Caché SQL Optimization Guide

Version 2018.1
2020-11-13

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

Caché SQL Optimization Guide
Caché Version 2018.1 2020-11-13

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 Figures

About This Book

This book describes how to optimize performance of Caché SQL. Caché SQL provides standard relational access to data stored within a Caché database.

The book addresses the following topics:

- "Defining and Building Indices" describes how to define and build indices.
- "Optimizing SQL Tables" describes ways that you can optimize access to the data in your Caché SQL tables.
- "Cached Queries" describes how Caché SQL retains recent queries in a cache, enabling rapid execution and sharing of defined queries.
- "Optimizing SQL Queries" describes ways that you can optimize the performance of your Caché SQL queries.
- "Interpreting an SQL Query Plan" gives tips on understanding the results of a "show plan" for an SQL query.
- "SQL Statements" describes how you can view all executed SQL statements; queries provide performance statistics.
- "Frozen Plans" describes how you can view and freeze SQL query plans.

For a detailed outline, see the Table of Contents.

This book supplements the following:

- *Using Caché SQL* provides in-depth material on SQL components and features, how to execute SQL queries, and support for error and transaction processing.
- *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 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 SQL Performance Optimization

Caché SQL supports several features that optimize SQL performance.

1.1 Table Definition Optimization

SQL performance is fundamentally dependent upon good data architecture. Dividing your data into multiple tables and establishing relationships between those tables is essential to efficient SQL. How to define tables and their relations to each other is beyond the scope of this book.

This book describes the following operations to optimize a table definition. These operations require that the table be defined, but do not require the table to be populated with data:

- Indices: you can define an index for a table field or group of fields. You can define several different type of indices: standard, bitmap, bitslice, and bitmap extent. SQL optimization uses the defined indices, rather than the data values themselves, to access specific records for a query, update, or delete operation.
- Frozen Plans: you can freeze Data Definition execution plans. (Frozen Plans are also used to freeze query plans).

1.2 Table Data Optimization

You can perform the following operations to optimize table access based on analysis of typical data in the table:

- Tune Table: examines typical table data and generates ExtentSize (number of rows), Selectivity (percent of rows having a specific value), and BlockCount metadata. The query optimizer uses this information to determine the most efficient query execution plan.
- Selectivity and Outlier Selectivity: determines the percent of rows having a specific value for a field, and whether one value is an outlier value, a value significantly more common than the other values for that field.

1.3 Query Optimization

You can perform the following operations to optimize the execution of a specific query. These query optimizations use existing table definition and table data optimizations:

- Runtime Statistics: used to measure the performance of query execution on your system.
- Show Plan displays the execution plan for a query.
- Cached Queries and Literal Substitution: maintains a cache of recent dynamic queries, allowing for re-execution of a query without repeating the overhead of preparing the query.
- SQL Statements and Frozen Plans allows you to retain a specific query execution plan.
- Index configuration and usage: used to specify how existing indices should be used.
- Index optimization hints: %ALLINDEX, %IGNOREINDEX
- JOIN optimization hints: %FIRSTTABLE, %FULL, %INORDER, %STARTTABLE
- Subquery optimization hints: %NOFLATTEN, %NOMERGE, %NOREDUCE, %NOSVSO
- Parallel query execution: FROM %PARALLEL
- Union optimization: UNION %PARALLEL, UNION/OR

1.4 Configuration Optimization

By default, the **Memory and Startup Settings** default to **Automatically** configured, and the **Maximum Per-Process Memory** defaults to 262144 kb. To optimize SQL running on a production system, you should change the default to **Manually** configured, and increase the **Maximum Per-Process Memory** setting. For further details, refer to Memory and Startup Settings in the "Configuring Caché" chapter of the *Caché System Administration Guide*.

2

Defining and Building Indices

This chapter describes how to define and build indices on field values within tables. It includes the following topics:

Overview: index types and attributes.

Data Access:

- Data Storage Types and Indices
- Index Global Names
- Master Map, RowID, and IDKEY
- Automatically-Defined Indices for Unique and Primary Key constraints
- Bitmap Extent Index

Defining Indices:

- Defining Indices
- Bitmap Indices
- Bitslice Indices

Performing Operations on Indices:

- Building Indices
  - On an Inactive System
  - On a READONLY Active System
  - On a READ and WRITE Active System
- Validating Indices
- Listing Indices
- Opening an Instance by Index Key
- Checking If an Instance Exists
- Deleting an Instance

This chapter describes how to define and build indices. It does not describe which fields to index or how to analyze the effectiveness of defined indices. See "What to Index" for information on which fields to index to optimize performance of specific queries. See "Index Usage Analysis" for tools to analyze index usage for all queries in the current namespace.

2.1 Overview

An index is a structure maintained by a persistent class that Caché can use to optimize queries and other operations.

You can define an index on the values of a field within a table, or the corresponding property within a class. (You can also define an index on the combined values of several fields/properties.) The same index is created, regardless of whether you defined it using SQL field and table syntax, or class property syntax. Caché automatically defines indices when certain types of fields (properties) are defined. You can define additional indices on any field in which data is stored or for which data can be reliably derived. Caché provides several types of indices. You can define more than one index for the same field (property), providing indices of different types for different purposes.

Caché populates and maintains indices (by default) whenever a data insert, update, or delete operation is carried out against the database, whether using SQL field and table syntax, or class property syntax. You can override this default (by using the %NOINDEX keyword) to rapidly make changes to the data, and then build or rebuild the corresponding index as a separate operation. You can define indices before populating a table with data. You can also define indices for a table that is already populated with data and then populate (build) the index as a separate operation.

Caché makes use of available indices when preparing and executing SQL queries. By default it selects which indices to use to optimize query performance. You can override this default to prevent the use of one or more indices for a specific query or for all queries, as appropriate. For information about optimizing index usage, refer to the Using Indices section of the "Optimizing Query Performance" chapter of this book.

2.1.1 Index Attributes

Every index has a unique name. This name is used for database administrative purposes (reporting, index building, dropping indices, and so on). Like other SQL entities, an index has both an SQL index name and a corresponding index property name; these names differ in permitted characters, case-sensitivity, and maximum length. If defined using the SQL **CREATE INDEX** command, the system generates a corresponding index property name. If defined using a persistent class definition, the SqlName keyword allows the user to specify a different SQL index name. The Management Portal SQL interface Catalog Details displays the SQL index name (**SQL Map Name**) and the corresponding index property name (**Index Name**) for each index.

The index type is defined by two index class keywords, Type and Extent. The types of indices available with Caché include:

- Standard Indices (Type = index) — A persistent array that associates the indexed value(s) with the RowID(s) of the row(s) that contains the value(s). Any index not explicitly defined as a bitmap index, bitslice index, or extent index is a standard index.
- Bitmap Indices (Type = bitmap) — A special kind of index that uses a series of bitstrings to represent the set of object ID values that correspond to a given indexed value; Caché includes a number of performance optimizations for bitmap indices.
- Bitslice Indices (Type = bitslice) — A special kind of index that enables very fast evaluation of certain expressions, such as sums and range conditions. Certain SQL queries automatically use bitslice indices.
- Extent Indices — An index of all of the objects in an extent. For more information, see the Extent index keyword page in the *Caché Class Definition Reference*.

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

2.1.2 Storage Type and Indices

The index functionality described here applies to data stored in a persistent class.

Caché SQL supports index functionality for data stored using the Caché default storage structure: %CacheStorage (CacheStorage-mapped classes).

Caché SQL also supports index functionality for data stored using %CacheSQLStorage (CacheSQLStorage-mapped classes). You can define an index for a CacheSQLStorage-mapped class using a functional index type. The index is defined in the same manner as an index in a class using default storage, with the following special considerations:

- The class must define the IdKey functional index, if it is not automatically system assigned.
- This functional index must be defined as an INDEX.

Refer to %Library.FunctionalIndex for further details.

Note that the %CacheStorage and %CacheSQLStorage class methods should not be called directly. Instead, you should invoke index functionality using the %Persistent class methods and the operations described in this chapter.

2.1.3 Index Global Names

The subscripted global used to store index data is generated using one of two strategies:

- %CLASSPARAMETER *USEEXTENTSET*=0 uses a global naming strategy that creates "traditional" global names, consisting of a user-specified name, an appended letter code, and the name of the index. These global names are comprehensible to the user, but can be long and are less efficient than hashed global names.
  - If USEEXTENTSET=0 and DEFAULTGLOBAL is not specified, the following example describes the generated global names: the Sample.MyTest persistent class would define a Master Map global named `^Sample.MyTestD` a Bitmap Extent index global name `^Sample.MyTestI("$MyTest")` (or `^Sample.MyTestI("DDLBEIndex")`), and for the defined index NameIDX, it defines a global named `^Sample.MyTestI("NameIDX")`. Note that these globals specify the persistent class name (which is case-sensitive), not the SQL table name. For a full list of the appended letter codes, refer to the *DEFAULTGLOBAL* parameter of the %Library.Persistent class.
  - If USEEXTENTSET=0 and *DEFAULTGLOBAL* is specified, the specified global name is substituted for the persistent class name. This allows you to specify a name that is shorter or clearer than the persistent class name. For example, if DEFAULTGLOBAL="MyGlobal" the globals would have names such as the following: `^MyGlobalD` and `^MyGlobalI("NameIDX")`.
- %CLASSPARAMETER *USEEXTENTSET*=1 uses a global naming strategy that creates hashed global names. This involves hashing the package name, hashing the class name, then appending a dot and a sequential integer suffix that identifies the index. These global names are less comprehensible to the user, but tend to be shorter and more efficient.

  The integer suffix is keyed solely to the index name; the field(s) associated with the index name and the index type have no effect on integer numbering. For example, ^EW3K.CgZk.1 is the Master Map, ^EW3K.CgZk.2 is the Bitmap Extent, ^EW3K.CgZk.3 is a defined standard index NameIDX for the LastName field, and ^EW3K.CgZk.4 is the defined index WorkIdIDX. If you delete NameIDX the global ^EW3K.CgZk.3 is deleted, creating a gap in the integer sequence. If you define LNameIDX for the LastName field, the global ^EW3K.CgZk.5 is created; however, if you later create bitmap index NameIDX for the FullName field, the index global will again be ^EW3K.CgZk.3. For further details on this index global naming convention, refer to the *USEEXTENTSET* parameter of the %Library.Persistent class.
  - If USEEXTENTSET=1 and DEFAULTGLOBAL is not specified, the package name and class name are hashed, as described above. The sequential integer suffix is appended.
  - If USEEXTENTSET=1 and DEFAULTGLOBAL is specified, the DEFAULTGLOBAL name is used rather than the hashed package name and class name. The sequential integer suffix is appended. For example, if DEFAULTGLOBAL="MyGlobal" the globals would have names such as the following: `^MyGlobal.1` and `^MyGlobal.3`.

