for example, the local array `:myvar()`. The local array is populated by Caché, using each field’s SqlColumnNumber as the numeric subscript. Note that SqlColumnNumber is the column number in the table definition, *not* the *select-list* sequence. (You cannot use a subscripted host variable for fields of a view.)

A host variable array must be a local array that has its lowest level subscript omitted. Therefore, `:myvar()`, `:myvar(5,)`, and `:myvar(5,2,)` are all valid host variable subscripted arrays.

- A host variable subscripted array may be used for input in an **INSERT**, **UPDATE**, or **INSERT OR UPDATE** statement VALUES clause. When used in an **INSERT** or **UPDATE** statement, a host variable array allows you to define which columns are being updated at runtime, rather than at compile time. For **INSERT** and **UPDATE** usage, refer to those commands in the *Caché SQL Reference*.
- A host variable subscripted array may be used for output in a **SELECT** or **DECLARE** statement INTO clause. Subscripted array usage in **SELECT** is shown in the examples that follow.

In the following example, the **SELECT** populates the Cdata array with the values of the specified fields. The elements of Cdata() correspond to the table column definition, *not* the **SELECT** elements. Therefore, the Name field is column 6, the Age field is column 2, and the date of birth (DOB) field is column 3 in Sample.Person:

ObjectScript

```
&sql(SELECT Name,Age,DOB
     INTO :Cdata()
     FROM Sample.Person)
IF SQLCODE=0 {
   WRITE !,"Name is: ",Cdata(6)
   WRITE !,"Age is: ",Cdata(2)
   WRITE !,"DOB is: ",$ZDATE(Cdata(3),1)
  }
ELSE {WRITE !,"SQLCODE=",SQLCODE }
```

The following example uses a subscripted array host variable to return all of the field values of a row:

ObjectScript

```
&sql(SELECT * INTO :Allfields()
     FROM Sample.Person)
IF SQLCODE=0 {
   SET x=1
   WHILE x '="" {
   WRITE !,x," field is ",Allfields(x)
   SET x=$ORDER(Allfields(x)) }
   }
ELSE {WRITE !,"SQLCODE=",SQLCODE }
```

Note that this **WHILE** loop is incremented using **$ORDER** rather than a simple `x=x+1`. This is because in many tables (such as Sample.Person) there may be hidden columns. These cause the column number sequence to be discontinuous.

If the SELECT list contains items that are not fields from that table, such as expressions or arrow-syntax fields, the INTO clause must also contain comma-separated non-array host variables. The following example combines a subscripted array host variable to return values that correspond to defined table columns, and host variables to return values that do not correspond to defined table columns:

ObjectScript

```
&sql(SELECT Name,Home_City,{fn NOW},Age,($HOROLOG-DOB)/365.25,Home_State
     INTO :Allfields(),:timestmp('now'),:exactage
     FROM Sample.Person)
IF SQLCODE=0 {
   SET x=$ORDER(Allfields(""))
   WHILE x '="" {
   WRITE !,x," field is ",Allfields(x)
   SET x=$ORDER(Allfields(x)) }
   WRITE !,"date & time now is ",timestmp("now")
   WRITE !,"exact age is ",exactage
   }
ELSE {WRITE !,"SQLCODE=",SQLCODE }
```

Note that the non-array host variables must match the non-column **SELECT** items in number and sequence.

The use of a host variable as a subscripted array is subject to the following restrictions:

- A subscripted list can only be used when selecting fields from a single table in the FROM clause. This is because when selecting fields from multiple tables, the SqlColumnNumber values may conflict.
- A subscripted list can only be used when selecting table fields. It cannot be used for expressions or aggregate fields. This is because these *select-list* items do not have an SqlColumnNumber value.

For further details on using a host variable array, refer to the INTO clause in the *Caché SQL Reference*.

12.4.3 NULL and Undefined Host Variables

If you specify an input host variable that is not defined, Embedded SQL treats its value as NULL.

ObjectScript

```
NEW x
&sql(SELECT Home_State,:x
     INTO :a,:b
     FROM Sample.Person)
IF SQLCODE=0 {
  WRITE !,"The length of Home_State is: ",$LENGTH(a)
  WRITE !,"The length of x is: ",$LENGTH(b) }
ELSE {WRITE !,"SQLCODE=",SQLCODE }
```

The SQL NULL is equivalent to the ObjectScript "" string (a zero-length string).

If you output a NULL to a host variable, Embedded SQL treats its value as the ObjectScript "" string (a zero-length string). For example, some records in Sample.Person have a NULL Spouse field. After executing this query:

ObjectScript

```
&sql(SELECT Name,Spouse
   INTO :name, :spouse
   FROM Sample.Person
   WHERE Spouse IS NULL)
 IF SQLCODE=0 {
   WRITE !,"Name: ",name," of length ",$LENGTH(name)," defined: ",$DATA(name)
   WRITE !,"Spouse: ",spouse," of length ",$LENGTH(spouse)," defined: ",$DATA(spouse) }
  ELSE {WRITE !,"SQLCODE=",SQLCODE }
```

The host variable, *spouse*, will be set to "" (a zero-length string) to indicate a NULL value.

In the rare case that a table field contains an SQL zero-length string (''), such as if an application explicitly set the field to an SQL '' string, the host variable will contain the special marker value, $CHAR(0) (a string of length 1, containing only a single, ASCII 0 character), which is the ObjectScript representation for the SQL zero-length string. Use of SQL zero-length strings is strongly discouraged.

The following example compares host variables output from an SQL NULL and an SQL zero-length string:

ObjectScript

```
&sql(SELECT '',Spouse
   INTO :zls, :spouse
   FROM Sample.Person
   WHERE Spouse IS NULL)
 IF SQLCODE=0 {
   WRITE "In ObjectScript"
   WRITE !,"ZLS is of length ",$LENGTH(zls)," defined: ",$DATA(zls)
   WRITE !,"NULL is of length ",$LENGTH(spouse)," defined: ",$DATA(spouse) }
  ELSE {WRITE !,"SQLCODE=",SQLCODE }
```

Note that this host variable NULL behavior is only true within server-based queries (Embedded SQL and Dynamic SQL). Within ODBC and JDBC, NULL values are explicitly specified using the ODBC or JDBC interface.

12.4.4 Validity of Host Variables

- Input host variables are never modified by Embedded SQL.
- Output host variables are only reliably valid after Embedded SQL when SQLCODE = 0.

For example, the following use of *OutVal* is *not* reliably valid:

ObjectScript

```
InvalidExample
   SET InVal = "1234"
   SET OutVal = "None"
   &sql(SELECT Name
        INTO :OutVal
        FROM Sample.Person
        WHERE %ID=:InVal)
   IF OutVal="None" {           ; Improper Use
   WRITE !,"No data returned"
   WRITE !,"SQLCODE=",SQLCODE }
   ELSE {
   WRITE !,"Name is: ",OutVal }
```

The value of *OutVal* set before invoking Embedded SQL should *not* be referenced by the **IF** command after returning from Embedded SQL.

Instead, you should code this example as follows, using the SQLCODE variable:

ObjectScript

```
ValidExample
   SET InVal = "1234"
   &sql(SELECT Name
        INTO :OutVal
        FROM Sample.Person
        WHERE %ID=:InVal)
   IF SQLCODE'=0 { SET OutVal="None"
      IF OutVal="None" {
      WRITE !,"No data returned"
      WRITE !,"SQLCODE=",SQLCODE } }
   ELSE {
   WRITE !,"Name is: ",OutVal }
```

The Embedded SQL sets the SQLCODE variable to 0 to indicate the successful retrieval of an output row. An SQLCODE value of 100 indicates that no row was found that matches the **SELECT** criteria. An SQLCODE negative number value indicates a SQL error condition.

12.4.5 Host Variables and Procedure Blocks

If your Embedded SQL is within a procedure block, all input and output host variables must be public. This can be done by declaring them in the PUBLIC section at the beginning of the procedure block, or by naming them with an initial % character (which automatically makes them public). You must also declare SQLCODE as public. For further details on the SQLCODE variable, see below.

In the following procedure block example, the host variables *zip*, *city*, and *state*, as well as the SQLCODE variable are declared as PUBLIC. The SQL system variables %ROWCOUNT, %ROWID, and %msg are already public, because their names begin with a % character. The procedure code then performs a **NEW** on SQLCODE, the other SQL system variables, and the *state* local variable:

ObjectScript

```
UpdateTest(zip,city)
  [SQLCODE,zip,city,state] PUBLIC {
  NEW SQLCODE,%ROWCOUNT,%ROWID,%msg,state
  SET state="MA"
  &sql(UPDATE Sample.Person
       SET Home_City = :city, Home_State = :state
       WHERE Home_Zip = :zip)
  QUIT %ROWCOUNT
  }
```

12.5 SQL Cursors

A cursor is a pointer to data that allows an Embedded SQL program to perform an operation on the record pointed to. By using a cursor, Embedded SQL can iterate through a result set. Embedded SQL can use a cursor to execute a query that returns data from multiple records. Embedded SQL can also use a cursor to update or delete multiple records.

You must first DECLARE an SQL cursor, giving it a name. In the **DECLARE** statement you supply a **SELECT** statement that identifies which records the cursor will point to. You then supply this cursor name to the OPEN cursor statement. You then repeatedly issue the FETCH cursor statement to iterate through the **SELECT** result set. You then issue a CLOSE cursor statement.

- A cursor-based query uses DECLARE cursorname CURSOR FOR SELECT to select records and (optionally) return select column values into output host variables. The FETCH statement iterates through the result set, using these variables to return selected column values.
- A cursor-based DELETE or UPDATE uses DECLARE cursorname CURSOR FOR SELECT to select records for the operation. No output host variables are specified. The FETCH statement iterates through the result set. The **DELETE** or **UPDATE** statement contains a **WHERE CURRENT OF** clause to identify the current cursor position in order to perform the operation on the selected record. For further details on cursor-based **DELETE** and **UPDATE**, refer to the WHERE CURRENT OF page in *Caché SQL Reference*.

Note that a cursor cannot span methods. Therefore, you must declare, open, fetch, and close a cursor within the same class method. It is important to consider this with all code that generates classes and methods, such as classes generated from a .CSP file.

The following example, uses a cursor to execute a query and display the results to the principal device:

ObjectScript

```
&sql(DECLARE C1 CURSOR FOR
   SELECT %ID,Name
   INTO :id, :name
   FROM Sample.Person
   WHERE Name %STARTSWITH 'A'
   ORDER BY Name
)

&sql(OPEN C1)
     QUIT:(SQLCODE'=0)
&sql(FETCH C1)

While (SQLCODE = 0) {
    Write id, ":  ", name,!
   &sql(FETCH C1)
}

&sql(CLOSE C1)
```

This example does the following:

1. It declares a cursor, *C1*, that returns a set of Person rows ordered by Name.
2. It opens the cursor.
3. It calls FETCH on the cursor until it reaches the end of the data. After each call to FETCH, the SQLCODE variable will be set to 0 if there is more data to fetch. After each call to FETCH, the values returned are copied into the host variables specified by the INTO clause of the DECLARE statement.
4. It closes the cursor.

12.5.1 The DECLARE Cursor Statement

The DECLARE statement specifies both the cursor name and the SQL SELECT statement that defines the cursor. The DECLARE statement must occur within a routine *before* any statements that use the cursor.

A cursor name must be unique within a class or routine. For this reason, a routine that is called recursively cannot contain a cursor declaration. In this situation, it may be preferable to use Dynamic SQL.

The following example declares a cursor named *MyCursor*:

ObjectScript

```
&sql(DECLARE MyCursor CURSOR FOR
   SELECT Name, DOB
   FROM Sample.Person
   WHERE Home_State = :state
   ORDER BY Name
)
```

A DECLARE statement may include an optional INTO clause that specifies the names of the local host variables that will receive data as the cursor is traversed. For example, we can add an INTO clause to the previous example:

ObjectScript

```
&sql(DECLARE MyCursor CURSOR FOR
   SELECT Name, DOB
   INTO :name, :dob
   FROM Sample.Person
   WHERE Home_State = :state
   ORDER BY Name
)
```

The INTO clause may contain a comma-separated list of host variables, a single host variable array, or a combination of both. If specified as a comma-separated list, the number of INTO clause host variables must exactly match the number of columns within the cursor's SELECT list or you will receive a "Cardinality Mismatch" error when the statement is compiled.

If the DECLARE statement does not include an INTO clause, then the INTO clause must appear within the FETCH statement.

12.5.2 The OPEN Cursor Statement

The OPEN statement prepares a cursor for subsequent execution:

ObjectScript

```
&sql(OPEN MyCursor)
```

Upon a successful call to OPEN, the SQLCODE variable will be set to 0.

You cannot FETCH data from a cursor without first calling OPEN.

Depending on the actual query used for the cursor, the OPEN statement may do very little actual work or it may perform some initialization work for the query.

12.5.3 The FETCH Cursor Statement

The FETCH statement fetches the data for the next row of the cursor (as defined by the cursor query):

ObjectScript

```
&sql(FETCH MyCursor)
```

You must DECLARE and OPEN a cursor before you can call FETCH on it.

A FETCH statement may contain an INTO clause that specifies the names of the local host variables that will receive data as the cursor is traversed. For example, we can add an INTO clause to the previous example:

ObjectScript

```
&sql(FETCH MyCursor INTO :a, :b)
```

The INTO clause may contain a comma-separated list of host variables, a single host variable array, or a combination of both. If specified as a comma-separated list, the number of INTO clause host variables must exactly match the number of columns within the cursor's SELECT list or you will receive an SQLCODE -76 "Cardinality Mismatch" error when the statement is compiled.

Commonly, the INTO clause is specified in the DECLARE statement, not the FETCH statement. If both the SELECT query in the DECLARE statement and the FETCH statement contain an INTO clause, the host variables specified by both statements are set.

If FETCH retrieves data, the SQLCODE variable is set to 0; if there is no data (or no more data) to FETCH, SQLCODE is set to 100 (No more data). Host variable values should only be used when SQLCODE=0.

Depending on the query, the first call to FETCH may perform additional tasks (such as sorting values within a temporary data structure).

12.5.4 The CLOSE Cursor Statement

The CLOSE statement terminates the execution of a cursor:

ObjectScript

```
&sql(CLOSE MyCursor)
```

The CLOSE statement cleans up any temporary storage used by the execution of a query. Programs that fail to call CLOSE will experience resource leaks (such as unneeded increase of the CACHETEMP temporary database).

Upon a successful call to CLOSE, the SQLCODE variable is set to 0. Therefore, before closing a cursor you should check whether the final FETCH set SQLCODE to 0 or 100.

12.6 Embedded SQL Variables

The following local variables have specialized uses in Embedded SQL. These local variable names are case-sensitive. At process initiation, these variables are undefined. They are set by Embedded SQL operations. They can also be set directly using the **SET** command, or reset to undefined using the **NEW** command. Like any local variable, a value persists for the duration of the process or until set to another value or undefined using **NEW**. For example, some successful Embedded SQL operations do not set %ROWID; following these operations, %ROWID remains set to its prior value.

- %msg
- %ROWCOUNT
- %ROWID
- SQLCODE

These local variables are not set by Dynamic SQL. (Note that the SQL Shell and the Management Portal SQL interface execute Dynamic SQL.) Instead, Dynamic SQL sets corresponding object properties.

The following ObjectScript special variables are used in Embedded SQL. These special variable names are not case-sensitive. At process initiation, these variables are initialized to a value. They are set by Embedded SQL operations. They cannot be set directly using the **SET** or **NEW** commands.

- $TLEVEL
- $USERNAME

As part of the defined Caché Embedded SQL interface, Caché may set any of these variables during Embedded SQL processing.

If the Embedded SQL is in a class method (with ProcedureBlock=ON), the system automatically places all of these variables in the PublicList and **NEW**s the SQLCODE, %ROWID, %ROWCOUNT, %msg, and all non-% variables used by the SQL statement. It does not **NEW** the %ok variable. You can pass these variables by reference to/from the method; variables passed by reference will not be **NEW**ed automatically in the class method procedure block.

If the Embedded SQL is in a routine, it is the responsibility of the programmer to NEW the %msg, %ok, %ROWCOUNT, %ROWID, and SQLCODE variables before invoking Embedded SQL. NEWing these variables prevents interference with prior settings of these variables. To avoid a <FRAMESTACK> error, you should not perform this **NEW** operation within an iteration cycle.

12.6.1 %msg

A variable that contains a system-supplied error message string. Caché SQL only sets %msg if it has set SQLCODE to a negative integer, indicating an error. If SQLCODE is set to 0 or 100, the %msg variable is unchanged from its prior value.

This behavior differs from the corresponding Dynamic SQL *%Message* property, which is set to the empty string when there is no current error.

In some cases, a specific SQLCODE error code may be associated with more than one %msg string, describing different conditions that generated the SQLCODE. %msg can also take a user-defined message string. This is most commonly used to issue a user-defined message from a trigger when trigger code explicitly sets %ok=0 to abort the trigger.

An error message string is generated in the NLS language in effect for the process when the SQL code is executed. The SQL code may be compiled in a different NLS language environment; the message will be generated according to the runtime NLS environment. See *$SYS.NLS.Locale.Language*.

12.6.2 %ROWCOUNT

An integer counter that indicates the number of rows affected by a particular statement.

- **INSERT**, **UPDATE**, **INSERT OR UPDATE**, **DELETE**, and **TRUNCATE TABLE** set %ROWCOUNT to the number of rows affected. An **INSERT** command with explicit values can only affect one row, and thus sets %ROWCOUNT to either 0 or 1. An **INSERT** query results, an **UPDATE**, or a **DELETE** can affect multiple rows, and can thus set %ROWCOUNT to 0 or a positive integer.
- **SELECT** with no declared cursor can only act upon a single row, and thus execution of a simple **SELECT** always sets %ROWCOUNT to either 1 (single row that matched the selection criteria retrieved) or 0 (no rows matched the selection criteria).
- **DECLARE cursorname CURSOR FOR SELECT** does not initialize %ROWCOUNT; %ROWCOUNT is unchanged following the **SELECT**, and remains unchanged following **OPEN cursorname**. The first successful **FETCH** sets %ROWCOUNT. If no rows matched the query selection criteria, **FETCH** sets %ROWCOUNT=0; if **FETCH** retrieves a row that matched the query selection criteria, it sets %ROWCOUNT=1. Each subsequent **FETCH** that retrieves a row increments %ROWCOUNT. Upon **CLOSE** or when **FETCH** issues an SQLCODE 100 (No Data, or No More Data), %ROWCOUNT contains the total number of rows retrieved.

This **SELECT** behavior differs from the corresponding Dynamic SQL *%ROWCOUNT* property, which is set to 0 upon completion of query execution, and is only incremented when the program iterates through the result set returned by the query.

If a **SELECT** query returns only aggregate functions, every **FETCH** sets %ROWCOUNT=1. The first **FETCH** always completes with SQLCODE=0, even when there is no data in the table; any subsequent **FETCH** completes with SQLCODE=100 and sets %ROWCOUNT=1.

The following Embedded SQL example declares a cursor and uses **FETCH** to fetch each row in the table. When the end of data is reached (SQLCODE=100) %ROWCOUNT contains the number of rows retrieved:

ObjectScript

```
   SET name="LastName,FirstName",state="##"
   &sql(DECLARE EmpCursor CURSOR FOR
        SELECT Name, Home_State
        INTO :name,:state FROM Sample.Person
        WHERE Home_State %STARTSWITH 'M')
   WRITE !,"BEFORE: Name=",name," State=",state
   &sql(OPEN EmpCursor)
      QUIT:(SQLCODE'=0)
   FOR { &sql(FETCH EmpCursor)
        QUIT:SQLCODE
        WRITE !,"Row fetch count: ",%ROWCOUNT
        WRITE " Name=",name," State=",state
}
   WRITE !,"Final Fetch SQLCODE: ",SQLCODE
   &sql(CLOSE EmpCursor)
   WRITE !,"AFTER: Name=",name," State=",state
   WRITE !,"Total rows fetched: ",%ROWCOUNT
```

The following Embedded SQL example performs an **UPDATE** and sets the number of rows affected by the change:

ObjectScript

```
 &sql(UPDATE MyApp.Employee
     Set Salary = (Salary * 1.1)
     WHERE Salary < 50000)
 Write "Employees: ", %ROWCOUNT,!
```

Keep in mind that all Embedded SQL statements (within a given process) modify the %ROWCOUNT variable. If you need the value provided by %ROWCOUNT, be sure to get its value before executing additional Embedded SQL statements. Depending on how Embedded SQL is invoked, you may have to **NEW** the %ROWCOUNT variable before entering Embedded SQL.

Also note that *explicitly* rolling back a transaction will not affect the value of %ROWCOUNT. For example, the following will report that changes have been made, even though they have been rolled back:

ObjectScript

```
 TSTART // start an explicit transaction
  NEW SQLCODE,%ROWCOUNT,%ROWID
 &sql(UPDATE MyApp.Employee
     Set Salary = (Salary * 1.1)
     WHERE Salary < 50000)

 TROLLBACK // force a rollback; this will NOT modify %ROWCOUNT
 Write "Employees: ", %ROWCOUNT,!
```

Implicit transactions (such as if an **UPDATE** fails a constraint check) are reflected by %ROWCOUNT.

12.6.3 %ROWID

When you initialize a process, %ROWID is undefined. When you issue a **NEW %ROWID** command, %ROWID is reset to undefined. %ROWID is set by the Embedded SQL operations described below. If the operation is not successful, or completes successfully but does not fetch or modify any rows, the %ROWID value remains unchanged from its prior value:

either undefined, or set to a value by a previous Embedded SQL operation. For this reason, it is important to NEW %ROWID before each Embedded SQL operation.

%ROWID is set to the RowID of the last row affected by the following operations:

- **INSERT**, **UPDATE**, **INSERT OR UPDATE**, **DELETE**, or **TRUNCATE TABLE**: After a single-row operation, the %ROWID variable contains the system-assigned value of the RowID (Object ID) assigned to the inserted, updated, or deleted record. After a multiple-row operation, the %ROWID variable contains the system-assigned value of the RowID (Object ID) of the last record inserted, updated, or deleted. If no record is inserted, updated, or deleted, the %ROWID variable value is unchanged.
- Cursor-based **SELECT**: The **DECLARE cursorname CURSOR** and **OPEN cursorname** statements do not initialize %ROWID; the %ROWID value is unchanged from its prior value. The first successful **FETCH** sets %ROWID. Each subsequent **FETCH** that retrieves a row resets %ROWID to the current RowID. **FETCH** sets %ROWID if it retrieves a row of an updateable cursor. An updateable cursor is one in which the top FROM clause contains exactly one element, either a single table name or an updateable view name. If the cursor is not updateable, %ROWID remains unchanged. If no rows matched the query selection criteria, **FETCH** does not change the prior the %ROWID value (if any). Upon **CLOSE** or when **FETCH** issues an SQLCODE 100 (No Data, or No More Data), %ROWID contains the RowID of the last row retrieved.

  Cursor-based **SELECT** with a DISTINCT keyword or a GROUP BY clause does not set %ROWID. The %ROWID value is unchanged from its previous value (if any).

  Cursor-based **SELECT** with an aggregate function does not set %ROWID if it returns only aggregate function values. If it returns both field values and aggregate function values, the %ROWID value for every **FETCH** is set to the RowID of the last row returned by the query.
- **SELECT** with no declared cursor does not set %ROWID. The %ROWID value is unchanged upon the completion of a simple **SELECT** statement.

In Dynamic SQL, the corresponding *%ROWID* property returns the RowID of the last record inserted, updated, or deleted. Dynamic SQL does not return a %ROWID property value when performing a **SELECT** query.

You can retrieve the current %ROWID from ObjectScript using the following method call:

ObjectScript

```
WRITE $SYSTEM.SQL.GetROWID()
```

Following an **INSERT**, **UPDATE**, **DELETE**, **TRUNCATE TABLE**, or Cursor-based **SELECT** operation, the LAST_IDENTITY SQL function returns the value of the IDENTITY field for the most-recently modified record. If the table does not have an IDENTITY field, this function returns the RowID for the most-recently modified record.

12.6.4 SQLCODE

After running an embedded SQL Query, you must check the SQLCODE before processing the output host variables.

If SQLCODE=0 the query completed successfully and returned data. The output host variables contain field values.

If SQLCODE=100 the query completed successfully, but output host variable values may differ. Either:

- The query returned one or more rows of data (SQLCODE=0), then reached the end of the data (SQLCODE=100), in which case output host variables are set to the field values of the last row returned. %ROWCOUNT>0.
- The query returned no data, in which case the output host variables are undefined. %ROWCOUNT=0.

If a query returns only aggregate functions, the first **FETCH** always completes with SQLCODE=0 and %ROWCOUNT=1, even when there is no data in the table. The second **FETCH** completes with SQLCODE=100 and %ROWCOUNT=1. If

there is no data in the table or no data matches the query conditions, the query sets output host variables to 0 or the empty string, as appropriate.

If SQLCODE is a negative number the query failed with an error condition. For a list of these error codes and additional information, refer to the SQLCODE Values and Error Messages chapter of the *Caché Error Reference*.

Depending on how Embedded SQL is invoked, you may have to NEW the SQLCODE variable before entering Embedded SQL. Within trigger code, setting SQLCODE to a nonzero value automatically sets %ok=0, aborting and rolling back the trigger operation.

In Dynamic SQL, the corresponding *%SQLCODE* property returns SQL error code values.

12.6.5 $TLEVEL

The transaction level counter. Caché SQL initializes **$TLEVEL** to 0. If there is no current transaction, **$TLEVEL** is 0.

- An initial START TRANSACTION sets **$TLEVEL** to 1. Additional **START TRANSACTION** statements have no effect on **$TLEVEL**.
- Each SAVEPOINT statement increments **$TLEVEL** by 1.
- A ROLLBACK TO SAVEPOINT pointname statement decrements **$TLEVEL**. The amount of decrement depends on the savepoint specified.
- A COMMIT resets **$TLEVEL** to 0.
- A ROLLBACK resets **$TLEVEL** to 0.

You can also use the %INTRANSACTION statement to determine if a transaction is in progress.

$TLEVEL is also set by ObjectScript transaction commands. For further details, refer to the $TLEVEL special variable in the *Caché ObjectScript Reference*.

12.6.6 $USERNAME

The SQL username is the same as the Caché username, stored in the ObjectScript $USERNAME special variable. The username can be used as the system-wide default schema or as an element in the schema search path.

12.7 Auditing Embedded SQL

Caché supports optional auditing of Embedded SQL statements. Embedded SQL auditing is performed when the following two requirements are met:

1. The %System/%SQL/EmbeddedStatement system audit event is enabled system-wide. By default, this system audit event is not enabled. To enable, go to Management Portal, **System Administration**, select **Security**, then **Auditing**, then **Configure System Events**.
2. The routine containing the Embedded SQL statement must contain the #sqlcompile audit macro preprocessor directive. If this directive is set to ON, any Embedded SQL statement following it in the compiled routine is audited when executed.

Auditing records information in the Audit Database. To view the Audit Database, go to the Management Portal, **System Administration**, select **Security**, then **Auditing**, then **View Audit Database**. You can set the **Event Name** filter to Embedded-Statement to limit the **View Audit Database** to Embedded SQL statements. The Audit Database lists Time (a local timestamp), User, PID (process ID), and the Description, which specifies the type of Embedded SQL statement. For example, `SQL SELECT Statement`.

By selecting the **Details** link for an event you can list additional information, including the **Event Data**. The Event Data includes the SQL statement executed and the values of any input arguments to the statement. For example:

```
SELECT TOP :n Name,ColorPreference INTO :name,:color FROM Sample.Stuff WHERE Name %STARTSWITH :letter
Parameter values:
n=5
letter="F"
```

Caché also supports auditing of Dynamic SQL statements (**Event Name**=DynamicStatement) and ODBC and JDBC statements (**Event Name**=XDBCStatement).

13

Using Dynamic SQL

This chapter discusses Dynamic SQL, queries and other SQL statements that are prepared and executed at runtime. It includes the following topics:

- An introduction to Dynamic SQL
- Comparing Dynamic SQL and Embedded SQL
- The %SQL.Statement class
- Creating an object instance and specifying its properties
- Preparing an SQL statement
- Executing an SQL statement
- Returning the entire result set
- Returning specific values from a result set
- Returning multiple result sets
- Working with metadata
- Auditing Dynamic SQL

This chapter describes Dynamic SQL programming using the %SQL.Statement class, which is the preferred implementation of Dynamic SQL. All statements about Dynamic SQL in this chapter, and throughout our documentation, refer specifically to the %SQL.Statement implementation, unless otherwise indicated. You can also create Dynamic SQL programs using the older %ResultSet.SQL class or the %Library.ResultSet class, as described in the Dynamic SQL Using the Older %ResultSet.SQL and %Library.ResultSet Classes chapter of this manual.

13.1 Introduction to Dynamic SQL

Dynamic SQL refers to SQL statements that are prepared and executed at runtime. Dynamic SQL lets you program within Caché in a manner similar to an ODBC or JDBC application (except that you are executing the SQL statement within the same process context as the database engine).

Dynamic SQL can be invoked from either an ObjectScript program or a Caché Basic program.

Dynamic SQL can be used to perform an SQL query. It can also be used to issue other SQL statements. The examples in this chapter perform a **SELECT** query. For Dynamic SQL program examples invoking CREATE TABLE, INSERT, UPDATE, DELETE, or CALL, refer to these commands in the *Caché SQL Reference*.

Dynamic SQL is used in the execution of the Caché SQL Shell, the Caché Management Portal Execute Query interface, the SQL Code Import methods, and the Data Import and Export Utilities. The maximum size of a row in Dynamic SQL (and applications that use it) is 32,767 characters. This limitation can be greatly expanded by configuring long string operations.

13.1.1 Dynamic SQL versus Embedded SQL

Dynamic SQL differs from Embedded SQL in the following ways:

- Dynamic SQL queries are prepared at program execution time, not compilation time. This means that the compiler cannot check for errors at compilation time and preprocessor macros cannot be used within Dynamic SQL. It also means that executing programs can create specialized Dynamic SQL queries in response to user or other input.
- Dynamic SQL can issue a **CREATE TABLE** or **CREATE VIEW** and perform an **INSERT** or **SELECT** on that table or view in the same routine. Embedded SQL, because it is compiled, cannot do this.
- Dynamic SQL executes slightly less efficiently than Embedded SQL, because it does not generate in-line code for queries. However, re-execution of a Dynamic SQL query is substantially faster than the first execution of the query because Dynamic SQL supports cached queries.
- Dynamic SQL can accept a literal value input to a query in two ways: input parameters specified using the "?" character, and input host variables (for example, :var). Embedded SQL uses input and output host variables (for example, :var).
- Dynamic SQL output values are retrieved using the API of the result set object (that is, the Data property). Embedded SQL uses host variables (for example, :var) with the **INTO** clause of a **SELECT** statement to output values.
- Dynamic SQL sets the %SQLCODE, %Message, %ROWCOUNT, and %ROWID object properties. Embedded SQL sets the corresponding SQLCODE, %msg, %ROWCOUNT, and %ROWID local variables. Dynamic SQL does not set %ROWID for a **SELECT** query; Embedded SQL sets %ROWID for a cursor-based **SELECT** query.
- Dynamic SQL can be invoked from either ObjectScript or Caché Basic. Embedded SQL can only be invoked from ObjectScript.
- Dynamic SQL provides an easy way to find query metadata (such as quantity and names of columns).
- Queries prepared by Dynamic SQL are maintained within the query cache so that subsequent calls to prepare the same query can reuse previously generated code. Embedded SQL generates in-line code at compilation time and does not need to use the query cache. Note that Dynamic SQL does not cache most non-query SQL statements, because these statements are commonly only used once. Refer to the "Cached Queries" chapter of the *Caché SQL Optimization Guide* for further details.
- Dynamic SQL performs SQL privilege checking; you must have the appropriate privileges to access or modify a table, field, etc. Embedded SQL does not perform SQL privilege checking. Refer to the SQL %CHECKPRIV statement for further details.
- Dynamic SQL cannot access a private class method. To access an existing class method, the method must be made public. This is a general SQL limitation. However, Embedded SQL gets around this limitation because the Embedded SQL operation itself is a method of the same class.

Dynamic SQL and Embedded SQL use the same data representation (logical mode by default, but this can be changed) and NULL handling.

13.2 The %SQL.Statement Class

The preferred interface for Dynamic SQL is the %SQL.Statement class. To prepare and execute Dynamic SQL statements, use an instance of %SQL.Statement. The result of executing a Dynamic SQL statement is an SQL statement result object that is an instance of the %SQL.StatementResult class. An SQL statement result object is either a unitary value, a result set, or a context object. In all cases, the result object supports a standard interface. Each result object initializes the %SQLCODE, %Message and other result object properties; The values these properties are set to depends on the SQL statement issued. For a successfully executed **SELECT** statement, the object is a result set (specifically, an instance of %SQL.IResultSet) and supports the expected result set functionality.

The following ObjectScript code prepares and executes a Dynamic SQL query:

ObjectScript

```
  /* Simple %SQL.Statement example */
  SET $NAMESPACE="SAMPLES"
  SET myquery = "SELECT TOP 5 Name,DOB FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  DO rset.%Display()
  WRITE !,"End of data"
```

The following is the same Dynamic SQL query in Caché Basic:

```
myquery = "SELECT TOP 5 Name,DOB FROM Sample.Person"
tStatement = New %SQL.Statement()
qStatus = tStatement.%Prepare(myquery)
If qStatus<>1 Then
   PrintLn "%Prepare failed: "
   PrintLn Piece(qStatus," ",3,10)
Else
  rset = tStatement.%Execute()
  CALL rset.%Display()
  PrintLn
  PrintLn "End of data"
End If
```

The examples in this chapter use methods associated with the %SQL.Statement and %SQL.StatementResult classes.

13.3 Creating an Object Instance

You can create an instance of the %SQL.Statement class using the **%New()** class method in ObjectScript:

ObjectScript

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

Or in Caché Basic:

Basic example:

```
tStatement = New %SQL.Statement()
```

At this point the result set object is ready to prepare an SQL statement. Once you have created an instance of the %SQL.Statement class, you can use that instance to issue multiple Dynamic SQL queries and/or **INSERT**, **UPDATE**, or **DELETE** operations.

%New() accepts three optional comma-separated parameters in the following order:

1. %SelectMode, which specifies the mode used for data input and data display.
2. %SchemaPath, which specifies the search path used to supply the schema name for an unqualified table name.
3. %Dialect, which specifies the Transact-SQL (TSQL) Sybase or MSSQL dialect. The default is Caché SQL.

There is also an %ObjectSelectMode property, which cannot be set as a **%New()** parameter. %ObjectSelectMode specifies the data type binding of fields to their related object properties.

In the following ObjectScript example, the %SelectMode is 2 (Display mode), and the %SchemaPath specifies "Sample" as the default schema:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New(2,"Sample")
```

In the following ObjectScript example, a %SelectMode is not specified (note the placeholder comma), and the %SchemaPath specifies a schema search path containing three schema names:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New(,"MyTests,Sample,Cinema")
```

13.3.1 %SelectMode Property

The *%SelectMode* property specifies one of the following modes: 0=Logical (the default), 1=ODBC, 2=Display. These modes specify how a data value is input and displayed. A mode is most commonly used for date and time values and for displaying %List data (a string containing an encoded list). Data is stored in Logical mode.

A **SELECT** query uses the %SelectMode value to determine the format used to display data.