If you use the CREATE TABLE command to define a table, *USEEXTENTSET* defaults to 1. Therefore, by default, **CREATE TABLE** creates hashed global names. This default behavior can be changed using the %CLASSPARAMETER keyword and the USEEXTENTSET and DEFAULTGLOBAL parameters. You can change the system-wide default using the **SetDDLUseExtentSet()** method.

If you define a persistent class that is projected to a table, *USEEXTENTSET* defaults to 0. Therefore, by default, traditional global names are used.

DEFAULTGLOBAL, if defined, serves as a default value. If the ExtentLocation, DataLocation, or IndexLocation storage keywords are defined, those values are used rather than the defaults described above. For further details, refer to Hashed Global Names in the "Introduction to Persistent Objects" chapter of *Defining and Using Classes*.

You can supply the global name to ZWRITE to display the index data.

2.1.4 Master Map

The system automatically defines a Master Map (Data/Master) for every table. The Master Map is not an index, it is a map that directly accesses the data itself using its map subscript field(s). By default, the master map subscript field is the system-defined RowID field. By default, this direct data access using the RowID field is represented with the SQL Map Name IDKEY.

By default, a user-defined primary key is not the IDKEY. This is because Master Map lookup using RowID integers is almost always more efficient than lookup by primary key values. However, if you specify that the primary key is the IDKEY, the primary key index is defined as the Master Map for the table and SQL Map Name is the primary key index name.

For a single-field primary key/IDKEY, the primary key index is the Master Map, but the Master Map data access column remains the RowID. This is because there is a one-to-one match between a record's unique primary key field value and its RowID value, and RowID is the presumed more efficient lookup. For a multi-field primary key/IDKEY, the Master Map is given the primary key index name, and the Master Map data access columns are the primary key fields.

You can view the Master Map definition through the Management Portal SQL Catalog Details tab.

For SQL and default storage, the Master Map data is stored in a global named ^package.classnameD. Note that this global name specifies the persistent class name, not the corresponding SQL table name, and that the global name is case-sensitive. You can use ZWRITE to display the Master Map data.

Data access using a Master Map is inefficient, especially for large tables. For this reason, it is recommended that the user define indices that can be used to access data fields specified in WHERE conditions, JOIN operations, and other operations.

2.2 Automatically-Defined Indices

The system automatically defines certain indices when you define a table. The following indices are automatically generated when you define a table and populated when you add or modify table data. If you define:

- A primary key that is not an IDKEY, the system generates a corresponding index of type Unique. The name of the primary key index may be user-specified or derived from the name of the table. For example, if you define an unnamed primary key, the corresponding index will be named tablenamePKEY#, where # is a sequential integer for each unique and primary key constraint.
- A UNIQUE field, Caché generates an index for each UNIQUE field with the name tablenameUNIQUE#, where # is a sequential integer for each unique and primary key constraint.
- A UNIQUE constraint, the system generates an index for each UNIQUE constraint with the specified name, indexing the fields that together define a unique value.

You can view these indices through the Management Portal SQL Catalog Details tab. The CREATE INDEX command can be used to add a UNIQUE field constraint; the DROP INDEX command can be used to remove a UNIQUE field constraint.

By default, the system generates the IDKEY index on the RowID field. Defining an IDENTITY field does not generate an index. However, if you define an IDENTITY field and make that field the primary key, Caché defines the IdKey index on the IDENTITY field and makes it the primary key index. This is shown in the following example:

```
CREATE TABLE Sample.MyStudents (
           FirstName VARCHAR(12),
           LastName VARCHAR(12),
           StudentID IDENTITY,
           CONSTRAINT StudentPK PRIMARY KEY (StudentID) )
```

Similarly, if you define an IDENTITY field and give that field a UNIQUE constraint, Caché explicitly defines an IdKey/Unique index on the IDENTITY field. This is shown in the following example:

```
CREATE TABLE Sample.MyStudents (
           FirstName VARCHAR(12),
           LastName VARCHAR(12),
           StudentID IDENTITY,
           CONSTRAINT StudentU UNIQUE (StudentID) )
```

These IDENTITY indexing operations only occur when there is no explicitly defined IdKey index and the table contains no data.

2.2.1 Bitmap Extent Index

A bitmap extent index is a bitmap index for the rows of the table, not for any specified field of the table. In a bitmap extent index, each bit represents a sequential RowID integer value, and the value of each bit specifies whether or not the corresponding row exists. InterSystems SQL uses this index to improve performance of COUNT(*), which returns the number of records (rows) in the table. A table can have, at most, one bitmap extent index. Attempting to create more than one bitmap extent index results in an SQLCODE -400 error with the %msg `ERROR #5445: Multiple Extent indices defined: DDLBEIndex`.

All tables defined using CREATE TABLE automatically define a bitmap extent index. This automatically-generated index is assigned the Index Name DDLBEIndex and the SQL MapName %%DDLBEIndex. A table defined as a class may have a bitmap extent index defined with an Index Name and SQL MapName of $*ClassName* (where *ClassName* is the name of the table's persistent class.)

You can use the **CREATE INDEX** command with the BITMAPEXTENT keyword to add a bitmap extent index to a table, or to rename an automatically-generated bitmap extent index. For further details, refer to CREATE INDEX.

You can view a table's bitmap extent index through the Management Portal SQL Catalog Details tab. Though a table can have only one bitmap extent index, a table that inherits from another table is listed with both its own bitmap extent index and the bitmap extent index of the table it extends from. For example, the Sample.Employee table extends the Sample.Person table; in the **Catalog Details** Maps/IndicesSample.Employee lists both a `$Employee` and `$Person` bitmap extent index.

In a table that undergoes many **DELETE** operations the storage for a bitmap extent index can gradually become less efficient. You can rebuild a Bitmap Extent index from the Management Portal by selecting the table's **Catalog Details** tab, Maps/Indices option and selecting **Rebuild Index**.

The %SYS.Maint.Bitmap utility methods compress the bitmap extent index, as well as bitmap indices and bitslice indices. For further details, see "Maintaining Bitmap Indices".

Invoking the **%BuildIndices()** method builds an existing bitmap extent index in any of the following cases: the **%BuildIndices()** *pIndexList* argument is not specified (build all defined indices); *pIndexList* specifies the bitmap extent index by name; or *pIndexList* specifies any defined bitmap index. See "Building Indices Programmatically".

2.3 Defining Indices

There are two ways to define indices:

- Defining Indices Using a Class Definition, which includes:
  - Properties That Can Be Indexed
  - Indices on Multiple Properties
  - Index Collation
  - Using the Unique, PrimaryKey, and IdKey Keywords with Indices
  - Defining iFind Indices
  - Storing Data with Indices
  - Indexing Collections
  - Indexing an Embedded Object (%SerialObject) Property
  - Notes on Indices Defined in Classes
- Defining Indices Using DDL

2.3.1 Defining Indices Using a Class Definition

Within Studio, you can add index definitions to a %Persistent class definition using either the New Index Wizard or by editing the text of the class definition. An index is defined on one or more index property expressions optionally followed by one or more optional index keywords. It takes the form:

```
INDEX index_name ON index_property_expression_list [index_keyword_list];
```

where:

- *index_name* is a valid identifier.
- *index_property_expression_list* is a list of the one or more comma-separated property expressions that serve as the basis for the index.
- *index_keyword_list* is an optional comma-separated list of index keywords, enclosed in square brackets. Used to specify the index Type for a bitmap or bitslice index. Also used to specify a Unique, IdKey, or PrimaryKey index. (An IdKey or PrimaryKey index is, by definition, also a Unique index.) The complete list of index keywords appears in the *Caché Class Definition Reference*.

The *index_property_expression_list* argument consists of one or more index property expressions. An index property expression consists of:

- The name of the property to be indexed.
- An optional `(ELEMENTS)` or `(KEYS)` expression, which provide a means of indexing on collection subvalues. If the index property is not a collection, the user can use the **BuildValueArray()** method to produce an array containing keys and elements. For more information on keys and elements, see the "Indexing Collections" section.
- An optional collation expression. This consists of a collation name followed optionally by a list of one or more comma-separated collation parameters. You cannot specify an index collation for a Unique, IdKey, or PrimaryKey index. A Unique or PrimaryKey index takes its collation from the property (field) that is being indexing. An IdKey index is

always EXACT collation. For a list of valid collation names, see the "Collation Types" section of the "Collation" chapter of *Using Caché SQL*.

For example, the following class definition defines two properties and an index based on each of them:

```
Class MyApp.Student Extends %Persistent [DdlAllowed]
{
 Property Name As %String;
 Property GPA As %Decimal;

 Index NameIDX On Name;
 Index GPAIDX On GPA;
}
```

A more complex index definition might be:

```
 Index Index1 On (Property1 As SQLUPPER(77), Property2 AS EXACT);
```

2.3.1.1 Properties That Can Be Indexed

The only properties that can be indexed are:

- Those that are stored in the database
- Those that can be reliably derived from stored properties

A property that can be reliably derived (and is not stored) must be defined with the SQLComputed keyword as true; the code specified by SQLComputeCode must be the only way to derive the property's value and the property cannot be set directly.

If it is possible to directly set the value of a derived property, such as is the case of a simple (non-collection) property defined as Transient and not also defined as Calculated, then directly setting the property's value overrides the computation defined in SQLComputeCode and the value cannot be derived reliably from stored properties; this type of derived property is referred to as *nondeterministic*. (The Calculated keyword actually means that no instance memory is allocated.) The general rule is that only derived properties defined as Calculated and SQLComputed can be indexed. There is, however, an exception for derived collections: a collection that is derived (SQLComputed) and is Transient (not stored) and is not also defined as Calculated (meaning no instance memory) can be indexed.

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

2.3.1.2 Indices on Multiple Properties

You can define indices on combinations of two or more properties (fields). Within a class definition, use the On clause of the index definition to specify a list of properties, such as:

```
Class MyApp.Employee Extends %Persistent [DdlAllowed]
{
 Property Name As %String;
 Property Salary As %Integer;
 Property State As %String(MAXLEN=2);

 Index MainIDX On(State,Salary);
}
```

An index on multiple properties may be useful if you need to perform queries that use a combination of field values, such as:

```
SELECT Name,State,Salary
  FROM Employee
  ORDER BY State,Salary
```

2.3.1.3 Index Collation

A Unique, PrimaryKey, or IdKey index cannot specify a collation type. For other types of indices, each property specified in an index definition can optionally have a collation type. The index collation type should match the property (field) collation type when the index is applied.

1. If an index definition includes an explicitly specified collation for a property, the index uses that collation.
2. If an index definition does not include an explicitly specified collation for a property, the index uses the collation explicitly specified in the property definition.
3. If the property definition does not include an explicitly specified collation, then the index uses the collation that is the default for the property data type.

For example, the Name property is defined as a string, and therefore has, by default, SQLUPPER collation. If you define an index on Name, it takes, by default, the property's collation, and the index would also be defined with SQLUPPER. The property collation and the index collation match.

However, if a comparison applies a different collation, for example, `WHERE %EXACT(Name)=%EXACT(:invar)`, the property collation type in this usage no longer matches the index collation type. A mismatch between the property comparison collation type and the index collation type may cause the index to not be used. Therefore, in this case, you might wish to define the index for the Name property with collation EXACT. If an ON clause of a JOIN statement specifies a collation type, for example, `FROM Table1 LEFT JOIN Table2 ON %EXACT(Table1.Name) = %EXACT(Table2.Name)`, a mismatch between the property collation type specified here and the index collation type may cause Caché to not use the index.

The following rules govern collation matches between an index and a property:

- Matching collation types always maximize use of an index.
- A mismatch of collation types, where the property is specified with EXACT collation (as shown above) and the index has some other collation allow the index to be used, but its use is less effective than matching collation types.
- A mismatch of collation types, where the property collation is not EXACT and the property collation does not match the index collation, causes the index to not be used.

To explicitly specify a collation for a property in an index definition, the syntax is:

```
Index IndexName On PropertyName As CollationName;
```

where

- *IndexName* is the name of the index
- *PropertyName* is the property being indexed
- *CollationName* is the type of collation being used for the index

For example:

```
Index NameIDX On Name As Exact;
```

Different properties can have different collation types. For example, in the following example the F1 property uses SQLUPPER collation while F2 uses EXACT collation:

```
Index Index1 On (F1 As SQLUPPER, F2 As EXACT);
```

For a list of recommended collation types, see the "Collation Types" section of the "Collation" chapter of *Using Caché SQL*.

Note: An index specified as Unique, PrimaryKey, or IdKey cannot specify an index collation. The index takes its collation from the property collations.

2.3.1.4 Using the Unique, PrimaryKey, and IdKey Keywords with Indices

As is typical with SQL, Caché supports the notions of a unique key and a primary key. Caché also has the ability to define an ID key, which is one that is a unique record ID for the instance of a class (row of a table). These features are implemented through the Unique, PrimaryKey, and IdKey keywords:

- Unique — Defines a UNIQUE constraint on the properties listed in the index's list of properties. That is, only a unique data value for this property (field) can be indexed. Uniqueness is determined based on the property's collation. For example, if the property collation is EXACT, values that differ in letter case are unique; if the property collation is SQLUPPER, values that differ in letter case are not unique. However, note that the uniqueness of indices is not checked for properties that are undefined. In accordance with the SQL standard, an undefined property is always treated as unique.
- PrimaryKey — Defines a PRIMARY KEY constraint on the properties listed in the index's list of properties.
- IdKey — Defines a unique constraint and specifies which properties are used to define the unique identity of an instance (row). An IdKey always has EXACT collation, even when it is of data type string.

  Note: Every class/table has an ID key. If it is not defined explicitly (such as with an index IdKey), the system generates an ID key value automatically.

The syntax of such keywords appears in the following example:

```
Class MyApp.SampleTable Extends %Persistent [DdlAllowed]
{
  Property Prop1 As %String;
  Property Prop2 As %String;
  Property Prop3 As %String;

  Index Prop1IDX on Prop1 [ Unique ];
  Index Prop2IDX on Prop2 [ PrimaryKey ];
  Index Prop3IDX on Prop3 [ IdKey ];
}
```

Note: The IdKey, PrimaryKey, and Unique keywords are only valid with standard indices. You cannot use them with bitmap or bitslice indices.

It is also valid syntax to specify both the IdKey and PrimaryKey keywords together, such as:

```
 Index IDPKIDX on Prop4 [ IdKey, PrimaryKey ];
```

This syntax specifies that the IDPKIDX index is both the IdKey for the class (table), as well as its primary key. All other combinations of these keywords are redundant.

For any index defined with one of these keywords, there is a method that allows you to open the instance of the class where the properties associated with the index have particular values; for more information, see the "Opening an Instance by Index Key" section.

For more information on the IdKey keyword, see the IdKey page of the *Caché Class Definition Reference*. For more information on the PrimaryKey keyword, see the PrimaryKey page of the *Caché Class Definition Reference*. For more information on the Unique keyword, see the Unique page of the *Caché Class Definition Reference*.

2.3.1.5 Defining iFind Indices

You can define iFind indices in a table class definition as follows:

```
Class Sample.TextBooks Extends %Persistent [DdlAllowed]
{
 Property BookName As %String;
 Property SampleText As %String(MAXLEN=5000);

 Index NameIDX On BookName [ IdKey ];
 Index ifindIDXB On (SampleText) As %iFind.Index.Basic;
 Index ifindIDXS On (SampleText) As %iFind.Index.Semantic;
 Index ifindIDXA On (SampleText) As %iFind.Index.Analytic;
}
```

For further details refer to iFind Search Tool.

2.3.1.6 Storing Data with Indices

You can specify that a copy of one or more data values be stored within an index using the index Data keyword:

```
Class Sample.Person Extends %Persistent [DdlAllowed]
{
 Property Name As %String;
 Property SSN As %String(MAXLEN=20);

 Index NameIDX On Name [Data = Name];
}
```

In this case, the index, NameIDX, is subscripted by the collated (uppercase) value of the various Name values. A copy of the actual (uncollated) value of the Name is stored within the index. These copies are maintained when changes are made to the Sample.Person table through SQL or to corresponding the Sample.Person class or its instances through objects.

Maintaining a copy of data along within an index can be helpful in cases where you frequently perform selective (selecting a few rows out of many) or ordered searches that return a few columns out of many.

For example, consider the following query against the Sample.Person table:

```
SELECT Name FROM Sample.Person ORDER BY Name
```

The SQL Engine could decide to satisfy this request entirely by reading from the NameIDX and never reading the master data for the table.

Note: You cannot store data values with a bitmap index.

2.3.1.7 Indexing a NULL

If the data has a NULL (no data present) for an indexed field, the corresponding index represents this using an index null marker. By default, the index null marker value is -1E14. Use of an index null marker provides that null values collate before all non-null values.

The %Library.BigInt data type can potentially store small negative numbers less than -1E14. By default, the %BigInt index null marker value is -1E14, and is therefore compatible with existing %BigInt indices. If indexed %BigInt data values are likely to include these extremely small negative numbers, you can change the index null marker value for a specific field as part of property definition using the INDEXNULLMARKER property parameter, as shown in the following example:

```
Property ExtremeNums As %Library.BigInt(INDEXNULLMARKER = "-1E19");
```

You can also change the index null marker default value in a data type class definition. Refer to %Library.DataType for details.

2.3.1.8 Indexing Collections

When a property is indexed, the value that is placed in the index is the entire collated property value. For collections, it is possible to define index properties that correspond to the element and key values of the collection by appending (ELEMENTS) or (KEYS) to the property name. (ELEMENTS) and (KEYS) allow you to specify that multiple values are produced from a single property value and each of these sub-values is indexed. When the property is a collection then the ELEMENTS

token references the elements of the collection by value and the KEYS token references them by position. When both ELEMENTS and KEYS are present in a single index definition then the index key value includes the key and associated element value.

For example, suppose there is an index based on FavoriteColors property of the Sample.Person class. The simplest form of an index on the items in this property's collection would be either of:

```
INDEX fcIDX1 ON (FavoriteColors(ELEMENTS));
```

or

```
INDEX fcIDX2 ON (FavoriteColors(KEYS));
```

where `FavoriteColor(ELEMENTS)` refers to the elements of the FavoriteColors property, since it is a collection. The general form is *propertyName*`(ELEMENTS)` or *propertyName*`(KEYS)`, where that collection's content is the set of elements contained in a property defined as a `List Of` or an `Array Of` some data type). For information on collections, see the chapter "Working with Collections" in *Using Caché Objects*.

To index literal properties (described in the "Defining and Using Literal Properties" chapter of *Using Caché Objects*), you can create an index value array as produced by a *propertyName***BuildValueArray()** method (described in the following section). As with collections proper, the `(ELEMENTS)` and `(KEYS)` syntax is valid with index value arrays.

If property-collection is projected as array, then the index must obey the following restrictions in order to be projected to the collection table. The index must include (KEYS). The index cannot reference any properties other than the collection itself and the object's ID value. If a projected index also defines DATA to be stored in the index, then the data properties stored must also be restricted to the collection and the ID. Otherwise the index is not projected. This restriction applies to an index on a collection property that is projected as an array; it does not apply to an index on a collection that is projected as a list. For further details, refer to "Controlling the SQL Projection of Collection Properties" in *Using Cache Objects*.

Indices that correspond to element or key values of a collection can also have all the standard index features, such as storing data with the index, index-specific collations, and so on.

Indexing Data Type Properties with (ELEMENTS) and (KEYS)

For the purposes of indexing data type properties, you can also create index value arrays using the **BuildValueArray()** method. This method parses a property value into an array of keys and elements; it does this by producing a collection of element values derived from the value of the property with which it is associated. When you use **BuildValueArray()** to create an index value array, its structure is suitable for indexing.

The **BuildValueArray()** method has the name *propertyName***BuildValueArray()** and its signature is:

```
ClassMethod propertynameBuildValueArray(value, ByRef valueArray As %Library.String) As %Status
```

where

- The name of the **BuildValueArray()** method derives from the property name in the typical way for composite methods.
- The first argument is the property value.
- The second argument is an array that is passed by reference. This is an array containing key-element pairs where the array subscripted by the key is equal to the element.
- The method returns a %Status value.

Consider this example:

```
/// DescriptiveWords is a comma-delimited string of words
Property DescriptiveWords As %String;

/// Index based on DescriptiveWords
Index dwIDX On DescriptiveWords(ELEMENTS);

/// The DescriptiveWordsBuildValueArray() method demonstrates how to index on subvalues of a property.
///
```

```
/// (If DescriptiveWords were defined as a collection, this method would not be necessary.)
ClassMethod DescriptiveWordsBuildValueArray(
     Words As %Library.String = "",
     ByRef wordArray As %Library.String)
   As %Status {
 If Words '= "" {
   For tPointer = 1:1:$Length(Words,",") {
     Set tWord = $Piece(Words,",",tPointer)
     If tWord '= "" {
       Set wordArray(tPointer) = tWord
     }
   }
 }
 Else {
   Set wordArray("TODO") = "Enter keywords for this person"
 }
 Quit $$$OK
}
```

In this case, the dwIDX index is based on the DescriptiveWords property. The **DescriptiveWordsBuildValueArray()** method takes the value specified by the *Words* argument, creates an index value array based on that value, and stores it in *wordArray*. Caché uses the implementation of **BuildValueArray()** internally; you do not call this method.

Note: It is not necessary to base any of the element/key values on the property value. The only recommendation is that the same array of elements and keys be created each time this method is passed a given value.