An **INSERT** or **UPDATE** operation uses the %SelectMode value to determine the permitted format(s) for data input.

%SelectMode is used for data display. SQL statements run internally in Logical mode. For example, an **ORDER BY** clause orders records based on their Logical values, regardless of the %SelectMode setting. SQL functions use Logical values, regardless of the %SelectMode setting. Methods projected as SQLPROC also run in Logical mode. SQL routines called as functions in an SQL statement need to return the function value in Logical format.

- For a **SELECT** query, %SelectMode specifies the format used for displaying the data. Setting %SelectMode to ODBC or Display also affects the data format used for specifying comparison predicate values. Some predicate values must be specified in the %SelectMode format, other predicate values must be specified in Logical format, regardless of the %SelectMode. For details, refer to Overview of Predicates in the *Caché SQL Reference*.
  - Time data type data in %SelectMode=1 (ODBC) can display fractional seconds, which is not the same as actual ODBC time. The Caché Time data type supports fractional seconds. The corresponding ODBC TIME data type (TIME_STRUCT standard header definition) does not support fractional seconds. The ODBC TIME data type truncates a supplied time value to whole seconds. ADO DotNet and JDBC do not have this restriction.
  - %List data type data in %SelectMode=0 (Logical) does not display the internal storage value, because %List data is encoded using non-printing characters. Instead, Dynamic SQL displays a %List data value as a $LISTBUILD statement, such as the following: `$lb("White","Green")`. See %Print() Method for an example. %List data type data in %SelectMode=1 (ODBC) displays list elements separated by commas; this elements separator is specified as the *CollectionOdbcDelimiter* parameter. %List data type data in %SelectMode=2 (Display) displays list elements separated by `$CHAR(10,13)` (Line Feed, Carriage Return); this elements separator is specified as the *CollectionDisplayDelimiter* parameter.

- For an **INSERT** or **UPDATE** operation, %SelectMode specifies the format for input data that will be converted to Logical storage format. For this data conversion to occur, the SQL code must have been compiled with a select mode of RUNTIME (the default) so that a Display or ODBC %SelectMode is used when the **INSERT** or **UPDATE** is executed. For permitted input values for dates and times, refer to the date and time data types. For further details, refer to the INSERT or UPDATE statement in the *Caché SQL Reference*.

You can specify %SelectMode either as the first parameter of the **%New()** class method, or set it directly, as shown in the following two examples:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New(2)
```

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SelectMode=2
```

The following example returns the current value of %SelectMode:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()
WRITE !,"default select mode=",tStatement.%SelectMode
SET tStatement.%SelectMode=2
WRITE !,"set select mode=",tStatement.%SelectMode
```

You can determine the SelectMode default setting for the current process using the **$SYSTEM.SQL.GetSelectMode()** method. You can change the SelectMode default setting for the current process using the **$SYSTEM.SQL.SetSelectMode(n)** method, when *n* can be 0=Logical, 1=ODBC, or 2=Display. Setting %SelectMode overrides this default for the current object instance; it does not change the SelectMode process default.

For further details on SelectMode options, refer to "Data Display Options" in the "Caché SQL Basics" chapter of this book.

13.3.2 %SchemaPath Property

The *%SchemaPath* property specifies the search path used to supply the schema name for an unqualified table name, view name, or stored procedure name. A schema search path is used for data management operations such as **SELECT**, **CALL**, **INSERT**, and **TRUNCATE TABLE**; it is ignored by data definition operations such as **DROP TABLE**.

The search path is specified as a quoted string containing a schema name or a comma-separated series of schema names. Caché searches the listed schemas in left-to-right order. Caché searches each specified schema until it locates the first matching table, view, or stored procedure name. Because schemas are searched in the specified order, there is no detection of ambiguous table names. Only schema names in the current namespace are searched.

The schema search path can contain both literal schema names and the CURRENT_PATH, CURRENT_SCHEMA, and DEFAULT_SCHEMA keywords.

- CURRENT_PATH specifies the current schema search path, as defined in a prior %SchemaPath property. This is commonly used to add schemas to the beginning or end of an existing schema search path.
- CURRENT_SCHEMA specifies the current schema container class name if the %SQL.Statement call is made from within a class method. If a #sqlcompile path macro directive is defined in a class method, the CURRENT_SCHEMA is the schema mapped to the current class package. Otherwise, CURRENT_SCHEMA is the same as DEFAULT_SCHEMA.
- DEFAULT_SCHEMA specifies the system-wide default schema. This keyword enables you to search the system-wide default schema as a item within the schema search path, before searching other listed schemas. The system-wide default

schema is always searched after searching the schema search path if all the schemas specified in the path have been searched without a match.

The %SchemaPath is the first place Caché searches schemas for a matching table name. If %SchemaPath is not specified, or does not list a schema that contains a matching table name, Caché uses the system-wide default schema.

You can specify a schema search path either by specifying the %SchemaPath property, or by specifying the second parameter of the **%New()** class method, as shown in the following two examples:

ObjectScript

```
SET path="MyTests,Sample,Cinema"
SET tStatement = ##class(%SQL.Statement).%New(,path)
```

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SchemaPath="MyTests,Sample,Cinema"
```

You can set %SchemaPath at any point prior to the **%Prepare()** method which uses it.

The following example returns the current value of %SchemaPath:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()
WRITE !,"default path=",tStatement.%SchemaPath
SET tStatement.%SchemaPath="MyTests,Sample,Cinema"
 WRITE !,"set path=",tStatement.%SchemaPath
```

You can use the **%ClassPath()** method to set %SchemaPath to the search path defined for the specified class name:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%SchemaPath=tStatement.%ClassPath("Sample.Person")
WRITE tStatement.%SchemaPath
```

13.3.3 %Dialect Property

The *%Dialect* property specifies the SQL statement dialect. You can specify either Sybase or MSSQL. This setting causes the SQL statement to be processed using the specified Transact-SQL dialect.

The Sybase and MSSQL dialects support a limited subset of SQL statements in these dialects. They support the **SELECT**, **INSERT**, **UPDATE**, **DELETE**, and **EXECUTE** statements. They support the **CREATE TABLE** statement for permanent tables, but not for temporary tables. **CREATE VIEW** is supported. **CREATE TRIGGER** and **DROP TRIGGER** are supported. However, this implementation does not support transaction rollback should the **CREATE TRIGGER** statement partially succeed but then fail on class compile. **CREATE PROCEDURE** and **CREATE FUNCTION** are supported.

The Sybase and MSSQL dialects support the **IF** flow-of-control statement. This command is not supported in the Caché SQL dialect.

The default is Caché SQL, which can be represented either by a null or “CACHE”.

You can specify %Dialect either as the third parameter of the **%New()** class method, or set it directly as a property, or set it using a method, as shown in the following three examples:

Setting %Dialect in **%New()** class method:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New(,,"Sybase")
WRITE "language mode set to=",tStatement.%Dialect
```

Setting the %Dialect property directly:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()
SET defaultdialect=tStatement.%Dialect
WRITE "default language mode=",defaultdialect,!
SET tStatement.%Dialect="Sybase"
WRITE "language mode set to=",tStatement.%Dialect,!
SET tStatement.%Dialect="Cache"
 WRITE "language mode reset to default=",tStatement.%Dialect,!
```

Setting the %Dialect property using the **%DialectSet()** instance method, which returns an error status:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatus = tStatement.%DialectSet("Sybase")
  IF tStatus'=1 {WRITE "%DialectSet failed:" DO $System.Status.DisplayError(tStatus) QUIT}
WRITE "language mode set to=",tStatement.%Dialect
```

The **%DialectSet()** method returns a %Status value: Success returns a status of 1. Failure returns an object expression that begins with 0, followed by encoded error information. For this reason, you cannot perform a `tStatus=0` test for failure; you can perform a `$$$ISOK(tStatus)=0` macro test for failure.

13.3.4 %ObjectSelectMode Property

The *%ObjectSelectMode* property is a boolean value. If %ObjectSelectMode=0 (the default) all columns in the SELECT list are bound to properties with literal types in the result set. If %ObjectSelectMode=1 then columns in the SELECT list are bound to properties with the type defined in the associated property definition.

%ObjectSelectMode allows you to specify how columns whose type class is a swizzleable class will be defined in the result set class generated from a SELECT statement. If %ObjectSelectMode=0 the property corresponding to the swizzleable column will be defined in result sets as a simple literal type corresponding to the SQL table's ROWID type. If %ObjectSelectMode=1 the property will be defined with the column's declared type. That means that accessing the result set property will trigger swizzling.

%ObjectSelectMode cannot be set as a parameter of **%New()**.

The following example returns the %ObjectSelectMode default value, sets %ObjectSelectMode, then returns the new %ObjectSelectMode value:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery = "SELECT TOP 5 %ID AS MyID,Name,Age FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
WRITE !,"default ObjectSelectMode=",tStatement.%ObjectSelectMode
SET tStatement.%ObjectSelectMode=1
WRITE !,"set ObjectSelectMode=",tStatement.%ObjectSelectMode
```

%ObjectSelectMode=1 is principally used when returning values from a result set using the field name property. This is further described with examples in Fieldname Property in the "Returning Specific Values from the Result Set" section of this chapter.

13.4 Preparing an SQL Statement

Preparing an SQL statement validates the statement, prepares it for subsequent execution, and generates metadata about the SQL statement.

There are three ways to prepare a statement:

- %Prepare(), which prepares an SQL statement (a query, for example) for a subsequent **%Execute()**.
- %PrepareClassQuery(), which prepares a call statement to an existing query. Once prepared, this query can be executed using a subsequent **%Execute()**.
- %ExecDirect(), which both prepares and executes an SQL statement. **%ExecDirect()** is described in "Executing an SQL Statement".

Preparing an SQL statement places the statement in the query cache. This allows the same query to be executed multiple times without the need to re-prepare the SQL statement. A cached query can be executed one or more times by any process; it can be executed with different input parameter values.

Each time you prepare an SQL statement, Caché searches the query cache to determine if the same SQL statement has already been prepared and cached. If not, it places the prepared statement in the query cache. If the prepared statement already exists in the query cache, no new cached query is created. For this reason, it is important not to code a prepare statement within a loop structure.

13.4.1 %Prepare()

You can prepare an SQL statement using the **%Prepare()** instance method of the %SQL.Statement class. The **%Prepare()** method takes, as its first parameter, the SQL statement. This can be specified as a quoted string or a variable that resolves to a quoted string.

For example, in ObjectScript:

ObjectScript

```
SET qStatus = tStatement.%Prepare("SELECT Name,Age FROM Sample.Person")
```

Or in Caché Basic:

Basic example:

```
qStatus = tStatement.%Prepare("SELECT Name,Age FROM Sample.Person")
```

More complex queries can be specified using a subscripted array passed by reference, as shown in the following example:

ObjectScript

```
SET myquery = 3
SET myquery(1) = "SELECT %ID AS id, Name, DOB, Home_State"
SET myquery(2) = "FROM Person WHERE Age > 80"
SET myquery(3) = "ORDER BY 2"
SET qStatus = tStatement.%Prepare(.myquery)
```

A query can contain duplicate field names and field name aliases.

A query supplied to **%Prepare()** can contain input host variables, as shown in the following example:

ObjectScript

```
SET minage = 80
SET myquery = 3
SET myquery(1) = "SELECT %ID AS id, Name, DOB, Home_State"
SET myquery(2) = "FROM Person WHERE Age > :minage"
SET myquery(3) = "ORDER BY 2"
SET qStatus = tStatement.%Prepare(.myquery)
```

Caché substitutes the defined literal value for each input host variable when the SQL statement is executed. Note however, that if this code is called as a method, the *minage* variable must be made Public. By default, methods are ProcedureBlocks; this means that a method (such as **%Prepare()**) cannot see variables defined by its caller. You can either override this default by specifying the class as [Not ProcedureBlock], specifying the method as [ProcedureBlock = 0], or by specifying [PublicList = minage].

Note: It is good program practice to always confirm that an input variable contains an appropriate value before inserting it into SQL code.

You can also supply literal values to a query using ? input parameters. Caché substitutes a literal value for each ? input parameter using the corresponding parameter value you supply to the **%Execute()** method. Following a **%Prepare()**, you can use the %GetImplementationDetails() method to list the input host variables and the ? input parameters in the query.

The **%Prepare()** method returns a %Status value: Success returns a status of 1 (the query string is valid; referenced tables exist in the current namespace). Failure returns an object expression that begins with 0, followed by encoded error information. For this reason, you cannot perform a `status=0` test for failure; you can perform a `$$$ISOK(status)=0` macro test for failure.

The **%Prepare()** method uses the %SchemaPath property defined earlier to resolve unqualified names.

Note: Dynamic SQL performance can be significantly improved by using fully qualified names whenever possible.

You can specify input parameters in the SQL statement by using the "? " character:

ObjectScript

```
SET myquery="SELECT TOP ? Name,Age FROM Sample.Person WHERE Age > ?"
SET qStatus = tStatement.%Prepare(myquery)
```

You specify the value for each ? input parameter in the **%Execute()** instance method when you execute the query. An input parameter must take a literal value or an expression that resolves to a literal value. An input parameter cannot take a field name value or a field name alias. An input parameter must be declared PUBLIC for a **SELECT** statement to reference it directly.

A query can contain field aliases. In this case, the Data property accesses the data using the alias, not the field name.

You are not limited to **SELECT** statements within Dynamic SQL: you can use the **%Prepare()** instance method to prepare other SQL statements, including the **CALL**, **INSERT**, **UPDATE**, and **DELETE** statements.

You can display information about the currently prepared statement using the **%Display()** instance method, as shown in the following example:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET tStatement = ##class(%SQL.Statement).%New(,"Sample")
SET myquery = 3
SET myquery(1) = "SELECT TOP ? Name,DOB,Home_State"
SET myquery(2) = "FROM Person"
SET myquery(3) = "WHERE Age > 60 AND Age < 65"
SET qStatus = tStatement.%Prepare(.myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
DO tStatement.%Display()
WRITE !,"End of %Prepare display"
```

This information consists of the Implementation Class, the Arguments (a comma-separated list of the actual arguments, either literal values or ? input parameters), and the Statement Text.

13.4.2 %PrepareClassQuery()

You can prepare an existing SQL query using the **%PrepareClassQuery()** instance method of the %SQL.Statement class. The **%PrepareClassQuery()** method takes two parameters: the class name of the existing query, and the query name. Both are specified as a quoted string or a variable that resolves to a quoted string.

For example, in ObjectScript:

ObjectScript

```
SET qStatus = tStatement.%PrepareClassQuery("User.queryDocTest","DocTest")
```

Or in Caché Basic:

Basic example:

```
qStatus = tStatement.%PrepareClassQuery("User.queryDocTest","DocTest")
```

The **%PrepareClassQuery()** method returns a %Status value: Success returns a status of 1. Failure returns an object expression that begins with 0, followed by encoded error information. For this reason, you cannot perform a `qStatus=0` test for failure; you can perform a `$$$ISOK(qStatus)=0` macro test for failure.

The **%PrepareClassQuery()** method uses the %SchemaPath property defined earlier to resolve unqualified names.

%PrepareClassQuery() executes using a CALL statement. Because of this, the executed class query must have an SqlProc parameter.

The following example shows **%PrepareClassQuery()** invoking the ByName query defined in the Sample.Person class, passing a string to limit the names returned to those that start with that string value:

ObjectScript

```
SET statemt=##class(%SQL.Statement).%New()
SET cqStatus=statemt.%PrepareClassQuery("Sample.Person","ByName")
  IF cqStatus'=1 {WRITE "%PrepareClassQuery failed:" DO $System.Status.DisplayError(cqStatus) QUIT}
SET rset=statemt.%Execute("L")
DO rset.%Display()
```

The following example shows **%PrepareClassQuery()** invoking an existing query:

ObjectScript

```
 SET tStatement=##class(%SQL.Statement).%New()
 SET cqStatus=tStatement.%PrepareClassQuery("%SYS.GlobalQuery","Size")
   IF cqStatus'=1 {WRITE "%PrepareClassQuery failed:" DO $System.Status.DisplayError(cqStatus) QUIT}

 SET install=$SYSTEM.Util.InstallDirectory()
 SET rset=tStatement.%Execute(install_"mgr\Samples")
 DO rset.%Display()
```

The following example shows **%Prepare()** preparing a CREATE QUERY statement, and then **%PrepareClassQuery()** invoking this class query:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
/* Creating the Query */
SET myquery=4
  SET myquery(1)="CREATE QUERY DocTest() SELECTMODE RUNTIME PROCEDURE "
  SET myquery(2)="BEGIN "
  SET myquery(3)="SELECT TOP 5 Name,Home_State FROM Sample.Person ; "
```

```
  SET myquery(4)="END"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 { WRITE !,"Created a query",! }
ELSEIF rset.%SQLCODE=-361 { WRITE !,"Query exists: ",rset.%Message,! }
ELSE { WRITE !,"CREATE QUERY error: ",rset.%SQLCODE," ",rset.%Message   QUIT}
/* Calling the Query */
WRITE !,"Calling a class query"
SET cqStatus = tStatement.%PrepareClassQuery("User.queryDocTest","DocTest")
 IF cqStatus'=1 {WRITE !,"%PrepareClassQuery failed:" DO $System.Status.DisplayError(cqStatus) QUIT}

SET rset = tStatement.%Execute()
WRITE "Query data",!,!
WHILE rset.%Next() {
   DO rset.%Print() }
WRITE !,"End of data"
/* Deleting the Query */
&sql(DROP QUERY DocTest)
IF SQLCODE=0 { WRITE !,"Deleted the query" }
```

To display a row of data retrieved by a stored query you can use the **%Print()** method, as shown in this example. To display specific column data that was retrieved by a stored query you must use either the **%Get("fieldname")** or the **%GetData(colnum)** method. See "Iterating through a Result Set".

If the query is defined to accept arguments, you can specify input parameters in the SQL statement by using the "? " character. You specify the value for each ? input parameter in the **%Execute()** method when you execute the query. An input parameter must be declared PUBLIC for a **SELECT** statement to reference it directly.

You can display information about the currently prepared query using the **%Display()** method, as shown in the following example:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
/* Creating the Query */
SET myquery=4
  SET myquery(1)="CREATE QUERY DocTest() SELECTMODE RUNTIME PROCEDURE "
  SET myquery(2)="BEGIN "
  SET myquery(3)="SELECT TOP 5 Name,Home_State FROM Sample.Person ; "
  SET myquery(4)="END"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 { WRITE !,"Created a query",! }
ELSEIF rset.%SQLCODE=-361 { WRITE !,"Query exists: ",rset.%Message }
ELSE { WRITE !,"CREATE QUERY error: ",rset.%SQLCODE," ",rset.%Message   QUIT}
/* Preparing and Displying Info about the Query */
WRITE !,"Preparing a class query"
SET cqStatus = tStatement.%PrepareClassQuery("User.queryDocTest","DocTest")
 IF cqStatus'=1 {WRITE !,"%PrepareClassQuery failed:" DO $System.Status.DisplayError(cqStatus) QUIT}

DO tStatement.%Display()
WRITE !,"End of %Prepare display"
/* Deleting the Query */
&sql(DROP QUERY DocTest)
IF SQLCODE=0 { WRITE !,"Deleted the query" }
```

This information consists of the Implementation Class, the Arguments (a comma-separated list of the actual arguments, either literal values or ? input parameters), and the Statement Text.

For further details, refer to "Defining and Using Class Queries" in *Using Caché Objects*.

13.4.3 Results of a Successful Prepare

Following a successful prepare (**%Prepare()**, **%PrepareClassQuery()**, or **%ExecDirect()**) you can invoke the %SQL.Statement **%Display()** instance method or **%GetImplementationDetails()** instance method to return the details of the currently prepared statement. For example:

%Display():

ObjectScript

```
SET myquery = "SELECT TOP 5 Name,Age FROM Sample.Person WHERE Age > 21"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
DO tStatement.%Display()
SET rset = tStatement.%Execute()
```

%GetImplementationDetails():

ObjectScript

```
SET myquery = "SELECT TOP ? Name,Age FROM Sample.Person WHERE Age > 21 AND Name=:fname"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET bool = tStatement.%GetImplementationDetails(.pclassname,.ptext,.pargs)
 IF bool=1 {WRITE "Implementation class= ",pclassname,!
           WRITE "Statement text= ",ptext,!
           WRITE "Arguments= ",$LISTTOSTRING(pargs),!  }  // returns "?,?,c,21,v,fname
 ELSE {WRITE "%GetImplementationDetails() failed",!}
SET rset = tStatement.%Execute()
```

These methods provide the following information:

- **Implementation class**: the class name corresponding to the cached query. For example: `%sqlcq.SAMPLES.cls49`.
- **Arguments**: A list of the query arguments in the order specified. If an argument is enclosed in double parentheses to suppress literal substitution the argument is not included in the argument list.

  %Display() displays a comma-separated list of the query arguments. Each argument can be a literal value, the name of an input host variables (without the colon), or a question mark (?) for an input parameter. If there are no arguments, this item displays `<<none>>`. A predicate that specifies multiple values, such as **IN** or **%INLIST** lists each value as a separate argument.

  %GetImplementationDetails() returns the query arguments as a %List structure. Each argument is represented by a pair of elements, a type and a value: Type c (constant) is followed by a literal value; Type v (variable) is followed by the name of an input host variable (without the colon); Type ? is an input parameter, and is followed by a second question mark. If there are no arguments, the arguments list is an empty string. A predicate that specifies multiple values, such as **IN** or **%INLIST** lists each value as a separate type and value pair.
- **Statement Text**: the query text, exactly as specified. Comments and letter case are preserved, host variables and input parameters are shown as written, the default schema is not shown. For **%Prepare()** for example, `SELECT TOP :n Name FROM Clients`. For **%PrepareClassQuery()** for example, `call Sample.SP_Sample_By_Name(?)`.

For other metadata information generated for a prepared query, refer to SQL Metadata.

13.4.4 The preparse() Method

You can use the **preparse()** method to return a %List structure of the query arguments without having to prepare the SQL query. The query arguments are returned in the same format as **%GetImplementationDetails()**.

The **preparse()** method also returns the query text. However, unlike **%Display()** and **%GetImplementationDetails()** which return the query text exactly as specified, the **preparse()** method replaces each query argument with a ? character, removes comments, and normalizes whitespace. It does not supply a default schema name. The **preparse()** method in the following example returns a parsed version of the query text and a %List structure of the query arguments:

ObjectScript

```
SET myq=2
SET myq(1)="SELECT TOP ? Name /* first name */, Age "
SET myq(2)="FROM Sample.MyTable WHERE Name='Fred' AND Age > :years -- end of query"
DO ##class(%SQL.Statement).preparse(.myq,.stripped,.args)
WRITE "preparsed query text: ",stripped,!
WRITE "arguments list: ",$LISTTOSTRING(args)
```

13.5 Executing an SQL Statement

There are two ways to execute an SQL statement using the %SQL.Statement class:

- **%Execute()**, which executes an SQL statement previous prepared using **%Prepare()** or **%PrepareClassQuery()**.
- **%ExecDirect()**, which both prepares and executes an SQL statement.

You can also execute an SQL statement without creating an object instance by using the **%SYSTEM.SQL.Execute()** method. This method both prepares and executes the SQL statement. It creates a cached query. The **Execute()** method is shown in the following Terminal example:

```
USER>SET topnum=5
USER>SET rset=$SYSTEM.SQL.Execute("SELECT TOP :topnum Name,Age FROM Sample.Person")

USER>DO rset.%Display()
```

13.5.1 %Execute()

After preparing a query, you can execute it by calling the **%Execute()** instance method of the %SQL.Statement class. In the case of a non-**SELECT** statement, **%Execute()** invokes the desired operation (such as performing an **INSERT**). In the case of a **SELECT** query, **%Execute()** generates a result set for subsequent traversal and data retrieval.

For example, in ObjectScript:

ObjectScript

```
SET rset = tStatement.%Execute()
```

Or in Caché Basic:

Basic example:

```
rset = tStatement.%Execute()
```

The **%Execute()** method sets the %SQL.StatementResult class properties *%SQLCODE* and *%Message* for all SQL statements. **%Execute()** sets other %SQL.StatementResult properties as follows:

- **INSERT**, **UPDATE**, **INSERT OR UPDATE**, **DELETE**, and **TRUNCATE TABLE** statements set *%ROWCOUNT* to the number of rows affected by the operation, and *%ROWID* to the Id of the last record inserted, updated, or deleted.
- A **SELECT** statement sets the *%ROWCOUNT* property to 0 when it creates the result set. %ROWCOUNT is incremented when the program iterates through the contents of the result set, for example by using the **%Next()** method. **%Next()** returns 1 to indicate that it is positioned on a row or 0 to indicate that it is positioned after the last row (at the end of the result set). If the cursor is positioned after the last row, the value of %ROWCOUNT indicates the number of rows contained in the result set.

If a **SELECT** query returns only aggregate functions, every **%Next()** sets %ROWCOUNT=1. The first **%Next()** always sets %SQLCODE=0, even when there is no data in the table; any subsequent **%Next()** sets %SQLCODE=100 and sets %ROWCOUNT=1.

A **SELECT** also sets the *%CurrentResult* and the *%ResultColumnCount*. **SELECT** does not set %ROWID.

For further details, refer to the corresponding SQL System Variables in the "Using Embedded SQL" chapter of this manual. If you are executing TSQL code with %Dialect set to Sybase or MSSQL, errors are reported both in the standard protocols for that SQL dialect and in the Caché %SQLCODE and %Message properties.

13.5.1.1 %Execute() with Input Parameters

The **%Execute()** method can take one or more parameters that correspond to the input parameters (indicated by "?") within the prepared SQL statement. The **%Execute()** parameters correspond to the sequence in which the "?" characters appear within the SQL statement: the first parameter is used for the first "?", the second parameter for the second "?", and so on. Multiple **%Execute()** parameters are separated by commas. You can omit a parameter value by specifying the placeholder comma. The number of **%Execute()** parameters must correspond to the "?" input parameters. If there are fewer or more **%Execute()** parameters than corresponding "?" input parameters, execution fails with the %SQLCODE property set to an SQLCODE -400 error.

You can use an input parameter to supply a literal value or an expression to the SELECT list and to the other query clauses, including the TOP clause and the WHERE clause. You cannot use an input parameter to supply a column name or a column name alias to the SELECT list or to the other query clauses.

The maximum number of input parameters when specified as explicit **%Execute()** parameters is 255. The maximum number of input parameters when specified using a variable length array **%Execute(vals...)** is 380.

Following a Prepare, you can use Prepare arguments metadata to return the count and required data types for ? input parameters. You can use the %GetImplementationDetails() method to return a list of ? input parameters in a prepared query and the query text with the ? input parameters shown in context.

The following ObjectScript example executes a query with two input parameters. It specifies the input parameter values (21 and 26) in the **%Execute()** method.

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET tStatement = ##class(%SQL.Statement).%New(1)
SET tStatement.%SchemaPath = "MyTests,Sample,Cinema"
SET myquery=2
SET myquery(1)="SELECT Name,DOB,Age FROM Person"
SET myquery(2)="WHERE Age > ? AND Age < ? ORDER BY Age"
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(21,26)
WRITE !,"Execute OK: SQLCODE=",rset.%SQLCODE,!!
DO rset.%Display()
WRITE !,"End of data: SQLCODE=",rset.%SQLCODE
```

The following ObjectScript example executes the same query. The **%Execute()** method formal parameter list uses a variable length array (dynd...) to specify an indefinite number of input parameter values; in this case, the subscripts of the *dynd* array. The *dynd* variable is set to 2 to indicate two subscript values.

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET tStatement = ##class(%SQL.Statement).%New(1)
SET tStatement.%SchemaPath = "MyTests,Sample,Cinema"
SET myquery=2
SET myquery(1)="SELECT Name,DOB,Age FROM Person"
SET myquery(2)="WHERE Age > ? AND Age < ? ORDER BY Age"
SET dynd=2,dynd(1)=21,dynd(2)=26
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(dynd...)
WRITE !,"Execute OK: SQLCODE=",rset.%SQLCODE,!!
DO rset.%Display()
WRITE !,"End of data: SQLCODE=",rset.%SQLCODE
```

You can issue multiple **%Execute()** operations on a prepared result set. This enables you to run a query multiple times, supplying different input parameter values. It is not necessary to close the result set between **%Execute()** operations, as shown in the following example:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery="SELECT Name,SSN,Age FROM Sample.Person WHERE Name %STARTSWITH ?"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute("A")
DO rset.%Display()
WRITE !,"End of A data",!!
SET rset = tStatement.%Execute("B")
DO rset.%Display()
WRITE !,"End of B data"
```

13.5.1.2 Handling %Execute Errors Using TRY/CATCH

You can execute Dynamic SQL within a TRY block structure, passing runtime errors to the associated CATCH block exception handler. For **%Execute()** errors, you can use the %Exception.SQL class to create an exception instance, which you can then THROW to the **CATCH** exception handler.

The following example creates an SQL exception instance when an **%Execute()** error occurs. In this case, the error is a cardinality mismatch between the number of ? input parameters (1) and the number of **%Execute()** parameters (3). It throws the %SQLCODE and %Message property values (as Code and Data) to the **CATCH** exception handler. The exception handler uses the **%IsA()** instance method to test the exception type, then displays the **%Execute()** error:

ObjectScript

```
TRY {
SET $NAMESPACE="SAMPLES"
SET myquery = "SELECT TOP ? Name,DOB FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(7,9,4)
   IF rset.%SQLCODE=0 { WRITE !,"Executed query",! }
   ELSE { SET badSQL=##class(%Exception.SQL).%New(,rset.%SQLCODE,,rset.%Message)
          THROW badSQL }
DO rset.%Display()
WRITE !,"End of data"
RETURN
}
CATCH exp { WRITE "In the CATCH block",!
            IF 1=exp.%IsA("%Exception.SQL") {
              WRITE "SQLCODE: ",exp.Code,!
              WRITE "Message: ",exp.Data,! }
            ELSE { WRITE "Not an SQL exception",! }
            RETURN
}
```

13.5.2 %ExecDirect()

The %SQL.Statement class provides the **%ExecDirect()** class method, that both prepares and executes a query in a single operation. It can prepare either a specified query (like %Prepare()) or an existing class query (like %PrepareClassQuery()).

%ExecDirect() prepares and executes a specified query:

ObjectScript

```
SET myquery=2
SET myquery(1)="SELECT Name,Age FROM Sample.Person"
SET myquery(2)="WHERE Age > 21 AND Age < 30 ORDER BY Age"
SET rset = ##class(%SQL.Statement).%ExecDirect(,.myquery)
  IF rset.%SQLCODE=0 { WRITE !,"ExecDirect OK",!! }
  ELSE { WRITE !,"ExecDirect SQLCODE=",rset.%SQLCODE,!,rset.%Message  QUIT}
DO rset.%Display()
WRITE !,"End of data: SQLCODE=",rset.%SQLCODE
```

%ExecDirect() prepares and executes an existing class query:

ObjectScript

```
SET mycallq = "?=CALL Sample.PersonSets('A','NH')"
SET rset = ##class(%SQL.Statement).%ExecDirect(,mycallq)
  IF rset.%SQLCODE=0 { WRITE !,"ExecDirect OK",!! }
  ELSE { WRITE !,"ExecDirect SQLCODE=",rset.%SQLCODE,!,rset.%Message  QUIT}
DO rset.%Display()
WRITE !,"End of data: SQLCODE=",rset.%SQLCODE
```

You can specify input parameter values as the third and subsequent parameters of the **%ExecDirect()** class method, as shown in the following example:

ObjectScript

```
SET myquery=2
SET myquery(1)="SELECT Name,Age FROM Sample.Person"
SET myquery(2)="WHERE Age > ? AND Age < ? ORDER BY Age"
SET rset = ##class(%SQL.Statement).%ExecDirect(,.myquery,12,20)
  IF rset.%SQLCODE'=0 {WRITE !,"1st ExecDirect SQLCODE=",rset.%SQLCODE,!,rset.%Message  QUIT}
DO rset.%Display()
WRITE !,"End of teen data",!!
SET rset2 = ##class(%SQL.Statement).%ExecDirect(,.myquery,19,30)
  IF rset2.%SQLCODE'=0 {WRITE !,"2nd ExecDirect SQLCODE=",rset2.%SQLCODE,!,rset2.%Message  QUIT}
DO rset2.%Display()
WRITE !,"End of twenties data"
```

The **%ExecDirect()** input parameters correspond to the sequence in which the "?" characters appear within the SQL statement: the third parameter is used for the first "?", the fourth parameter for the second "?", and so on. You can omit a parameter value by specifying a placeholder comma. If there are fewer **%ExecDirect()** input parameters than corresponding "?" input parameters, the default value (if one exists) is used.

In the following example, the first **%ExecDirect()** specifies all three "?" input parameters, the second **%ExecDirect()** specifies only the second ? input parameter, and omits the first and third. It takes the **Sample.PersonSets()** default ('MA') for the third input parameter:

ObjectScript

```
SET mycall = "?=CALL Sample.PersonSets(?,?)"
SET rset = ##class(%SQL.Statement).%ExecDirect(,mycall,"","A","NH")
  IF rset.%SQLCODE'=0 {WRITE !,"1st ExecDirect SQLCODE=",rset.%SQLCODE,!,rset.%Message  QUIT}
DO rset.%Display()
WRITE !,"End of A people data",!!
SET rset2 = ##class(%SQL.Statement).%ExecDirect(,mycall,,"B")
  IF rset2.%SQLCODE'=0 {WRITE !,"2nd ExecDirect SQLCODE=",rset2.%SQLCODE,!,rset2.%Message  QUIT}
DO rset2.%Display()
WRITE !,"End of B people data"
```

%ExecDirect() can invoke the %SQL.Statement **%Display()** instance method or **%GetImplementationDetails()** instance method to return the details of the currently prepared statement. Because **%ExecDirect()** can prepare and execute either a specified query or an existing class query, you can use the **%GetImplementationDetails()** *pStatementType* parameter to determine which kind of query was prepared:

ObjectScript

```
SET mycall = "?=CALL Sample.PersonSets('A',?)"
SET rset = ##class(%SQL.Statement).%ExecDirect(tStatement,mycall,,"NH")
  IF rset.%SQLCODE'=0 {WRITE !,"ExecDirect SQLCODE=",rset.%SQLCODE,!,rset.%Message  QUIT}
SET bool = tStatement.%GetImplementationDetails(.pclassname,.ptext,.pargs,.pStatementType)
IF bool=1 {IF pStatementType=1 {WRITE "Type= specified query",!}
           ELSEIF pStatementType=45 {WRITE "Type= existing class query",!}
           WRITE "Implementation class= ",pclassname,!
           WRITE "Statement text= ",ptext,!
           WRITE "Arguments= ",$LISTTOSTRING(pargs),!!  }
ELSE {WRITE "%GetImplementationDetails() failed"}
DO rset.%Display()
WRITE !,"End of data"
```

For further details, see Results of a Successful Prepare.

13.6 Returning the Full Result Set

Executing a statement with either **%Execute()** or **%ExecDirect()** returns an object that implements the %SQL.StatementResult interface. This object can be a unitary value, a result set, or a context object that is returned from a **CALL** statement.

13.6.1 %Display() Method

You can display the entire result set (the contents of the result object) by calling the **%Display()** instance method of the %SQL.StatementResult class.