Setting values for DescriptiveWords property of various instances and examining those values involves activity such as the following:

```
SAMPLES>SET empsalesoref = ##class(MyApp.Salesperson).%OpenId(3)

SAMPLES>SET empsalesoref.DescriptiveWords = "Creative"

SAMPLES>WRITE empsalesoref.%Save()
1
SAMPLES>SET empsalesoref = ##class(MyApp.Salesperson).%OpenId(4)

SAMPLES>SET empsalesoref.DescriptiveWords = "Logical,Tall"

SAMPLES>WRITE empsalesoref.%Save()
1
```

This results in sample index content such as:

DescriptiveWords(ELEMENTS)	ID	Data
" CREATIVE"	3	""
" ENTER KEYWORDS FOR THIS PERSON"	1	""
" ENTER KEYWORDS FOR THIS PERSON"	2	""
" LOGICAL"	4	""
" TALL"	4	""

Note: This table displays index content in an abstracted form. The actual form of storage on disk is subject to change.

Projecting an Index on array(ELEMENTS) to a Child Table

In order to project a parent table index on array(ELEMENTS) to a child table, the child class/table must have all of the necessary columns to properly maintain the index. It must include information about the key, which is part of the child table RowID. Every index row must provide complete RowID information, which enables returning to the corresponding master map row.

Lacking this complete RowID information, an **INSERT** to a child table cannot populate the associated parent table index on array(ELEMENTS).

2.3.1.9 Indexing an Embedded Object (%SerialObject) Property

To index a property in an embedded object, you create an index in the persistent class referencing that embedded object. The property name must specify the name of the referencing field in the table (%Persistent class) and the property in the embedded object (%SerialObject), as shown in the following example:

```
Class Sample.Person Extends (%Persistent) [ DdlAllowed ]
{  Property Name As %String(MAXLEN=50);
   Property Home As Sample.Address;
   Index StateInx On Home.State;
}
```

Here *Home* is a property in Sample.Person that references the embedded object Sample.Address, which contains the *State* property, as shown in the following example:

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

Only the data values in the instance of the embedded object associated with the persistent class property reference are indexed. You cannot index a %SerialObject property directly.

You can also define an index on an embedded object property using the SQL CREATE INDEX statement, as shown in the following example:

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

For further details, refer to Introduction to Serial Objects in *Defining and Using Classes* and Embedded Object (%SerialObject) in the "Defining Tables" chapter of *Using Caché SQL*.

2.3.1.10 Notes on Indices Defined in Classes

When working with indices in class definitions, here are some points to keep in mind:

- Index definitions are only inherited from the primary (first) superclass.
- If you use Studio to add (or remove) an index definition for a class that has data stored within the database, you must manually populate the index by using one of the procedures described in "Building Indices."

2.3.2 Defining Indices Using DDL

If you are using DDL statements to define tables, you can also use the following DDL commands to create and remove indices:

- CREATE INDEX
- DROP INDEX

The DDL index commands do the following:

1. They update the corresponding class and table definitions on which an index is being added or removed. The modified class definition is recompiled.
2. They add or remove index data in the database as needed: The CREATE INDEX command populates the index using the data currently stored within the database. Similarly, the DROP INDEX command deletes the index data (that is, the actual index) from the database.

2.4 Bitmap Indices

A bitmap index is a special type of index that uses a series of bitstrings to represent the set of ID values that correspond to a given indexed data value. You can define a bitmap index for a field if the table's ID field is defined as a positive integer (see restrictions).

Bitmap indices have the following important features:

- Bitmaps are highly compressed: bitmap indices can be significantly smaller than standard indices. This reduces disk and cache usage considerably.
- Bitmaps operations are optimized for transaction processing: you can use bitmap indices within tables with no performance penalty as compared with using standard indices.
- Logical operations on bitmaps (counting, AND, and OR) are optimized for high performance.
- The SQL Engine includes a number of special optimizations that can take advantage of bitmap indices.

Subject to the restrictions listed below, bitmap indices operate in the same manner as standard indices. Indexed values are collated and you can index on combinations of multiple fields.

This chapter addresses the following topics related to bitmap indices:

- Bitmap Index Operation
- Defining Bitmap Indices by Using a Class Definition
- Defining Bitmap Indices Using DDL
- Generating a Bitmap Extent Index
- Choosing an Index Type
- Restrictions on Bitmap Indices
- Maintaining Bitmap Indices
- SQL Manipulation of Bitmap Chunks

2.4.1 Bitmap Index Operation

Bitmap indices work in the following way. Suppose you have a Person table containing a number of columns:

Figure 2–1: Person Table

Person					
RowID	Name	Age	State	Job	...
1	Smith	24	NY	Lawyer	...
2	Jones	35	NY	Doctor	...
3	Presley	48	CA	Farmer	...
4	Nixon	72	NY	Singer	...
...	...	...	...	...	...

Each row in this table has a system-assigned ID number (a set of increasing integer values). A bitmap index uses a set of bitstrings (a string containing 1 and 0 values). Within a bitstring, the ordinal position of a bit corresponds to the ID (row number) of the indexed table. For a given value, say where State is "NY", there is a string of bits with a 1 for every position that corresponds to a row containing "NY" and a 0 in every other position.

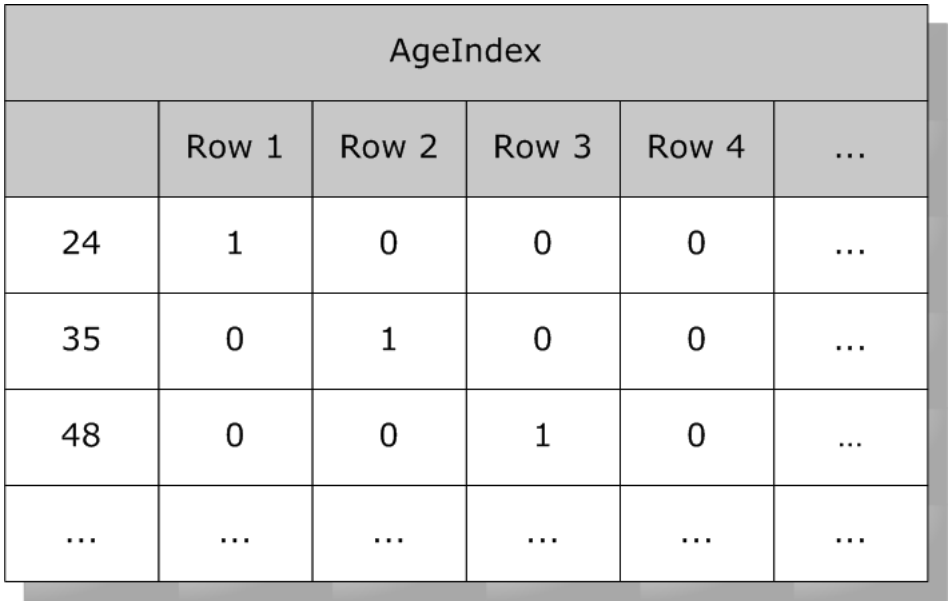
For example, a bitmap index on State might look like this:

Figure 2–2: State Bitmap Index

StateIndex					
	Row 1	Row 2	Row 3	Row 4	...
CA	0	0	1	0	...
NY	1	1	0	1	...
WY	0	0	0	0	...
...	...	...	...	...	...

While an index on Age might look like this:

Figure 2–3: Age Bitmap Index

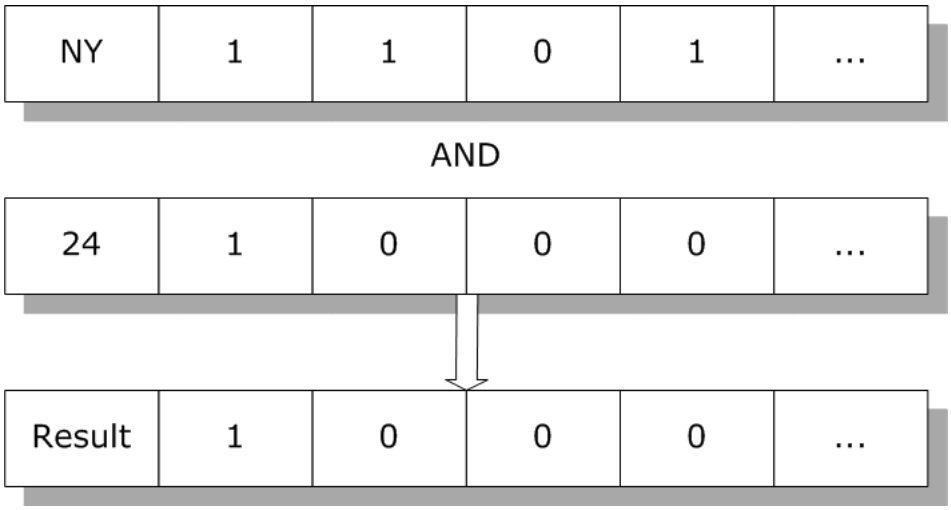


AgeIndex					
	Row 1	Row 2	Row 3	Row 4	...
24	1	0	0	0	...
35	0	1	0	0	...
48	0	0	1	0	...
...	...	...	...	...	...

Note: The Age field shown here can be an ordinary data field or a field whose value can be reliably derived (Calculated and SQLComputed).

In addition to using bitmap indices for standard operations, the SQL engine can use bitmap indices to efficiently perform special set-based operations using combinations of multiple indices. For example, to find all instances of Person that are 24 years old and live in New York, the SQL Engine can simply perform the logical AND of the Age and State indices:

Figure 2–4: Using Multiple Indices



The resulting bitmap contains the set of all rows that match the search criteria. The SQL Engine uses this to return data from these rows.

The SQL Engine can use bitmap indices for the following operations:

- ANDing of multiple conditions on a given table.
- ORing of multiple conditions on a given table.
- RANGE conditions on a given table.

- COUNT operations on a given table.

2.4.2 Defining a Bitmap Index Using a Class Definition

Within Studio, you can add bitmap index definitions to a class definition using either the New Index Wizard or by editing the text of the class definition in the same way that you would create a standard index. The only difference is that you need to specify the index Type as being "bitmap":

```
Class MyApp.SalesPerson Extends %Persistent [DdlAllowed]
{
 Property Name As %String;
 Property Region As %Integer;

 Index RegionIDX On Region [Type = bitmap];
}
```

2.4.3 Defining Bitmap Indices Using DDL

If you are using DDL statements to define tables, you can also use the following DDL commands to create and remove bitmap indices:

- CREATE INDEX
- DROP INDEX

This is identical to creating standard indices, except that you must add the BITMAP keyword to the CREATE INDEX statement:

```
CREATE BITMAP INDEX RegionIDX ON TABLE MyApp.SalesPerson (Region)
```

2.4.4 Generating a Bitmap Extent Index

When compiling a class that contains a bitmap index, the class compiler generates a bitmap extent index if there are any bitmap indices present in the class and no bitmap extent index is defined for that class. The class inherits the bitmap extent index from the primary superclass if it exists, either defined or generated. When building indices for a class, the bitmap extent index is built either if it is asked to be built or if another bitmap index is being built and the bitmap extent index structure is empty.

Caché does not generate a bitmap extent index unless there are bitmap indices present. A bitmap extent index is defined as: `type = bitmap`, `extent = true`. That means a bitmap extent index inherited from a primary superclass is considered to be a bitmap index and will trigger a bitmap extent index to be generated in the subclass, if no bitmap extent index is explicitly defined in that subclass.

Caché does not generate a bitmap extent index in a superclass based on future possibility. This means that Caché does not ever generate a bitmap extent index in a persistent class unless an index whose `type = bitmap` is present. A presumption that some future subclass might introduce an index with `type = bitmap` is not sufficient.

Note: Special care is required during the process of adding a bitmap index to a class on a production system (where users are actively using a particular class, compiling said class, and subsequently building the bitmap index structure for it). On such a system, the bitmap extent index may be populated in the interim between the compile completing and the index build proceeding. This can cause the index build procedure to not implicitly build the bitmap extent index, which leads to a partially complete bitmap extent index.

2.4.5 Choosing an Index Type

The following is a general guideline for choosing between bitmap and standard indices. In general, use standard indices for indexing on all types of keys and references:

- Primary key
- Foreign key
- Unique keys
- Relationships
- Simple object references

Otherwise, bitmap indices are generally preferable (assuming that the table uses system-assigned numeric ID numbers).

Other factors:

- Separate bitmap indices on each property usually have better performance than a bitmap index on multiple properties. This is because the SQL engine can efficiently combine separate bitmap indices using AND and OR operations.
- If a property (or a set of properties that you really need to index together) has more than 10,000-20,000 distinct values (or value combinations), consider standard indices. If, however, these values are very unevenly distributed so that a small number of values accounts for a substantial fraction of rows, then a bitmap index could be much better. In general, the goal is to reduce the overall size required by the index.

2.4.6 Restrictions on Bitmap Indices

Bitmap indices have the following restrictions:

- You can only define a bitmap index for a field in a record that has a positive integer value ID. Therefore, you can only define bitmap indices in tables (classes) that either use system-assigned numeric ID values, or use an IdKey to define custom ID values when the IdKey is based on a single property with type %Integer and MINVAL > 0, or type %Numeric with SCALE = 0 and MINVAL > 0.

  You can use the **$SYSTEM.SQL.SetBitmapFriendlyCheck()** method to set a system-wide configuration parameter to check at compile time for this restriction, determining whether a defined bitmap index is allowed in a %CacheSQLStorage class. This check only applies to classes that use %CacheSQLStorage. You can use **$SYSTEM.SQL.GetBitmapFriendlyCheck()** to determine the current configuration of this option.
- You cannot define a bitmap index for a field in a record that uses a multi-field ID key.
- You cannot define a bitmap index on a UNIQUE column.
- You cannot use bitmap indices on child tables within parent-child relationships.
- You cannot store data values within a bitmap index.

For a table containing more than 1 million records, a bitmap index is less efficient than a standard index when the number of unique values exceeds 10,000. Therefore, for a large table it is recommended that you avoid using a bitmap index for any field that contains (or is likely to contain) more than 10,000 unique values; for a table of any size, avoid using a bitmap index for any field that is likely to contain more than 20,000 unique values. These are general approximations, not exact numbers.

2.4.6.1 Application Logic Restrictions

A bitmap structure can be represented by an array of bit strings, where each element of the array represents a "chunk" with a fixed number of bits. Because undefined is equivalent to a chunk with all 0 bits, the array can be sparse. An array element

that represents a chunk of all 0 bits need not exist at all. For this reason, application logic should avoid depending on the $BITCOUNT(str,0) count of 0-valued bits.

Because a bit string contains internal formatting, application logic should never depend upon the physical length of a bit string or upon equating two bit strings that have the same bit values. Following a rollback operation, a bit string is restored to its bit values prior to the transaction. However, because of internal formatting, the rolled back bit string may not equate to or have the same physical length as the bit string prior to the transaction.

2.4.7 Maintaining Bitmap Indices

In a volatile table (one that undergoes many **INSERT** and **DELETE** operations) the storage for a bitmap index can gradually become less efficient. To maintain bitmap indices, you can run the %SYS.Maint.Bitmap utility methods to compress the bitmap indices, restoring them to optimal efficiency. You can use the **OneClass()** method to compress the bitmap indices for a single class. Or you can use the **Namespace()** method to compress the bitmap indices for an entire namespace. These maintenance methods can be run on a live system.

The results of running the %SYS.Maint.Bitmap utility methods are written to the process that invoked the method. These results are also written to the class %SYS.Maint.BitmapResults.

2.4.8 SQL Manipulation of Bitmap Chunks

InterSystems SQL provides the following extensions to directly manipulate bitmap indices:

- %CHUNK function
- %BITPOS function
- %BITMAP aggregate function
- %BITMAPCHUNK aggregate function
- %SETINCHUNK predicate condition

All of these extensions follow the InterSystems SQL conventions for bitmap representation, representing a set of positive integers as a sequence of bitmap chunks, of up to 64,000 integers each.

These extensions enable easier and more efficient manipulation of certain conditions and filters, both within a query and in embedded SQL. In embedded SQL they enable simple input and output of bitmaps, especially at the single chunk level. They support the processing of complete bitmaps, which are handled by %BITMAP() and the %SQL.Bitmap class. They also enable bitmap processing for non-RowID values, such as foreign key values, parent-reference of a child table, either column of an association, etc.

For example, to output the bitmap for a specified chunk:

```
SELECT %BITMAPCHUNK(Home_Zip) FROM Sample.Person
WHERE %CHUNK(Home_Zip)=2
```

To output all the chunks for the whole table:

```
SELECT %CHUNK(Home_Zip),%BITMAPCHUNK(Home_Zip) FROM Sample.Person
GROUP BY %CHUNK(Home_Zip) ORDER BY 1
```

2.4.8.1 %CHUNK function

%CHUNK(f) returns the chunk assignment for a bitmap indexed field *f* value. This is calculated as f\64000+1. **%CHUNK(f)** for any field or value *f* that is not a bitmap indexed field always returns 1.

2.4.8.2 %BITPOS function

%BITPOS(f) returns the bit position assigned to a bitmap indexed field *f* value within its chunk. This is calculated as f#64000+1 . **%BITPOS(f)** for any field or value *f* that is not a bitmap indexed field returns 1 more than its integer value. A string has an integer value of 0.

2.4.8.3 %BITMAP aggregate function

The aggregate function **%BITMAP(f)** combines many *f* values into a %SQL.Bitmap object, in which the bit corresponding to *f* in the proper chunk is set to 1 for each value *f* in the result set. *f* in all of the above would normally be a positive integer field (or expression), usually (but not necessarily) the RowID.

2.4.8.4 %BITMAPCHUNK aggregate function

The aggregate function %BITMAPCHUNK(f) combines many values of the field *f* into an InterSystems SQL standard bitmap string of 64,000 bits, in which bit f#64000+1=%BITPOS(f) is set to 1 for each value f in the set. Note that the bit is set in the result regardless of the value of %CHUNK(f) . %BITMAPCHUNK() yields NULL for the empty set, and like any other aggregate it ignores NULL values in the input.

2.4.8.5 %SETINCHUNK predicate condition

The condition (f %SETINCHUNK bm) is true if and only if ($BIT(bm,%BITPOS(f))=1) . bm could be any bitmap expression string, e.g. an input host variable :bm, or the result of a %BITMAPCHUNK() aggregate function, etc. Note that the <bm> bit is checked regardless of the value of %CHUNK(f) . If <bm> is not a bitmap or is NULL, the condition returns FALSE. (f %SETINCHUNK NULL) yields FALSE (not UNKNOWN).

2.5 Bitslice Indices

A bitslice index is used for a numeric data field when that field is used for a **SUM**, **COUNT**, or **AVG** aggregate calculation. A bitslice index represents each numeric data value as a binary bit string. Rather than indexing a numeric data value using a boolean flag (as in a bitmap index), a bitslice index represents each value in binary and creates a bitmap for each digit in the binary value to record which rows have a 1 for that binary digit. This is a highly specialized type of index that can substantially improve performance of **SUM**, **COUNT**, or **AVG** aggregate calculations. (A bitslice index is *not* used for **COUNT(*)** calculations.) Bitslice indices are not used for other aggregate functions or other SQL numeric operations. The SQL optimizer determines whether a defined bitslice index should be used.

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

In the following example, Salary would be a candidate for a bitslice index:

```
SELECT AVG(Salary) FROM SalesPerson
```

A bitslice index can be used for an aggregate calculation in a query that uses a WHERE clause. This is most effective if the WHERE clause is inclusive of a large number of records. In the following example, the SQL optimizer would probably use a bitslice index on Salary, if defined; if so, it would also use a bitmap index on Region, either using a defined bitmap or generating a bitmap tempfile for Region:

```
SELECT AVG(Salary) FROM SalesPerson WHERE Region=2
```

However, a bitslice index is not used when the WHERE condition cannot be satisfied by an index, but must be performed by reading the table that contains the field being aggregated. The following example would not use the bitslice index on Salary:

```
SELECT AVG(Salary) FROM SalesPerson WHERE Name LIKE '%Mc%'
```

A bitslice index can be defined for any field containing numeric values. InterSystems SQL uses a scale parameter to convert fractional numbers into bitstrings, as described in the ObjectScript $FACTOR function. A bitslice index can be defined for a field of data type string; in this case, non-numeric string data values are treated as 0 for the purposes of the bitslice index.

A bitslice index can only be defined for fields in records that have system-assigned row Ids with positive integer values. A bitslice index can only be defined for a single field name, not a concatenation of multiple fields. You cannot specify a WITH DATA clause.

The following example compares a bitslice index to a bitmap index. If you create a bitmap index for values 1, 5, and 22 for rows 1, 2, and 3, it creates an index for the values:

```
^gloI("bitmap",1,1)= "100"
^gloI("bitmap",5,1)= "010"
^gloI("bitmap",22,1)="001"
```

If you create a bitslice index for values 1, 5, and 22 for rows 1, 2, and 3, it first converts the values to bit values:

```
1 =   00001
5 =   00101
22 = 10110
```

It then creates an index for the bits:

```
^gloI("bitslice",1,1)="110"
^gloI("bitslice",2,1)="001"
^gloI("bitslice",3,1)="011"
^gloI("bitslice",4,1)="000"
^gloI("bitslice",5,1)="001"
```

In this example, the value 22 in a bitmap index required setting 1 global node; the value 22 in a bitslice index required setting 3 global nodes.