For example, in ObjectScript:

ObjectScript

```
DO rset.%Display()
```

Or in Caché Basic:

Basic example:

```
CALL rset.%Display()
```

Note that the **%Display()** method does not return a %Status value.

When displaying a query result set, **%Display()** concludes by displaying the row count: "5 Rows(s) Affected". (This is the %ROWCOUNT value after **%Display** has iterated through the result set.) Note that **%Display()** does not issue a line return following this row count statement.

13.6.2 %DisplayFormatted() Method

You can reformat and redirect the result set contents to a generated file by calling the **%DisplayFormatted()** instance method of the %SQL.StatementResult class, rather than calling **%Display()**.

You can specify the result set format either by specifying the string option `%DisplayFormatted("HTML")` or the corresponding integer code `%DisplayFormatted(1)`. The following formats are available: XML (integer code 0), HTML (integer code 1), PDF (integer code 2), TXT (integer code 99), or CSV (integer code 100). (Note that CSV format is not

implemented as a true comma-separated value output; instead, it uses tabs to separate the columns.) TXT formatting (integer code 99) concludes with the row count (for example "5 Rows(s) Affected"); the other formats do not include a row count. Caché generates a file of the specified type, appending the appropriate file name extension.

You can specify or omit a result set file name:

- If you specify a destination file (for example, `%DisplayFormatted(99,"myresults")`) a file with that name and the appropriate suffix (file name extension) is generated in the mgr directory in the subdirectory for the current namespace. For example, C:\InterSystems\Cache\mgr\user\myresults.txt. If the specified file with that suffix already exists, Caché overwrites it with new data.
- If you do not specify a destination file (for example, `%DisplayFormatted(99)`) a file with a randomly-generated name and the appropriate suffix (file name extension) is generated in the mgr directory in the Temp subdirectory. For example, C:\InterSystems\Cache\mgr\Temp\w4FR2gM7tX2Fjs.txt. Each time a query is run a new destination file is generated.

These examples show Windows filenames; Caché supports equivalent locations on other operating systems.

If the specified file cannot be opened, this operation times out after 30 seconds with an error message; this commonly occurs when the user does not have WRITE privileges to the specified directory (file folder).

If data cannot be rendered in the specified format, the destination file is created but no result set data is written to it. Instead, an appropriate message is written to the destination file. For example, a stream field OID contains characters that conflict with XML and HTML special formatting characters. This XML and HTML stream field issue can be resolved by using the XMLELEMENT function on stream fields; for example, `SELECT Name,XMLELEMENT("Para",Notes)`.

You can optionally supply the name of a translate table that **%DisplayFormatted()** will use when performing the specified format conversion.

In the case of multiple result sets in a result set sequence, the content of each result set is written to its own file.

The optional third **%DisplayFormatted()** argument specifies that messages are stored in a separate result set. Upon successful completion a message like the following is returned:

```
Message
21 row(s) affected.
```

The following Windows example creates two PDF (integer code 2) result set files in C:\InterSystems\Cache\mgr\samples\. It creates the *mess* result set for messages, then uses **%Display()** to display messages to the Terminal:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery=2
SET myquery(1)="SELECT Name,Age FROM Sample.Person"
SET myquery(2)="WHERE Age > ? AND Age < ? ORDER BY Age"
SET rset = ##class(%SQL.Statement).%ExecDirect(,.myquery,12,20)
  IF rset.%SQLCODE'=0 {WRITE !,"1st ExecDirect SQLCODE=",rset.%SQLCODE,!,rset.%Message  QUIT}
DO rset.%DisplayFormatted(2,"Teenagers",.mess)
DO mess.%Display()
WRITE !,"End of teen data",!!
SET rset2 = ##class(%SQL.Statement).%ExecDirect(,.myquery,19,30)
  IF rset2.%SQLCODE'=0 {WRITE !,"2nd ExecDirect SQLCODE=",rset2.%SQLCODE,!,rset2.%Message  QUIT}
DO rset2.%DisplayFormatted(2,"Twenties",.mess)
DO mess.%Display()
WRITE !,"End of twenties data"
```

13.6.3 Paginating a Result Set

You can use a view ID (%VID) to paginate a result set. The following example returns pages from the result set, each page containing 5 rows:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET q1="SELECT %VID AS RSRow,* FROM "
SET q2="(SELECT Name,Home_State FROM Sample.Person WHERE Home_State %STARTSWITH 'M') "
SET q3="WHERE %VID BETWEEN ? AND ?"
SET myquery = q1_q2_q3
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus=tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
FOR i=1:5:25 {
    WRITE !!,"Next Page",!
    SET rset=tStatement.%Execute(i,i+4)
    DO rset.%Display()
    }
```

13.7 Returning Specific Values from the Result Set

To return specific values from a query result set, you must iterate through the result set one row at a time. To iterate through a result set, use the **%Next()** instance method. (For unitary values, there are no rows in the result object, so **%Next()** returns 0 — and not an error.) You can then either display the results of the whole current row using the **%Print()** method, or retrieve the value of a specified column in the current row.

The **%Next()** method fetches the data for the next row within the query results and places this data in the Data property of the result set object. **%Next()** returns 1 to indicate that it is positioned on a row in the query result. **%Next()** returns 0 to indicate that it is positioned after the last row (at the end of the result set). Each invocation of **%Next()** that returns 1 increments %ROWCOUNT; if the cursor is positioned after the last row (**%Next()** returns 0), the %ROWCOUNT indicates the number of rows in the result set.

If a **SELECT** query returns only aggregate functions, every **%Next()** sets %ROWCOUNT=1. The first **%Next()** returns 1 and sets %SQLCODE=0 and %ROWCOUNT=1, even when there is no data in the table; any subsequent **%Next()** returns 0 and sets %SQLCODE=100 and %ROWCOUNT=1.

After fetching a row from the result set, you can display data from that row using any of the following:

- rset.%Print() to return all of the data values for the current row from a query result set.
- rset.name to return a data value by property name, field name, alias property name, or alias field name from a query result set.
- rset.%Get("fieldname") to return a data value by field name or alias field name from either a query result set or a stored query.
- rset.%GetData(n) to return a data value by column number from either a query result set or a stored query.

13.7.1 %Print() Method

The **%Print()** instance method retrieves the current record from the result set. By default, **%Print()** inserts a blank space delimiter between data field values. **%Print()** does not insert a blank space before the first field value or after the last field value in a record; it issues a line return at the end of the record. If a data field value already contains a blank space, that field value is enclosed in quotation marks to differentiate it from the delimiter. For example, if **%Print()** is returning city names, it would return them as follows: `Chicago "New York" Boston Atlanta "Los Angeles" "Salt Lake City" Washington`. **%Print()** quotes field values that contain the delimiter as part of the data value even when the **%Print()** delimiter is never used; for example if there is only one field in the result set.

You can optionally specify a **%Print()** parameter that provides a different delimiter to be placed between the field values. Specifying a different delimiter overrides the quoting of data strings that contain blank spaces. This **%Print()** delimiter can be one or more characters. It is specified as a quoted string. It is generally preferable that the **%Print()** delimiter be a

character or string not found in the result set data. However, if a field value in the result set contains the **%Print()** delimiter character (or string), that field value is returned enclosed in quotation marks to differentiate it from the delimiter.

If a field value in the result set contains a line feed character, that field value is returned delimited by quotation marks.

The following ObjectScript example iterates through the query result set using **%Print()** to display each result set record, separating values with a "^|^" delimiter. Note how **%Print()** displays data from the FavoriteColors field which is an encoded list of elements:

ObjectScript

```
  SET $NAMESPACE="SAMPLES"
  SET q1="SELECT TOP 5 Name,DOB,Home_State,FavoriteColors "
  SET q2="FROM Sample.Person WHERE FavoriteColors IS NOT NULL"
  SET myquery = q1_q2
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  WHILE rset.%Next() {
     WRITE "Row count ",rset.%ROWCOUNT,!
     DO rset.%Print("^|^")
     }
  WRITE !,"End of data"
  WRITE !,"Total row count=",rset.%ROWCOUNT
```

In Caché Basic:

```
q1 = "SELECT TOP 5 Name,DOB,Home_State,FavoriteColors "
q2 = "FROM Sample.Person WHERE FavoriteColors IS NOT NULL"
myquery = q1 & q2
tStatement = New %SQL.Statement()
qStatus = tStatement.%Prepare(myquery)
If qStatus<>1 Then
   PrintLn "%Prepare failed:"
   PrintLn Piece(qStatus," ",3,10)
Else
 rset = tStatement.%Execute()
 While (rset.%Next())
    PrintLn "Row count ",rset.%ROWCOUNT
    rtn=rset.%Print("^|^")
 Wend
 PrintLn "End of data"
 PrintLn "Total row count=",rset.%ROWCOUNT
End If
```

The following example shows how field values that contain the delimiter are returned enclosed in quotation marks. In this example, the capital letter A is used as the field delimiter; therefore, any field value (name, street address, or state abbreviation) that contains a capital A literal is returned delimited by quotation marks.

ObjectScript

```
  SET $NAMESPACE="SAMPLES"
  SET myquery = "SELECT TOP 25 Name,Home_Street,Home_State,Age FROM Sample.Person"
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare(myquery)
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  WHILE rset.%Next() {
     DO rset.%Print("A")
     }
  WRITE !,"End of data"
  WRITE !,"Total row count=",rset.%ROWCOUNT
```

13.7.2 rset.name Property

When Caché generates a result set, it creates a result set class that contains a unique property corresponding to each field name and field name alias in the result set.

You can use the `rset.name` property to return a data value by property name, field name, property name alias, or field name alias.

- Property Name: If no field alias is defined, specify the field property name as `rset.PropName`. The result set field property name is taken from the corresponding property name in the table definition class.
- Field Name: If no field alias is defined, specify the field name (or the property name) as `rset."fieldname"`. This is the SqlFieldName specified in the table definition. Caché uses this field name to locate the corresponding property name. In many cases, the property name and the field name (SqlFieldName) are identical.
- Alias Property Name: If a field alias is defined, specify the alias property name as `rset.AliasProp`. An alias property name is generated from the column name alias in the **SELECT** statement. You cannot specify a field property name for a field with a defined alias.
- Alias Name: If a field alias is defined, specify this alias name (or the alias property name) as `rset."alias"`. This is the column name alias in the **SELECT** statement. You cannot specify a field name for a field with a defined alias.
- Aggregate, Expression, or Subquery: Caché assigns these *select-items* a field name of Aggregate_n, Expression_n, or Subquery_n (where the integer *n* corresponds to the sequence of the *select-item* list specified in the query). You can retrieve these *select-item* values using the field name (`rset."SubQuery_7"` not case-sensitive), the corresponding property name (`rset.Subquery7` case-sensitive), or by a user-defined field name alias. You can also just specify the *select-item* sequence number using rset.%GetData(n).

When specifying a property name, you must use correct letter case; when specifying a field name, correct letter case is not required.

This invocation of `rset.name` using the property name has the following consequences:

- Letter Case: Property names are case-sensitive. Field names are not case-sensitive. Dynamic SQL can automatically resolve differences in letter case between a specified field or alias name and the corresponding property name. However, letter case resolution takes time. To maximize performance, you should specify the exact letter case of the property name or the alias.
- Non-alphanumeric Characters: A property name can only contain alphanumeric characters (except for an initial % character). If the corresponding SQL field name or field name alias contains non-alphanumeric characters (for example, `Last_Name`) you can do either of the following:
  - Specify the field name delimited with quotation marks. For example, `rset."Last_Name"`). This use of delimiters does not require that delimited identifiers be enabled. Letter case resolution is performed.
  - Specify the corresponding property name, eliminating the non-alphanumeric characters. For example, `rset.LastName` (or `rset."LastName"`). You must specify the correct letter case for the property name.
- % Property Names: Generally, property names beginning with a % character are reserved for system use. If a field property name or alias begins with a % character and that name conflicts with a system-defined property, the system-defined property is returned. For example, for `SELECT Notes AS %Message`, invoking `rset.%Message` will not return the Notes field values; it returns the *%Message* property defined for the statement result class. You can use rset.%Get("%Message") to return the field value.
- Column Alias: If an alias is specified, Dynamic SQL always matches the alias rather than matching the field name or field property name. For example, for `SELECT Name AS Last_Name`, the data can only be retrieved using `rset.LastName` or `rset."Last_Name"`, not by using `rset.Name`.
- Duplicate Names: Names are duplicate if they resolve to the same property name. Duplicate names can be multiple references to the same field in a table, alias references to different fields in a table, or references to fields in different tables. For example `SELECT p.DOB,e.DOB` specifies two duplicate names, even though those names refer to fields in different tables.

If the **SELECT** statement contains multiple instances of the same field name or field name alias, `rset.PropName` or `rset."fieldname"` always return the first one specified in the **SELECT** statement. For example, for `SELECT c.Name,p.Name FROM Sample.Person AS p,Sample.Company AS c` using `rset.Name` retrieves the company name field data; `SELECT c.Name,p.Name AS Name FROM Sample.Person AS p,Sample.Company AS c` using `rset."name"` also retrieves the company name field data. If there are duplicate `Name` fields in the query the last character of the field name (`Name`) is replaced by a character (or characters) to create a unique property name. Thus a duplicate `Name` field name in a query has a corresponding unique property name, beginning with `Nam0` (for the first duplicate) through `Nam9` and continuing with capital letters `NamA` through `NamZ`.

For a user-specified query prepared using **%Prepare()** you can use the property name by itself. For a stored query prepared using **%PrepareClassQuery()**, you must use the **%Get("fieldname")** method.

The following example returns the values of three fields specified by property names: two field values by property name and the third field value by alias property name. In these cases, the specified property name is identical to the field name or field alias:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery = "SELECT TOP 5 Name,DOB AS bdate,FavoriteColors FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New(1)
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
   WRITE "Row count ",rset.%ROWCOUNT,!
   WRITE rset.Name
   WRITE " prefers ",rset.FavoriteColors
   WRITE " birth date ",rset.bdate,!!
   }
WRITE !,"End of data"
WRITE !,"Total row count=",rset.%ROWCOUNT
```

In the above example, one of the fields returned is the FavoriteColors field, which contains %List data. To display this data, the **%New(1)** class method sets the %SelectMode property parameter to 1 (ODBC), causing this program to display %List data as a comma-separated string and the birth date in ODBC format:

The following example returns the Home_State field. Because a property name cannot contain an underscore character, this example specifies the field name (the SqlFieldName) delimited with quotation marks ("Home_State"). You could also specify the corresponding generated property name without quotation marks (HomeState). Note that the delimited field name ("Home_State") is not case-sensitive, but the generated property name (HomeState) is case-sensitive:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery = "SELECT TOP 5 Name,Home_State FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New(2)
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
   WRITE "Row count ",rset.%ROWCOUNT,!
   WRITE rset.Name
   WRITE " lives in ",rset."Home_State",!
     }
WRITE !,"End of data"
WRITE !,"Total row count=",rset.%ROWCOUNT
```

13.7.2.1 Swizzling a Fieldname Property with %ObjectSelectMode=1

The following example is prepared with %ObjectSelectMode=1, which causes fields whose type class is a swizzleable type (a persistent class, a serial class, or a stream class) to automatically swizzle when returning a value using the field name property. The result of swizzling a field value is the corresponding object reference (oref). Caché does not perform this swizzling operation when accessing a field using the **%Get()** or **%GetData()** methods. In this example, `rset.Home` is swizzled, while `rset.%GetData(2)`, which refers to the same field, is not swizzled:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery = "SELECT TOP 5 Name,Home FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New(0)
SET tStatement.%ObjectSelectMode=1
WRITE !,"set ObjectSelectMode=",tStatement.%ObjectSelectMode,!
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
   WRITE "Row count ",rset.%ROWCOUNT,!
   WRITE rset.Name
   WRITE " ",rset.Home,!
   WRITE rset.%GetData(1)
   WRITE " ",$LISTTOSTRING(rset.%GetData(2)),!!
     }
WRITE !,"End of data"
WRITE !,"Total row count=",rset.%ROWCOUNT
```

The following example uses %ObjectSelectMode=1 to derive Home_State values for the selected records from the unique record ID (%ID). Note that the Home_State field is not selected in the original query:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery = "SELECT TOP 5 %ID AS MyID,Name,Age FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET tStatement.%ObjectSelectMode=1
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
   WRITE rset.Name
   WRITE " Home State:",rset.MyID.Home.State,!
   }
WRITE !,"End of data"
WRITE !,"Total row count=",rset.%ROWCOUNT
```

If configured, the system generates a <SWIZZLE FAIL> error if the swizzled property is defined but cannot be referenced. This can occur if the referenced property has been unexpectedly deleted from disk or is locked by another process. To determine the cause of the swizzle failure look in %objlasterror immediately after the <SWIZZLE FAIL> error and decode this %Status value.

By default, <SWIZZLE FAIL> is not configured. You can set this behavior globally by setting `SET ^%SYS("ThrowSwizzleError")=1`, or by using the Caché Management Portal. From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **Objects**. On this screen you can set the <SWIZZLE FAIL> option.

13.7.3 %Get("fieldname") Method

You can use the **%Get("fieldname")** instance method to return a data value by field name or field name alias. Dynamic SQL resolves letter case as needed. If the specified field name or field name alias does not exist, the system generates a <PROPERTY DOES NOT EXIST> error.

The following example returns values for the Home_State field and the Last_Name alias from the query result set.

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery = "SELECT TOP 5 Home_State,Name AS Last_Name FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New(2)
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
  WRITE rset.%Get("Home_State")," : ",rset.%Get("Last_Name"),!
  }
WRITE !,"End of data"
WRITE !,"Total row count=",rset.%ROWCOUNT
```

You must use the **%Get("fieldname")** instance method to retrieve individual data items by field property name from an existing query prepared using **%PrepareClassQuery()**. If the field property name does not exist, the system generates a <PROPERTY DOES NOT EXIST> error.

The following example returns the Nsp (namespace) field values by field property name from a built-in query. Because this query is an existing stored query, this field retrieval requires the use of the **%Get("fieldname")** method. Note that because "Nsp" is a property name, it is case-sensitive:

ObjectScript

```
SET tStatement = ##class(%SQL.Statement).%New(2)
SET qStatus = tStatement.%PrepareClassQuery("%SYS.Namespace","List")
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
   WRITE "Namespace: ",rset.%Get("Nsp"),!
   }
WRITE !,"End of data"
WRITE !,"Total row count=",rset.%ROWCOUNT
```

Duplicate Names: Names are duplicate if they resolve to the same property name. Duplicate names can be multiple references to the same field, references to different fields in a table, or references to fields in different tables. If the **SELECT** statement contains multiple instances of the same field name or field name alias, **%Get("fieldname")** always returns the last instance of a duplicate name as specified in the query. This is the opposite of `rset.PropName`, which returns the first instance of a duplicate name as specified in the query. This is shown in the following example:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery = "SELECT c.Name,p.Name FROM Sample.Person AS p,Sample.Company AS c"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
WRITE "Prop=",rset.Name," Get=",rset.%Get("Name"),! }
WRITE !,rset.%ROWCOUNT," End of data"
```

13.7.4 %GetData(n) Method

The **%GetData(n)** instance method returns data for the current row indexed by the integer count column number of the result set. You can use **%GetData(n)** with either a specified query prepared using **%Prepare()** or a stored query prepared using **%PrepareClassQuery()**.

The integer *n* corresponds to the sequence of the *select-item* list specified in the query. The ID field is not given an integer *n* value, unless explicitly specified in the *select-item* list. If *n* is higher than the number of *select-item*s in the query, or 0, or a negative number, Dynamic SQL returns no value and issues no error.

%GetData(n) is the only way to return a specific duplicate field name or duplicate alias; `rset.Name` returns the first duplicate, `%Get("Name")` returns the last duplicate.

In ObjectScript:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery="SELECT TOP 5 Name,SSN,Age FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
WHILE rset.%Next() {
   WRITE "Years:",rset.%GetData(3)," Name:",rset.%GetData(1),!
   }
 WRITE "End of data"
 WRITE !,"Total row count=",rset.%ROWCOUNT
```

In Caché Basic:

```
myquery = "SELECT TOP 5 Name,SSN,Age FROM Sample.Person"
tStatement = New %SQL.Statement()
qStatus = tStatement.%Prepare(myquery)
If qStatus<>1 Then
   PrintLn "%Prepare failed:"
   PrintLn Piece(qStatus," ",3,10)
Else
 rset = tStatement.%Execute()
 While (rset.%Next())
    Print "Years:",rset.%GetData(3)
    PrintLn " Name:",rset.%GetData(1)
 Wend
 PrintLn "End of data"
 PrintLn "Total row count=",rset.%ROWCOUNT
End If
```

13.8 Returning Multiple Result Sets

A CALL statement can return multiple dynamic result sets as a collection referred to as a *result set sequence* (RSS).

The following example uses the **%NextResult()** method to return multiple result sets separately:

ObjectScript

```
  SET $NAMESPACE="SAMPLES"
  SET mycall = "CALL Sample.CustomSets()"
  SET rset = ##class(%SQL.Statement).%ExecDirect(,mycall)
     IF rset.%SQLCODE'=0 {WRITE !,"ExecDirect SQLCODE=",rset.%SQLCODE,!,rset.%Message  QUIT}
  SET rset1=rset.%NextResult()
  DO rset1.%Display()
  WRITE !,"End of 1st Result Set data",!!
  SET rset2=rset.%NextResult()
  DO rset2.%Display()
  WRITE !,"End of 2nd Result Set data"
```

13.9 SQL Metadata

Dynamic SQL provides the following types of metadata:

- After a Prepare, metadata describing the type of query.
- After a Prepare, metadata describing the select-items in the query (**Columns** and **Extended Column Info**).
- After a Prepare, metadata describing the query arguments: ? parameters, `:var` parameters, and constants. (**Statement Parameters**, **Formal Parameters**, and **Objects**)
- After an Execute, metadata describing the query result set.

%SQL.StatementMetadata property values are available following a Prepare operation (**%Prepare()**, **%PrepareClassQuery()**, or **%ExecDirect()**).

- You can return %SQL.StatementMetadata properties directly for the most recent **%Prepare()**.
- You can return the %SQL.Statement %Metadata property containing the oref for the %SQL.StatementMetadata properties. This enables you to return metadata for multiple Prepare operations.

A **SELECT** or **CALL** statement returns all of this metadata. An **INSERT**, **UPDATE**, or **DELETE** returns Statement Type Metadata and the **Formal Parameters**.

13.9.1 Statement Type Metadata

Following a Prepare using the %SQL.Statement class, you can use the %SQL.StatementMetadata statementType property to determine what type of SQL statement was prepared, as shown in the following example. This example uses the %SQL.Statement %Metadata property to preserve and compare the metadata for two Prepare operations:

ObjectScript

```
 SET tStatement = ##class(%SQL.Statement).%New()
 SET myquery1 = "SELECT TOP ? Name,Age,AVG(Age),CURRENT_DATE FROM Sample.Person"
 SET myquery2 = "CALL Sample.SP_Sample_By_Name(?)"
 SET qStatus = tStatement.%Prepare(myquery1)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET meta1 = tStatement.%Metadata
   SET qStatus = tStatement.%Prepare(myquery2)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET meta2 = tStatement.%Metadata
WRITE "Statement type query 1: ",meta1.statementType,!
WRITE "Statement type query 2: ",meta2.statementType,!
WRITE "End of metadata"
```

The Class Reference entry for the *statementType* property lists the statement type integer codes. The most common codes are 1 (a SELECT query) and 45 (a CALL to a stored query).

You can return the same information using the **%GetImplementationDetails()** instance method, as described in Results of a Successful Prepare.

After executing a query, you can return the statement type name (for example, SELECT) from the result set.

13.9.2 Select-item Metadata

Following a Prepare of a **SELECT** or **CALL** statement using the %SQL.Statement class, you can return metadata about each select-item column specified in the query, either by displaying all of the metadata or by specifying individual metadata items. This column metadata includes ODBC data type information, as well as client type and InterSystems Objects property origins and class type information.

The following example returns the number of columns specified in the most recently prepared query:

ObjectScript

```
SET myquery = "SELECT %ID AS id,Name,DOB,Age,AVG(Age),CURRENT_DATE,Home_State FROM Sample.Person"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
WRITE "Number of columns=",tStatement.%Metadata.columnCount,!
WRITE "End of metadata"
```

The following example returns the column name (or column alias), ODBC data type, maximum data length (precision), and scale for each select-item field:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET myquery=2
SET myquery(1)="SELECT Name AS VendorName,LastPayDate,MinPayment,NetDays,"
SET myquery(2)="AVG(MinPayment),$HOROLOG,%TABLENAME FROM Sample.Vendor"
SET rset = ##class(%SQL.Statement).%New()
SET qStatus = rset.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET x=rset.%Metadata.columns.Count()
SET x=1
WHILE rset.%Metadata.columns.GetAt(x) {
  SET column=rset.%Metadata.columns.GetAt(x)
  WRITE !,x," ",column.colName," is data type ",column.ODBCType
  WRITE " with a size of ",column.precision," and scale = ",column.scale
  SET x=x+1 }
WRITE !,"End of metadata"
```

The following example displays all of the column metadata using the %SQL.StatementMetadata **%Display()** instance method:

ObjectScript

```
  SET tStatement = ##class(%SQL.Statement).%New()
  SET qStatus = tStatement.%Prepare("SELECT %ID AS id,Name,DOB,Age,AVG(Age),CURRENT_DATE,Home_State
FROM Sample.Person")
    IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  DO tStatement.%Metadata.%Display()
  WRITE !,"End of metadata"
```

This returns two table listings of the selected fields. The first columns metadata table lists column definition information:

Display Header	%SQL.StatementColumn Property	Description
Column Name	colName	The SQL name of the column. If the column is given an alias, the column alias, not the field name, is listed here. Names and aliases are truncated to 12 characters. For an expression, aggregate, literal, host variable, or subquery, the assigned “Expression_n”, “Aggregate_n”, “Literal_n”, “HostVar_n”, or “Subquery_n” label is listed (with *n* being the SELECT item sequence number). If you have assigned an alias to an expression, aggregate, literal, host variable, or subquery, the alias is listed here.
Type	ODBCType	The integer code for the ODBC data type. These codes are listed in the Integer Codes for Data Types section of the Data Types reference page in the *InterSystems SQL Reference*. Note that these ODBC data type codes are not the same as the CType data type codes.
Prec	precision	The precision or maximum length, in characters.
Scale	scale	The maximum number of fractional decimal digits. Returns 0 for integer or non-numeric values.
Null	isNullable	An integer value that indicates whether the column is defined as Non-NULL (0), or if NULL is permitted (1). The RowID returns 0. If the SELECT item is an aggregate or subquery that could result in NULL, or if it specifies the NULL literal, this item is set to 1. If the SELECT item is an expression or host variable, this item is set to 2 (cannot be determined).
Label	label	The column name or column alias (same as Column Name).
Table	tableName	The SQL table name. The actual table name is always listed here, even if you have given the table an alias. If the SELECT item is an expression or an aggregate no table name is listed. If the SELECT item is a subquery, the subquery table name is listed.
Schema	schemaName	The table’s schema name. If no schema name was specified, returns the system-wide default schema. If the SELECT item is an expression or an aggregate no schema name is listed. If the SELECT item is a subquery no schema name is listed.
CType	clientType	The integer code for the client data type. See the %SQL.StatementColumn *clientType* property for a list of values.

The second columns metadata table lists extended column information. The Extended Column Info table lists each column with twelve boolean flags (SQLRESULTCOL), specified as Y (Yes) or N (No):

Boolean Flag	%SQL.StatementColumn Property	Description
1: AutoIncrement	isAutoIncrement	The RowID and IDENTITY fields returns Y.
2: CaseSensitive	isCaseSensitive	A string data type field with %EXACT collation returns Y. A property that references a %SerialObject embedded object returns Y.
3: Currency	isCurrency	A field defined with a data type of %Library.Currency, such as the MONEY data type.
4: ReadOnly	isReadOnly	An Expression, Aggregate, Literal, HostVar, or Subquery returns Y. The RowID, IDENTITY, and RowVersion fields returns Y.
5: RowVersion	isRowVersion	The RowVersion field returns Y.
6: Unique	isUnique	A field defined as having a unique value constraint. The RowID and IDENTITY fields returns Y.
7: Aliased	isAliased	The system supplies an alias to a non-field select-item. Therefore, an Expression, Aggregate, Literal, HostVar, or Subquery returns Y, whether or not the user replaced the system alias by specifying a column alias. This flag is not affected by user-specified column aliases.
8: Expression	isExpression	An Expression returns Y.
9: Hidden	isHidden	If the table is defined with %PUBLICROWID or SqlRowIdPrivate=0 (the default), the RowID field returns N. Otherwise, the RowID field returns Y. A property that references a %SerialObject embedded object returns Y.
10: Identity	isIdentity	The RowID field returns Y.
11: KeyColumn	isKeyColumn	A field defined as a primary key field or the target of a foreign key constraint. The RowID field returns Y.
12: RowID	isRowId	The RowID and IDENTITY fields returns Y.

The Extended Column Info metadata table lists the **Column Name** (the SQL name or column alias), the **Linked Prop** (linked persistent class property) and **Type Class** (data type class) for each of the selected fields. Note that the **Linked Prop** lists the persistent class name (not the SQL table name) and the property name (not the column alias).

- For an ordinary table field (`SELECT Name FROM Sample.Person`): **Linked Prop**=Sample.Person.Name, **Type Class**=%Library.String.
- For the table's RowID (`SELECT %ID FROM Sample.Person`): **Linked Prop**= [none], **Type Class**=Sample.Person.
- For an Expression, Aggregate, Literal, HostVar, or Subquery (`SELECT COUNT(Name) FROM Sample.Person`): **Linked Prop**= [none], **Type Class**=%Library.BigInt.
- For a referenced %SerialObject embedded object property (`SELECT Home_State FROM Sample.Person`). **Linked Prop**=Sample.Address.State, **Type Class**=%Library.String.
- For a field referencing a %SerialObject embedded object (`SELECT Home FROM Sample.Person`). **Linked Prop**=Sample.Person.Home, **Type Class**=Sample.Address.

In this example, the Home_State field in Sample.Person references the State property of the %SerialObject class Sample.Address.