Note that an **INSERT** or **UPDATE** requires setting a bit in all *n* bitslices, rather than setting a single bitstring. These additional global set operations can affect performance of **INSERT** and **UPDATE** operations that involve populating bitslice indices. Populating and maintaining a bitslice index using **INSERT**, **UPDATE**, or **DELETE** operations is slower than populating a bitmap index or a regular index. Maintaining multiple bitslice indices, and/or maintaining a bitslice index on a field that is frequently updated may have a significant performance cost.

In a volatile table (one that undergoes many **INSERT**, **UPDATE**, and **DELETE** operations) the storage for a bitslice index can gradually become less efficient. The %SYS.Maint.Bitmap utility methods compress both bitmap indices and bitslice indices, restoring efficiency. For further details, see "Maintaining Bitmap Indices".

2.6 Building Indices

The current database access determines how you should rebuild an existing index:

- Inactive System (no other processes accessing data during index build or rebuild)
- READONLY active system (other processes able to query the data during index build or rebuild)
- READ and WRITE active system (other processes able to modify the data and to query the data during index build or rebuild)

2.6.1 Building Indices on an Inactive System

The system automatically generates methods (provided by the %Persistent class) that build (that is, provide values for) or purge (that is, remove values for) every index defined for a class (table). You can use these methods in either of two ways:

- Interactively, via the Management Portal.
- Programmatically, as method calls.

Building an index does the following:

1. Removes the current contents of the index.
2. Scans (reads every row) of the main table and adds index entries for each row in the table. If possible, the special $SortBegin and $SortEnd functions are used to make sure that building of large indices is efficient. When building a standard index, this use of **$SortBegin**/**$SortEnd** can use space in the CACHETEMP database, in addition to caching data in memory. Therefore, when building a very large standard index, Caché can require space in CACHETEMP roughly equivalent to the size of the final index.

Note: Methods to build indices are only provided for classes (tables) that use Caché default storage structure. Classes mapped to legacy storage structures do not support index building as it assumed that the legacy application manages the creation of indices.

2.6.1.1 Building Indices with the Management Portal

You can build existing indices (rebuild indices) for a table by doing the following:

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. After selecting a namespace, select the **Schema** drop-down list on the left side of the screen. This displays a list of the schemas in the current namespace with boolean flags indicating whether there are any tables or any views associated with each schema.
2. Select a schema from this list; it appears in the **Schema** box. Just above it is a drop-down list that allows you to select Tables, System Tables, Views, Procedures, or All of these that belong to the schema. Select either Tables or All, then open the Tables folder to list the tables in this schema. If there are no tables, opening the folder displays a blank page. (If you have not selected Tables or All, opening the Tables folder lists the tables for the entire namespace.)
3. Select one of the listed Tables. This displays the **Catalog Details** for the table.
   - To rebuild all indices: click the **Actions** drop-down list and select **Rebuild Table's Indices**.
   - To rebuild a single index: click the **Indices** button to display the existing indices. Each listed index has the option to **Rebuild Index**.

CAUTION: Do not rebuild indices while the table's data is being accessed by other users. To rebuild indices on an active system, see below.

2.6.1.2 Building Indices Programmatically

The preferred way of building indices for an inactive table is to use the **%BuildIndices()** method provided with the persistent class for the table. To build an index (or indices) programmatically, use the **%Library.Persistent.%BuildIndices()** method.

- Build All Indices: Invoking **%BuildIndices()** with no arguments builds (provides values for) all the indices defined for a given class (table):

```
SET sc = ##class(MyApp.SalesPerson).%BuildIndices()
IF sc=1 {WRITE !,"Successful index build" }
ELSE {WRITE !,"Index build failed",!
      DO $System.Status.DisplayError(sc) QUIT}
```

- Build Specified Indices: Invoking **%BuildIndices()** with a **$List** of index names as the first argument builds (provides values for) the specified defined indices for a given class (table):

```
SET sc = ##class(MyApp.SalesPerson).%BuildIndices($ListBuild("NameIDX","SSNKey"))
IF sc=1 {WRITE !,"Successful index build" }
ELSE {WRITE !,"Index build failed",!
      DO $System.Status.DisplayError(sc) QUIT}
```

- Build All Indices Except: Invoking **%BuildIndices()** with a **$List** of index names as the seventh argument builds (provides values for) all defined indices for a given class (table) except for the specified indices:

```
SET sc = ##class(MyApp.SalesPerson).%BuildIndices("",,,,,,$ListBuild("NameIDX","SSNKey"))
IF sc=1 {WRITE !,"Successful index build" }
ELSE {WRITE !,"Index build failed",!
      DO $System.Status.DisplayError(sc) QUIT}
```

The **%BuildIndices()** method does the following:

1. Invokes the $SortBegin function on any (non-bitmap) indices to be rebuilt (this initiates a high performance sorting operation for these indices).
2. Loops over the main data for the class (table), gathers the values used by indices, and adds these values to the indices (with appropriate collation transformation).
3. Invokes the $SortEnd function to finish the process of sorting the indices.

If the indices already have values, you must call **%BuildIndices()** with two arguments, where the second argument has a value of 1. Specifying 1 for this argument causes the method to purge the values before rebuilding them. For example:

```
SET sc = ##class(MyApp.SalesPerson).%BuildIndices(,1)
IF sc=1 {WRITE !,"Successful index build" }
ELSE {WRITE !,"Index build failed",!
      DO $System.Status.DisplayError(sc) QUIT}
```

which purges and rebuilds all the indices. You can also purge and rebuild a subset of the indices, such as in:

```
SET sc = ##class(MyApp.SalesPerson).%BuildIndices($ListBuild("NameIDX","SSNKey"),1)
IF sc=1 {WRITE !,"Successful index build" }
ELSE {WRITE !,"Index build failed",!
      DO $System.Status.DisplayError(sc) QUIT}
```

CAUTION: Do not rebuild indices while the table's data is being accessed by other users. To rebuild indices on an active system, see below.

%BuildIndices() also provides arguments that specify the kind of locking to perform and whether to disable journaling during lock building.

By default, **%BuildIndices()** builds index entries for all IDs. However, you can specify a range of IDs and **%BuildIndices()** will build the index entries only for IDs in that range, inclusive. For example, if you use INSERT with the %NOINDEX restriction to add a series of new records into a table, you can later use **%BuildIndices()** with an ID range to build index entries for those new records.

%BuildIndices() returns a %Status value. If **%BuildIndices()** fails due to a problem retrieving data, the system generates an SQLCODE error and a message (%msg) that include the %ROWID where the error was encountered.

2.6.2 Building Indices on a READONLY Active System

If a table is currently in use for query operations only (READONLY), you can build new indices or rebuild existing indices without interrupting query operations. If all the classes for which you wish to build one or more indices are currently READONLY, use the same series of operations described in "Building Indices on a READ and WRITE Active System", with the following difference: when you use **%BuildIndices()** set *pLockFlag*=3 (shared extent lock).

2.6.3 Building Indices on a READ and WRITE Active System

If a persistent class (table) is currently in use and is available for READ and WRITE access (query and data modification), you can build new indices or rebuild existing indices without interrupting these operations. If the class for which you wish to rebuild one or more indices is currently READ and WRITE accessible, the preferred way of building indices is to use the **%BuildIndices()** method provided with the persistent class for the table.

CAUTION: The following information applies to Dynamic SQL queries, not Embedded SQL. Embedded SQL checks the MapSelectability setting when it's compiled, not at runtime. Therefore, turning off MapSelectability for an index does not have any effect on Embedded SQL queries which have already been compiled. As a result, Embedded SQL queries could still try to use the disabled index and will give incorrect results.

The following is the preferred series of operations for building one or more indices during concurrent READ and WRITE access:

1. Make the index that you wish to build unavailable to queries (READ access). This is done using **SetMapSelectability()**. This makes the index unavailable for use by the Query Optimizer. This operation should be performed both when rebuilding an existing index and when creating a new index. For example:

   ```
   WRITE $SYSTEM.SQL.SetMapSelectability("Sample.MyStudents","StudentNameIDX",0)
   ```

   where:

   - The first argument is the Schema.Table name, which is the SqlTableName, not the persistent class name. For example, the default schema is SQLUser, not User. This value is case-sensitive.
   - The second argument is the SQL index map name. This is typically the name of the index, and refers to the name under which the index is stored on disk. For a new index, this is the name you will use when you create the index. This value is not case-sensitive.
   - The third argument is the MapSelectability flag, where 0 defines the index map as non-selectable (off), and 1 defines the index map as selectable (on). Specify 0.

   You can determine if an index is non-selectable by invoking the **GetMapSelectability()** method. This method returns 0 if you have explicitly flagged an index as non-selectable. In all other cases it returns 1; it does not perform validation checks for the existence of the table or the index. Note that the Schema.Table name is the SqlTableName, and is case-sensitive.

   SetMapSelectability() and **GetMapSelectability()** apply to index maps in the current namespace only. If this table is mapped to multiple namespaces, and the index needs to be built in each namespace, **SetMapSelectability()** should be called in each namespace.

2. Establish concurrent operations for the duration of the index build:

   - For a new index: Create the index definition in the class (or the new SQL Index Map specification in the %CacheSQLStorage of the class). Compile the class. At this point, the index exists in the table definition; this means that object saves, SQL INSERT operations, and SQL UPDATE operations are all filed in the index. However, because of the **SetMapSelectability()** call made in step 1, this index map is not chosen for any data retrieval. **SetMapSelectability()** prevents queries from using the extent index, but the data map will be projected to SQL

to use the index global and the data global. For a new index this is appropriate because the index has not yet been populated. Population of the extent index needs to occur before queries can be run against the table.

- For an existing index: Purge any cached queries that reference the table. The first operation that the index build performs is to kill the index. Therefore, you cannot rely on any code that is optimized to use the index while the index is being rebuilt.

3. Use the **%BuildIndices()** method of your class with *pLockFlag*=2 to build the index or indices.
4. Once you have completed building the index, enable the map for selectability by the Query Optimizer. Set the third argument, the MapSelectability flag to 1, as shown in the following example:

```
WRITE $SYSTEM.SQL.SetMapSelectability("Sample.MyStudents","StudentNameIDX",1)
```

5. Once again, purge any cached queries that reference the table. This will eliminate cached queries that were created during this process that could not use the index, and are thus less optimal than the same queries using the index.

This completes the process. The index is fully populated and the Query Optimizer is able to consider the index.

Note: **%BuildIndices()** can only be used to rebuild indices for tables that have positive integer ID values. You can also use **%BuildIndices()** to rebuild indices in a child table if the parent table has positive integer ID values. For other tables, use the **%ValidateIndices()** method as described in Validate Indices. Because **%ValidateIndices()** is the slowest method for building indices, it should only be used when there is no other option.

2.6.3.1 The %ConstructIndicesParallel() Method

The **%ConstructIndicesParallel()** method is a class method of the %Library.IndexBuilder class that builds the indices specified by the *INDEXBUILDERFILTER* parameter. It allows you to specify which sets of instances to build indices for (based on ID/row number of instances in the class/table). It also allows you to specify the number of background jobs for populating the index, as well as locking and journaling behavior during its operation.

%ConstructIndicesParallel() requires changes to the class to inherit from %Library.IndexBuilder and define the *INDEXBUILDERFILTER* class parameter. The *INDEXBUILDERFILTER* class parameter is recognized by the **%Library.IndexBuilder.%ConstructIndicesParallel()** method and specifies which indices to build or rebuild. *INDEXBUILDERFILTER* is a string value of an index name or a comma-separated list of index names. Each specified index is built or rebuilt; if *INDEXBUILDERFILTER* is not defined or set is to "", then all the indices in the class are built.

For example, suppose you add the NameIDX index to class MyApp.SalesPerson and would like to build only that index. You would then define the *INDEXBUILDERFILTER* parameter as follows:

```
Parameter INDEXBUILDERFILTER = "NameIDX";
```

When building indices on an active READ/WRITE system the following argument values are mandatory: either (Sortbegin=0 AND LockFlag=2) or (LockFlag=1). See the class documentation for **%ConstructIndicesParallel()** for details on its arguments.

%ConstructIndicesParallel() displays output regarding its progress. When all jobs have completed their work, the method returns. Here is an example of the output:

```
SAMPLES>SET sc=##class(MyApp.SalesPerson).%ConstructIndicesParallel(,,,1,,1,1)
Building 32 chunks and will use parallel build algorithm with 2 drone processes.
SortBegin is requested.
Started drone process: 48938
Started drone process: 48939
Expected time to complete is 17 secs to build 32 chunks of 64,000 objects using 2 processes.
Waiting for processes to complete....done.
Elapsed time using 2 processes was 28.126064.
SAMPLES>
```

Note: Large-scale data modification during index rebuild: If a process is modifying large numbers of rows in the table within a single transaction while an index rebuild is in progress, lock table contention issues may arise.

Populating an Index in Sections

When creating an index of this kind, you do not need to populate the entire index at once. Prior to enabling the map for selectability, the index is not visible, so queries will not operate on its partial data.

For example, suppose you can only run the index builder for a short period each night. If there are 300,000,000 rows in the table, you can build 100,000,000 rows each night over a period of three nights. On the first night, you would make a call such as:

```
SET sc=##class(Sample.Person).%ConstructIndicesParallel(,1,100000000,0,0,2,0)
```

On the second night, you would then make this call:

```
SET sc=##class(Sample.Person).%ConstructIndicesParallel(,100000001,200000000,0,0,2,0)
```

And on the third night, you would make this call:

```
SET sc=##class(Sample.Person).%ConstructIndicesParallel(,200000001,-1,0,0,2,0)
```

2.7 Validating Indices

You can validate indices using the either of the following methods:

- **$SYSTEM.OBJ.ValidateIndices()** validates the indices for a table, and also validates any indices in collection child tables for that table.
- **%Library.Storage.%ValidateIndices()** validates the indices for a table. Collection child table indices must be validated with separate **%ValidateIndices()** calls.

Both methods check the data integrity of one or more indices for a specified table, and optionally correct any index integrity issues found. They perform index validation in two steps:

1. Confirm that an index entity is properly defined for every row (object) in the table (class).
2. Traverse each index and for every entry indexed, make sure there is a value and matching entry in the table (class).

If either method finds discrepancies, it can correct the index structure and/or contents. It can validate, and optionally correct, standard indices, bitmap indices, bitmap extent indices, and bitslice indices.

%ValidateIndices() is commonly run from the Terminal. It displays output to the current device. This method can be applied to a specified %List of index names, or to all indices defined for the specified table (class). It operates only on those indices that originated in specified class; if an index originated in a superclass, that index can be validated by calling **%ValidateIndices()** on the superclass. **%ValidateIndices()** is not supported for READONLY classes.

The following example uses **%ValidateIndices()** to validate and correct all indices for the table Sample.Person:

```
ZNSPACE "Samples"
SET status=##class(Sample.Person).%ValidateIndices("",1,2,1)
IF status=1 {WRITE !,"Successful index validation/correction" }
ELSE {WRITE !,"Index validation/correction failed",!
      DO $System.Status.DisplayError(status) QUIT}
```

In this example, the first argument ("") specifies that all indices are to be validated; the second argument (1) specifies that index discrepancies should be corrected; the third argument (2) specifies exclusive locking on the entire table; and the fourth argument (1) specifies using multiple processes (if available) to perform validation. The method returns a %Status value.

2.7.1 Validating Indices by Name

The first argument of **%ValidateIndices()** or the second argument of **$SYSTEM.OBJ.ValidateIndices()** specifies which indices are to be validated as a %List structure. The IdKey index is always validated, regardless of the first argument value. You can validates all of the table's indices by specifying an empty string value (""). You can validates individual indices for the table by specifying a list structure. The following example validates the IdKey index and the two specified indices: NameIDX and SSNKey:

```
ZNSPACE "Samples"
SET IndList=$LISTBUILD("NameIDX","SSNKey")
SET status=##class(Sample.Person).%ValidateIndices(IndList,1,2,1)
SET status=##class(Sample.Person).%ValidateIndices(IndList,1,2,1)
IF status=1 {WRITE !,"Successful index validation/correction" }
ELSE {WRITE !,"Index validation/correction failed",!
      DO $System.Status.DisplayError(status) QUIT}
```

For either method, if the index list contains a non-existent index name, the method performs no index validation and returns a %Status error. If the index list contains a duplicate valid index name, the method validates the specified indices, ignoring the duplicate and issuing no error.

2.8 Listing Indices

The INFORMATION.SCHEMA.INDEXES persistent class displays information about all column indices in the current namespace. It returns one record for each indexed column. It provides a number of index properties, including the name of the index and the schema name, table name, and column name that the index maps to. Each column record also provides the ordinal position of that column in the index map; this value is 1 unless the index maps to multiple columns.

The following example returns the index name, and the corresponding table schema name, table name, and column name, and the ordinal position of the column in the index definition for all non-system indices in the current namespace:

```
SELECT Index_Name,Table_Schema,Table_Name,Column_Name,Ordinal_Position FROM INFORMATION_SCHEMA.INDEXES
 WHERE NOT Table_Schema %STARTSWITH '%'
```

2.9 Open, Exists, and Delete Methods

The Caché indexing facility supports the following operations:

- Opening an Instance by Index Key
- Checking If an Instance Exists
- Deleting an Instance

2.9.1 Opening an Instance by Index Key

For ID key, primary key, or unique indices, the *indexname***Open()** method (where *indexname* is the name of the index) allows you to open the object whose index property value or values match supplied value or values. Because this method has one argument corresponding to each property in the index, the method has three or more arguments:

- The first argument(s) each correspond to the properties in the index.
- The penultimate argument specifies the concurrency value with which the object is to be opened (with the available concurrency settings listed in the "Object Concurrency" appendix of *Using Caché Objects*).

- The final argument can accept a %Status code, in case the method fails to open an instance.

The method returns an OREF if it locates a matching instance.

For example, suppose that a class includes the following index definition:

```
Index SSNKey On SSN [ Unique ];
```

then, if the referenced object has been stored to disk and has a unique ID value, you can invoke the method as follows:

```
SET person = ##class(Sample.Person).SSNKeyOpen("111-22-3333",2,.sc)
```

Upon successful completion, the method has set the value of *person* to the OREF of the instance of Sample.Person whose SSN property has a value of `111-22-3333`.

The second argument to the method specifies the concurrency value, which here is 2 (shared). The third argument holds an optional %Status code; if the method does not find an object that matches the supplied value, then an error message is written to the status parameter *sc*.

This method is implemented as the **%Library.CacheIndex.Open()** method; this method is analogous to the **%Persistent.Open()** and **%Persistent.OpenId()** methods, except that it uses the properties in the index definition instead of the OID or ID argument.

2.9.2 Checking If an Instance Exists

The *indexname***Exists()** method (where *indexname* is the name of the index) checks if an instance exists with the index property value or values specified by the method's arguments. The method has one argument corresponding to each property in the index; its final, optional argument can receive the object's ID, if one matches the supplied value(s). The method returns a boolean, indicating success (1) or failure (0). This method is implemented as the **%Library.CacheIndex.Exists()** method.

For example, suppose that a class includes the following index definition:

```
Index SSNKey On SSN [ Unique ];
```

then, if the referenced object has been stored to disk and has a unique ID value, you can invoke the method as follows:

```
SET success = ##class(Sample.Person).SSNKeyExists("111-22-3333",.id)
```

Upon successful completion, *success* equals 1 and *id* contains the ID matching the object that was found.

This method returns values for all indices except:

- bitmap indices, or a bitmap extent index.
- when the index includes an `(ELEMENTS)` or `(KEYS)` expression. For more information on such indices, see the section "Indexing Collections."

2.9.3 Deleting an Instance

The *indexname***Delete()** method (where *indexname* is the name of the index) is meant for use with a Unique, PrimaryKey, and or IdKey index; it deletes the instance whose key value matches the supplied key property/column values. There is one optional argument, which you can use to specify a concurrency setting for the operation. The method returns a %Status code. It is implemented as the **%Library.CacheIndex.Delete()** method.

3

Optimizing Tables

There are a number of things you can do to ensure the maximum performance of Caché SQL tables. The optimizations can have a significant effect on any queries run against this table. The following performance optimizing considerations are discussed in this chapter:

- ExtentSize, Selectivity, and BlockCount to specify table data estimates before populating the table with data; this metadata is used to optimize future queries.
- Tune Table to analyze representative table data in a populated table; this generated metadata is used to optimize future queries.

3.1 ExtentSize, Selectivity, and BlockCount

When the Query Optimizer decides the most efficient way to execute a specific SQL query, three of the things it considers are:

- *ExtentSize* row count for each table used within the query.
- *Selectivity* the percentage of distinct values calculated for each column used by the query.
- *BlockCount* count for each SQL map used by the query.

In order to ensure that the Query Optimizer can make the correct decisions, it is important that these values are set correctly.

- You can explicitly set any of these statistics during class (table) definition, prior to populating the table with data.
- After populating the table with representative data, you can run Tune Table to calculate these statistics.
- After running Tune Table, you can override a calculated statistic by specifying an explicit value.

You can compare your explicitly set statistics to the Tune Table generated results. If the assumptions made by Tune Table result in less-than-optimal results from the Query Optimizer, you can use an explicitly set statistic rather than a Tune Table generated statistic.

In Studio the Class Editor window displays the class source code. At the bottom of the source code it displays the Storage definition, which includes the class *ExtentSize*, and the *Selectivity* (and, where appropriate, the *OutlierSelectivity*) for each property.

3.1.1 ExtentSize

The *ExtentSize* value for a table is simply the number of rows (roughly) stored within the table.

At development time, you can provide an initial *ExtentSize* value. If you do not specify an *ExtentSize*, the default is 100,000.