The following example returns the metadata for a called stored procedure with one formal parameter, which is also a statement parameter:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET mysql = "CALL Sample.SP_Sample_By_Name(?)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.mysql)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
DO tStatement.%Metadata.%Display()
WRITE !,"End of metadata"
```

It returns not only column (field) information, but also values for Statement Parameters, Formal Parameters, and Objects.

The following example returns the metadata for a with three formal parameters. One of these three parameters is designated with a question mark (?) making it a statement parameter:

ObjectScript

```
SET $NAMESPACE="SAMPLES"
SET mycall = "CALL personsets(?,'MA')"
SET tStatement = ##class(%SQL.Statement).%New(0,"sample")
SET qStatus = tStatement.%Prepare(mycall)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
DO tStatement.%Metadata.%Display()
WRITE !,"End of metadata"
```

Note that this metadata returns no column information, but the Statement Parameters, Formal Parameters lists contain the column names and data types.

13.9.3 Query Arguments Metadata

Following a Prepare using the %SQL.Statement class, you can return metadata about query arguments: input parameters (specified as a question mark (?)), input host variables (specified as `:varname`), and constants (literal values). The following metadata can be returned:

- Count of ? parameters: *parameterCount* property
- ODBC data types of ? parameters: %SQL.StatementMetadata **%Display()** instance method Statement Parameters list.
- List of ?, v (`:var`), and c (constant) parameters: **%GetImplementationDetails()** instance method, as described in Results of a Successful Prepare.

- ODBC data types of ?, v (`:var`), and c (constant) parameters: *formalParameters* property.

  %SQL.StatementMetadata **%Display()** instance method Formal Parameters list.

- Text of query showing these arguments: **%GetImplementationDetails()** instance method, as described in Results of a Successful Prepare.

The statement metadata **%Display()** method lists the Statement Parameters and Formal parameters. For each parameter it lists the sequential parameter number, ODBC data type, precision, scale, whether it is nullable (2 means that a value is always supplied), and its corresponding property name (colName), and column type.

Note that some ODBC data types are returned as negative integers. For a table of ODBC data type integer codes, see the Data Types reference page in the *InterSystems SQL Reference*.

The following example returns the ODBC data types of each of the query arguments (?, `:var`, and constants) in order. Note that the TOP argument is returned as data type 12 (VARCHAR) rather than 4 (INTEGER) because it is possible to specify TOP ALL:

ObjectScript

```
SET myquery = 4
SET myquery(1) = "SELECT TOP ? Name,DOB,Age+10 "
SET myquery(2) = "FROM Sample.Person"
SET myquery(3) = "WHERE %ID BETWEEN :startid :endid AND DOB=?"
SET myquery(4) = "ORDER BY $PIECE(Name,',',?)"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET prepmeta = tStatement.%Metadata
WRITE "Number of ? parameters=",prepmeta.parameterCount,!
SET formalobj = prepmeta.formalParameters
SET i=1
WHILE formalobj.GetAt(i) {
   SET prop=formalobj.GetAt(i)
   WRITE prop.colName," type= ",prop.ODBCType,!
   SET i=i+1 }
WRITE "End of metadata"
```

Following an Execute, arguments metadata is not available from the query result set metadata. In a result set all parameters are resolved. Therefore parameterCount = 0, and formalParameters contains no data.

13.9.4 Query Result Set Metadata

Following an Execute using the %SQL.Statement class, you can return result set metadata by invoking:

- %SQL.StatementResult class properties.
- %SQL.StatementResult **%GetMetadata()** method, accessing %SQL.StatementMetadata class properties.

13.9.4.1 %SQL.StatementResult Properties

Following an Execute query operation, %SQL.StatementResult returns:

- The *%StatementType* property returns an integer code that corresponds to the SQL statement most recently executed. The following is a partial list of these integer codes: 1 = **SELECT**; 2 = **INSERT**; 3 = **UPDATE**; 4 = **DELETE** or **TRUNCATE TABLE**; 9 = **CREATE TABLE**; 15 = **CREATE INDEX**; 45 = **CALL**. For a complete list of these values, refer to %SQL.StatementResult in the *InterSystems Class Reference*.
- The *%StatementTypeName* calculated property returns the command name of the SQL statement most recently executed, based on the %StatementType. This name is returned in uppercase letters. Note that a **TRUNCATE TABLE** operation is returned as DELETE. An **INSERT OR UPDATE** is returned as INSERT, even when it performed an update operation.
- The *%ResultColumnCount* property returns the number of columns in the result set rows.

The following example shows these properties:

ObjectScript

```
SET myquery = "SELECT TOP ? Name,DOB,Age FROM Sample.Person WHERE Age > ?"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute(10,55)
IF rset.%SQLCODE=0 {
WRITE "Statement type=",rset.%StatementType,!
WRITE "Statement name=",rset.%StatementTypeName,!
WRITE "Column count=",rset.%ResultColumnCount,!
WRITE "End of metadata" }
ELSE { WRITE !,"SQLCODE=",rset.%SQLCODE," ",rset.%Message }
```

13.9.4.2 %SQL.StatementResult %GetMetadata()

Following an Execute, you can use the %SQL.StatementResult **%GetMetadata()** method to access the %SQL.StatementMetadata class properties. These are the same properties accessed by the %SQL.Statement %Metadata property following a Prepare.

The following example shows the properties:

ObjectScript

```
SET myquery=2
SET myquery(1)="SELECT Name AS VendorName,LastPayDate,MinPayment,NetDays,"
SET myquery(2)="AVG(MinPayment),$HOROLOG,%TABLENAME FROM Sample.Vendor"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(.myquery)
  IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
IF rset.%SQLCODE=0 {
SET rsmeta=rset.%GetMetadata()
SET x=rsmeta.columns.Count()
SET x=1
WHILE rsmeta.columns.GetAt(x) {
  SET column=rsmeta.columns.GetAt(x)
  WRITE !,x," ",column.colName," is data type ",column.ODBCType
  WRITE " with a size of ",column.precision," and scale = ",column.scale
  SET x=x+1 }
}
ELSE { WRITE !,"SQLCODE=",rset.%SQLCODE," ",rset.%Message }
WRITE !,"End of metadata"
```

Note that the result set metadata does not provide arguments metadata. This is because the Execute operation resolves all parameters. Therefore, in a result set, parameterCount = 0, and formalParameters contains no data.

13.10 Auditing Dynamic SQL

Caché supports optional auditing of Dynamic SQL statements. Dynamic SQL auditing is performed when the %System/%SQL/DynamicStatement system audit event is enabled. By default, this system audit event is not enabled.

If you enable %System/%SQL/DynamicStatement, the system automatically audits every %SQL.Statement dynamic statement that is executed system-wide. Auditing records information in the Audit Database.

To view the Audit Database, go to the Management Portal, **System Administration**, select **Security**, then **Auditing**, then **View Audit Database**. You can set the **Event Name** filter to DynamicStatement to limit the **View Audit Database** to Dynamic SQL statements. The Audit Database lists Time (a local timestamp), User, PID (process ID), and the Description of the event. The Description specifies the type of Dynamic SQL statement. For example, `SQL SELECT Statement (%SQL.Statement)` or `SQL CREATE VIEW Statement (%SQL.Statement)`.

By selecting the **Details** link for an event you can list additional information, including the **Event Data**. The Event Data includes the SQL statement executed and the values of any arguments to the statement. For example:

```
SELECT TOP ? Name , Age FROM Sample . MyTest WHERE Name %STARTSWITH ?
/*#OPTIONS {"DynamicSQLTypeList":",1"} */
Parameter values:
%CallArgs(1)=5
%CallArgs(2)="Fred"
```

The total length of Event Data, which includes the statement and parameters, is 3,632,952 characters. If the statement and parameters are longer than 3632952, the Event Data will be truncated.

Caché also supports auditing of ODBC and JDBC statements (**Event Name**=XDBCStatement), and auditing of Embedded SQL statements (**Event Name**=EmbeddedStatement).

14

Dynamic SQL Using Older Result Set Classes

The %SQL.Statement class is the preferred way to perform Dynamic SQL. Dynamic SQL using this class is described in the previous chapter Using Dynamic SQL.

You can also use the older %ResultSet.SQL class or the %Library.ResultSet class to query the database. In most cases, the %SQL.Statement class is preferable for new Dynamic SQL code. The %ResultSet.SQL and %Library.ResultSet classes are described here for compatibility with existing code.

14.1 Dynamic SQL Using %ResultSet.SQL

The following ObjectScript example prepares and executes a Dynamic SQL query using the %ResultSet.SQL class:

ObjectScript

```
/* %ResultSet.SQL example */
ZNSPACE "SAMPLES"
SET myquery="SELECT TOP 5 Name,SSN FROM Sample.Person ORDER BY Name"
SET rset=##class(%ResultSet.SQL).%Prepare(myquery,.err,"")
  WHILE rset.%Next() {
  WRITE rset.Name,", ",rset.SSN,!
  }
WRITE "End of data"
```

The following %ResultSet.SQL class example uses the **%Print()** instance method to print the current row data for all selected columns in the column order specified in the query:

ObjectScript

```
ZNSPACE "SAMPLES"
SET myquery="SELECT TOP 10 Name,SSN FROM Sample.Person ORDER BY Name"
SET rset=##class(%ResultSet.SQL).%Prepare(myquery,.err,"")
  WHILE rset.%Next() {
  DO rset.%Print("^")
  }
WRITE "End of data"
```

This example uses the ^ character as a delimiter between column values. This use of a specified delimiter character is optional.

14.2 Dynamic SQL Using %Library.ResultSet

The following ObjectScript code prepares and executes a Dynamic SQL query using the %Library.ResultSet class and its **Prepare()** and **Execute()** methods:

ObjectScript

```
/* %Library.ResultSet example */
ZNSPACE "SAMPLES"
SET myquery="SELECT TOP 5 Name,SSN FROM Sample.Person ORDER BY Name"
SET rset=##class(%ResultSet).%New()
SET qStatus=rset.Prepare(myquery)
  IF qStatus'=1 {WRITE "Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET sc=rset.Execute()
   WHILE rset.Next() {
   WRITE rset.Data("Name"),", ",rset.Data("SSN"),!
   }
 WRITE "End of data",!
 WRITE "Row count=",%ROWCOUNT
```

Note: The **%ResultSet.%New()** class method originally required an argument of `"%DynamicQuery:SQL"` to create a new result set. You can now call it either with no argument at all, as in the previous example, or with the `"%DynamicQuery:SQL"` argument, as in the following example.

The following %Library.ResultSet example shows the use of column name aliases. The column name is specified by the SQL query. If you have two columns with the same name, you cannot retrieve them both via the *Data* property. You can provide unique column names by using aliases within your SQL statement:

ObjectScript

```
ZNSPACE "SAMPLES"
SET q1="SELECT TOP 10 P.Name AS pn,E.Name AS en"
SET q2=" FROM Sample.Person AS P,Sample.Employee AS E"
SET myquery=q1_q2
SET rset=##class(%ResultSet).%New("%DynamicQuery:SQL")
SET qStatus=rset.Prepare(myquery)
  IF qStatus'=1 {WRITE "Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET sc=rset.Execute()
   WHILE rset.Next() {
   WRITE rset.Data("pn"),", ",rset.Data("en"),!
   }
 WRITE "End of data",!
 WRITE "Row count=",%ROWCOUNT
```

14.2.1 %Library.ResultSet Supports SQL Result Properties

%Library.ResultSet supports the properties %SQLCODE, %ROWCOUNT, %ROWID and %Message. It sets %SQLCODE and %Message from status values returned by the query. The **Execute()** method populates %ROWID and %ROWCOUNT from the public variables %ROWID and %ROWCOUNT. **Execute()** initializes %ROWCOUNT to zero if the query is not dynamic. The **Next()** method also populates %ROWCOUNT.

14.2.2 %Library.ResultSet Does Not Support CALL

%Library.ResultSet does not support the **CALL** statement for invoking a dynamic query. If the SQL statement is a **CALL**, an "Invalid Statement Type" message is returned by the **Prepare()** method. The %SQL.Statement class supports the **CALL** statement.

If the called routine is a function, %Library.ResultSet can use **SELECT** to invoke it, as shown in the following example:

ObjectScript

```
ZNSPACE "SAMPLES"
SET rs=##class(%ResultSet).%New()
DO $SYSTEM.OBJ.DisplayError(rs.Prepare("SELECT Sample.Stored_Procedure_Test(?,?)"))
WRITE rs.%Execute("Doe,John",""),!
DO rs.%Display()
WRITE !,"End of display"
```

14.3 Input Parameters

Input parameters are specified in a query using a question mark (?). Values are supplied to these input parameters by a method.

- %ResultSet.SQL specifies the input parameter values in the **%Prepare()** method as the 4th and subsequent arguments. There is no limit on the number of input parameters. You can use input parameters to supply values to the TOP clause and the WHERE clause; you cannot use input parameters to supply values to the SELECT list.
- %Library.ResultSet specifies the input parameter values in the **Execute()** method as arguments. There is a limit of 16 input parameters. You can use input parameters to supply values to the TOP clause and the WHERE clause; you cannot use input parameters to supply values to the SELECT list.
- %SQL.Statement specifies the input parameter values in the **%Execute()** method as arguments. There is no limit on the number of input parameters. You can use input parameters to supply values to the TOP clause, the WHERE clause, and to supply expressions to the SELECT list; you cannot use input parameters to supply column names to the SELECT list.

The two following ObjectScript examples both execute the same query with two input parameters. The first uses %ResultSet.SQL and specifies the input parameter values (21 and 26) as the 4th and 5th arguments of the **Prepare()** method. The second uses %Library.ResultSet, and specifies the input parameter values (21 and 26) in the **Execute()** method.

ObjectScript

```
/* %ResultSet.SQL example */
ZNSPACE "SAMPLES"
SET myquery="SELECT Name,Age FROM Sample.Person WHERE Age > ? AND Age < ?"
SET rset=##class(%ResultSet.SQL).%Prepare(myquery,.err,"",21,26)
  WHILE rset.%Next() {
  WRITE rset.Name,", ",rset.Age,!
  }
WRITE "End of data"
```

ObjectScript

```
/* %Library.ResultSet example */
ZNSPACE "SAMPLES"
SET myquery="SELECT Name,Age FROM Sample.Person WHERE Age > ? AND Age < ?"
SET rset=##class(%ResultSet).%New("%DynamicQuery:SQL")
SET qStatus=rset.Prepare(myquery)
  IF qStatus'=1 {WRITE "Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET sc=rset.Execute(21,26)
   WHILE rset.Next() {
   WRITE rset.Data("Name"),", ",rset.Data("Age"),!
   }
 WRITE "End of data",!
 WRITE "Row count=",%ROWCOUNT
```

The following Caché Basic example uses %Library.ResultSet to execute a Dynamic SQL query with two input parameters:

```
myquery="SELECT Name,Age FROM Sample.Person WHERE Age > ? AND Age < ?"
result = New %Library.ResultSet()
' prepare the query
result.Prepare(myquery)
' find everyone with ages within the range specified below
result.Execute(21,26)
While (result.Next())
    PrintLn result.Data("Name") & ", " & result.Data("Age")
Wend
    PrintLn "End of data"
    PrintLn "Row count=",%ROWCOUNT
```

Note that public variables, such as SQLCODE, are not supported by Caché Basic subroutines. All variables in a Caché Basic subroutine are private variables.

14.4 Closing a Query

When you are done with a Dynamic SQL query you can close it (release any resources used by the query) in two different ways:

- By destroying the result set object (such as letting it go out of scope).
- By explicitly calling the **Close()** instance method of the %Library.ResultSet class. Calling the **Close()** method closes the current result set cursor, This allows you to execute and fetch from the same query without having to re-prepare it.

The following %Library.ResultSet example shows how using **Close()** enables you to start a new result set cursor:

ObjectScript

```
ZNSPACE "SAMPLES"
SET myquery="SELECT Name,SSN FROM Sample.Person WHERE Name %STARTSWITH ?"
SET rset=##class(%ResultSet).%New("%DynamicQuery:SQL")
SET qStatus=rset.Prepare(myquery)
  IF qStatus'=1 {WRITE "Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET sc=rset.Execute("A")
   WHILE rset.Next() {
   WRITE rset.Data("Name"),", ",rset.Data("SSN"),!
   }
 WRITE "End of 'A' data",!!
 SET sc=rset.Close()
 SET sc=rset.Execute("B")
   WHILE rset.Next() {
   WRITE rset.Data("Name"),", ",rset.Data("SSN"),!
   }
 WRITE "End of 'B' data"
```

14.5 %Library.ResultSet Metadata

%Library.ResultSet supports static metadata; %SQL.Statement supports dynamic metadata. ZEN Reports programming requires the use of the %Library.ResultSet class, because it requires static metadata.

After preparing a query, you can return metadata about that query. You can either return individual metadata items by using methods of the %Library.ResultSet class, or you can return a table of metadata by using the **%GetMetadata()** method.

To return a table of query metadata values, use **%GetMetadata()** with its **%Display()** method, as shown in the following example:

ObjectScript

```
ZNSPACE "SAMPLES"
SET q1="SELECT Name,SSN AS GovtNum,Age"
SET q2=" FROM Sample.Person WHERE Name %STARTSWITH ?"
SET myquery=q1_q2
SET rset=##class(%ResultSet).%New("%DynamicQuery:SQL")
SET qStatus=rset.Prepare(myquery)
  IF qStatus'=1 {WRITE "Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
WRITE !,rset.%GetMetadata().%Display()
```

The resulting metadata table lists the following items:

Column Name	The name of the column. If the column is given an alias, the alias is listed here. If the SELECT item is an expression or an aggregate, the assigned "Expression_n" or "Aggregate_n" label is listed (with *n* being the SELECT item sequence number). If you have assigned an alias to an expression or aggregate, the alias is listed here.
Type	The integer code for the ODBC data type. These codes are listed in the Integer Codes for Data Types section of the Data Types reference page in the *Caché SQL Reference*. Note that these ODBC data type codes are not the same as the client data type codes returned by %Library.ResultSet class method **GetColumnType(n)**.
Precision	The maximum data size, in characters.
Scale	The maximum number of fractional decimal digits.
Null	A boolean value that indicates whether the column is defined as Non-NULL (0), or if NULL is permitted (1). If the SELECT item is an expression or aggregate that could result in NULL, this item is set to 1. If the SELECT item is an expression with a system-supplied value (such as a system variable or a function that returns the current date, or returns Pi), this item is set to 2.
Label	The column name or alias.
Table	The table name. The actual table name is always listed here, even if you have given the table an alias. If the SELECT item is an expression or an aggregate no table name is listed.
Schema	The table's schema name. If the SELECT item is an expression or an aggregate no schema name is listed.

Each column is then listed with twelve Extended Column Info (SQLRESULTCOL) boolean flags, specified as Y (Yes) or N (No): 1:AutoIncrement, 2:CaseSensitive, 3:Currency, 4:ReadOnly, 5:RowVersion, 6:Unique, 7:Aliased, 8:Expression, 9:Hidden, 10:Identity, 11:KeyColumn, 12:RowId.

You can either return individual metadata items by using methods of the %Library.ResultSet class. These metadata item methods include:

Method	Description
GetColumnCount()	Returns the number of columns selected in the query.
GetColumnName(n)	Returns the name (or name alias) of a column. The *n* integer specifies the desired column by column sequence number in the query.
GetColumnType(n)	Returns an integer code for the client data type of a column specified in the query. The *n* integer specifies the desired column by column sequence number in the query. A table of these client data type integer codes is found in the %Library.ResultSet class documentation. Note that these client data type codes are not the same as the more widely used ODBC data type integer codes (described below). Also note that a column that contains list structured data (such as FavoriteColors in Sample.Person) returns a column data type of 10 (VARCHAR).
GetParamCount()	Returns the number of input parameters (question marks) specified in the query.
GetStatementType()	Returns an integer code for the SQL statement type of the query. For example, a 1=**SELECT**, 2=**INSERT**, etc. A table of these integer codes is found in the %Library.ResultSet class documentation.

The following ObjectScript example shows the use of these query metadata methods:

ObjectScript

```
ZNSPACE "SAMPLES"
SET q1="SELECT Name,SSN AS GovtNum,Age"
SET q2=" FROM Sample.Person WHERE Name %STARTSWITH ?"
SET myquery=q1_q2
SET rset=##class(%ResultSet).%New("%DynamicQuery:SQL")
SET qStatus=rset.Prepare(myquery)
  IF qStatus'=1 {WRITE "Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
WRITE !,rset.GetStatementType() /* returns 1 (SELECT) */
WRITE !,rset.GetColumnCount()  /* returns 3 */
WRITE !,rset.GetColumnName(1)  /* returns Name */
WRITE !,rset.GetColumnName(2)  /* returns GovtNum */
WRITE !,rset.GetColumnType(1)  /* returns 10 (VARCHAR)  */
WRITE !,rset.GetColumnType(3)  /* returns 5 (INTEGER)  */
WRITE !,rset.GetParamCount()  /* returns 1 */
```

14.6 %ResultSet.SQL Metadata

To return a table of query metadata values from %ResultSet.SQL, use the **%PrepareMetaData()** class method, as shown in the following example:

ObjectScript

```
ZNSPACE "SAMPLES"
SET q1="SELECT ID,Name,CURRENT_DATE AS Now,DOB,Age,AVG(Age) AS AvgAge,SSN"
SET q2=" FROM Sample.Person"
SET myquery=q1_q2
SET rset=##class(%ResultSet.SQL).%PrepareMetaData(myquery)
DO rset.%Display()
```

15

Using the SQL Shell Interface

One way to test SQL statements is to execute them from the Caché Terminal using the SQL Shell. This interactive SQL Shell allows you to execute SQL statements dynamically. The SQL Shell uses Dynamic SQL, which means that queries are prepared and executed at runtime. It accesses resources and performs operations within the current namespace.

Unless otherwise indicated, SQL Shell commands and SQL code are not case-sensitive.

The following topics are documented in this chapter:

- Other Ways of Executing SQL from the Terminal prompt, from Management Portal, or from a program.
- Invoking the SQL Shell from the Terminal, inputting, preparing, and executing an SQL statement.
- Using the GO command to re-execute an SQL statement.
- Using input parameters to interactively supply values to an SQL statement at execution time.
- Executing ObjectScript commands from within the SQL Shell.
- Executing an SQL stored procedure using the SQL CALL statement.
- Executing an SQL script file using the Shell's RUN command.
- Storing and recalling SQL statements by number or by assigned name.
- Setting SQL Shell configuration parameters.
- Displaying SQL statement metadata, Show Plan, and Show Statement.
- Timing SQL statement performance.
- Executing Transact-SQL statements (Sybase or MSSQL) from the SQL Shell.

15.1 Other Ways of Executing SQL

You can execute a single line of SQL code from the Terminal command line without invoking the SQL Shell by using the **$SYSTEM.SQL.Execute()** method. The following examples show how this method is used from the Terminal prompt:

```
SAMPLES>SET result=$SYSTEM.SQL.Execute("SELECT TOP 5 name,dob,ssn FROM Sample.Person")
SAMPLES>DO result.%Display()
SAMPLES>SET result=$SYSTEM.SQL.Execute("CALL Sample.PersonSets('M','MA')")
SAMPLES>DO result.%Display()
```

If the SQL statement contains an error, the **Execute()** method completes successfully; the **%Display()** method returns the error information, such as the following:

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

[SQLCODE: <-29>:<Field not found in the applicable tables>]
[%msg: < Field 'GAME' not found in the applicable tables^ SELECT TOP ? game ,>]
0 Rows Affected
SAMPLES>
```

The **Execute()** method also provides optional SelectMode, Dialect, and ObjectSelectMode parameters.

Caché supports numerous other ways to write and execute SQL code, as described in other chapters of this manual. These include:

- Embedded SQL: SQL code embedded within ObjectScript code.
- Dynamic SQL: using %SQL.Statement class methods to execute statements from within ObjectScript or Caché Basic code.
- Management Portal SQL Interface: executing Dynamic SQL from the Caché Management Portal using the **Execute Query** interface.

15.2 Invoking the SQL Shell

You can use the **$SYSTEM.SQL.Shell()** method to invoke the SQL Shell from the Terminal prompt, as follows:

ObjectScript

```
DO $SYSTEM.SQL.Shell()
```

Alternatively, you can invoke the SQL Shell as an instantiated instance using the %SQL.Shell class, as follows:

ObjectScript

```
DO ##class(%SQL.Shell).%Go("Cache")
```

or

ObjectScript

```
SET sqlsh=##class(%SQL.Shell).%New()
DO sqlsh.%Go("Cache")
```

Regardless of how invoked, the SQL Shell returns the SQL Shell prompt (*nsp*>>), where *nsp* is the name of the current namespace. At this prompt you can use either of the following Shell modes:

- Single line mode: at the prompt type a line of SQL code. To end the SQL statement, press **Enter**. By default, this both prepares and executes the SQL code (this is known as Immediate execute mode). For a query, the result set is displayed on the terminal screen. For other SQL statements, the SQLCODE and row count values are displayed on the terminal screen.
- Multiline mode: at the prompt press **Enter**. This puts you in multiline mode. You can type multiple lines of SQL code, each new line prompt indicating the line number. (A blank line does not increment the line number.) To conclude a multiline SQL statement, type GO and press **Enter**. By default, this both prepares and executes the SQL code. For a query, the result set is displayed on the terminal screen. For other SQL statements, the SQLCODE and row count values are displayed on the terminal screen.

Multiline mode provides the following commands, which you type at the multiline prompt and then press **Enter**: `L` or `LIST` to list all SQL code entered thus far. `C` or `CLEAR` to delete all SQL code entered thus far. `C n` or `CLEAR n` (where *n* is a line number integer) to delete a specific line of SQL code. `G` or `GO` to prepare and execute the SQL code and return to single line mode. `Q` or `QUIT` to delete all SQL code entered thus far and return to single line mode. These commands are not case-sensitive. Issuing a command does not increment the line number of the next multiline prompt. Typing `?` at the multiline prompt lists these multiline commands.

To prepare an SQL statement, the SQL Shell first validates the statement, including confirming that the specified tables exist in the current namespace and the specified fields exist in the table. If not, it displays the appropriate SQLCODE.

The SQL Shell performs SQL privilege checking; you must have the appropriate privileges to access or modify a table, field, etc. For further details, refer to the "Users, Roles, and Privileges" chapter of this manual.

If the statement is valid and you have appropriate privileges, the SQL Shell echoes your SQL statement, assigning a sequential number to it. These numbers are assigned sequentially for the duration of the terminal session, regardless of whether you change namespaces and/or exit and re-enter the SQL Shell. These assigned statement numbers permit you to recall prior SQL statements, as described below.

You can also invoke the SQL Shell from the Terminal prompt with `DO Shell^%apiSQL`.

To list all the available SQL Shell commands, enter **?** at the SQL prompt.

To terminate an SQL Shell session and return to the Terminal prompt, enter either the **Q** or **QUIT** command or the **E** command at the SQL prompt. SQL Shell commands are not case-sensitive.

The following is a sample SQL Shell session using the default parameter settings:

```
USER>SET $NAMESPACE="SAMPLES"
SAMPLES>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
SAMPLES>>SELECT TOP 5 Name,Home_State FROM Sample.Person ORDER BY Home_State
1. SELECT TOP 5 Name,Home_State FROM Sample.Person ORDER BY Home_State

Name               Home_State
Djokovic,Josephine W.  AK
Klingman,Aviel P.      AK
Quine, Sam X.          AK
Xiang,Robert C.        AL
Roentgen,Alexandria Q. AR

5 Row(s) Affected
----------------------------------------------
SAMPLES>>SELECT GETDATE()
2. SELECT GETDATE()

Expression_1
2009-09-29 11:41:42

1 Row(s) Affected
----------------------------------------------
SAMPLES>>QUIT

SAMPLES>
```

The following is a multiline SQL Shell session using the default parameter settings:

```
USER>SET $NAMESPACE="SAMPLES"
SAMPLES>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
SAMPLES>>   << entering multiline statement mode >>
        1>>SELECT TOP 5
        2>>Name,Home_State
        3>>FROM Sample.Person
        4>>ORDER BY Home_State
        5>>GO
```

```
1. SELECT TOP 5
   Name,Home_State
   FROM Sample.Person
   ORDER BY Home_State

Name               Home_State
Djokovic,Josephine W.  AK
Klingman,Aviel P.      AK
Quine, Sam X.          AK
Xiang,Robert C.        AL
Roentgen,Alexandria Q. AR

5 Row(s) Affected
---------------------------------------------
SAMPLES>>
```

15.2.1 GO Command

The SQL Shell **GO** command executes the most recent SQL statement. In single line mode, **GO** re-executes the SQL statement most recently executed. When in multiline mode, the **GO** command is used to execute the multiline SQL statement and exit multiline mode. A subsequent **GO** in single line mode re-executes the SQL statement.

15.2.2 Input Parameters

The SQL Shell supports the use of input parameters using the "?" character in the SQL statement. Each time you execute the SQL statement, you are prompted to specify values for these input parameters. You must specified these values in the same sequence that the "?" characters appear in the SQL statement: the first prompt supplies a value to the first "?", the second prompt supplies a value to the second "?", and so on.

There is no limit on the number of input parameters. You can use input parameters to supply values to the TOP clause, the WHERE clause, and to supply expressions to the SELECT list; you cannot use input parameters to supply column names to the SELECT list.

You can specify a host variable as an input parameter value. At the input parameter prompt, specify a value prefaced by a colon (:). This value may be a public variable, an ObjectScript special variable, a numeric literal, or an expression. The SQL Shell then prompts you with "is this a literal (Y/N)?". Specifying N (No) at this prompt (or just pressing Enter) means that the input value is parsed as a host variable. For example, `:myval` would be parsed as the value of the local variable *myval*; `:^myval` would be parsed as the value of the global variable ^*myval*; `:$HOROLOG` would be parsed as the value of the **$HOROLOG** special variable; `:3` would be parsed as the number 3; `:10-3` would be parsed as the number 7. Specifying Y (Yes) at this prompt means that the input value, including the colon preface, is supplied to the input parameter as a literal.

15.2.3 Executing ObjectScript Commands

Within the SQL Shell, you may wish to issue an ObjectScript command. For example, to change the Caché namespace by using the **SET $NAMESPACE** command to the namespace containing the SQL table or stored procedure you wish to reference. You can use the SQL Shell **COS** command to issue an ObjectScript command line, consisting of one or more ObjectScript commands, as shown in the following example:

```
%SYS>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
---------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
[SQL]%SYS>>COS SET oldns=$NAMESPACE SET $NAMESPACE="USER" WRITE "changed the namespace"
changed the namespace
[SQL]USER>>COS SET $NAMESPACE=oldns WRITE "reverted to old namespace"
reverted to old namespace
[SQL]%SYS>>
```

The rest of the command line following the **COS** command is treated as ObjectScript code. You can specify a **COS** command while in SQL Shell single-line mode or in SQL Shell multiline mode. The following example executes a **SELECT** query on a table defined in the USER namespace:

```
%SYS>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
[SQL]%SYS>>  << entering multiline statement mode >>
        1>>COS SET $NAMESPACE="USER"

        1>>SELECT TOP 5 Name,Home_State
        2>>FROM Sample.Person
        3>>GO
   /* SQL query results */
[SQL]USER>>
```

Note that the **COS** statement does not advance the line count.

In SQL Shell multiline mode, a **COS** command is executed upon line return, but an SQL statement is not issued until you specify GO. Thus, the following example is functionally identical to the previous example:

```
%SYS>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
[SQL]%SYS>>  << entering multiline statement mode >>
        1>>SELECT TOP 5 Name,Home_State
        2>>FROM Sample.Person
        3>>COS SET $NAMESPACE="USER" WRITE "changed namespace"
changed namespace
        3>>GO
   /* SQL query results */
[SQL]USER>>
```

The following example uses a **COS** command to define a host variable:

```
USER>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
[SQL]USER>>  << entering multiline statement mode >>
        1>>SELECT TOP :n Name,Home_State
        2>>FROM Sample.Person
        3>>COS SET n=5

        3>>GO
```

15.2.4 CALL Command

You can use the SQL Shell to issue the SQL CALL statement to call an SQL stored procedure, as shown in the following example:

```
SAMPLES>>CALL Sample.PersonSets('G','NY')
```

The SQL Shell issues an SQLCODE -428 error if the specified stored procedure does not exist in the current namespace.

The SQL Shell issues an SQLCODE -370 error if you specify more input parameters than are defined in the stored procedure. You can specify parameter values to the stored procedure using any combination of literals ('string'), host variables (:var), and input parameters (?).

- You can use host variables in a **CALL** statement, as shown in the following example:

  ```
  SAMPLES>>COS SET a="G",b="NY"
  SAMPLES>>CALL Sample.PersonSets(:a,:b)
  ```

- You can use input parameters (“?” characters) in a **CALL** statement, as shown in the following example:

  ```
  SAMPLES>>CALL Sample.PersonSets(?,?)
  ```

  The SQL Shell prompts you for a value for each of these input parameters when the **CALL** statement is executed.

15.2.5 Executing an SQL Script File

The SQL Shell **RUN** command executes an SQL script file. The type of script file is determined by the DIALECT setting. The DIALECT default is CACHE (Caché SQL). For further details, see RUN Command later in this chapter.

15.3 Storing and Recalling SQL Statements

15.3.1 Recall by Number

The SQL Shell automatically stores each successful SQL statement issued during the terminal session in a local cache and assigns it a sequential number. These numbers are used for recalling prior SQL statements during the current Terminal process. SQL Shell only assigns numbers to SQL statements that are successful; if an error occurs during preparation of an SQL statement, no number is assigned. These number assignments are not namespace-specific. The following are the available recall by number commands:

- **#**: You can use **#** to list all of the prior cached SQL statements with their assigned numbers.
- **#n**: You can recall and execute a prior SQL statement by specifying **#n** at the SQL Shell prompt, where *n* is an integer that SQL Shell assigned to that statement.
- **#0**: You can recall and execute the most recently prepared SQL statement by specifying **#0** at the SQL Shell prompt. **#0** recalls the most recently prepared SQL statement, not necessarily the most recently executed SQL statement. Therefore, recalling and executing SQL statements has no effect on which SQL statement is recalled by **#0**.

Recalling an SQL statement by number does not assign a new number to the statement. SQL Shell assigns numbers sequentially for the duration of the Terminal session; exiting and re-entering the SQL Shell or changing namespaces have no effect on number assignment or the validity of prior assigned numbers.

To delete all number assignments, use **#CLEAR** and confirm this action at the displayed prompt. This deletes all prior number assignments and restarts number assignment with 1.

15.3.2 Recall by Name

You can optionally assign a name to an SQL statement, then recall the statement by name. These names are used for recalling prior SQL statements issued from any of the current user's Terminal processes. There are two ways to save and recall an SQL statement by name:

- Save to a global using **SAVEGLOBAL**; recall from a global using **OPEN**.
- Save to a file using **SAVE**; recall from a file using **LOAD**.

15.3.2.1 Saving to a Global

To assign a global name to the most recent SQL statement, use the SQL Shell command **SAVEGLOBAL name**, which can be abbreviated as **SG name**. You can then use the SQL Shell command **OPEN name** to recall the SQL statement from

the global. If EXECUTEMODE is IMMEDIATE, the SQL Shell both recalls and executes the statement. If EXECUTEMODE is DEFERRED, the statement will be prepared but will not be executed until you specify the **GO** command.

Each time you use **OPEN name** to recall an SQL statement by global name, the SQL Shell assigns a new number to the statement. Both the old and new numbers remain valid for recall by number.

A *name* can contain any printable characters except the blank space character. Letters in a *name* are case-sensitive. A *name* can be of any length. A *name* is specific to the current namespace. You can save the same SQL statement multiple times with different names; all of the saved names remain valid. If you attempt to save an SQL statement using a name already assigned, SQL Shell prompts you whether you wish to overwrite the existing name, reassigning it to the new SQL statement.

Global names are assigned for the current namespace. You can list all assigned global names for the current namespace using the SQL Shell **L** (or **LIST**) command. Once assigned, a *name* is available to all of the current user's Terminal processes. An assigned *name* persists after the Terminal process that created it has ended. If there are no *name* assignments, **LIST** returns a "No statements saved" message.

To delete a global *name* assignment, use **CLEAR name**. To delete all global *name* assignments for the current namespace, use **CLEAR** and confirm this action at the displayed prompt.

15.3.2.2 Saving to a File

To assign a file name to the most recent SQL statement, use the SQL Shell command **SAVE name**. You can then use the SQL Shell command **LOAD name** to recall the SQL statement. If EXECUTEMODE is IMMEDIATE, the SQL Shell both recalls and executes the statement. Each time you use **LOAD name** to recall an SQL statement by file name, the SQL Shell assigns a new number to the statement. Both the old and new numbers remain valid for recall by number.

A *name* can contain any printable characters except the blank space character. Letters in a *name* are case-sensitive. A *name* can be of any length. A *name* is specific to the current namespace. You can save the same SQL statement multiple times with different names; all of the saved names remain valid. If you attempt to save an SQL statement using a name already assigned, SQL Shell prompts you whether you wish to overwrite the existing name, reassigning it to the new SQL statement.

Names are assigned for the current namespace. Once assigned, a *name* is available to all of the current user's Terminal processes. An assigned *name* persists after the Terminal process that created it has ended.

15.4 SQL Shell Parameters

The SQL Shell provides the following configurable parameters:

- Commandprefix: (TSQL) specifies a prefix for SQL Shell commands.
- Dialect: (TSQL) specifies the version of SQL to use.
- Displayfile
- Displaymode: specifies the format for query output.
- Displaypath
- Displaytranslate[table]
- Echo: specifies whether result set data should be echoed to the Terminal.
- Executemode: specifies whether or not to defer SQL execution.
- Log: specifies that Shell activity should be logged to a file.
- Messages: specifies whether error messages or query metrics and the cached query name should be displayed to the Terminal.

- Path: sets the schema search path for an unqualified table name.
- Selectmode: specifies whether data should be displayed in Logical, ODBC, or Display mode. Also determines if input data is converted from a display format to logical storage format.

The parameters labelled (TSQL) are principally used for executing Sybase or MSSQL Transact-SQL code from the SQL Shell. They are described in the "Transact-SQL Support" section at the end of this chapter.

15.4.1 Displaying, Setting, and Saving SQL Shell Parameters

SQL Shell configuration parameters are specific to the current SQL Shell invocation on the current Terminal process. Settings apply across namespaces. However, if you exit the SQL Shell, all SQL Shell parameters reset to default values. Caché provides system default values; you can establish different default values using **SET SAVE**, as described below.

The SQL Shell **SET** command (with no arguments) displays the current shell configuration parameters, as shown in the following example. In this example, the **SET** shows the system default values, which are the values established when you invoke the SQL Shell:

```
USER>>SET

commandprefix = ""
dialect = CACHE
displayfile =
displaymode = currentdevice
displaypath =
displaytranslatetable =
echo = on
executemode = immediate
log = off
messages = on
path = SQLUser
selectmode = logical
USER>>
```

To display the current setting for a single configuration parameter, specify **SET** *param*. For example, **SET SELECTMODE** returns the current selectmode setting.

You can use the SQL Shell **SET** command to set a shell configuration parameter. A set value persists for the duration of the SQL Shell invocation; each time you invoke the SQL Shell, the parameters reset to default values. **SET** can use either of the following syntax forms:

```
SET param value
SET param = value
```

Both *param* and *value* are not case-sensitive. Spaces are permitted, but not required, before and after the equal sign.

The SQL Shell **SET SAVE** command saves the current shell configuration parameter settings as the user defaults. These defaults are applied to all subsequent SQL Shell invocations from the current process. They are also applied as SQL shell defaults to any subsequently invoked SQL Shell on any user Terminal process. They remain in effect until specifically reset. Using **SET SAVE** does not affect currently running SQL Shell invocations.

The SQL Shell **SET CLEAR** command clears (resets to system defaults) the current shell configuration parameter settings for the current process. Caché applies this reset to defaults to subsequent SQL Shell invocations by the current process, or any new Terminal process invoked by the current user. **SET CLEAR** does not affect currently running SQL Shell invocations.

15.4.2 Setting DISPLAYMODE and DISPLAYTRANSLATE

You can use **SET DISPLAYMODE** to specify the format used to display query data, as shown in the following example:

```
SAMPLES>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
SAMPLES>>SET DISPLAYMODE XML

displaymode = xml
SAMPLES>>
```

The DISPLAYMODE default is CURRENTDEVICE, which displays the query data on the Terminal in TXT format. You can specify SET DISPLAYMODE = CUR to restore the CURRENTDEVICE default.

The other available options are TXT, HTML, PDF, XML, and CSV. The selection of a format determines the file type. Caché creates a file of this type, writes the query data to the file, and, when possible, launches the appropriate program to display this query data file. For all options except TXT, a second file is created to record result set messages. By default, SQL Shell creates these files in the Caché mgr\Temp\ directory and assigns a randomly generated file name with the appropriate file type suffix. The generated Message file name is the same as the data file name, except for the appended string “Messages”. For the HTML, PDF, and XML options, the Messages file has the same file type suffix as the query data file. For the CSV option, the Messages file has the TXT file type suffix.

The following is an example of the files created when DISPLAYMODE = TXT:

```
C:\InterSystems\Cache\mgr\Temp\sGm7qLdVZn5VbA.txt
C:\InterSystems\Cache\mgr\Temp\sGm7qLdVZn5VbAMessages.txt
```

Each time you run a query, the SQL Shell creates a new pair of files with randomly generated file names.

If DISPLAYMODE is TXT or CSV, you can optionally specify the name of a translate table to apply when performing format conversion. You can specify either SET DISPLAYTRANSLATE or SET DISPLAYTRANSLATETABLE. Translate table name values are case-sensitive.

If DISPLAYMODE is set to a value other than CURRENTDEVICE, any query result set data containing a control character results in a generated Warning message. Generally, control characters only appear in query result set data when it is in Logical mode. For example, data in a List structure contains control characters when displayed in Logical mode. For this reason, it is recommended that when you set DISPLAYMODE to a value other than CURRENTDEVICE that you also set SELECTMODE to either DISPLAY or ODBC.

15.4.2.1 Setting DISPLAYFILE and DISPLAYPATH

If DISPLAYMODE is set to a value other than CURRENTDEVICE, you can specify the target file location using the DISPLAYFILE and DISPLAYPATH parameters:

- DISPLAYFILE: set this parameter to a simple file name with no suffix; for example, SET DISPLAYFILE = myfile. You can also set this parameter to a partially-qualified path, which Caché appends to the DISPLAYPATH value or the default directory, creating subdirectories as needed; for example, SET DISPLAYFILE = mydir\myfile. If DISPLAYPATH is set, the system creates a file with this file name in the specified directory; if DISPLAYPATH is not set, the system creates a file with this file name in the Caché mgr\Temp\ directory.
- DISPLAYPATH: set this parameter to an existing fully-qualified directory path structure ending in a slash (“/”) or backslash (“\”), depending on operating system platform. If DISPLAYFILE is set, the system creates a file with the DISPLAYFILE name in this directory; if DISPLAYFILE is not set, the system creates a file with a randomly-generated name in this directory. If the DISPLAYPATH directory does not exist, Caché ignores DISPLAYPATH and DISPLAYFILE settings and instead uses the default directory and default randomly-generated file name.

When necessary, the system automatically adds a slash (or backslash) to the end of your DISPLAYPATH value and/or removes a slash (or backslash) from the beginning of your DISPLAYFILE value to create a valid fully-qualified directory path.

The following example sets DISPLAYMODE, DISPLAYFILE, and DISPLAYPATH:

```
SAMPLES>>SET DISPLAYMODE XML

displaymode = xml
SAMPLES>>SET DISPLAYFILE = myfile

displayfile = myfile
SAMPLES>>SET DISPLAYPATH = C:\temp\mydir\

displaypath = C:\temp\mydir\
SAMPLES>>
```

When you execute a query the SQL Shell will generate the following files. The first contains the query data. The second contains any messages resulting from the query execution:

```
C:\temp\mydir\myfile.xml
C:\temp\mydir\myfileMessages.xml
```

If you specify neither DISPLAYFILE or DISPLAYPATH, the system creates files in the Mgr\Temp\ directory for your Caché installation (for example, C:\InterSystems\Cache\Mgr\Temp\) with a randomly generated file name.

If DISPLAYMODE is not set to CURRENTDEVICE, each time you run a query with DISPLAYFILE set, any existing data in the named file and the corresponding Messages file is replaced by the new query data. Each time you run a query with DISPLAYFILE not set, the SQL Shell creates a new file with a randomly generated file name and a new corresponding Messages file.

If DISPLAYMODE is set to CURRENTDEVICE, the DISPLAYFILE and DISPLAYPATH parameters have no effect.

15.4.3 Setting EXECUTEMODE

The SQL Shell supports immediate and deferred SQL statement execution. Immediate execution prepares and executes the specified SQL statement when you press Enter. Deferred execution prepares the statement when you press Enter, but does not execute it until you specify **GO** at the SQL prompt.

The available options are `SET EXECUTEMODE IMMEDIATE` (the default), `SET EXECUTEMODE DEFERRED`, and `SET EXECUTEMODE` to display the current mode setting. The following example sets the execute mode:

```
SAMPLES>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
SAMPLES>>SET EXECUTEMODE DEFERRED

Executemode = deferred
SAMPLES>>
```

Deferred execution allows you to prepare multiple SQL queries, then recall them by name or number for execution. To execute a prepared SQL statement, recall the desired statement (from the appropriate namespace) then specify **GO**.

The following example shows the preparation of three queries in Deferred mode. The first two are saved and assigned a recall name; the third is not assigned a name, but can be recalled by number:

```
SAMPLES>>SELECT TOP 5 Name,Home_State FROM Sample.Person
1. SELECT TOP 5 Name,Home_State FROM Sample.Person
SAMPLES>>SAVE 5sample
Query saved as: 5sample
SAMPLES>>SELECT TOP 5 Name,Home_State FROM Sample.Person ORDER BY Home_State
2. SELECT TOP 5 Name,Home_State FROM Sample.Person ORDER BY Home_State
SAMPLES>>SAVE 5ordered
Query saved as: 5ordered
SAMPLES>>SELECT Name,Home_State FROM Sample.Person ORDER BY Home_State
3. SELECT Name,Home_State FROM Sample.Person ORDER BY Home_State
SAMPLES>>
```

The following example shows the deferred mode execution of two of the queries defined in the previous example. Note that this example recalls one query by name (upon recall the SQL Shell gives it a new number), and one query by number:

```
SAMPLES>>OPEN 5ordered
SELECT TOP 5 Name,Home_State FROM Sample.Person ORDER BY Home_State
4. SELECT TOP 5 Name,Home_State FROM Sample.Person ORDER BY Home_State
---------------------------------------------
SAMPLES>>GO

Name               Home_State
Djokovic,Josephine W.  AK
Klingman,Aviel P.      AK
Quine, Sam X.          AK
Xiang,Robert C.        AL
Roentgen,Alexandria Q. AR

5 Row(s) Affected
---------------------------------------------
SAMPLES>>#3
SELECT Name,Home_State FROM Sample.Person ORDER BY Home_State
3. SELECT Name,Home_State FROM Sample.Person ORDER BY Home_State
---------------------------------------------
SAMPLES>>GO
.
.
.
```

15.4.4 Setting ECHO

You can use **SET ECHO** to specify whether to echo the query results to the SQL Shell. If you specify SET ECHO=OFF, the query is prepared, a cached query is defined, and the query is executed. No query results are displayed to the Terminal. This is shown in the following example:

```
[SQL]USER>>set echo=off

echo = off
[SQL]USER>>SELECT Name,Age FROM Sample.MyTest
4.      SELECT Name,Age FROM Sample.MyTest


statement prepare time(s)/globals/lines/disk: 0.0002s/5/155/0ms
          execute time(s)/globals/lines/disk: 0.0001s/0/105/0ms
                            cached query class: %sqlcq.USER.cls3
---------------------------------------------------------------------------
[SQL]USER>>
```

If you specify SET ECHO=ON (the default) the query results are displayed to the Terminal. This is shown in the following example:

```
[SQL]USER>>set echo=on

echo = on
[SQL]USER>>SELECT Name,Age FROM Sample.MyTest
5.      SELECT Name,Age FROM Sample.MyTest

Name    Age
Fred Flintstone 41
Wilma Flintstone        38
Barney Rubble   40
Betty Rubble    42

4 Rows(s) Affected
statement prepare time(s)/globals/lines/disk: 0.0002s/5/155/0ms
          execute time(s)/globals/lines/disk: 0.0002s/5/719/0ms
                        cached query class: %sqlcq.USER.cls3
---------------------------------------------------------------------------
[SQL]USER>>
```

SET ECHO is only meaningful if DISPLAYMODE=CURRENTDEVICE (the default).

SET ECHO and **SET MESSAGES** specify what is displayed on the Terminal; they do not affect the prepare or execution of the query. If both SET MESSAGES=OFF and SET ECHO=OFF, the query is prepared, a cached query is created, and query execution creates a query result set, but nothing is returned to the Terminal.

15.4.5 Setting MESSAGES

You can use **SET MESSAGES** to specify whether to display the query error message (if unsuccessful), or query execution information (if successful):

- If query execution is unsuccessful: If you specify SET MESSAGES=OFF, nothing is displayed to the Terminal. If you specify SET MESSAGES=ON (the default) the query error message is displayed, such as the following: `ERROR #5540: SQLCODE: -30 Message: Table 'SAMPLE.NOTABLE' not found`.
- If query execution is successful: If you specify SET MESSAGES=OFF, only the query results and the line `n Rows(s) Affected` are displayed to the Terminal. If you specify SET MESSAGES=ON (the default) the query results and the line `n Rows(s) Affected` are followed by the statement prepare metrics, the statement execution metrics, and the name of the generated cached query.

  Prepare and Execute metrics are measured in elapsed time (in fractional seconds), total number of global references, total number of lines executed, and disk read latency (in milliseconds).

The information displayed when SET MESSAGES=ON is not changed by setting DISPLAYMODE. Some DISPLAYMODE options create both a query result set file and a messages file. This messages file contains result set messages, not the query prepare and execute messages displayed to the Terminal when SET MESSAGES=ON.

SET MESSAGES and **SET ECHO** specify what is displayed on the Terminal; they do not affect the prepare or execution of the query. If both SET MESSAGES=OFF and SET ECHO=OFF, a successful query is prepared, a cached query is created, and query execution creates a query result set, but nothing is returned to the Terminal.

15.4.6 Setting LOG

You can use **SET LOG** to specify whether to log SQL Shell activity to a file. The available options are:

- **SET LOG OFF**: The default. Caché does not log activity for the current SQL Shell.
- **SET LOG ON**: Caché logs SQL Shell activity to the default log file.
- **SET LOG** *pathname*: Caché logs SQL Shell activity to the file specified by *pathname*.

SET LOG ON creates a log file in Cache\mgr\namespace, where *namespace* is the name of the current namespace for the process. This default log file is named xsqlnnnn.log, where *nnnn* is the process ID (pid) number for the current process.

By default, a log file is specific to the current process and the current namespace. To log SQL Shell activity from multiple processes and/or from multiple namespaces in the same log, specify SET LOG *pathname* for each process and/or namespace using the same *pathname*.

A log file can be suspended and resumed. Once a log file has been created, SET LOG OFF suspends writing to that log file. SET LOG ON resumes writing to the default log file. `Log restarted: date time` is written to the log file when logging resumes. SET LOG ON always activates the default log file. Thus, if you suspend writing to a specified *pathname* log file, you must specify SET LOG *pathname* when resuming.

Activating a log file creates a copy of SQL Shell activity displayed on the terminal; it does not redirect SQL Shell terminal output. The SQL Shell log records SQL errors for failed SQL execution and the SQL code and resulting row count for successful SQL execution. The SQL Shell log does not record result set data.

If a log is already active, specifying SET LOG ON has no effect. If a log is already active, specifying SET LOG *pathname* suspends the current log and activates the log specified by *pathname*.

15.4.7 Setting PATH

You can use **SET PATH schema** to set the schema search path, which SQL uses to supply the correct schema name for an unqualified table name. *schema* can be a single schema name, or a comma-separated list of schema names, as shown in the following example:

```
[SQL]USER>>SET PATH cinema,sample,user
```

SET PATH with no argument deletes the current schema search path, reverting to the system-wide default schema name.

If **SET PATH schema** is not specified, or the table is not found in the specified schemas, SQL Shell uses the system-wide default schema name. For further details on schema search paths, refer to the #sqlcompile path macro in the "ObjectScript Macros and the Macro Preprocessor" chapter of *Using Caché ObjectScript*.

15.4.8 Setting SELECTMODE

You can use **SET SELECTMODE** to specify the mode used to display query data.

```
SAMPLES>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
SAMPLES>>SET SELECTMODE DISPLAY

selectmode = display
SAMPLES>>
```

The available options are DISPLAY, LOGICAL, and ODBC. LOGICAL is the default. To determine the current mode, specify **SET SELECTMODE** without a value:

```
SAMPLES>>SET SELECTMODE

selectmode = logical
SAMPLES>>
```

%List data is encoded using non-printing characters. Therefore, when selectmode=logical, SQL Shell displays a %List data value as a $LISTBUILD statement, such as the following: `$lb("White","Green")`. Time data type data supports fractional seconds. Therefore, when selectmode=odbc, SQL Shell displays fractional seconds, which does not correspond to the ODBC standard. The actual ODBC TIME data type truncates fractional seconds.

For further details on SelectMode options, refer to "Data Display Options" in the "Caché SQL Basics" chapter of this book.

You can also use **SET SELECTMODE** to specify whether input data will be converted from display format to logical storage format. For this data conversion to occur, the SQL code must have been compiled with a select mode of RUNTIME. At execution time, **SET SELECTMODE** must be set to LOGICAL (the default). For further details, refer to the INSERT or UPDATE statement in the *Caché SQL Reference*.

15.5 SQL Metadata and Performance Metrics

15.5.1 Displaying Metadata, Show Plan, and Show Statement

The SQL Shell supports the following additional commands:

- **M** or **METADATA** to display metadata information about the current query. For details, refer to Select-item Metadata in the "Using Dynamic SQL" chapter.
- **SHOW PLAN**, **SHOW PL** (or simply **SHOW**) to display show plan information about the current query. The show plan can be used for debugging and optimizing the performance of a query. It specifies how the query executes, including the use of indexes and a cost value for the query. A show plan can be returned for the following statements: SELECT, DECLARE, non-cursor UPDATE or DELETE, and INSERT...SELECT.
- **SHOW STATEMENT** or **SHOW ST** to display the prepared SQL statement. This information consists of the Implementation Class, the Arguments (a comma-separated list of the actual arguments, such as the TOP clause and WHERE clause argument values), and the Statement Text.

For further details on Caché SQL Shell commands, enter **?** at the SQL prompt, or refer to %SYSTEM.SQL.Shell() in the *InterSystems Class Reference*.

For further details on interpreting a query plan, see "Interpreting an SQL Query Plan" in the *Caché SQL Optimization Guide*.

15.5.2 SQL Shell Performance

Following the successful execution of an SQL statement, the SQL Shell displays four statement prepare values (**times(s)/globals/lines/disk**) and four statement execute values (**times(s)/globals/lines/disk**):

- The statement **prepare time** is the time it took to prepare the dynamic statement. This includes the time it took to generate and compile the statement. It includes the time it took to find the statement in the statement cache. Thus, if a statement is executed, then recalled by number or recalled by name, the prepare time on the recalled statement is near zero. If a statement is prepared and executed, then re-executed by issuing the GO command, the prepare time on the re-execution is zero.
- The elapsed **execute time** is the elapsed time from the call to **%Execute()** until the return from **%Display()**. It does not include wait time for input parameter values.

The statement **globals** is the count of global references, **lines** is the count of lines of code executed, and **disk** is the disk latency time in milliseconds. The SQL Shell keeps separate counts for the Prepare operation and the Execute operation.

These performance values are only displayed when DISPLAYMODE is set to currentdevice, and MESSAGES is set to ON. These are the SQL Shell default settings.

15.6 Transact-SQL Support

By default, the SQL Shell executes Caché SQL code. However, the SQL Shell can be used to execute Sybase or MSSQL code.

15.6.1 Setting DIALECT

By default, the SQL Shell parses code as Caché SQL. You can use SET DIALECT to configure the SQL Shell to execute Sybase or MSSQL code. To change the current dialect, SET DIALECT to Sybase, MSSQL, or Cache. The default is Dialect=Cache.

he following is an example of the executing a MSSQL program from the SQL Shell:

```
SAMPLES>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------
The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
SAMPLES>>SET DIALECT MSSQL

dialect = MSSQL
SAMPLES>>SELECT TOP 5 name + '-' + ssn FROM Sample.Person
1.      SELECT TOP 5 name + '-' + ssn FROM Sample.Person

Expression_1
Zweifelhofer,Maria H.-559-20-7648
Vonnegut,Bill A.-552-41-2071
Clinton,Terry E.-757-30-8013
Bachman,Peter U.-775-59-3756
Avery,Emily N.-833-18-9563

5 Rows(s) Affected
statement prepare time: 0.2894s, elapsed execute time: 0.0467s.
---------------------------------------------------------------------------
SAMPLES>>
```

The Sybase and MSSQL dialects support a limited subset of SQL statements in these dialects. They support the **SELECT**, **INSERT**, **UPDATE**, and **DELETE** statements. They support the **CREATE TABLE** statement for permanent tables, but not for temporary tables. **CREATE VIEW** is supported. **CREATE TRIGGER** and **DROP TRIGGER** are supported. However, this implementation does not support transaction rollback should the **CREATE TRIGGER** statement partially succeed but then fail on class compile. **CREATE PROCEDURE** and **CREATE FUNCTION** are supported.

15.6.2 Setting COMMANDPREFIX

You can use **SET COMMANDPREFIX** to specify a prefix (usually a single character) that must be appended to subsequent SQL Shell commands. This prefix is not used on SQL statements issued from the SQL Shell prompt. The purpose of this prefix is to prevent ambiguity between SQL Shell commands and SQL code statements. For example, SET is an SQL Shell command; SET is also an SQL code statement in Sybase and MSSQL.

By default, there is no command prefix. To establish a command prefix, SET COMMANDPREFIX=*prefix*, with *prefix* specified without quotation marks. To revert to having no command prefix, SET COMMANDPREFIX="". The following example shows the command prefix / (the slash character) being set, used, and reverted:

```
SAMPLES>DO $SYSTEM.SQL.Shell()
SQL Command Line Shell
----------------------------------------------------

The command prefix is currently set to: <<nothing>>.
Enter q to quit, ? for help.
SAMPLES>>SET COMMANDPREFIX=/

commandprefix = /
SAMPLES>>/SET LOG=ON
```

```
log = xsql4148.log
SAMPLES>>  << entering multiline statement mode >>
        1>>SELECT TOP 3 Name,Age
        2>>FROM Sample.Person
        3>>/GO
9.      SELECT TOP 3 Name,Age
        FROM Sample.Person

Name    Age
Frith,Jose M.   13
Finn,William D. 15
Ximines,Uma Y.  44

3 Rows(s) Affected
statement prepare time: 0.0010s, elapsed execute time: 0.0014s.
---------------------------------------------------------------------------
SAMPLES>>/SET COMMANDPREFIX

commandprefix = /
SAMPLES>>/SET COMMANDPREFIX=""

commandprefix = ""
SAMPLES>>SET COMMANDPREFIX

commandprefix =
SAMPLES>>
```

When a command prefix is set, the command prefix is required for all SQL Shell commands, except ?, #, and GO; these three SQL Shell commands can be issued with or without the command prefix.

The SQL Shell displays the current command prefix as part of the SQL Shell initialization, when you issue a SET or a SET COMMANDPREFIX command, and at the end of the ? commands option display.

15.6.3 RUN Command

The SQL Shell RUN command executes an SQL script file. You must SET DIALECT before issuing a RUN command to specify either Caché SQL, Sybase TSQL, or Microsoft SQL (MSSQL); the default dialect is Caché SQL. You can either invoke `RUN scriptname` or just invoke `RUN` and be prompted for the script file name.

RUN loads the script file, then prepares and executes each statement contained in the file. Statements in the script file must be delimited, usually either with a GO line, or with a semicolon (;). The RUN command prompts you to specify the delimiter.

The SQL script file results are displayed on the current device and, optionally, in a log file. Optionally, a file containing statements that failed to prepare can be produced.

The RUN command returns prompts to specify these options, as shown in the following example:

```
USER>>SET DIALECT=Sybase

dialect = Sybase
USER>>RUN

Enter the name of the SQL script file to run: SybaseTest

Enter the file name that will contain a log of statements, results and errors (.log): SyTest.log
     SyTest.log

Many script files contain statements not supported by Cache' SQL.
Would you like to log the statements not supported to a file so they
can be dealt with manually, if applicable?   Y=> y
Enter the file name in which to record non-supported statements (_Unsupported.log): SyTest_Unsupported.log


Please enter the end-of-statement delimiter (Default is 'GO'):  GO=>

Pause how many seconds after error?   5 => 3

Sybase Conversion Utility (v3)
Reading source from file:
Statements, results and messages will be logged to: SyTest.log
.
.
.
```

15.6.4 TSQL Examples

The following SQL Shell example creates a Sybase procedure AvgAge. It executes this procedure using the Sybase EXEC command. It then changes the dialect to Caché and executes the same procedure using the Caché SQL CALL command.

```
SAMPLES>>SET DIALECT Sybase

dialect = Sybase
SAMPLES>>  << entering multiline statement mode >>
        1>>CREATE PROCEDURE AvgAge
        2>>AS SELECT AVG(Age) FROM Sample.Person
        3>>GO
12.     CREATE PROCEDURE AvgAge
        AS SELECT AVG(Age) FROM Sample.Person


statement prepare time: 0.1114s, elapsed execute time: 0.4364s.
---------------------------------------------------------------------------
SAMPLES>>EXEC AvgAge
13.     EXEC AvgAge



Dumping result #1
Aggregate_1
44.35

1 Rows(s) Affected
statement prepare time: 0.0956s, elapsed execute time: 1.1761s.
---------------------------------------------------------------------------

SAMPLES>>SET DIALECT=Cache

dialect = CACHE
SAMPLES>>CALL AvgAge()
14.     CALL AvgAge()



Dumping result #1
Aggregate_1
44.35

1 Rows(s) Affected
statement prepare time: 0.0418s, elapsed execute time: 0.0040s.
---------------------------------------------------------------------------
SAMPLES>>
```

16

Using the Management Portal SQL Interface

This chapter describes how to perform SQL operations from the Caché Management Portal. The Management Portal interface uses Dynamic SQL, which means that queries are prepared and executed at runtime. The Management Portal interface is intended as an aid for developing and testing SQL code against small data sets. It is not intended to be used as an interface for SQL execution in a production environment.

The Management Portal also provides various options to configure SQL. For further details, refer to SQL configuration settings described in *Caché Advanced Configuration Settings Reference*.

16.1 Management Portal SQL Facilities

Caché allows you to examine and manipulate data using SQL tools from the Caché Management Portal. The starting point for this is the Management Portal **System Explorer** option. From there you select the **SQL** option (**[System] > [SQL]**). This displays the SQL interface, which allow you to:

- Execute SQL Statements — write and run SQL statements against existing table definitions and data. You can either write the SQL code directly into a text box (including SELECT, INSERT, UPDATE, DELETE, CREATE TABLE and other SQL statements), retrieve a statement from the SQL history into the text box, drag and drop a table into the text box to generate a query (SELECT statement), or compose a query (SELECT statement) using the Query Builder interface.
- Filtering Schema Contents — on the left side of the screen display the SQL schemas for the current namespace or a filtered subset of these schemas, with each schema's tables, views, procedures, and cached queries. You can select an individual table, view, procedure, or cached query to display its Catalog Details.
- Wizards — execute a wizard to perform data import, data export, or data migration. Execute a wizard to link to tables or views or to link to stored procedures. Execute a wizard to map FileMan files to InterSystems classes.
- Actions — define a view; print out the details of a table definition; improve the performance of a query by running Tune Table and/or rebuilding indices; or perform clean up by purging unwanted cached queries and/or dropping unwanted table, view, or procedure definitions.
- Open Table — display the current data in the table in Display mode. This is commonly not the complete data in the table: both the number of records and the length of data in a column are restricted to provide a manageable display.
- Documentation — Allows you to view the list of SQL error codes and the list of SQL reserved words. If you select a table, allows you display Class Documentation (the *Class Reference* page for that table).

16.1.1 Selecting a Namespace

All SQL operations occur within a specific namespace. Therefore, you must first specify which namespace you wish to use by clicking the **Switch** option at the top of the SQL interface page. This displays the list of available namespaces, from which you can make your selection.

You can set your Management Portal default namespace. From the Management Portal select **System Administration**, **Security**, **Users** (**[System] > [Security Management] > [Users]**). Click the name of the desired user. This allows you to edit the user definition. From the General tab, select a **Startup Namespace** from the dropdown list. Click **Save**.

16.2 Executing SQL Statements

From the Management Portal select **System Explorer**, then **SQL** (**[System] > [SQL]**). Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces. To execute an SQL query, there are three options:

- Execute Query: write and execute an SQL statement. The SQL statement can be a **SELECT** query, or it can be a Caché SQL DDL or DML statement; the statement is validated on the Caché server when it executes.
- Show History: recall a previously run SQL statement, and either re-run it, or modify it and then run it. All executed statements are listed, including those that did not successfully execute.
- **Query Builder**: invoke the SQL Query Builder (which is exclusively for creating **SELECT** statements). Within the SQL Query Builder, create an SQL SELECT query by choosing tables, columns, WHERE clause predicates, and other query components. You can then run the query by clicking **Execute Query**.

16.2.1 Writing SQL Statements

The **Execute Query** text box allows you to write not only **SELECT** and **CALL** queries, but most SQL statements, including DDL statements such as **CREATE TABLE**, and DML statements such as **INSERT**, **UPDATE**, and **DELETE**.

You can specify SQL code in the **Execute Query** text box using the following:

- Type (or paste) the SQL code into the text box. The SQL code area does not colorize SQL text or provide any syntax or existence validation. However, it does provide automatic spelling verification. You can erase the contents of the text box using the **X** icon.
- Use the Show History list to select a prior SQL statement. The selected statement is copied into the text box. Upon execution, this statement moves to the top of the Show History list. Note that Show History lists all previously executed statements, including those that failed execution.
- Use Table Drag and Drop to construct SQL code in the text box.
- You can use the **Query Builder**, rather than the **Execute Query** text box, to specify and execute a SELECT query. A SELECT query executed using **Query Builder** is not shown in **Execute Query** or listed in **Show History**.

SQL code in the **Execute Query** text box can include:

- ? Input Parameters. If you specify input parameters, such as `TOP ?` or `WHERE Age BETWEEN ? AND ?`, the **Execute** button displays the **Enter Parameter Value for Query** window, with entry fields for each input parameter in the order specified in the query. For further details on ? input parameters, refer to Executing an SQL Statement in the "Using Dynamic SQL" chapter of this manual.

- Whitespace Characters. You can specify multiple blank spaces, single and multiple line returns. The tab key is disabled; when copying code into the SQL code area, existing tabs are converted to single blank spaces. Line returns and multiple blank spaces are not retained.
- Comments. The SQL code area supports single-line and multiline comments. Comments are retained and shown in the Show History display. Comments are not shown in the Show Plan Statement Text display or in cached queries.
- Queries that Return Multiple Result Sets.

After writing SQL code in the text box, you can click the **Show Plan** button to check the SQL code without executing the SQL code. If the code is valid, **Show Plan** displays a Query Plan. If the code is invalid, **Show Plan** displays an SQLCODE error value and message. You can also use the **Show Plan** button to display this information for the most-recently-executed SQL code.

To execute the SQL code, click the **Execute** button.

16.2.1.1 Table Drag and Drop

You can generate a query by dragging a table (or view) from the Tables list (or Views list) on the left side of the screen and dropping it into the **Execute Query** text box. This generates a **SELECT** with a *select-item* list of all of the non-hidden fields in the table and a FROM clause specifying the table. You can then further modify this query and execute it using the **Execute** button.

You can also drag and drop a procedure name from the Procedures list on the left side of the screen.

16.2.2 Execute Query Options

The SQL execution interface has the following options:

- The Select Mode drop-down list with a **SELECT** specifies the format that the query should use to supply data values (for example, in the WHERE clause) and to display data values in the query result set. The options are **Display Mode** (the default), **ODBC Mode**, and **Logical Mode**. For further details on these options, refer to "Data Display Options" in the "Caché SQL Basics" chapter of this book.

  The Select Mode drop-down list with an **INSERT** or **UPDATE** allows you to specify whether input data will be converted from display format to logical storage format. For this data conversion to occur, the SQL code must have been compiled with a select mode of RUNTIME. At execution time, the Select Mode drop-down list must be set to **Logical Mode**. For further details, refer to the INSERT or UPDATE statement in the *Caché SQL Reference*.

  Select Mode is meaningful for data types whose Logical storage format differs from the desired display format (Display or ODBC), such as Caché dates and times and Caché %List structured data.
- The **Max** field allows you to limit how many rows of data to return from a query. It can be set to any positive integer, including 0. Once you set **Max**, that value is used for all queries for the duration of the session, unless explicitly changed. The default is 1000. The maximum value is 100,000, which is the default if you enter no value (set **Max** to null), enter a value greater than 100,000, or a non-numeric value. You can also limit the number of rows of data to return by using a TOP clause. **Max** has no effect on other SQL statements, such as **DELETE**.
- The **Show Plan** button displays the Query Text and the Query Plan including the relative cost (overhead) of the current query plan for the query in the page's text box. You can invoke Show Plan from either the **Execute Query** or **Show History** interface. A query plan is generated when a query is Prepared (compiled); this occurs when you write a query and select the **Show Plan** button. You do not have to execute a query to show its query plan. **Show Plan** displays an SQLCODE and error message when invoked for an invalid query.

If you click the **more** option, the SQL execution interface displays the following additional options:

- **Dialect**: the dialect of SQL code. Available values are Cache, Sybase, and MSSQL. The default is Cache. Sybase and MSSQL are described in the Caché Transact-SQL (TSQL) Migration Guide .

- **Row Number**: a check box specifying whether to include a row count number for each row in the result set display. Row Number is a sequential integer assigned to each row in the result set. This is simply a numbering of the returned rows, it does not correspond either the RowID or the %VID. The row number column header name is #. The default is to display row numbers.

16.2.3 SQL Statement Results

After writing SQL code in the **Execute Query** text box, you can execute the code by clicking the **Execute** button. This either successfully executes the SQL statement and displays the results below the code window, or the SQL code fails. If the SQL code fails, it displays an error message (in red) below the code window; pressing the **Show Plan** button displays the SQLCODE error and error message.

The result set is returned as a table with a row counter displayed as the first column (#), if the Row Number box is checked. The remaining columns are displayed in the order specified. The RowID (ID field) may be displayed or hidden. Each column is identified by the column name (or the column alias, if specified). An aggregate, expression, subquery, host variable, or literal SELECT item is either identified by a column alias (if specified), or by the word `Aggregate_`, `Expression_`, `Subquery_`, `HostVar_`, or `Literal_` followed by the SELECT item sequence number (by default).

If a row column contains no data (NULL) the result set displays a blank table cell. Specifying an empty string literal displays a `HostVar_` field with a blank table cell. Specify NULL displays a `Literal_` field with a blank table cell.

If a selected field is a date, time, timestamp, or %List-encoded field, the displayed value depends on the Display Mode.

If the specified query returns more than one result set, **Execute Query** displays these result sets as named tabs: **Result #1**, **Result #2**, and so forth.

- If the query executes successfully, it displays performance information and the name of the cached query routine. If there is resulting data to display, this appears below the performance information. The execution information includes the `Row count`, the `Performance`, the `Cached Query` showing the cached query name, and `Last update` specifying the timestamp for the last execution of the query.
  - `Row count`: For a DDL statement such as CREATE TABLE, displays `Row count: 0` if the operation was successful; displays no value for `Row count` if the operation failed. For a DML statement such as INSERT, UPDATE, or DELETE, displays the number of rows affected.

    For a SELECT, displays the number of rows returned as a result set. Note that the number of rows returned is governed by the **Max** setting, which may be lower than the number of rows which could have been selected. For multiple result sets, the number of rows for each result set are listed, separated by the / character. A query that specifies one or more aggregate functions (and no selected fields) always displays `Row count: 1` and returns the results of expressions, subqueries, and aggregate functions, even if the FROM clause table contains no rows. A query that specifies no aggregate functions and selects no rows always displays `Row count: 0` and returns no results, even if the query specifies only expressions and subqueries that do not reference the FROM clause table. A query with no FROM clause always displays `Row count: 1` and returns the results of expressions, subqueries, and aggregate functions.
  - `Performance`: measured in elapsed time (in fractional seconds), total number of global references, total number of lines executed, and disk read latency (in milliseconds). If a cached query exists for the query these performance metrics are for executing the cached query. Therefore, the first execution of a query will have substantially higher performance metrics than subsequent executions. If the specified query returns more than one result set, these performance metrics are totals for all of the queries.

    To analyze these performance metrics for the cached query in greater depth, refer to Examining Routine Performance Using ^%SYS.MONLBL.
  - `Cached Query`: the automatically generated cached query class name. For example, `%sqlcq.USER.cls2` indicating the second cached query in the USER namespace. Each new query is assigned a new cached query name with the next consecutive integer. By clicking this cached query name, you can display information about

the cached query and further links to display its Show Plan or to Execute the cached query. (For a DDL statement, see SQL Commands That Are Not Cached.)

Closing the Management Portal or stopping Caché does not delete cached queries or reset cached query numbering. To purge cached queries from the current namespace, invoke the **%SYSTEM.SQL.Purge()** method.

Not all SQL statements result in a cached query. A query that is the same as an existing cached query, except for literal substitution values (such as the TOP clause value and predicate literals) does not create a new cached query. Some SQL statements are not cached, including DDL statements and privilege assignment statements. Non-query SQL statements, such as **CREATE TABLE**, also display a cached query name. However, this cached query name is created then immediately deleted; the next SQL statement (query or non-query) reuses the same cached query name.

– `Last update`: the date and time that the last **Execute Query** (or other SQL operation) was performed. This timestamp is reset each time the query is executed, even when repeatedly executing the identical query.

Successful execution also provides a **Print** link that displays the Print Query window, which gives you the options to either print or export to a file the query text and/or the query result set. The clickable **Query** and **Result** toggles enable you to display or hide the query text or the query result set. The displayed query result set includes the namespace name, the result set data and row count, a timestamp, and the cached query name. (Note that the timestamp is the time when the Print Query window was invoked, not the time when the query was executed.) The Print Query window **Print** button prints a screenshot of the Print Query window. The Export to File check box displays options to specify an export file format (xml, hdml, pdf, txt, csv) and an export file pathname. The **Export** option ignores the **Query** and **Result** toggles and always exports only the result set data and the row count; (the query text, namespace, timestamp, and cached query name are not included).

- If unsuccessful, it displays an error message. You can click the **Show Plan** button to display the corresponding SQLCODE error value and message.

16.2.4 Show History

Click **Show History** to list prior SQL statements executed during the current session. Show History lists all SQL statements invoked from this interface, both those successfully executed and those whose execution failed. By default, SQL statements are listed by Execution Time, with the most recently executed appearing at the top of the list. You can click on any of the column headings to order the SQL statements in ascending or descending order by column values. Executing an SQL Statement from the **Show History** listing updates its Execution Time (local date and time stamp), and increments its Count (number of times executed).

You can filter the **Show History** listing, as follows: in the **Filter** box specify a string then press the Tab key. Only those history items that contain that string will be included in the refreshed listing. The filter string can either be a string found in the SQL Statement column (such as a table name), or it can be a string found in the Execution Time column (such as a date). The filter string is not case-sensitive. A filter string remains in effect until you explicitly change it.

You can modify and execute an SQL statement from **Show History** by selecting the statement, which causes it to be displayed in the **Execute Query** text box. In **Execute Query** you can modify the SQL code and then click **Execute**. Making any change to an SQL statement retrieved from **Show History** causes it to be stored in **Show History** as a new statement; this include changes that do not affect execution, such as changing letter case, whitespace, or comments. Whitespace is not shown in **Show History**, but it is preserved when an SQL statement is retrieved from **Show History**.

You can execute (re-run) an unmodified SQL statement directly from the **Show History** list by clicking the **Execute** button found to the right of the SQL statement in the Show History listing.

Note that the **Show History** listing is not the same as the list of cached queries. **Show History** lists all invoked SQL statements from the current session, including those that failed during execution.

16.2.5 Other SQL Interfaces

Caché supports numerous other ways to write and execute SQL code, as described in other chapters of this manual. These include:

- Embedded SQL: SQL code embedded within ObjectScript code.
- Dynamic SQL: using %SQL.Statement class methods (or other result set class methods) to execute statements from within ObjectScript or Caché Basic code.
- SQL Shell: executing Dynamic SQL from the Terminal using the SQL Shell interface.

16.3 Filtering Schema Contents

The left side of the Management Portal SQL interface allows you to view the contents of a schema (or multiple schemas that match a filter pattern).

1. Specify which namespace you wish to use by clicking the **Switch** option at the top of the SQL interface page. This displays the list of available namespaces, from which you can make your selection.
2. Apply a **Filter** or select a schema from the **Schema** drop-down list.

   You can use the **Filter** field to filter the lists by typing a search pattern. You can filter for schemas, or for table/view/procedure names (items) within a schema or within multiple schemas. A search pattern consists of the name of a schema, a dot (.), and the name of an item — each name composed of some combination of literals and wildcards. Literals are not case-sensitive. The wildcards are:

   - asterisk (*) meaning 0 or more characters of any type.
   - underscore (_) meaning a single character of any type.
   - an apostrophe (') inversion prefix meaning “not” (everything except).
   - a backslash (\) escape character: _ means a literal underscore character.

   For example, `S*` returns all schemas that begin with S. `S*.Person` returns all Person items in all schemas that begin with S. `*.Person*` returns all items that begin with Person in all schemas. You can use a comma-separated list of search patterns to select all items that fulfil any one of the listed patterns (OR logic). For example, `*.Person*,*.Employee*` selects all Person and Employee items in all schemas.

   To apply a **Filter** search pattern, click the refresh button, or press the Tab key.

   A **Filter** search pattern remains in effect until you explicitly change it. The “x” button to the right of the **Filter** field clears the search pattern.
3. Selecting a schema from the **Schema** drop-down list overrides and resets any prior **Filter** search pattern, selecting for a single schema. Specifying a **Filter** search pattern overrides any prior **Schema**.
4. Optionally, use the drop-down “applies to” list to specify which categories of item to list: Tables, Views, Procedures, Cached Queries, or all of the above. The default is All. Any category that was specified in the “applies to” drop-down list is limited by **Filter** or **Schema**. Those categories not specified in “applies to” continue to list all of the items of that category type in the namespace.
5. Optionally, click the **System** check box to include system items (items whose names begin with %). The default is to not include system items.

6. Expand the list for a category to list its items for the specified **Schema** or specified **Filter** search pattern. When you expand a list, any category that contains no items does not expand.
7. Click on an item in an expanded list to display its **Catalog Details** on the right side of the SQL interface.

   If the selected item is a Table or a Procedure, the **Catalog Details Class Name** information provides a link to the corresponding *Class Reference* documentation.

16.3.1 Browse Tab

The Browse tab provides a convenient way to quickly view all the schemas in a namespace, or a filtered subset of the schemas in the namespace. You can select **Show All Schemas** or **Show Schemas with Filter**, which applies the filter specified on the left side of the Management Portal SQL interface. By clicking on the Schema Name heading, you can list the schemas in ascending or descending alphabetical order.

Each listed schema provides links to lists of its associated Tables, Views, Procedures, and Queries (cached queries). If the schema has no items of that type, a hyphen (rather than a named link) is shown in that schema list column. This enables you to quickly get information about the contents of schemas.

Clicking a Tables, Views, Procedures, or Queries link displays a table of basic information about those items. By clicking on a table heading, you can sort the list by that column's values in ascending or descending order. The Procedures table always includes Extent procedures, regardless of the Procedures setting on the left side of the Management Portal SQL interface.

You can get more information on individual Tables, Views, Procedures, and Cached Queries using the Catalog Details tab. Selecting a Table or View from the Browse tab *does not* activate the **Open Table** link for that table.

16.4 Catalog Details

The Management Portal provides **Catalog Details** information for each Table, View, Procedure, and Cached Query. The filtering schema contents (left side) component of the Management Portal SQL interface allows you to select an individual item to display its **Catalog Details**.

16.4.1 Catalog Details for a Table

The following **Catalog Details** options are provided for each table:

- Table Info: Table Type: either TABLE, GLOBAL TEMPORARY, or SYSTEM TABLE (system tables are only displayed if the System check box is selected), Owner name, Last Compiled timestamp, External and Readonly boolean values, Class Name, Extent Size, the name of the Child Table(s) and/or the Parent Table (if relevant) and one or more References fields to other tables (if relevant), whether it uses the %CacheStorage default storage class, and whether it Supports Bitmap Indices.

  Class Name is a link to the corresponding entry in the *InterSystems Class Reference* documentation. The **Class Name** is a unique package.class name derived from the table name by removing punctuation characters, as described in Identifiers and Class Entity Names.

  References only appears in Table Info if there is one or more references from a field in the current table to another table. These references to other tables are listed as links to the Table Info for the referenced table.

  This option also provides a modifiable value for the **Number of rows to load when table is opened**. This sets the maximum number of rows to display in Open Table. The available range is from 1 to 10,000; the default is 100. The Management Portal corrects a value outside the available range to a valid value: 0 corrects to 100; a fractional number rounds up to the next higher integer; a number greater than 10,000 corrects to 10,000.

- Fields: a list of the fields in the table showing: Field Name, Datatype, Column #, Required, Unique, Collation, Hidden, MaxLen, MaxVal, MinVal, Stream, Container, xDBC Type, Reference To, Version Column, Selectivity, Outlier Selectivity, Outlier Value, and Average Field Size.
- Maps/Indices: a list of the indices defined for the table showing: Index Name, SQL Map Name, Columns, Type, Block Count, Map Inherited. **Index Name** is the index property name and follows property naming conventions; when generated from an SQL index name, punctuation characters (such as underscores) in the SQL index name are stripped out. The **SQL Map Name** is the SQL name for the index. A generated **SQL Map Name** is the same as the Constraint Name, and follows the same naming conventions (described below). **Columns** specifies a field or a comma-separated list of fields specified for the index; it may specify the index collation type and full schema.table.field reference, as in the following example: `$$SQLUPPER({Sample.People.Name})`. **Type** can be one of the following: Bitmap Extent, Data/Master, Index (standard index or bitslice index), Bitmap index, and Unique constraint. The **Block Count** contains both the count and how that count was determined: set explicitly by the class author (Defined), computed by TuneTable (Measured), or estimated by the class compiler (Estimated). If **Map Inherited?** is Yes, this map was inherited from a superclass.

  This option also provides a link for each index to rebuild the index.
- Triggers: a list of the triggers defined for the table showing: Trigger Name, Time Event, Order, Code.
- Constraints: a list of the constraints for fields of the table showing: Constraint Name, Constraint Type, and Constraint Data (field name(s) listed in parentheses). Constraints include primary key, foreign key, and unique constraints. A primary key is, by definition, unique; it is only listed once. This option list constraints by constraint name; a constraint involving multiple fields is listed once with Constraint Data displaying a comma-separated list of the component fields. The Constraint Type can be UNIQUE, PRIMARY KEY, Implicit PRIMARY KEY, FOREIGN KEY, or Implicit FOREIGN KEY.

  You can also list constraints by invoking INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE. This list constraints by field name. The following example returns the name of the field and the name of the constraint for all UNIQUE, PRIMARY KEY, FOREIGN KEY and CHECK constraints:

  SQL

  ```
  SELECT Column_Name,Constraint_Name FROM INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE WHERE
  TABLE_SCHEMA='Sample' AND TABLE_NAME='Person'
  ```

  If the table is defined with %PUBLICROWID and no explicit primary key is defined, the RowID field is listed with a Constraint Type of `Implicit PRIMARY KEY` with the Constraint Name `RowIDField_As_PKey`.

  For explicit constraints, the Constraint Name is generated as follows:

  – Constraint specified in the field definition: For example, `FullName VARCHAR(48) UNIQUE` or `FullName VARCHAR(48) PRIMARY KEY`. The Constraint Name value for the field is a generated value with the syntax `TABLENAME_CTYPE#`, where CTYPE is UNIQUE, PKEY, or FKEY, and # is a sequential integer assigned to unnamed constraints in the order specified in the table definition. For example, if FullName has the 2nd unnamed unique constraint (excluding the ID field) in the MyTest table, the generated Constraint Name for FullName would be `MYTEST_UNIQUE2`; if FullName is the primary key and the 3rd unnamed constraint (excluding the ID field) specified in the MyTest table, the generated Constraint Name for FullName would be `MYTEST_PKEY3`.

  – CONSTRAINT keyword named constraint clause: For example, `CONSTRAINT UFullName UNIQUE(FirstName,LastName)` or `CONSTRAINT PKName PRIMARY KEY(FullName)`), the Constraint Name is the specified unique constraint name. For example, FirstName and LastName in the MyTest table would each have the Constraint Name `UFullName`; FullName would have the Constraint Name `PKName`.

  – Unnamed constraint clause: For example, `UNIQUE(FirstName,LastName)` or `PRIMARY KEY (FullName)`. the Constraint Name value is a generated value with the syntax `TABLENAMECType#`, where CType is Unique, PKey, or FKey, and # is a sequential integer assigned to unnamed constraints in the order specified in the table definition. For example, if FirstName and LastName have the 2nd unnamed unique constraint (excluding the ID field) in the MyTest table, the generated Constraint Name for FirstName and LastName would be `MYTESTUnique2`;

if FullName is the primary key and the 3rd unnamed constraint (excluding the ID field) specified in the MyTest table, the generated Constraint Name for FullName would be `MYTESTPKey3`. (Note mixed uppercase/lowercase and absence of an underscore.)

If a field is involved in more than one uniqueness constraint, it is listed separately for each Constraint Name.

- Cached Queries: a list of the cached queries for the table showing: Routine name, Query text, Creation Time, Source, Query Type.
- **Table's SQL Statements**: a list of the SQL Statements generated for this table. Same information as namespace-wide SQL Statements display.

16.4.2 Catalog Details for a View

Management Portal SQL interface also provides **Catalog Details** for views, procedures, and cached queries:

The following **Catalog Details** options are provided for each view:

- View Info: **Owner** name, **Last Compiled** timestamp. This timestamp updates when you use the **Edit View** link and save changes.

  Defined as Read Only and **View is Updateable** booleans: if view definition included WITH READ ONLY, these are set to 1 and 0 respectively. Otherwise, if the view is defined from a single table they are set to 0 and 1; if the view is defined from joined tables they are set to 0 and 0. You can change this option using the **Edit View** link.

  Class Name is a unique package.class name derived from the view name by removing punctuation characters, as described in Identifiers and Class Entity Names.

  Check Option is only listed if the view definition included the WITH CHECK OPTION clause. It can be LOCAL or CASCADED. You can change this option using the **Edit View** link.

  Class Type is VIEW. It provides an **Edit View** link to edit the view definition.

  View Text is the SELECT statement used to define the view. You can change the view definition using the **Edit View** link.

  The list of fields includes the Field Name, Data Type, MAXLEN Parameter, MAXVAL Parameter, MINVAL Parameter, BLOB (%Stream.GlobalCharacter or %Stream.GlobalBinary field), Length, Precision, and Scale.

- **View's SQL Statements**: a list of the SQL Statements generated for this view. Same information as namespace-wide SQL Statements display.

16.4.3 Catalog Details for a Stored Procedure

The following **Catalog Details** options are provided for each procedure:

- Stored Procedure Info:

  Class Name is a unique package.class name derived from the procedure name by pre-pending a type identifier ('func', 'meth', 'proc', or 'query') to the class name (for example, the SQL function MyProc becomes funcMyProc) and removing punctuation characters, as described in Identifiers and Class Entity Names. **Class Document** is a link to the corresponding entry in the *InterSystems Class Reference*. **Procedure Type** (for example, function). **Method or Query Name** the name of the generated class method or class query; this name is generated described in Identifiers and Class Entity Names. The **Run Procedure** link provides an option to interactively run the procedure.

- **Stored Procedure's SQL Statements**: a list of the SQL Statements generated for this stored procedure. Same information as namespace-wide SQL Statements display.

16.4.4 Catalog Details for a Cached Query

Cached Query provides the full text of the query, an option to show the query execution plan, and an option to interactively execute the cached query.

16.5 Open Table

If you select a table or view on the left side of the Management Portal SQL interface, the Catalog Details for that table or view are displayed. The **Open Table** link at the top of the page also becomes active. **Open Table** displays the actual data in the table (or accessed via the view). The data is shown in Display format.

By default, the first 100 rows of data are displayed; this default is modifiable by setting the **Number of rows to load when table is opened** in the Catalog Details tab Table Info. If there are more rows in the table than this number of rows to load value, the `More data...` indicator is shown at the bottom of the data display. If there are fewer rows in the table than this number of rows to load value, the `Complete` indicator is shown at the bottom of the data display.

If the data in a column is too long to be displayed, the first 100 characters of the data for that column are displayed followed by an ellipsis (`...`) indicating additional data.

A column of data type %Stream.GlobalCharacter displays the actual data (up to 100 characters) as a string. A column of data type %Stream.GlobalBinary displays as `<binary>`.

16.6 Actions

- **Create View** — Displays a page for creating a view. Instructions for using this option are provided in the "Defining and Using Views" chapter of this book.
- **Print Catalog** — Allows you to print complete information about a table definition. Clicking **Print Catalog** displays a print preview. By clicking Indices, Triggers, and/or Constraints on this print preview you can include or exclude this information from the catalog printout.
- **Purge Cached Queries** — Provides three options for purging cached queries: purge all cached queries for the current namespace, purge all cached queries for the specified table, or purge only selected cached queries.
- **Tune Table Information** — Run the Tune Table facility against the selected table. This calculates the selectivity of each table column against the current data. A selectivity value of 1 indicates a column that defined as unique (and therefore has all unique data values). A selectivity value of 1.0000% indicates a column not defined as unique for which all current data values are unique values. A percentage value greater that 1.0000% indicates the relative number of duplicate values for that column in the current data. By using these selectivity values you can determine what indexes to define and how to use these indexes to optimize performance.
- **Tune all tables in schema** — Run the Tune Table facility against all of the tables belonging to a specified schema in the current namespace.
- **Rebuild Table's Indices** — Rebuild all indexes for the specified table.
- **Drop this item** — Drop (delete) the specified table definition, view definition, procedure, or cached query. You must have the appropriate privileges to perform this operation. **Drop** cannot be used on a table created by defining a persistent class, unless the table class definition includes [DdlAllowed]. Otherwise, the operation fails with an SQLCODE -300 error with the %msg `DDL not enabled for class 'Schema.tablename'`.

If a class is defined as a linked table, the Drop action drops the linked table on the local system, even if the linked table class is not defined as DdlAllowed. Drop does not drop the actual table this link references that resides on the server.

16.7 Wizards

- **Data Import Wizard** — Runs a wizard to Import data from a text file into a Caché class.
- **Data Export Wizard** — Runs a wizard to export data from a Caché class into a text file.
- **Data Migration Wizard** — Runs a wizard to migrate data from an external source and create a Caché class definition to store it.
- **Link Table Wizard** — Runs a wizard to link to tables or views in external sources as if it were native Caché data.
- **Link Procedure Wizard** — Runs a wizard to link to procedures in external sources.
- **FileMan Wizard** — Runs a wizard to map FileMan files to InterSystems classes.

17

Importing SQL Code

This chapter describes how to import SQL code from a text file into Caché SQL. When you import SQL code, Caché prepares and executes each line of SQL using the %Library.ResultSet dynamic SQL class. If it encounters a line of code it cannot parse, SQL import skips over that line of code and continues to prepare and execute subsequent lines until it reaches the end of the file. All SQL code import operations import to the current namespace.

SQL Import is primarily used to import Data Definition Language (DDL) statements, such as CREATE TABLE, and to populate tables using INSERT, UPDATE, and DELETE statements. SQL import does prepare and execute SELECT statements, but does not create a result set.

SQL import can be used to import Caché SQL code. It can also be used for code migration, to import SQL code from other vendors (FDBMS, Informix, InterBase, MSSQLServer, MySQL, Oracle, Sybase). Code from other vendors is converted to Caché SQL code and executed. SQL import cannot import all SQL statements into Caché SQL. It imports those statements and clauses that are compatible with the Caché implementation of the SQL standard. Incompatible features are commonly parsed, but ignored.

Successfully executed SQL statements create a corresponding cached query, where appropriate.

You perform SQL code import by invoking the appropriate method from the %SYSTEM.SQL class. When importing SQL code, these methods can create two other files: an Errors.log file which records errors in parsing SQL statements, and an Unsupported.log file, which contains the literal text of lines that the method does not recognize as an SQL statement.

This chapter describes importing different types of SQL code:

- Importing Caché SQL
- Importing non-Caché SQL: FDBMS, Informix, InterBase, MSSQLServer, MySQL, Oracle, Sybase

For further details on %Library.ResultSet refer to "Dynamic SQL Using %Library.ResultSet".

17.1 Importing Caché SQL

You can import Caché SQL code from a text file using either of the following methods:

- **DDLImport()** is a general-purpose SQL import method. This method runs as a background (non-interactive) process. To import Caché SQL you specify "CACHE" as the first parameter.
- **Cache()** is a Caché SQL import method. This method runs interactively from the Terminal. It prompts you to specify the location of the import text file, the location to create the Errors.log file and the Unsupported.log file, and other information.

The following example imports the Caché SQL code file mysqlcode.txt, executing its SQL statements in the current namespace:

ObjectScript

```
DO $SYSTEM.SQL.DDLImport("CACHE",$USERNAME,"c:\temp\mysqlcode.txt",,1)
```

By default, **DDLImport()** creates an errors log file. This example creates a file named `mysqlcode_Errors.log` in the same directory as the SQL code file. The fifth parameter is a boolean specifying whether or not to create a file that lists unsupported statements. The default is 0. In this example, the fifth parameter is set to 1, creating a file named `mysqlcode_Unsupported.log` in the same directory as the SQL code file. These log files are created even when there is nothing written to them.

When executing **DDLImport()** from the Terminal, it first lists the input file, the error log file, and the unsupported log file. Then for each SQL command it displays a listing such as the following:

```
SQL statement to process (number 1):
     CREATE TABLE Sample.MyStudents (StudentName VARCHAR(32),
     StudentDOB DATE)
      Preparing SQL statement...
      Executing SQL statement...
  DONE
```

If an error occurs in any SQL command, the Terminal display the error, as shown in the following example:

```
SQL statement to process (number 3):
       INSERT INTO Sample.MyStudents (StudentName,StudentDOB) SELECT Name,
       DOB FROM Sample.Person WHERE Age <= '21'
    Preparing SQL statement...
ERROR #5540: SQLCODE: -30 Message:  Table 'SAMPLE.PERSON' not found
       Pausing 5 seconds - read error message!  (Type Q to Quit)
```

If you do not Quit within 5 seconds, **DDLImport()** proceeds to execute the next SQL command. The error is recorded in the errors log file with a timestamp, the user name, and the namespace name.

17.1.1 Import File Format

An SQL text file must be an unformatted file such as a .txt file. Each SQL statement must begin on its own line. An SQL statement may be broken into multiple lines and indentation is permitted. By default, each SQL statement must be followed by a GO statement on its own line.

The following is an example of a valid Caché SQL import file text:

```
 CREATE TABLE Sample.MyStudents (StudentName VARCHAR(32),StudentDOB DATE)
GO
 CREATE INDEX NameIdx ON TABLE Sample.MyStudents (StudentName)
GO
 INSERT INTO Sample.MyStudents (StudentName,StudentDOB) SELECT Name,
 DOB FROM Sample.Person WHERE Age <= '21'
GO
 INSERT INTO Sample.MyStudents (StudentName,StudentDOB)
      VALUES ('Jones,Mary',60123)
GO
 UPDATE Sample.MyStudents SET StudentName='Smith-Jones,Mary' WHERE StudentName='Jones,Mary'
GO
 DELETE FROM Sample.MyStudents WHERE StudentName %STARTSWITH 'A'
GO
```

By setting the **DDLImport("CACHE")** *deos* seventh parameter, this method can accept (but does not require) a specified end-of-statement delimiter, commonly a semicolon (;), at the end of each SQL statement. The default is to not support an end-of-statement delimiter. The “GO” statement on the line following an SQL statement is always supported, but is not required if *deos* specifies an end-of-statement delimiter.

17.1.2 Supported SQL Statements

Not all valid Caché SQL code statements can be imported. The following is a list of supported Caché SQL commands:

- **CREATE TABLE**, **ALTER TABLE**, **DROP TABLE**
- **CREATE VIEW**, **ALTER VIEW**, **DROP VIEW**
- **CREATE INDEX** all index types, except bitslice
- **CREATE USER**, **DROP USER**
- **CREATE ROLE**
- **GRANT**, **REVOKE**
- **INSERT**, **UPDATE**, **INSERT OR UPDATE**, **DELETE**
- **SET OPTION**
- **SELECT** for optimizer plan mode only

17.2 Code Migration: Importing non-Caché SQL

You can import SQL code that is in the SQL format used by other vendors. Code from other vendors is converted to Caché SQL code and executed. The following methods are provided:

- **DDLImport()** is a general-purpose SQL import method. This method runs as a background (non-interactive) process. Refer to Importing Caché SQL for general information on using this method.

  To import SQL in a specific format you specify the name of that format as the first parameter: FDBMS, Informix, InterBase, MSSQLServer, MySQL, Oracle, or Sybase.

- Individual interactive methods are provided to import the following types of SQL: **FDBMS()**, **Informix()**, **InterBase()**, **MSSQLServer()**, **Oracle()**, and **Sybase()**. These methods runs interactively from the Terminal. It prompts you to specify the location of the import text file, the location to create the Errors.log file and the Unsupported.log file, and other information.

- **DDLImportDir()** allow you to import SQL code from multiple files in a directory. This method runs as a background (non-interactive) process. It supports Informix, MSSQLServer, and Sybase. All files to be imported must have a .sql extension suffix.

- **ImportDir()** allow you to import SQL code from multiple files in a directory. Provides more options than **DDLImportDir()**. This method runs as a background (non-interactive) process. It supports MSSQLServer, and Sybase. You can specify a list of allowed file extension suffixes.

18

Using Triggers

This chapter describes how you can define triggers in Caché SQL. Triggers are lines of code that are executed in response to certain SQL events. This chapter includes the following topics:

- Defining Triggers
- Types of Triggers
- Trigger Code
- Pulling Triggers
- Triggers and Object Access
- Triggers and Transactions
- Listing Triggers

18.1 Defining Triggers

There are several ways to define a trigger for a specific table:

- Include a trigger definition in the persistent class definition that projects to an SQL table. For example, this definition of the MyApp.Person class includes a definition of the LogEvent trigger, which is invoked after each call to INSERT data into the MyApp.Person table:

  Class Definition

  ```
  Class MyApp.Person Extends %Persistent [DdlAllowed]
  {
      // ... Definitions of other class members

      Trigger LogEvent [ Event = INSERT, Time = AFTER ]
      {
          // trigger code to log an event
      }
   }
  ```

  For further details, refer to Trigger Definitions.

- Use the SQL CREATE TRIGGER command to create a trigger. This generates a trigger object definition in the corresponding persistent class. SQL trigger names follow identifier naming conventions. Caché uses the SQL trigger name to generate a corresponding trigger class entity name.

You must have the %CREATE_TRIGGER administrative-level privilege to create a trigger. You must have the %DROP_TRIGGER administrative-level privilege to drop a trigger.

Note: This chapter describes Caché SQL triggers. Caché MultiValue triggers are completely separate from Caché SQL triggers. An SQL update will not fire a MultiValue trigger; a MultiValue update will not fire an SQL trigger.

The maximum number of user-defined triggers for a class is 200.

Note: Caché SQL does not support triggers on tables projected by collections. A user cannot define such a trigger, and the projection of a collection as a child table does not consider triggers involving that base collection.

18.2 Types of Triggers

A trigger is defined by the following:

- The type of event that causes it to execute. A trigger may be either a single-event trigger or a multiple-event trigger. A single-event trigger is defined to execute when an INSERT, an UPDATE, or a DELETE event occurs on the specified table. A multiple-event trigger is defined to execute when any one of the multiple specified events occurs on the specified table. You can define an INSERT/UPDATE, an UPDATE/DELETE, or an INSERT/UPDATE/DELETE multiple-event trigger. The type of event is specified in a class definition by the required Event trigger keyword. You cannot specify a multi-event trigger using the SQL **CREATE TRIGGER** command.
- The time that the trigger executes: Before or After the event occurs. This is specified in a class definition by the optional Time trigger keyword. The default is Before.
- You can associate multiple triggers with the same event and time; in this case, you can control the order in which multiple triggers are fired using the Order trigger keyword. Triggers with a lower Order value are fired first. If multiple triggers have the same Order value, then the order in which they are fired is not specified.
- The optional Foreach trigger keyword provides additional granularity. This keyword controls whether the trigger is fired once per row (`Foreach = row`), once per row or object access (`Foreach = row/object`), or once per statement (`Foreach = statement`). A trigger defined with no Foreach trigger keyword is fired once per row. If a trigger is defined with `Foreach = row/object`, then the trigger is also called at specific points during object access, as described later in this chapter. You can list the Foreach value for each trigger using the ACTIONORIENTATION property of INFORMATION.SCHEMA.TRIGGERS

For a full list of trigger keywords, see the Class Definition Reference.

The following are the available triggers and their equivalent callback methods:

- BEFORE INSERT (equivalent to %OnBeforeSave())
- AFTER INSERT (equivalent to %OnAfterSave())
- BEFORE UPDATE (equivalent to %OnBeforeSave())
- AFTER UPDATE (equivalent to %OnAfterSave())
- BEFORE UPDATE OF specified column(s)
- AFTER UPDATE OF specified column(s)
- BEFORE DELETE (equivalent to %OnDelete())
- AFTER DELETE (equivalent to %OnAfterDelete())

Note: When a trigger is executed, it cannot directly modify the value of a property in the table that is being processed. This is because Caché executes trigger code after field (property) value validation code. For example, a trigger cannot set a LastModified field to the current timestamp in the row being processed. However, the trigger code can issue an **UPDATE** to a field value in the table. The **UPDATE** performs its own field value validation.

For further details, refer to CREATE TRIGGER in the *InterSystems SQL Reference*.

18.2.1 AFTER Triggers

An AFTER trigger executes after an INSERT, UPDATE, or DELETE event occurs:

- If SQLCODE=0 (event completed successfully) Caché executes the AFTER trigger.
- If SQLCODE is a negative number (event failed) Caché does not executes the AFTER trigger.
- If SQLCODE=100 (no row was found to insert, update, or delete) Caché executes the AFTER trigger.

18.2.2 Recursive Triggers

Trigger execution can be recursive. For example, if table T1 has a trigger that performs an insert into table T2 and table T2 has a trigger that performs an insert into table T1. Recursion can also occur when table T1 has a trigger that calls a routine/procedure and that routine/procedure performs an insert into T1. Handling of trigger recursion depends on the type of trigger:

- Row and Row/Object triggers: Caché *does not* prevent row triggers and row/object triggers from being executed recursively. It is the programmer's responsibility to handle trigger recursion. A runtime <FRAMESTACK> error may occur if the trigger code does not handle recursive execution.
- Statement triggers: Caché prevents an AFTER statement trigger from being executed recursively. Caché will not issue an AFTER trigger if it detects that the trigger has been called previously in the execution stack. No error is issued; the trigger is simply not executed a second time.

  Caché *does not* prevent a BEFORE statement trigger from being executed recursively. It is the programmer's responsibility to handle BEFORE trigger recursion. A runtime <FRAMESTACK> error may occur if the BEFORE trigger code does not handle recursive execution.

18.3 Trigger Code

Each trigger contains one or more lines of code that perform a triggered action. This code is invoked by the SQL Engine whenever the event associated with the trigger occurs. If the trigger is defined using CREATE TRIGGER, this action code can be written in either ObjectScript or SQL. (Caché converts code written in SQL to ObjectScript in the class definition.) If the trigger is defined using Studio, this action code must be written in ObjectScript.

Because the code for a trigger is not generated as a procedure, all local variables in a trigger are public variables. This means all variables in triggers should be explicitly declared with a **NEW** statement; this protects them from conflicting with variables in the code that invokes the trigger.

18.3.1 %ok, %msg, and %oper System Variables

- **%ok**: A variable used only in trigger code. If trigger code succeeds, it sets %ok=1. If trigger code fails, it sets %ok=0. If during trigger execution an SQLCODE error is issued, Caché sets %ok=0. When %ok=0, the trigger code aborts and the trigger operation and the operation that invoked the trigger are rolled back. If INSERT or UPDATE trigger

code fails and there is a foreign key constraint defined for the table, Caché releases the lock on the corresponding row in the foreign key table.

Trigger code can explicitly set %ok=0. This creates a runtime error that aborts execution of the trigger and rolls back the operation. Commonly, before setting %ok=0, trigger code explicitly sets the %msg variable to a user-specified string describing this user-defined trigger code error.

The %ok variable is a public variable which must be explicitly NEWed. %ok is unchanged from its prior value upon the completion of a non-trigger code **SELECT**, **INSERT**, **UPDATE**, or **DELETE** statement. %ok is only defined by the execution of trigger code.

- **%msg**: Trigger code can explicitly set the `%msg` variable to a string describing the cause of the runtime error. SQLCODE errors set the `%msg` variable.
- **%oper**: A variable used only in trigger code. Trigger code can refer to the variable `%oper`, which contains the name of the event that fired the trigger (`INSERT`, `UPDATE`, or `DELETE`).

18.3.2 {fieldname} Syntax

Within trigger code, you can refer to field values (for the fields belonging to the table the trigger is associated with) using a special {*fieldname*} syntax. For example, the following definition of the LogEvent trigger in the MyApp.Person class includes a reference to the ID field, as `{ID}`:

Class Definition

```
Class MyApp.Person Extends %Persistent [DdlAllowed]
{
    // ... Definitions of other class members

    /// This trigger updates the LogTable after every insert
    Trigger LogEvent [ Event = INSERT, Time = AFTER ]
    {
        // get row id of inserted row
        NEW id,SQLCODE,%msg,%ok,%oper
        SET id = {ID}

        // INSERT value into Log table
        &sql(INSERT INTO LogTable
            (TableName, IDValue)
            VALUES ('MyApp.Person', :id))
        IF SQLCODE<0 {SET baderr="SQLCODE ERROR:"_SQLCODE_" "_%msg
                      SET %ok=0
                    RETURN baderr }

      }
   // ... Definitions of other class members

}
```

This {*fieldname*} syntax supports unitary fields. It does not support %SerialObject collection properties. For example, if a table references the embedded serial object class Address, which contains the property City, the trigger syntax `{Address_City}` is a valid reference to a field. The trigger syntax `{Address}` is a reference to a collection property, and cannot be used.

18.3.3 Macros within Trigger Code

Your trigger code can contain a macro definition that references a field name (using {*fieldname*} syntax). However, if your trigger code contains a #include preprocessor directive for a macro that references a field name (using {*fieldname*} syntax), the field name cannot be accessed. This is because Caché translates {*fieldname*} references in the trigger code *before* the code is passed to the macro preprocessor. If a {*fieldname*} reference is in the #include file, it is not “seen” in the trigger code, and is therefore not translated.

The work-around for this situation is to define the macro with an argument, then pass the {*fieldname*} in to the macro in the trigger. For example, the #include file could contain a line such as the following:

ObjectScript

```
#define dtThrowTrigger(%val) SET x=$GET(%val,"?")
```

And then within the trigger invoke the macro supplying the {*fieldname*} syntax as an argument:

ObjectScript

```
  $$$dtThrowTrigger({%%ID})
```

18.3.4 {name*O}, {name*N}, and {name*C} Trigger Code Syntax

Three syntax shortcuts are available in **UPDATE** trigger code.

You can reference the old (pre-update) value using the following syntax:

```
{fieldname*O}
```

where *fieldname* is the name of the field and the character after the asterisk is the letter "O" (for Old). For an **INSERT** trigger, `{fieldname*O}` is always the empty string ("").

You can reference the new (post-update) value using the following syntax:

```
{fieldname*N}
```

where *fieldname* is the name of the field and the character after the asterisk is the letter "N" (for New). This `{fieldname*N}` syntax can be used only to reference a value to be stored; it cannot be used to change the value. You cannot set `{fieldname*N}` in trigger code. Computing the value of a field on **INSERT** or **UPDATE** should be achieved by other means, such as SqlComputeOnChange.

You can test whether a field value has been changed (updated) using the following syntax:

```
{fieldname*C}
```

where *fieldname* is the name of the field and the character after the asterisk is the letter "C" (for Changed). `{fieldname*C}` evaluates to 1 if the field has been changed and 0 if it has not been changed. For an **INSERT** trigger, Caché sets `{fieldname*C}` to 1.

For a class with stream properties, an SQL trigger reference to the stream property `{Stream*N}` and `{Stream*O}` returns the OID for the stream, if the SQL statement (**INSERT** or **UPDATE**) did not insert/update the stream property itself. However, if the SQL statement did insert/update the stream property, `{Stream*O}` remains the OID, but the `{Stream*N}` value is set to one of the following:

- BEFORE trigger returns the value of the stream field in whatever format it was passed to the **UPDATE** or **INSERT**. This could be the literal data value that was entered into the stream property, or the OREF or OID of a temporary stream object.
- AFTER trigger returns the Id of the stream as the `{Stream*N}` value. This is the Id value Caché stored in the ^classnameD global for the stream field. This value is in the appropriate Id format based on the CLASSNAME type parameter for the stream property.

If a stream property is updated using Caché objects, the `{Stream*N}` value is always an OID.

Note: For a trigger for child-tables created by an array collection of serial objects, trigger logic works with object access/save but does not work with SQL access (**INSERT** or **UPDATE**).

18.3.5 Additional Trigger Code Syntax

Trigger code written in ObjectScript can contain the pseudo-field reference variables {%%CLASSNAME}, {%%CLASSNAMEQ}, {%%OPERATION}, {%%TABLENAME}, and {%%ID}. These pseudo-fields are translated into a specific value at class compilation time. For further details, refer to CREATE TRIGGER in the *InterSystems SQL Reference*.

You can use class methods from within trigger code, SQL computed code, and SQL map definitions since class methods do not depend on having an open object. You must use the `##class(classname).Methodname()` syntax to invoke a method from within trigger code. You cannot use the ..Methodname() syntax, because this syntax requires a current open object.

You can pass the value of a field of the current row as an argument of the class method, but the class method itself cannot use field syntax.

18.4 Pulling Triggers

A defined trigger is “pulled” (executed) if the corresponding DML command for that table is invoked.

A Row or Row/Object trigger is pulled for each row that the DML command successfully inserts, updates, or deletes.

A Statement trigger is pulled once for each INSERT, UPDATE, or DELETE statement that successfully executes, regardless of whether the statement actually changes any rows of the table data.

- An INSERT statement pulls the corresponding INSERT trigger. To prevent pulling of this corresponding trigger, specify the %NOTRIGGER keyword.
- An UPDATE statement pulls the corresponding UPDATE trigger. To prevent pulling of this corresponding trigger, specify the %NOTRIGGER keyword.
- An INSERT OR UPDATE statement pulls the corresponding INSERT trigger or UPDATE trigger, depending on the type of DDL operation performed. To prevent pulling of either type of trigger, specify the %NOTRIGGER keyword.
- A DELETE statement pulls the corresponding DELETE trigger. To prevent pulling of this corresponding trigger, specify the %NOTRIGGER keyword.
- A TRUNCATE TABLE statement does not pull a DELETE trigger.

18.5 Triggers and Object Access

If a trigger is defined with `Foreach = row/object`, then the trigger is also called at specific points during object access, depending on the Event and Time keywords of the trigger definition, as follows:

Event	Time	Trigger is also called at this time
INSERT	BEFORE	Just before **%Save()** for a new object
INSERT	AFTER	Just after **%Save()** for a new object
UPDATE	BEFORE	Just before **%Save()** for an existing object
UPDATE	AFTER	Just after **%Save()** for an existing object
DELETE	BEFORE	Just before **%DeleteId()** for an existing object
DELETE	AFTER	Just after **%DeleteId()** for an existing object

As a consequence, it is not necessary to also implement callback methods in order to keep SQL and object behavior synchronized,

For information on Foreach trigger keyword, see the *Caché Class Definition Reference*.

18.5.1 Not Pulling Triggers During Object Access

By default, SQL objects are stored using %Storage.Persistent. Caché also supports %CacheSQLStorage storage.

When saving or deleting objects in a class that uses %CacheSQLStorage, all statement (`Foreach = statement`), row (`Foreach = row`), and row/object (`Foreach = row/object`) triggers are pulled. A trigger defined with no Foreach trigger keyword is a row trigger. Pulling all triggers is the default behavior.

However, when saving or deleting objects in a class using %CacheSQLStorage, you can specify that only triggers defined as `Foreach = row/object` should be pulled. Triggers defined as `Foreach = statement` or `Foreach = row` are not pulled. This done by specifying the class parameter OBJECTSPULLTRIGGERS = 0. The default is OBJECTSPULLTRIGGERS = 1.

This parameter only applies to classes defined as using %CacheSQLStorage.

18.6 Triggers and Transactions

A trigger executes trigger code within a transaction. It sets the transaction level, then executes the trigger code. Upon successful completion of trigger code, the trigger commits the transaction.

Note: A consequence of triggers using transactions is that if a trigger invokes code that commits a transaction, completion of the trigger fails because the transaction level has already been decremented to 0. This situation can occur when invoking an Ensemble Business Service.

With an AFTER INSERT statement level ObjectScript trigger, if the trigger sets %ok=0 the insert of the row fails with an SQLCODE -131 error. Transaction rollback may occur, as follows:

- If AUTO_COMMIT=ON, the transaction for the INSERT will be rolled back.
- If AUTO_COMMIT=OFF, it is up to the application to either rollback or commit the transaction for the INSERT.
- If NO_AUTO_COMMIT mode was used, no transaction was started, so the INSERT cannot be rolled back.

The AUTO_COMMIT mode is established using the SET TRANSACTION %COMMITMODE option, or the **$SYSTEM.SQL.SetAutoCommit()** method.

The trigger can set an error message in the %msg variable in the trigger. This message will be returned to the caller, giving information why the trigger failed.

The %ok and %msg system variables are described in the System Variables section of the "Using Embedded SQL" chapter of this manual.

18.7 Listing Triggers

Triggers defined for a specified table are listed in the Management Portal SQL interface Catalog Details. This lists basic information for each trigger. To list more detailed information, use INFORMATION.SCHEMA.TRIGGERS.

The INFORMATION.SCHEMA.TRIGGERS class lists the defined triggers in the current namespace. For each trigger INFORMATION.SCHEMA.TRIGGERS lists various properties, including the name of the trigger, the associated schema and table name, the EVENTMANIPULATION property (INSERT, UPDATE, DELETE, INSERT/UPDATE, INSERT/UPDATE/DELETE), the ACTIONTIMING property (BEFORE, AFTER), the CREATED property (trigger creation timestamp), and the ACTIONSTATEMENT property, which is the generated SQL trigger code.

The CREATED property derives the trigger creation timestamp from when the class definition was last modified. Therefore, subsequent use of this class (for example, to define other triggers) may result in unintended updating of the CREATED property value.

You can access this INFORMATION.SCHEMA.TRIGGERS information from an SQL query, as shown in the following example:

SQL

```
SELECT
TABLE_NAME,TRIGGER_NAME,CREATED,EVENT_MANIPULATION,ACTION_TIMING,ACTION_ORIENTATION,ACTION_STATEMENT
FROM INFORMATION_SCHEMA.TRIGGERS WHERE TABLE_SCHEMA='Sample'
```

19

Defining and Using Stored Procedures

This chapter describes how to define and use stored procedures in Caché SQL. It discusses the following:

- An overview of the types of stored procedures
- How to define stored procedures
- How to use stored procedures
- How to list stored procedures and their parameters.

19.1 Overview

An SQL routine is an executable unit of code that can be invoked by the SQL query processor. There are two types of SQL routines: functions and stored procedures. Functions are invoked from any SQL statement that supports `functionname()` syntax. Stored procedures can only be invoked by a **CALL** statement. Functions accept some number of input directed arguments and return a single result value. Stored procedures accept some number of input, input-output, and output arguments. A stored procedure can be a user-defined function, returning a single value. A function can also be invoked by a **CALL** statement.

Like most relational database systems, Caché allows you to create SQL stored procedures. A Stored Procedure (SP) provides a callable routine that is stored in the database and can be invoked within an SQL context (for example, by using the **CALL** statement or via ODBC or JDBC).

Unlike relational databases, Caché lets you define stored procedures as methods of classes. In fact, a stored procedure is nothing more than a class method that is made available to SQL. Within a stored procedure, you can use the full range of Caché object-based features.

- You can defined a stored procedure as a query that returns a single result set of data by querying the database.
- You can define a stored procedure as a function procedure that can serve as a user-defined function, returning a single value.
- You can define a stored procedure as a method that can modify the database data and return either a single value or one or more result sets.

You can determine if a procedure already exists using the **$SYSTEM.SQL.ProcedureExists()** method. This method also returns the procedure type: “function” or “query”.

19.2 Defining Stored Procedures

As with most aspects of Caché SQL, there are two ways of defining stored procedures: using DDL and using classes. These are described in the following sections.

19.2.1 Defining a Stored Procedure Using DDL

Caché SQL supports the following commands to create a query:

- CREATE PROCEDURE can create a query that is always projected as a stored procedure. A query can return a single result set.
- CREATE QUERY creates a query that can optionally be projected as a stored procedure. A query can return a single result set.

Caché SQL supports the following commands to create a method or function:

- CREATE PROCEDURE can create a method that is always projected as a stored procedure. A method can return a single value, or one or more result sets.
- CREATE METHOD can create a method that can optionally be projected as a stored procedure. A method can return a single value, or one or more result sets.
- CREATE FUNCTION can create a function procedure that can optionally be projected as a stored procedure. A function can return a single value.

The block of executable code specified within these commands can be written either in Caché SQL or ObjectScript. You can include Embedded SQL within an ObjectScript code block.

19.2.2 SQL to Class Name Transformations

When you use DDL to create a stored procedure, the name you specify is transformed into a class name. If the class does not exist, the system creates it.

- If the name is unqualified and no FOR clause is provided: the system-wide default schema name is used as the package name, followed by a dot, followed by a generated class name consisting of the string ‘func’, ‘meth’, ‘proc’, or ‘query’, followed by the SQL name stripped of punctuation characters. For example, the unqualified procedure name `Store_Name` results in a class name such as the following: `User.procStoreName`. This procedure class contains the method **StoreName()**.
- If the name is qualified and no FOR clause is provided: the name of the schema is converted to a package name, followed by a dot, followed by the string ‘func’, ‘meth’, ‘proc’, or ‘query’, followed by the SQL name stripped of punctuation characters. If necessary, the specified package name is converted to a valid package name.

  If the name is qualified and a FOR clause is provided: the qualified class name specified in the FOR clause overrides the schema name specified in the function, method, procedure, or query name.
- SQL stored procedure names follow identifier naming conventions. Caché strips punctuation characters from the SQL name to generate unique class entity names for the procedure class and its class methods.

The following rules govern the transformation of a schema name to valid package name:

- If the schema name contains an underscore, this character is converted to a dot, denoting a subpackage. For example, the qualified name `myprocs.myname` creates the package `myprocs`. The qualified name `my_procs.myname` creates the package `my` containing the subpackage `procs`.

The following example shows how the punctuation differs in a class name and its SQL invocation. It defines a method with a class name containing two dots. When invoked from SQL, the example replace the first dot with an underscore character:

Class Definition

```
Class tmp.test.sql Extends %RegisteredObject
 {  ClassMethod myfunc(dummy As %String) As %String [ SqlProc ]
    { /* method code */
      Quit "abc" }
 }
```

SQL

```
SELECT tmp_test.sql_myfunc(Name)
FROM Sample.Person
```

19.2.3 Defining a Method Stored Procedure using Classes

Class methods can be exposed as Stored Procedures. These are ideal for actions that do not return data, such as a Stored Procedure that calculates a value and stores it in the database. Almost all classes can expose methods as Stored Procedures; the exception is generator classes, such as a data type class ([ClassType = datatype]). Generator classes do not have a runtime context. It is only valid to use a datatype context within the runtime of some other entity, such as a property.

To define a method stored procedure, simply define a class method and set its SqlProc keyword:

Class Definition

```
Class MyApp.Person Extends %Persistent [DdlAllowed]
{

/// This procedure finds total sales for a territory
ClassMethod FindTotal(territory As %String) As %Integer [SqlProc]
{
    // use embedded sql to find total sales
    &sql(SELECT SUM(SalesAmount) INTO :total
            FROM Sales
            WHERE Territory = :territory
    )

    Quit total
}
}
```

After this class is compiled, the **FindTotal()** method will be projected to SQL as the stored procedure **MyApp.Person_FindTotal()**. You can change the name that SQL uses for the procedure using the SqlName keyword of the method.

The method uses a procedure context handler to pass the procedure context back and forth between the procedure and its caller (for example, the ODBC server). This procedure context handler is automatically generated by Caché (as %qHandle:%SQLProcContext) using the %sqlcontext object.

%sqlcontext consists of properties for the SQLCODE error status, the SQL row count, an error message, and so forth, which are set using the corresponding SQL variables, as follows:

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

There is no need to do anything with these values, but their values will be interpreted by the client. The %sqlcontext object is reset before each execution.

The method should return no value.

The maximum number of user-defined methods for a class is 2000.

For instance, suppose there is a **CalcAvgScore()** method:

```
ClassMethod CalcAvgScore(firstname As %String,lastname As %String) [sqlproc]
{
  New SQLCODE,%ROWID
  &sql(UPDATE students SET avgscore =
    (SELECT AVG(sc.score)
     FROM scores sc, students st
     WHERE sc.student_id=st.student_id
       AND st.lastname=:lastname
       AND st.firstname=:firstname)
     WHERE students.lastname=:lastname
       AND students.firstname=:firstname)

  IF ($GET(%sqlcontext)'= "") {
    SET %sqlcontext.%SQLCODE = SQLCODE
    SET %sqlcontext.%ROWCOUNT = %ROWCOUNT
  }
  QUIT
}
```

19.2.4 Defining a Query Stored Procedure using Classes

Many Stored Procedures that return data from the database can be implemented through the standard query interface. This approach works well as long as the procedure can be written in embedded SQL. Note the use of the Embedded SQL host variable to supply a value to the WHERE clause in the following example:

Class Definition

```
Class MyApp.Person Extends %Persistent [DdlAllowed]
{

    /// This procedure result set is the persons in a specified Home_State, ordered by Name
    Query ListPersons(state As %String = "") As %SQLQuery [ SqlProc ]
    {
        SELECT ID,Name,Home_State
        FROM Sample.Person
        WHERE Home_State = :state
        ORDER BY Name
    }
}
```

To expose a query as a Stored Procedure, either change the value of the SQLProc field to True in the Studio Inspector's entry for the query or add the following "[SqlProc]" string to the query definition:

```
Query QueryName() As %SQLQuery( ... query definition ... )
    [ SqlProc ]
```

After this class is compiled, the **ListPersons** query will be projected to SQL as the stored procedure **MyApp.Person_ListPersons**. You can change the name that SQL uses for the procedure using the SqlName keyword of the query.

When **MyApp.Person_ListPersons** is called from SQL, it will automatically return the result set defined by the query's SQL statement.

The following example is a stored procedure using a result set:

Class Definition

```
Class apc.OpiLLS.SpCollectResults1 [ Abstract ]
{

/// This SP returns a number of rows (pNumRecs) from WebService.LLSResults, and updates a property for
 each record
Query MyQuery(pNumRecs As %Integer) As %Query(ROWSPEC = "Name:%String,DOB:%Date") [ SqlProc ]
{
}

/// You put initial code here in the Execute method
ClassMethod MyQueryExecute(ByRef qHandle As %Binary, pNumRecs As %Integer) As %Status
{
```

```
    SET mysql="SELECT TOP ? Name,DOB FROM Sample.Person"
    SET rset=##class(%SQL.Statement).%ExecDirect(,mysql,pNumRecs)
        IF rset.%SQLCODE'=0 {QUIT rset.%SQLCODE}
    SET qHandle=rset
    QUIT $$$OK
}

/// This code is called by the SQL framework for each row, until no more rows are returned
ClassMethod MyQueryFetch(ByRef qHandle As %Binary, ByRef Row As %List,
                         ByRef AtEnd As %Integer = 0) As %Status [ PlaceAfter = NewQuery1Execute ]
{
     SET rset=qHandle
     SET tSC=$$$OK

     FOR {
        ///Get next row, quit if end of result set
        IF 'rset.%Next() {
                SET Row = "", AtEnd = 1
                SET tSC=$$$OK
                QUIT
                }
        SET name=rset.Name
        SET dob=rset.DOB
        SET Row = $LISTBUILD(name,dob)
        QUIT
        }
        QUIT tSC
}

ClassMethod MyQueryClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = NewQuery1Execute ]
{
        KILL qHandle   //probably not necesary as killed by the SQL Call framework
        QUIT $$$OK
}

}
```

If it is possible to write the query as a simple SQL statement and create it through the Query Wizard, it is not necessary to know anything about the underlying methods that implement the query.

Behind the scenes, for each query the class compiler generates methods based on the name of the Stored Procedure, including:

- *stored-procedure-name*Execute()
- *stored-procedure-name*Fetch()
- *stored-procedure-name*FetchRows()
- *stored-procedure-name*GetInfo()
- *stored-procedure-name*Close()

If the query is of type %SQLQuery, the class compiler automatically inserts some embedded SQL into the generated methods. **Execute()** declares and opens a stored cursor for the SQL. **Fetch()** is called repeatedly until it returns an empty row (SET Row=""). You can, optionally, also have **Fetch()** return an AtEnd=1 boolean flag to indicate that the current Fetch constitutes the last row and the next Fetch is expected to return an empty row. However, an empty row (Row="") should always be used as the test to determine when the result set has ended; Row="" should always be set when setting AtEnd=1.

FetchRows() is logically equivalent to repeated calls to **Fetch()**. **GetInfo()** is called to return details of the signature for the Stored Procedure. **Close()** closes the cursor.

All these methods are called automatically when a Stored Procedure is invoked from a client, but could in theory be called directly from ObjectScript running on the server.

To pass an object from the **Execute()** to a **Fetch()**, or from a **Fetch()** to the next invocation of **Fetch()**, you can set the query handler to the object reference (oref) of the object you wish to pass. To pass multiple objects, you can set qHandle as an array:

ObjectScript

```
SET qHandle(1)=oref1,qHandle(2)=oref2
```

It is possible to create a result set stored procedure that is based on custom-written code (not an SQL statement).

The maximum number of user-defined queries for a class is 200.

19.2.5 Customized Class Queries

For complex queries, or for Stored Procedures that do not fit the query model, it is often necessary to customize the query by replacing some or all of its methods. You can use %Library.Query, as described in this section.

It is often easier to implement the query if you choose type %Query (%Library.Query) instead of %SQLQuery (%Library.SQLQuery). This generate the same five methods, but now the **FetchRows()** is simply a repeated invocation of **Fetch()** (%SQLQuery has some optimization that causes other behavior). **GetInfo()** simply gets information from the signature, so it is very unlikely that the code will need to be changed. This reduces the problem to creating class methods for each of the other three. Note that when the class is compiled, the compiler detects the presence of these methods, and does not overwrite them.

The methods need specific signatures: They all take a Qhandle (query handler) of type %Binary. This is a pointer to a structure holding the nature and state of the query. This is passed by reference to **Execute()** and **Fetch()** and by value to **Close()**:

```
ClassMethod SP1Close(qHandle As %Binary) As %Status
{
   // ...
}

ClassMethod SP1Execute(ByRef qHandle As %Binary,
    p1 As %String) As %Status
{
   // ...
}

ClassMethod SP1Fetch(ByRef qHandle As %Binary,
    ByRef Row As %List, ByRef AtEnd As %Integer=0) As %Status
{
   // ...
}

Query SP1(p1 As %String)
   As %Query(CONTAINID=0,ROWSPEC="lastname:%String") [sqlproc ]
{
}
```

The code usually includes declaration and use of an SQL cursor. Cursors generated from queries of type %SQLQuery automatically have names such as Q14. You must ensure that your queries are given distinct names.

The class compiler must find a cursor declaration, before making any attempt to use the cursor. Therefore the DECLARE statement (usually in Execute) must be in the same MAC routine as the Close and Fetch and must come before either of them. Editing the source directly, use the method keyword PLACEAFTER in both the Close and the Fetch definitions to make sure this happens.

Error messages refer to the internal cursor name, which typically has an extra digit. Therefore an error message for cursor Q140 probably refers to Q14.

19.3 Using Stored Procedures

You can use stored procedures in two distinct ways:

- You can invoke a stored procedure using the SQL **CALL** statement; see the CALL statement in the *Caché SQL Reference* for more details.
- You can use a stored function (that is, a method-based stored procedure that returns a single value) as if it were a built-in function within an SQL query.

Note: When executing a stored procedure that takes an SQL function as a argument, invoke the stored procedure using CALL, as in the following example:

SQL

```
CALL sp.MyProc(CURRENT_DATE)
```

A **SELECT** query does not support executing a stored procedure with an SQL function argument. **SELECT** does support executing a stored function with an SQL function argument.

xDBC does not support executing a stored procedure with an SQL function argument using either **SELECT** or **CALL**.

19.3.1 Stored Functions

A stored function is a method-based stored procedure that returns a single value. For example, the following class defines a stored function, **Square**, that returns the square of a given value:

Class Definition

```
Class MyApp.Utils Extends %Persistent [DdlAllowed]
{
ClassMethod Square(val As %Integer) As %Integer [SqlProc]
{
    Quit val * val
}
}
```

A stored function is simply a class method with the SqlProc keyword specified.

Note: For a stored function, the ReturnResultsets keyword must either be not specified (the default) or prefaced by the keyword `Not`.

You can use a stored function within an SQL query as if it were a built-in SQL function. The name of the function is the SQL name of the stored function (in this case "Square") qualified by the schema (package) name in which it was defined (in this case "MyApp").

The following query uses the **Square** function:

SQL

```
SELECT Cost, MyApp.Utils_Square(Cost) As SquareCost FROM Products
```

If you define multiple stored functions within the same package (schema), you must make sure that they have unique SQL names.

The following example defines a table named Sample.Wages that has two defined data fields (properties) and two defined stored functions, TimePlus and DTime:

Class Definition

```
Class Sample.Wages Extends %Persistent [ DdlAllowed ]
{
  Property Name As %String(MAXLEN = 50) [ Required ];
  Property Salary As %Integer;
  ClassMethod TimePlus(val As %Integer) As %Integer [ SqlProc ]
  {
   QUIT val * 1.5
  }
  ClassMethod DTime(val As %Integer) As %Integer [ SqlProc ]
  {
   QUIT val * 2
  }
}
```

The following query uses these stored procedures to return the regular salary, time-and-a-half, and double time salary rates for each employee in the same table, Sample.Wages:

SQL

```
SELECT Name,Salary,
       Sample.Wages_TimePlus(Salary) AS Overtime,
       Sample.Wages_DTime(Salary) AS DoubleTime FROM Sample.Wages
```

The following query uses these stored procedures to return the regular salary, time-and-a-half, and double time salary rates for each employee in a different table, Sample.Employee:

SQL

```
SELECT Name,Salary,
       Sample.Wages_TimePlus(Salary) AS Overtime,
       Sample.Wages_DTime(Salary) AS DoubleTime FROM Sample.Employee
```

19.3.2 Privileges

To execute a procedure, a user must have EXECUTE privilege for that procedure. Use the GRANT command or the %SYSTEM.SQL **GrantObjPriv()** method to assign EXECUTE privilege for a specified procedure to a specified user.

You can determine if a specified user has EXECUTE privilege for a specified procedure by invoking the **$SYSTEM.SQL.CheckPriv()** method.

To list all the procedures for which a user has EXECUTE privilege, go to the Management Portal. From **System Administration** select **Security**, then select either **Users** (**[System] > [Security Management] > [Users]**) or **Roles** (**[System] > [Security Management] > [Roles]**). Select **Edit** for the desired user or role, then select the **SQL Procedures** tab. Select the desired **Namespace** from the drop-down list.

19.4 Listing Procedures

The INFORMATION.SCHEMA.ROUTINES persistent class displays information about all routines and procedures in the current namespace.

When specified in Embedded SQL, INFORMATION.SCHEMA.ROUTINES requires the **#include %occInclude** macro preprocessor directive. This directive is not required for Dynamic SQL.

The following example returns the routine name, method or query name, routine type (PROCEDURE or FUNCTION), routine body (SQL=class query with SQL, EXTERNAL=not a class query with SQL), the return data type, and the routine definition for all routines in the schema “Sample” in the current namespace:

SQL

```
SELECT ROUTINE_NAME,METHOD_OR_QUERY_NAME,ROUTINE_TYPE,ROUTINE_BODY,SQL_DATA_ACCESS,IS_USER_DEFINED_CAST,
DATA_TYPE||' '||CHARACTER_MAXIMUM_LENGTH AS Returns,NUMERIC_PRECISION||':'||NUMERIC_SCALE AS
PrecisionScale,
ROUTINE_DEFINITION
FROM INFORMATION_SCHEMA.ROUTINES WHERE ROUTINE_SCHEMA='Sample'
```

The INFORMATION.SCHEMA.PARAMETERS persistent class displays information about input and output parameters for all routines and procedures in the current namespace.

The following example returns the routine name, parameter name, whether it is an input or output parameter, and the parameter data type information for all routines in the schema “Sample” in the current namespace:

SQL

```
SELECT SPECIFIC_NAME,PARAMETER_NAME,PARAMETER_MODE,ORDINAL_POSITION,
DATA_TYPE,CHARACTER_MAXIMUM_LENGTH AS MaxLen,NUMERIC_PRECISION||':'||NUMERIC_SCALE AS PrecisionScale
FROM INFORMATION_SCHEMA.PARAMETERS WHERE SPECIFIC_SCHEMA='Sample'
```

You can display much of the same information for a single procedure using the Catalog Details tab in the Management Portal SQL Interface. The Catalog Details for a procedure include the procedure type (query or function), class name, method or query name, the description, and the number of input and output parameters. The Catalog Details Stored Procedure Info display also provides an option to run the stored procedure.

20

Storing and Using Stream Data (BLOBs and CLOBs)

InterSystems SQL supports the ability to store stream data as either BLOBs (Binary Large Objects) or CLOBs (Character Large Objects) within an Caché database. This chapter discusses the following topics:

- Defining stream data fields
- Inserting data into stream data fields
- Querying stream field data
- DISTINCT, GROUP BY and ORDER BY clauses
- Predicate conditions and streams
- Aggregate functions and streams
- Scalar functions and streams
- Stream field concurrency locking
- Using stream fields within Caché methods
- Using stream fields from ODBC
- Using stream fields from JDBC

20.1 Stream Fields and SQL

InterSystems SQL supports two kinds of stream fields:

- Character streams, used for large quantities of text.
- Binary streams, used for images, audio, or video.

20.1.1 BLOBs and CLOBs

InterSystems SQL supports the ability to store BLOBs (Binary Large Objects) and CLOBs (Character Large Objects) within the database as stream objects. BLOBs are used to store binary information, such as images, while CLOBs are used to store

character information. BLOBs and CLOBs can store up to 4 Gigabytes of data (the limit imposed by the JDBC and ODBC specifications).

The operation of the BLOBs and CLOBs is identical in every respect except how they handle character encoding conversion (such as Unicode to multibyte) when accessed via an ODBC or JDBC client: the data in a BLOB is treated as binary data and is never converted to another encoding while the data in a CLOB is treated as character data and is converted as necessary.

If a binary stream file (BLOB) contains the single non-printing character $CHAR(0), it is considered to be an empty binary stream. It is equivalent to the "" empty binary stream value: it exists (is not null), but has a length of 0.

From the object point of view, BLOBs and CLOBs are represented as stream objects. For more information, see the chapter "Working with Streams" of *Defining and Using Classes*.

20.1.2 Defining Stream Data Fields

InterSystems SQL supports a variety of data type names for stream fields. These InterSystems data type names are synonyms that correspond to the following:

- Character streams: data type LONGVARCHAR, which maps to the %Stream.GlobalCharacter class and the ODBC/JDBC data type -1.
- Character streams: data type LONGVARBINARY, which maps to the %Stream.GlobalBinary class and the ODBC/JDBC data type -4.

Some InterSystems stream data types allow you to specify a data precision value. This value is a no-op and has no effect on the permitted size of the stream data. It is provided to allow the user to document the anticipated size of future data.

For data type mappings of stream data types, refer to the Data Types reference page in *InterSystems SQL Reference*.

For how to define fields (properties) of a table (persistent class), refer to Defining a Table by Creating a Persistent Class and Defining a Table by Using DDL. When defining a stream property of a persistent class, you can optionally specify the LOCATION parameter; see Declaring Stream Properties in the "Working with Streams" chapter of *Defining and Using Classes*.

The following example defines a table containing two stream fields:

SQL

```
CREATE TABLE Sample.MyTable (
    Name VARCHAR(50) NOT NULL,
    Notes LONGVARCHAR,
    Photo LONGVARBINARY)
```

20.1.2.1 Stream Field Constraints

The definition of a stream field is subject to the following field data constraints:

A stream field can be defined as NOT NULL.

A stream field can take a DEFAULT value, an ON UPDATE value, or a COMPUTECODE value.

A stream field cannot be defined as UNIQUE, a primary key field, or an IdKey. Attempting to do so results in an SQLCODE -400 fatal error with a %msg such as the following: `ERROR #5414: Invalid index attribute: Sample.MyTable::MYTABLEUNIQUE2::Notes, Stream property is not allowed in a unique/primary key/idkey index > ERROR #5030: An error occurred while compiling class 'Sample.MyTable'`.

A stream field cannot be defined with a specified COLLATE value. Attempting to do so results in an SQLCODE -400 fatal error with a %msg such as the following: `ERROR #5480: Property parameter not declared: Sample.MyTable:Photo:COLLATION > ERROR #5030: An error occurred while compiling class 'Sample.MyTable'`.

20.1.3 Inserting Data into Stream Data Fields

There are three ways to INSERT data into stream fields:

- %Stream.GlobalCharacter fields: you can insert character stream data directly. For example,

  SQL

  ```
  INSERT INTO Sample.MyTable (Name,Notes)
      VALUES ('Fred','These are extensive notes about Fred')
  ```

- %Stream.GlobalCharacter and %Stream.GlobalBinary fields: you can insert stream data using an OREF. You can use the **Write()** method to append a string to the character stream, or the **WriteLine()** method to append a string with a line terminator to the character stream. By default, the line terminator is $CHAR(13,10) (carriage return/line feed); you can change the line terminator by setting the LineTerminator property. In the following example, the first part of the example creates a character stream consisting of two strings and their terminators, then inserts it into a stream field using Embedded SQL. The second part of the example returns the character stream length and displays the character stream data showing the terminators:

  ObjectScript

  ```
  CreateAndInsertCharacterStream
    SET gcoref=##class(%Stream.GlobalCharacter).%New()
    DO gcoref.WriteLine("First Line")
    DO gcoref.WriteLine("Second Line")
    &sql(INSERT INTO Sample.MyTable (Name,Notes)
      VALUES ('Fred',:gcoref))
  DisplayTheCharacterStream
    KILL ^CacheStream
    WRITE gcoref.%Save(),!
    ZWRITE ^CacheStream
  ```

- %Stream.GlobalCharacter and %Stream.GlobalBinary fields: you can insert stream data by reading it from a file. For example,

  ObjectScript

  ```
    SET myf="C:\InterSystems\Cache\mgr\temp\IMG_0190.JPG"
    OPEN myf:("RF"):10
    USE myf:0
    READ x(1):10
    &sql(INSERT INTO Sample.MyTable (Name,Photo) VALUES ('George',:x(1)))
    CLOSE myf
  ```

  For further details, refer to Sequential File I/O in *I/O Device Guide*.

String data that is inserted as a DEFAULT value or a computed value is stored in the format appropriate for the stream field.

20.1.4 Querying Stream Field Data

A query *select-item* that selects a stream field returns the fully formed OID (object ID) value of the stream object, as shown in the following example:

```
SELECT Name,Photo,Notes
FROM Sample.MyTable WHERE Photo IS NOT NULL
```

An OID is a %List formatted data address such as the following:
`$lb("1","%Stream.GlobalCharacter","^Sample.MyTableS")`.

- The first element of the OID is a sequential positive integer (starting with 1) that is assigned to each inserted stream data value in a table. For example, if Row 1 is inserted with values for the stream fields Photo and Notes, these are

assigned 1 and 2. If Row 2 is inserted with a value for Notes, that is assigned 3. If Row 3 is inserted with a value for Photo and Notes, those are assigned 4 and 5. The assignment sequence is the order that the fields are listed in the table definition, not the order they are specified in the **INSERT** command. By default, a single integer sequence is used which corresponds to the stream location global counter. However, a table may have multiple stream counters, as described below.

An **UPDATE** operation does not change the initial integer value. A **DELETE** operation may create gaps in the integer sequence, but does not change these integer values. Using **DELETE** to delete all records does not reset this integer counter. Using TRUNCATE TABLE to delete all records resets this integer counter if all of the table stream fields use the default StreamLocation value. **TRUNCATE TABLE** cannot be used to reset the stream integer counter for a embedded object (%SerialObject) class.

- The second element of the OID is the stream data type, either %Stream.GlobalCharacter or %Stream.GlobalBinary.
- The third element of the OID is a global specifying the package name and persistent class name that correspond to the table, with the letter S appended: `^Sample.MyTableS`. If the table was created as a persistent class, this is the StreamLocation storage keyword `<StreamLocation>^Sample.MyTableS</StreamLocation>` value.

  The default stream location is a global such as `^Sample.MyTableS`. This global is used to count the inserts to all stream properties (fields) that do not have a custom LOCATION. For example, if all stream properties in Sample.MyTable use the default stream location, when ten stream data values have been inserted into stream properties of Sample.MyTable, the `^Sample.MyTableS` global contains the value 10. This global contains the most recently assigned value of the stream data inserts counter. If no stream field data has been inserted, or **TRUNCATE TABLE** has been used to delete all table data, this global is undefined.

  When defining a persistent class stream field property, you can define a custom LOCATION, such as the following: `Property Note2 As %Stream.GlobalCharacter (LOCATION="^MyCustomGlobalS");`. In this situation, the `^MyCustomGlobalS` global serves as the stream data inserts counter for the stream property (or properties) that specify this LOCATION; stream properties that do not specify a LOCATION use the default stream location global (`^Sample.MyTableS`) as the stream data inserts counter. Each global counts the inserts for the stream properties associated with that location. If no stream field data has been inserted the location global is undefined. **TRUNCATE TABLE** does not reset stream counters if one or more stream properties defined a LOCATION.

  Subscripts of these stream location global variables contain the data for each stream field. For example, `^Sample.MyTableS(3)` represents the third inserted stream data item. `^Sample.MyTableS(3,0)` is the length of the data. `^Sample.MyTableS(3,1)` is the actual stream data value.

Note: The OID for a stream field is not the same as the OID returned for a RowID or a reference field. The %OID function returns the OID for a RowID or a reference field; **%OID** cannot be used with a stream field. Attempting to use a stream field as an argument to **%OID** results in an SQLCODE -37 error.

Use of a stream field in the WHERE clause or HAVING clause of a query is highly restricted. You cannot use an equality condition or other relational operator (=, !=, <, >), or a Contains operator ( ]) or Follows operator ([ ) with a stream field. Attempting to use these operators with a stream field results in an SQLCODE -313 error. Refer to Predicate Conditions and Streams for valid predicates using a stream field.

20.1.4.1 Result Set Display

- Dynamic SQL executed either from a program or from the SQL Shell returns the OID in the format `$lb("6","%Stream.GlobalCharacter","^Sample.MyTableS")`.
- Dynamic SQL executed from the Management Portal SQL Execute interface returns the OID in the format `6%Stream.GlobalCharacter^Sample.MyTableS`.
- Embedded SQL returns the same OID, but as an encoded %List. You can use the $LISTTOSTRING function to display the OID as a string with its elements separated by commas: `6,%Stream.GlobalBinary,^Sample.MyTableS`.

20.1.5 DISTINCT, GROUP BY, and ORDER BY

Every stream data field OID value is unique, even when the data itself contains duplicates. These **SELECT** clauses operate on the stream OID value, not the data value. Therefore, when applied to a stream field in a query:

- A DISTINCT clause has no effect on duplicate stream data values. A **DISTINCT** clause reduces the number records where the stream field is NULL to one NULL record.
- A GROUP BY clause has no effect on duplicate stream data values. A **GROUP BY** clause reduces the number records where the stream field is NULL to one NULL record.
- An ORDER BY clause orders stream data values by their OID value, not their data value. An **ORDER BY** clause lists records where the stream field is NULL before listing records with a stream field data value.

20.1.6 Predicate Conditions and Streams

The IS [NOT] NULL predicate can be applied to the data value of a stream field, as shown in the following example:

```
SELECT Name,Notes
FROM Sample.MyTable WHERE Notes IS NOT NULL
```

The BETWEEN, EXISTS, IN, %INLIST, LIKE, %MATCHES, and %PATTERN predicates can be applied to the OID value of the stream object, as shown in the following example:

```
SELECT Name,Notes
FROM Sample.MyTable WHERE Notes %MATCHES '*1[0-9]*GlobalChar*'
```

Attempting to use any other predicate condition on a stream field results in an SQLCODE -313 error.

20.1.7 Aggregate Functions and Streams

The COUNT aggregate function takes a stream field and counts the rows containing non-null values for the field, as shown in the following example:

```
SELECT COUNT(Photo) AS PicRows,COUNT(Notes) AS NoteRows
FROM Sample.MyTable
```

However, **COUNT(DISTINCT)** is not supported for stream fields.

No other aggregate functions are supported for stream fields. Attempting to use a stream field with any other aggregate function results in an SQLCODE -37 error.

20.1.8 Scalar Functions and Streams

InterSystems SQL cannot apply any function to a stream field, except the **%OBJECT**, **CHARACTER_LENGTH** (or **CHAR_LENGTH** or **DATALENGTH**), **SUBSTRING**, **CONVERT**, **XMLCONCAT**, **XMLELEMENT**, **XMLFOREST**, and **%INTERNAL** functions. Attempting to use a stream field as an argument to any other SQL function results in an SQLCODE -37 error.

- The %OBJECT function opens a stream object (takes an OID) and returns the oref (object reference), as shown in the following example:

  ```
  SELECT Name,Notes,%OBJECT(Notes) AS NotesOref
  FROM Sample.MyTable WHERE Notes IS NOT NULL
  ```

- The CHARACTER_LENGTH, CHAR_LENGTH, and DATALENGTH functions take a stream field and return the actual data length, as shown in the following example:

```
SELECT Name,DATALENGTH(Notes) AS NotesNumChars
FROM Sample.MyTable WHERE Notes IS NOT NULL
```

- The SUBSTRING function takes a stream field and returns the specified substring of the stream field's actual data value, as shown in the following example:

```
SELECT Name,SUBSTRING(Notes,1,10) AS Notes1st10Chars
FROM Sample.MyTable WHERE Notes IS NOT NULL
```

- The CONVERT function can be used to convert a stream data type to VARCHAR, as shown in the following example:

```
SELECT Name,CONVERT(VARCHAR(100),Notes) AS NotesTextAsStr
FROM Sample.MyTable WHERE Notes IS NOT NULL
```

  `CONVERT(datatype,expression)` syntax supports stream data conversion. If the VARCHAR precision is less than the length of the actual stream data, it truncates the returned value to the VARCHAR precision. If the VARCHAR precision is greater than the length of the actual stream data, the returned value has the length of the actual stream data. No padding is performed.

  `{fn CONVERT(expression,datatype)}` syntax does not support stream data conversion; it issues an SQLCODE -37 error.

- The %INTERNAL function can be used on a stream field, but performs no operation.

20.2 Stream Field Concurrency Locking

Caché protects stream data values from concurrent operations by another process by taking out a lock on the stream data.

Caché takes out an exclusive lock before performing a write operation. The exclusive lock is released immediately after the write operation completes.

Caché takes out a shared lock out when the first read operation occurs. A shared lock is only acquired if the stream is actually read, and is released immediately after the entire stream has been read from disk into the internal temporary input buffer.

20.3 Using Stream Fields within Caché Methods

You cannot use a BLOB or CLOB value using Embedded SQL or Dynamic SQL directly within a Caché method; instead you use SQL to find the stream identifier for a BLOB or CLOB and then create an instance of the %AbstractStream object to access the data.

20.4 Using Stream Fields from ODBC

The ODBC specification does not provide for any recognition or special handling for BLOB and CLOB fields. InterSystems SQL represents CLOB fields within ODBC as having type LONGVARCHAR (-1). BLOB fields are represented as having

type LONGVARBINARY (-4). For ODBC/JDBC data type mappings of stream data types, refer to Integer Codes for Data Types in the Data Types reference page in *InterSystems SQL Reference*.

The ODBC driver/server uses a special protocol to access BLOB and CLOB fields. Typically you have to write special code within ODBC application to use CLOB and BLOB fields; the standard reporting tools typically do not support them.

20.5 Using Stream Fields from JDBC

Within a Java program you can retrieve or set data from a BLOB or CLOB using the standard JDBC BLOB and CLOB interfaces. For example:

```
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT MyCLOB,MyBLOB FROM MyTable");
rs.next();      // fetch the Blob/Clob

java.sql.Clob clob = rs.getClob(1);
java.sql.Blob blob = rs.getBlob(2);

// Length
System.out.println("Clob length = " + clob.length());
System.out.println("Blob length = " + blob.length());

// ...
```

Note: When finished with a BLOB or CLOB, you must explicitly call the **free()** method to close the object in Java and send a message to the server to release stream resources (objects and locks). Just letting the Java object go out of scope does not send a message to clean up the server resources.

21

Users, Roles, and Privileges

Caché has both system-level security, and an additional set of SQL-related security features. Caché security provides an additional level of security capabilities beyond its database-level protections. Some of the key differences between SQL and system-level security are:

- SQL protections are more granular than system-level protections. You can define privileges for tables, views, and stored procedures.
- SQL privileges can be granted to users as well as to roles. System-level privileges are only assigned to roles.
- Holding an SQL privilege implicitly grants any related system privileges that are required to perform the SQL action. (Conversely, system-level privileges do not imply table-level privileges.) The different types of privileges are described in the "SQL Privileges and System Privileges" section.

InterSystems SQL enforces privilege checking for ODBC, JDBC, Dynamic SQL, and the Caché SQL Shell interface. Embedded SQL statements do not perform privilege checking; it is assumed that applications using Embedded SQL will check for privileges before using Embedded SQL statements.

This chapter discusses the following topics:

- SQL Privileges and System-level Privileges
- Users
- Roles
- SQL Privileges

21.1 SQL Privileges and System Privileges

To manipulate tables or other SQL entities through SQL-specific mechanisms, a user must have the appropriate SQL privileges. System-level privileges are not sufficient. A user may be granted SQL privileges directly, or the user may belong to a role that has SQL privileges.

Note: Roles are shared by SQL and system level security: a single role can include both system and SQL privileges.

Consider the following example for an instance of Caché on a Windows machine:

- There is a persistent class in the USER namespace called User.MyPerson. This class is projected to SQL as the SQLUser.MyPerson table.

- There is a user called Test, who belongs to no roles (and therefore has no system privileges) and who has all privileges on the SQLUser.MyPerson table (and no other SQL privileges).
- There is a second user, called Test2. This user is assigned to the following roles: **%DB_USER** (and so can read or write data on the USER database); **%SQL** (and so has SQL access through the %Service_Bindings service); and, through a custom role, has privileges for using the Console and **%Development**.

If the Test user attempts to read or write data in the SQLUser.MyPerson table through any SQL-specific mechanism (such as one that uses ODBC), the attempt succeeds. This is because Caché makes the Test user a member of the **%SQL** role (which includes the **%Service_SQL:Use** privilege) and the **%DB_USER** role, so the user has the necessary privileges to establish the connection; this is visible in audit events that the connection generates, such as the %System/%Login/Login event. (If the Test user attempts to use Terminal object mechanisms, these attempts fail, because the user lacks sufficient privilege for these.)

If the Test2 user attempts to read or write data in the SQLUser.MyPerson table through any SQL-specific mechanism (such as one that uses ODBC), the attempt fails because the user does not have sufficient privileges for the table. (If the Test2 user attempts to view the same data in the Terminal using object mechanisms, the attempt succeeds — because the user is sufficiently privileged for this type of connection.)

21.2 Users

An InterSystems SQL user is the same as a user defined for InterSystems security. You can define a user using either SQL commands or the Management Portal.

- In SQL you use the CREATE USER statement to create a user. This simply creates a user name and user password. The newly created user has no roles. You must use the GRANT statement to assign privileges and roles to the user. You can use the ALTER USER and DROP USER statements to modify existing user definitions.
- In the Management Portal Select **System Administration** select **Security**, then select **Users**. Click the **Create New User** button at the top of the page. This takes you to the **Edit User** page where you can specify the user name, user password, and other parameters. Once you create a user, the other tabs become available, where you can specify which roles a user holds, which general SQL privileges the user holds, which table-level privileges the user holds, which views are available, and which stored procedures can be executed.

If a user has SQL table privileges, or general SQL privileges, then roles granted or revoked on the user's **Roles** tab do not affect a user's access to tables through SQL-based services, such as ODBC. This is because, in the SQL-based services, table-based privileges take precedence over resource-based privileges.

You can use %Library.SQLCatalogPriv class queries to list:

- All users SQLUsers()
- All privileges granted to a specified user SQLUserPrivs("username")
- All system privileges granted to a specified user SQLUserSysPrivs("username")
- All roles granted to a specified user SQLUserRole("username")

The following example lists the privileges granted to the current user:

ObjectScript

```
SET statemt=##class(%SQL.Statement).%New()
SET cqStatus=statemt.%PrepareClassQuery("%Library.SQLCatalogPriv","SQLUserPrivs")
  IF cqStatus'=1 {WRITE "%PrepareClassQuery failed:" DO $System.Status.DisplayError(cqStatus) QUIT}

SET rset=statemt.%Execute($USERNAME)
WRITE "Privileges for ",$USERNAME
DO rset.%Display()
```

21.2.1 User Name as Schema Name

Under some circumstances, a username can be implicitly used as an SQL schema name. This may pose problems if the username contains characters that are forbidden in an SQL identifier. For example, in a multiple domain configuration the username contains the "@" character.

Caché handles this situation differently depending on the setting of the *Delimited Identifiers* configuration parameter:

- If the use of delimited identifiers is enabled, no special processing occurs.
- If the use of delimited identifiers is disabled, then any forbidden characters are removed from the username to form a schema name. For example, the username "documentation@intersystems.com" would become the schema name "documentationintersystemscom".

This does not affect the value returned by the SQL **CURRENT_USER** function. It is always the same as *$USERNAME*.

21.3 Roles

SQL privileges are assigned to a user or role. A role enables you to set the same privileges for multiple users. Roles are shared by SQL and system level security: a single role can include both system privileges and SQL privileges.

The Management Portal, **System Administration**, **Security**, **Roles** page provides a list of role definitions for a Caché instance. To view or change details on a particular role, select the **Name** link for the role. On the **Edit Role** page that appears, there is information regarding the roles privileges, and which users or roles hold it.

The **General** tab lists a role's privileges for InterSystems security resources. If a role only holds SQL privileges, the General tab's Resources table lists the role's privileges as "None defined."

The **SQL Privileges** tab lists a role's privileges for InterSystems SQL resources, where a drop-down list of namespaces allows you to view each namespace's resources. Because privileges are listed by namespace, the listing for a role holding no privileges in a particular namespace displays "None."

Note: You should define privileges using roles and associate specific users with these roles. There are two reasons for this:

1. It is much more efficient for the SQL Engine to determine privilege levels by checking a relatively small role database than by checking individual user entries.
2. It is much easier to administer a system using a small set of roles as compared with a system with many individual user settings.

For example, you can define a role called "ACCOUNTING" with certain access privileges. As the Accounting Department grows, you can define new users and associate them with the ACCOUNTING role. If you need to modify the privileges for ACCOUNTING, you can do it once and it will automatically cover all the members of the Accounting Department.

A role can hold other roles. For example, the ACCOUNTING role can hold the BILLINGCLERK role. A user granted the ACCOUNTING role would have the privileges of both the ACCOUNTING role and the BILLINGCLERK role.

You can also define users and roles with the following SQL commands: CREATE USER, CREATE ROLE, ALTER USER, GRANT, DROP USER, and DROP ROLE.

You can use %Library.SQLCatalogPriv class queries to list:

- All roles SQLRoles()
- All privileges granted to a specified role SQLRolePrivileges("rolename")
- All roles or users granted to a specified role SQLRoleUser("rolename")
- All roles granted to a specified user SQLUserRole("username")

21.4 SQL Privileges

SQL privileges are assigned to a user or role. A role enables you to set the same privileges for multiple users.

InterSystems SQL supports two types of privileges: administrative and object.

- Administrative privileges are namespace-specific.

  Administrative privileges cover the creation, altering, and deleting of types of objects, such as the %CREATE_TABLE privilege required to create tables. The %ALTER_TABLE privilege is required not only to alter a table, but to create or drop an index and to create or drop a trigger.

  Administrative privileges also include %NOCHECK, %NOINDEX, %NOLOCK, and %NOTRIGGER, which determine whether the user can apply the corresponding keyword restrictions when performing an **INSERT**, **UPDATE**, **INSERT OR UPDATE**, or **DELETE**. Assigning the %NOTRIGGER administrative privilege is required for a user to perform a **TRUNCATE TABLE**.

- Object privileges are specific to a table, view, or stored procedure. They specify the type of access to specific named SQL objects (in the SQL sense of the word: a table, a view, a column, or a stored procedure). If the user is the Owner (creator) of the SQL object, the user is automatically granted all privileges for that object.

  Table-level object privileges provide access (%ALTER, DELETE, SELECT, INSERT, UPDATE, EXECUTE, REFERENCES) to the data in all columns of a table or view, both those columns that currently exist and any subsequently added columns.

  Column-level object privileges provide access to the data in only the specified columns of a table or view. You do not need to assign column-level privileges for columns with system-defined values, such as RowID and Identity.

  Stored procedure object privileges permit the assignment of EXECUTE privilege for the procedure to specified users or roles.

For further details, refer to the GRANT command.

21.4.1 Granting SQL Privileges

You can grant privileges in the following ways:

- Use the Management Portal. From **System Administration** select **Security**, then select either **Users** or **Roles**. Select the desired user or role, then select the appropriate tab: **SQL Privileges** for administrative privileges, **SQL Tables**, **SQL Views**, or **SQL Procedures** for object privileges.
- From SQL, use the GRANT command to grant specific administrative privileges or object privileges to a specified user or role (or list of users or roles). You can use the REVOKE command to remove privileges.

- From ObjectScript, use the **%SYSTEM.SQL.GrantObjPriv()** method to grant specific object privileges to a specified user (or list of users).

21.4.2 Listing SQL Privileges

- Use the Management Portal. From **System Administration** select **Security**, then select either **Users** or **Roles**. Select the desired user or role, then select the appropriate tab: **SQL Privileges** for administrative privileges, **SQL Tables**, **SQL Views**, or **SQL Procedures** for object privileges.
- From SQL, use the %CHECKPRIV command to determine if the current user has a specific administrative or object privilege.
- From ObjectScript, use the **$SYSTEM.SQL.CheckPriv()** method to determine if a specified user has a specific object privilege.

22

Using the Caché SQL Gateway

The Caché SQL Gateway provides access from Caché to external databases via JDBC and ODBC. This chapter discusses the following topics:

- Architecture of the Caché SQL Gateway — describes the internal structure and limitations of the SQL Gateway
- Creating Gateway Connections for External Sources — gives an overview of *logical connection definitions*, which are used by the SQL Gateway wizards to identify the external databases.
- The Link Table Wizard: Linking to a Table or View — describes the procedure for linking to tables or views in external sources so that you can access the data in the same way you access any Caché object.
- The Link Procedure Wizard: Linking to a Stored Procedure — describes the procedure for linking to stored procedures in external sources.
- Controlling Gateway Connections — describes methods used to manage SQL Gateway connections.
- The Data Migration Wizard: Migrating Data from an ODBC Source — describes how to migrate data from external ODBC sources and create an appropriate Caché class definition to store the data.

22.1 Architecture of the Caché SQL Gateway

Internally, the Caché SQL Gateway uses the following components:

- The *Connection Manager* maintains a list of *logical connection definitions* for Caché. Each definition has a logical name used in Caché, as well as connection details for a specific external ODBC or JDBC compliant database. The Caché SQL Gateway uses these logical names when it establishes connections (see Creating Gateway Connections for External Sources).
- The *Caché SQL Gateway API* is a set of functions used by a Caché program to communicate with a third-party RDBMS. These functions are implemented by means of a shared library, which is responsible for making the ODBC or JDBC calls.
- The *External Table Query Processor* is an extension to the Caché SQL Query Processor that handles queries targeted at external tables.
- The *SQL Dictionary* stores a list of all defined SQL tables. A given table is marked as "external" when its data is stored in a third-party RDBMS. When the Caché SQL Query Processor detects that the table (or tables) referenced within an SQL query are external, it invokes the External Table Query Processor, which generates a *query execution plan* by calling the Caché SQL Gateway API instead of accessing data stored within Caché.

22.1.1 Persisting External Tables in Caché

All object persistence in Caché is provided by means of a storage class (see "Storage Definitions and Storage Classes" in *Using Caché Objects*), which generates the code needed to save and retrieve a persistent object within a database. The SQL storage class (%CacheSQLStorage) provides object persistence by means of specially generated SQL queries.

A class that uses %CacheSQLStorage for persistence indicates that it is an "external" class by providing a value for its CONNECTION and EXTERNALTABLENAME class parameters. The class compiler creates an SQL table definition for the class, and generates the SQL queries for the object persistence code. These queries automatically make calls to the correct external database by means of the External Table Query Processor.

22.1.2 Restrictions on SQL Gateway Queries

When you use the Caché SQL Gateway, note the following restrictions:

- All the tables listed in the FROM clause of an SQL query must come from the same data source. Queries that join data from heterogeneous data sources are not allowed.
- SQL queries targeted at external databases cannot use the following Caché SQL extensions:
  - The "->" operator.
  - The %EXACT function, or the %SYSTEM.Util **Collation()** method with the collation flag set to EXACT.
  - The inclusion of other columns within a count (*) query.
  - Caché-specific operators that have % as the first character of their name.

22.2 Creating Gateway Connections for External Sources

Caché maintains a list of SQL Gateway connection definitions, which are logical names for connections to external data sources. Each connection definition consists of a logical name (for use within Caché), information on connecting to the data source, and a user name and password to use when establishing the connection. These connections are stored in the table %Library.sys_SQLConnection. You can export data from this table and import it into another Caché instance.

Each gateway connection consists of the following details:

- A logical name for the gateway connection. This name would be used, for example, within any Caché SQL queries.
- Optional login credentials to access the database.
- Optional information to control the JDBC or ODBC driver.
- Driver-specific connection details:
  - For JDBC: The full class name of the JDBC client driver, the driver class path (a list of JAR files to search when locating the JDBC driver), and the JDBC connection URL.
  - For ODBC: a DSN (data source name), defined in the usual way (see Using Caché as an ODBC Data Source on Windows and Using Caché as an ODBC Data Source on UNIX® in *Using Caché with ODBC*).

    Note: When creating an SQL gateway connection for use by the Link Table Wizard using Microsoft SQL Server DNS configuration, do not set the **Use regional settings** option. This option is intended only for applications that display data, not for applications that process data.

For detailed information on creating logical connection definitions for JDBC and ODBC, see:

- Creating JDBC SQL Gateway Connections for External Sources in *Using Caché with JDBC*
- Creating ODBC SQL Gateway Connections for External Sources in *Using Caché with ODBC*

22.3 The Link Table Wizard: Linking to a Table or View

The Management Portal provides a wizard that you can use to link to an external table in an ODBC- or JDBC-compliant database. When you have linked to an external table, you can:

- Access data stored in third-party relational databases within Caché applications using objects and/or SQL queries.
- Store persistent Caché objects in external relational databases.

For example, suppose you have an Employee table stored within an external relational database. You can use this table within Caché as an object by creating an Employee class that communicates (by executing SQL queries via JDBC or ODBC) with the external database.

From the perspective of a Caché application, the Employee class behaves in much the same way as any other persistent class: You can open instances, modify, and save them. If you issue SQL queries against the Employee class, they are automatically dispatched to the external database.

The use of the Caché SQL Gateway is independent of application logic; an application can be modified to switch between external databases and the built-in Caché database with minimal effort and no change to application logic.

Any class that uses the Caché SQL Gateway to provide object persistence is identical in usage to classes that using native persistence and can make full use of Caché features including Java, ActiveX, SQL, and Web access.

22.3.1 Using the Link Table Wizard

When you link to an external table or view, you create a persistent Caché class that is linked to that table or view. The new class stores and retrieves data from the external source using the SQL Gateway. You can specify information about both the Caché class and the corresponding SQL table in Caché.

Note: This wizard generates ObjectScript code with class names and class member names that you control. When you use this wizard, be sure to follow the rules for ObjectScript identifiers, including length limits (see the section on Naming Conventions in *Using Caché Objects*).

- If you have not yet created a gateway connection to the external database, do so before you begin (see Creating Gateway Connections for External Sources).
- From the Management Portal select **System Explorer**, then **SQL** (**[System] > [SQL]**). Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces.

  At the top of the page, click the `Wizards` drop-down list, and select **Link Table**.
- On the first page of the wizard, select one or more table or views, as follows:
  - `Select a destination namespace` — Select the Caché namespace to which the data will be copied.
  - `Schema Filter` — Specify a schema (class package) name that contains the table or view. You can specify a name with wildcards to return multiple schemas, or % to return all schemas. For example, C% will return all schemas in the namespace beginning with the letter C. Use of this filter is recommended, as it will shorten the return list of schemas to select from, and thus improve loading speed. You can select multiple items. In this case,

when you click Next, the next screen prompts you for a package name. Specify the name of the package to contain the classes and then click Finish.

- Table Filter — Specify the table or view to link to. You can specify a name with wildcards to return multiple tables and/or views, or % to return all tables/views.
- Table type — Select TABLE, VIEW, SYSTEM TABLE, or ALL. The default is TABLE.
- Select a SQL Gateway connection — Select the SQL Gateway connection to use.

- Click Next.
- On the second page, specify which fields should be available as object properties in Caché. Make changes as follows:
  - Highlight one or more fields and click the single arrow to move it or them from one list to another; click the double arrow to move all fields (selected or not) from one list to another.
  - In the selected list, use the up and down arrows to modify the order of the fields in the table that Caché projects for the given class. This does not affect the order of the properties in the class definition.
- Click Next.
- On the third page, specify information about the properties in the generated class. For each property, you can specify all the available options:
  - Read only — Select this check box to make the property read-only. This controls the ReadOnly keyword for the property.

    Tip: Use the select_all check box to select or clear all the check boxes in this column.
  - New Property Name — Specifies the name of the object property that will contain the data from this field.
  - New Column Name (SQL Field Name) — Specifies the SQL field name to use for this property. This controls the SqlFieldName keyword for the property.
- Click Next.
- On the last page, specify the following:
  - Primary Key — Select the primary key for the new Caché table from the list provided. In addition to the default key provided, you can click the "Browse" button to select one or more columns. You may select multiple columns; multiple columns are returned as a composite key separated by commas. You must specify a primary key.
  - New class name — Specify the name of the Caché class to create, including the package. The default package name is nullschema.
  - New table name — Specify the name of the SQL table to create in Caché. This controls the SqlTableName keyword for the class.
- Click Finish. The wizard displays the Background Jobs page with a link to the background tasks page.
- Click Close. Or click the given link to view the background tasks page. In either case, the wizard starts a background task to do the work.

The wizard stores a new class definition in the Caché database and compiles it. If data is present, it should be immediately visible in the external database (you can check by issuing SQL queries against the newly created Caché class/table). You can now use the new class as you would any other persistent class within Caché.

Note: **Closing the Link Table Connection**

By design, the code generated by the Link Table Wizard does not close the connections that it opens. This avoids problems such as conflicts between SQL statements that share the same connection. See "Controlling Gateway Connections" for more information.

22.3.2 Limitations When Using the Linked Table

As always, it is important to be aware of the particular limitations (syntactical or otherwise) and requirements of the database to which you are connecting. The following are a few examples:

- Informix: You cannot create a view inside of Caché that is based on a linked Informix table, because the generated SQL is not valid in Informix.
- Sybase: As part of query processing, Caché SQL can transform the expression of an outer join into an equivalent canonicalized form. The SQL92-standard CROSS JOIN syntax may be required to reconstruct this form as SQL in order to access a linked table. Because Sybase does not support SQL92-standard CROSS JOIN, some queries using outer joins on linked Sybase tables will fail to execute.

Before you try to use a linked table, you might want to examine the cached query that is generated for it, to ensure that the syntax is valid for the database you are using. To see the cached query for a given linked table:

- In the Management Portal, go to **[System] > [SQL]** and select `Browse SQL Schemas` in the `SQL Operations` column.
- Click the namespace you are interested in.
- Click the `Queries` link next to the package that contains the table.
- The system displays a table of the cached queries for this package. The `Query` column displays the full query.
- Optionally click the link for the query to see more details.

22.4 The Link Procedure Wizard: Linking to a Stored Procedure

The Management Portal provides a wizard that you can use to link to a stored procedure defined in an external ODBC- or JDBC-compliant database. When you link to the procedure, the system generates a method and a class to contain the method. When you link to an stored procedure, you create a class method that does the same action that the stored procedure does. This method is marked with the `SqlProc` keyword. The class method is generated within a new class, and you can specify information such as the class and package name. This method cannot accept a variable number of arguments. Default parameters are permitted, but the signature of the stored procedure is fixed.

Note: This wizard generates ObjectScript code with class names and class member names that you control. When you use this wizard, be sure to follow the rules for ObjectScript identifiers, including length limits (see the section on Naming Conventions in *Using Caché Objects*).

- If you have not yet created a gateway connection to the external database, do so before you begin (see Creating Gateway Connections for External Sources).
- From the Management Portal select **System Explorer**, then **SQL** (**[System] > [SQL]**). Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces.

At the top of the page, click the Wizards drop-down list, and select **Link Procedure**.

- On the first page of the wizard, select one or more procedures, as follows:
  - Select a destination namespace — Select the Caché namespace to which the data will be copied.
  - Schema Filter — Specify a schema (class package) name that contains the procedure. You can specify a name with wildcards to return multiple schemas, or % to return all schemas. For example, C% will return all schemas in the namespace beginning with the letter C. Use of this filter is recommended, as it will shorten the return list of schemas to select from, and thus improve loading speed.
  - Procedure Filter — Specify a procedure to link to. You can specify a name with wildcards to return multiple procedures, or % to return all procedures. You can select multiple procedures. In this case, when you click Next, the next screen prompts you for a package name. Specify the name of the package to contain the classes and then click Finish.
  - Select a SQL Gateway connection — Select the SQL Gateway connection to use.
- Click Next.
- On the second page, specify details about the class to generate in Caché:
  - New package name — Specify the name of the package to contain the class or classes.
  - New class name — Specify the name of the class to generate.
  - New procedure name — Specify the name of the procedure; specifically this controls the SqlName keyword of the method.
  - New method name — Specify the name of the method to generate.
  - Description method name — Optionally provide a description of the method; this is used as a comment for the class definition, to be displayed in the class reference.
- Click Finish. The wizard displays the Background Jobs page with a link to the background tasks page.
- Click Close. Or click the given link to view the background tasks page. In either case, the wizard starts a background task to do the work.

The wizard stores a new class definition within the Caché database and compiles it.

Note: **Closing the Link Procedure Connection**

By design, the code generated by the Link Procedure Wizard does not close the connections that it opens. This avoids problems such as conflicts between SQL statements that share the same connection. See "Controlling Gateway Connections" for more information.

22.5 Controlling Gateway Connections

In some cases, it may be necessary to manage connections created by code that links external tables or stored procedures (see "The Link Table Wizard" and "The Link Procedure Wizard"). SQL Gateway connections can be managed by the %SYSTEM.SQLGateway class, which provides methods such as the following:

- **DropAll()** — drop all open connections and unload the SQL Gateway library.
- **DropConnection()** — disconnect the specified JDBC or ODBC connection.
- **TestConnection()** — test a previously defined SQL Gateway connection (see "Creating Gateway Connections for External Sources") and write diagnostic output to the current device.

- Various methods for opening connections and controlling transactions. See the %SYSTEM.SQLGateway class documentation for full details.

These methods can be called with the special $SYSTEM object. For example, the following command would close a previously defined SQL Gateway connection named "MyConnectionName":

```
do $system.SQLGateway.DropConnection("MyConnectionName")
```

Note that SQL Gateway connection names are case-sensitive.

22.6 The Data Migration Wizard: Migrating Data from an ODBC or JDBC Source

The Management Portal provides a wizard that you can use to migrate data from an external table or view.

When you migrate data from a table or view in an external source, the system generates a persistent class to store data of that table or view and then copies the data. This wizard assumes that the class should have the same name as the table or view from which it comes; similarly, the property names are the same as in the table or view. After the class has been generated, it does not have any connection to external data source.

- If you have not yet created an SQL Gateway connection to the external database, do so before you begin (see Creating Gateway Connections for External Sources).
- From the Management Portal select **System Explorer**, then **SQL** (**[System] > [SQL]**). Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces.

  At the top of the page, click the `Wizards` drop-down list, and select **Data Migration**.
- On the first page of the wizard, select the table or view, as follows:
  - `Select a destination namespace` — Select the Caché namespace to which the data will be copied.
  - `Schema Filter` — Specify a schema (class package) name that contains the table or view. You can specify a name with wildcards to return multiple schemas, or % to return all schemas. For example, C% will return all schemas in the namespace beginning with the letter C. Use of this filter is recommended, as it will shorten the return list of schemas to select from, and thus improve loading speed.
  - `Table Filter` — Specify a table or view name. You can specify a name with wildcards to return multiple tables and/or views, or % to return all tables/views.
  - `Table type` — Select `TABLE`, `VIEW`, `SYSTEM TABLE`, or `ALL`. The default is `TABLE`.
  - `Select a SQL Gateway connection` — Select the SQL Gateway connection to use.
- Click `Next`.
- On the next page, you can optionally specify the following information for each class:
  - `New Schema` — Specify the package to contain the class or classes. Be sure to follow the rules for ObjectScript identifiers, including length limits (see the section on Naming Conventions in *Using Caché Objects*).

    Tip: To change the package name for all classes, type a value at the top of this column and then click `Change all`.
  - `Copy Definition` — Select this check box to generate this class, based on the table definition in the external source. If you have already generated the class, you can clear this check box.

  - `Copy Data` — Select this check box to copy the data for this class from the external source. When you copy data, the wizard overwrites any existing data in the Caché class.

- Click `Next`. The wizard displays the following optional settings:
  - `Disable validation` — If checked, data will be imported with `%NOCHECK` specified in the *restriction* parameter of the **INSERT** command.
  - `Disable journaling for the importing process` — If checked, journaling will be disabled for the process performing the data migration (not system-wide). This can make the migration faster, at the cost of potentially leaving the migrated data in an indeterminate state if the migration is interrupted by a system failure. Journaling is re-enabled at the end of the run, successful or not.
  - `Defer indices` — If checked, indices are built after the data is inserted. The wizard calls the class' **%SortBegin()** method prior to inserting the data in the table. This causes the index entries to be written to a temporary location for sorting. They are written to the actual index location when the wizard calls the **%SortEnd()** method after all rows have been inserted. Do not use Defer Indices if there are Unique indices defined in the table and you want the migration to catch any unique constraint violations. A unique constraint violation will not be caught if Defer Indices is used.
  - `Disable triggers` — If checked, data will be imported with `%NOTRIGGER` specified in the *restriction* parameter of the **INSERT** command.
  - `Delete existing data from table before importing` — If checked, existing data will be deleted rather than merged with the new data.

- Click `Finish`. The wizard opens a new window and displays the Background Jobs page with a link to the background tasks page. Click `Close` to start the import immediately, or click the given link to view the background tasks page. In either case, the wizard starts the import as a background task.
- In the Data Migration Wizard window, click `Done` to go back to the home page of the Management Portal.

22.6.1 Microsoft Access and Foreign Key Constraints

When you use the Data Migration Wizard with Microsoft Access, the wizard tries to copy any foreign key constraints defined on the Access tables. To do this, it queries the `MSysRelationships` table in Access. By default, this table is hidden and does not provide read access. If the wizard can't access `MSysRelationships`, it migrates the data table definitions to Caché without any foreign key constraints.

If you want the utility to migrate the foreign key constraints along with the table definitions, set Microsoft Access to provide read access for `MSysRelationships`, as follows:

- In Microsoft Access, make sure that system objects are displayed.
- Click `Tools > Options` and select the setting on the `View` tab.
- Click `Tools > Security > User and Group Permissions`. Then select the `Read` check box next to the table name.

A

Importing and Exporting SQL Data

In the Management Portal, there are tools for importing and exporting data:

- Importing Data from a Text File
- Exporting Data to a Text File

These tools use Dynamic SQL, which means that queries are prepared and executed at runtime. By default, the maximum size of a row that can be imported or exported is 32,767 characters. This limitation can be greatly expanded by configuring long string operations.

You can also import data using the %SQL.Import.Mgr class, and export data using the %SQL.Export.Mgr class.

A.1 Importing Data from a Text File

You can import data from a text file into a suitable Caché class. When you do so, the system creates and saves new rows in the table for that class. The class must already exist and must be compiled. To import data into this class:

1. From the Management Portal select **System Explorer**, then **SQL** (**[System] > [SQL]**). Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces.
2. At the top of the page, click the `Wizards` drop-down list, and select **Data Import**.
3. On the first page of the wizard, start by specifying the location of the external file. For **The import file resides on**, click the name of the server to use.
4. Then enter the complete path and filename of the file.
5. For **Select schema name**, click the Caché package into which you want to import the data.
6. For **Select table name**, click the class that will contain the newly created objects.
7. Then click **Next**.
8. On the second page of the wizard, click the columns that will contain the imported data.
9. Then click **Next**.
10. On the third page of the wizard, describe the format of the external file.
    - For **What delimiter separates your columns?**, click the option corresponding to the delimiter in this file.
    - Click the **First row contains column headers?** check box if the first line of the file does not contain data.

- For **String quote**, click the option that indicates the quote delimiter character this file uses to start and end string data.
- For **Date format**, click the option that indicates the date format in this file.
- For **Time format**, click the option that indicates the time format in this file.
- For **TimeStamp format**, click the option that indicates the timestamp format in this file.
- Click the **Disable validation?** check box if you do not want the wizard to validate the data upon import.
- Click the **Defer Index Building with %SortBegin/%SortEnd?** check box if you do not want the wizard to rebuild indices during import. If **Defer Index Building** is checked, the wizard calls the %SortBegin method for the class before inserted the imported data into the table. When the import is done the wizard calls the %SortEnd method. No validation is done (same as an INSERT with %NOCHECK). This is because indices cannot be checked for uniqueness during SQL insert when %SortBegin/%SortEnd is used. If **Defer Index Building** is checked, the imported data is assumed to be valid and will not be checked for validity.
- Optionally click **Preview Data** to see how the wizard will parse the data in this file.

11. Click **Next**.
12. Review your entries and click **Finish**. The wizard displays the **Data Import Result** dialog box.
13. Click **Close**. Or click the given link to view the background tasks page.

    In either case, the wizard starts a background task to do the work.

A.2 Exporting Data to a Text File

You can export data for a given class to a text file. To do so:

1. From the Management Portal select **System Explorer**, then **SQL**. Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces.
2. At the top of the page, click the `Wizards` drop-down list, and select **Data Export**.
3. On the first page of the wizard:
   - Enter the complete path and filename of the file that you are going to create to hold the exported data.
   - From the drop-down lists, select a Namespace, Schema Name, and Table Name from which you want to export the data.
   - Optionally select a character set from the **Charset** drop-down list; the default is Device Default.

   Then click **Next**.
4. On the second page of the wizard, select which columns to export. Then click **Next**.
5. On the third page of the wizard, describe the format of the external file.
   - For **What delimiter separates your columns?**, click the option corresponding to the delimiter in this file.
   - Click the **Export column headers?** check box if you want to export column headers as the first line of the file.
   - For **String quote**, click an option to indicate how to start and end string data in this file.
   - For **Date format**, click an option to indicate the date format to use in this file.
   - For **Time format**, click an option to indicate the time format to use in this file.

- Optionally click **Preview Data** to see what the results will look like.

Then click **Next**.

6. Review your entries and click **Finish**. The wizard displays the **Data Export Result** dialog box.
7. Click **Close**. Or click the given link to view the background tasks page.

   In either case, the wizard starts a background task to do the work.