Typically, you provide a rough estimate of what you expect the size of this table will be when populated with data. It is not important to have an exact number. This value is used to compare the relative costs of scanning over different tables; the most important thing is to make sure that the relative values of *ExtentSize* between associated tables represent an accurate ratio (that is, small tables should have a small value and large tables a large one).

- **CREATE TABLE** provides an %EXTENTSIZE parameter keyword to specify the expected number of rows in the table, as shown in the following example:

```
CREATE TABLE Sample.DaysInAYear (%EXTENTSIZE 366,
                                 MonthName VARCHAR(24),Day INTEGER,
                                 Holiday VARCHAR(24),ZodiacSign VARCHAR(24))
```

- A persistent class definition for a table can specify an ExtentSize parameter within the storage definition:

```
<Storage name="Default">
<Data name="MyClassDefaultData">
...
<ExtentSize>200</ExtentSize>
...
</Storage>
```

  In this example, the fragment is the storage definition for the *MyClass* class, which specifies a value of 200 for *ExtentSize*.

If your table has real (or realistic) data, you can automatically calculate and set its *ExtentSize* value using the Tune Table facility within the Management Portal; for details, see the following section on Tune Table.

3.1.2 Selectivity

Within a Caché SQL table (class), every column (property) has a *Selectivity* value associated with it. The *Selectivity* value for a column is the percentage of rows within a table that would be returned as a result of query searching for a typical value of the column. Selectivity is 1/D where D is the number of distinct values for the field.

Selectivity is based on roughly equal quantities of the distinct values. For example, suppose a table contains a Gender column whose values are roughly evenly distributed between "M" and "F". The *Selectivity* value for the Gender column would be 50%. The *Selectivity* value for a more distinguishing property, such as Street Name, is typically a small percentage.

A field in which all the values are the same has a selectivity of 100%. A field that is defined as Unique (all values different) has a selectivity of 1 (which should not be confused with a selectivity of 1.0000%).

At development time, you can provide this value by defining a *Selectivity* parameter within the storage definition that is part of the class definition for the table:

```
<Storage name="Default">
<Data name="MyClassDefaultData">
...
<Property name="Gender">
<Selectivity>50%</Selectivity>
</Property>
...
</Storage>
```

To view a class's storage definition, in Studio, from the **View** menu, select **View Storage**; Studio includes the storage at the bottom of the source code for the class.

Typically you provide an estimate of what you expect the *Selectivity* will be when used within an application. As with *ExtentSize*, it is not important to have an exact number. Many of the data type classes provided by Caché will provide reasonable default values for *Selectivity*.

You can also use the **$SYSTEM.SQL.SetFieldSelectivity()** method to set the *Selectivity* value for a specific field (property).

If your table has real (or realistic) data, you can automatically calculate and set its *Selectivity* values using the Tune Table facility within the Management Portal. Tune Table determines if a field has an outlier value, a value that is far more common than any other value. If so, Tune Table calculates a separate *Outlier Selectivity* percentage, and calculates *Selectivity* based on the presence of this outlier value. The presence of an outlier value may dramatically change the *Selectivity* value.

3.1.3 BlockCount

When you compile a persistent class, the class compiler computes approximate numbers of map blocks used by each SQL map based on the *ExtentSize* and the property definitions. You can view these *BlockCount* values in the **Map BlockCount** tab of the Tune Table facility. The *BlockCount* is identified in Tune Table as `Estimated by class compiler`. Note that if you change the *ExtentSize*, you must close and re-open the SQL Tune Table window to see this change reflected in the *BlockCount* values.

When you run Tune Table, it measures the actual block count for each SQL map. Unless specified otherwise, the Tune Table measured values replace the class compiler approximate values. These Tune Table measured values are represented in the class definition as negative integers, to distinguish them from specified BlockCount values. This is shown in the following example:

```
<SQLMap name="IDKEY">
 <BlockCount>-4</BlockCount>
</SQLMap>
```

Tune Table measured values are represented in Tune Table as positive integers, identified as `Measured by TuneTable`.

You can define explicit BlockCount values in the class definition. You can explicitly specify a block count as a positive integer, as shown in the following example:

```
<SQLMap name="IDKEY">
 <BlockCount>12</BlockCount>
</SQLMap>
```

When you define a class you can omit defining the BlockCount for a map, explicitly specify a BlockCount as a positive integer, or explicitly define the BlockCount as NULL.

- If you do not specify a BlockCount, or specify a BlockCount of 0, the class compiler estimates the block count. Running Tune Table replaces the class compiler estimated value.
- If you specify an explicit positive integer BlockCount, running Tune Table does not replace this explicit BlockCount value. Explicit class definition block count values are represented in Tune Table as positive integers, identified as `Defined in class definition`. These block count values are not changed by subsequently running Tune Table.
- If you specify an explicit BlockCount of NULL, the SQL Map uses the BlockCount value estimated by the class compiler. Because BlockCount is “defined” in the class definition, running Tune Table does not replace this estimated BlockCount value.

The size of all Caché SQL map blocks is 2048 bytes (2K bytes).

Tune Table does not measure BlockCount in the following circumstances:

- If the table is a child table projected by an array or a list collection. The BlockCount values for these types of child tables are the same as BlockCount for the data map of the parent table.
- If a global map is a remote global (a global in a different namespace). The estimated BlockCount used during class compilation is used instead.

3.2 Tune Table

Tune Table is a utility that examines the data in a table and returns statistics about the ExtentSize (the number of rows in the table), the relative distribution of distinct values in each field, and the Average Field Size (average length of values in each field). It also generates the BlockCount for each SQL map. You can specify that Tune Table use this information to update the metadata associated with a table and each of its fields. The query optimizer can subsequently use these statistics to determine the most efficient execution plan for a query.

Using Tune Table on an external table will only calculate the ExtentSize. Tune Table cannot calculate field Selectivity values, Average Field Size, or map BlockCount values for an external table.

3.2.1 When to Run Tune Table

You should run Tune Table on each table after that table has been populated with a representative quantity of real data. Commonly, you only need to run Tune Table once, as a final step in application development, before the data goes "live." Tune Table is not a maintenance utility; it *should not* be run periodically on live data.

Note: There are rare cases where running Tune Table can decrease SQL performance. While Tune Table can be run on live data, it is recommended that you run Tune Table on a test system with real data, rather than on a production system.

Generally, Tune Table should not be re-run when table data is added, modified, or deleted, unless there is an order-of-magnitude change to the characteristics of the current data, as follows:

- Relative Table Sizes: Tune Table assumes that it is analyzing a representative subset of the data. This subset can be only a small percentage of the full data set, if it is a representative subset. Tune Table results remain relevant as the number of rows in a table changes, provided that the ExtentSizes of tables involved in joins or other relationships maintain roughly the same relative sizes. ExtentSize needs to be updated if the ratio between joined tables changes by an order of magnitude. This is important for JOIN statements, because the SQL optimizer uses ExtentSize when optimizing the table join order. As a general rule, a smaller table is joined before a larger table, regardless of the join order specified in the query. Therefore, you would want to re-run Tune Table on one or more tables if the ratio of rows in tableA to tableB changes from 1000:2000 to 10000:2000, but not if it changes to 2100:4000.
- Even Value Distribution: Tune Table assumes that every data value is equally likely. If it detects an outlier value, it assumes that every data value other than the outlier value is equally likely. Tune Table establishes Selectivity by analyzing the current data values for each field. Equal likelihood in real data is always a rough approximation; normal variation in the number of distinct data values and their relative distribution should not warrant re-running Tune Table. However, an order-of-magnitude change in the number of possible values for a field (the ratio of distinct values to records), or the overall likelihood of a single field value can result in inaccurate Selectivity. Dramatically changing the percentage of records with a single field value can cause Tune Table to designate an outlier value or to remove outlier value designation, significantly changing the calculated Selectivity. If the Selectivity of a field no longer reflects the actual distribution of data values, you should re-run Tune Table.
- A significant Caché upgrade, or a new site installation may warrant re-running Tune Table.

3.2.2 Running Tune Table

You can run Tune Table using the Management Portal, or by invoking the **$SYSTEM.SQL.TuneTable()** method. Before you run Tune Table, the table must contain some data.

Tune Table purges cached queries that reference the table(s) being tuned. The **TuneTable()** method provides a recompile cached queries option to regenerate these cached queries using the new Tune Table calculated values.

After running the Tune Table facility, the resulting *ExtentSize* and *Selectivity* values are saved in the class's storage definition. To view the storage definition, in Studio, from the **View** menu, select **View Storage**; Studio includes the storage at the bottom of the source code for the class.

3.2.2.1 Tune Table from the Management Portal

To run Tune Table from the Management Portal:

1. Select **System Explorer**, then **SQL** (**[System] > [SQL]**). Select a namespace by clicking the **Switch** option at the top of the page, then selecting a namespace from the displayed list. (You can set the Management Portal default namespace for each user.)
2. Select a **Schema** from the drop-down list on the left side of the screen, or use a **Filter**. For further details on how to use **Schema** and **Filter**, refer to Browsing SQL Schemas in the "Using the Management Portal SQL Interface" of this manual.
3. Do one of the following:
   - Tune a Single Table: Expand the **Tables** category and select a table from the list. Once you have selected a table, click the **Actions** drop-down list and select **Tune Table Information**. This displays the table's current ExtentSize and Selectivity information. If Tune Table has never been run, ExtentSize=100000, no Selectivity, Outlier Selectivity, Outlier Value, or Average Field Size information is shown (other than the ID having a selectivity of 1), and the Map BlockCount information is listed as `Estimated by class compiler`.

     From the **Selectivity** tab, select the **Tune Table** button. This runs Tune Table on the table, calculating the ExtentSize, Selectivity, Outlier Selectivity, Outlier Value, and Average Field Size values based on the data in the table. The Map BlockCount information is listed as `Measured by Tune Table`.
   - Tune All Tables in the Schema: click the **Actions** drop-down list and select **Tune All Tables in Schema**. This displays the Tune Table box. Select the **Finish** button to run Tune Table on all tables in the schema. When Tune Table completes this box displays a **Done** button. Select **Done** to exit the Tune Table box.

The SQL Tune Table window has two tabs: **Selectivity** and **Map BlockCount**. These tabs display the current values generated by Tune Table. They also allow you to manually set different values than the values generated by Tune Table.

The **Selectivity** tab contains the following fields:

- Current Table Extentsize. This field has an **edit** button that allows you to enter a different Table Extentsize.
- **Keep class up to date** check box. Any changes to statistics generated by Tune Table, or by user input value from the Tune Table interface, or from Tune Table methods are immediately represented in the class definition:
  - If this box is not checked (No), the up-to-date flag on the modified class definition is not set. This indicates that the class definition is out of date and should be recompiled. This is the default.
  - If this box is checked (Yes), the class definition remains flagged as up-to-date. This is the preferred option when making changes to statistics on a live system, because it makes it less likely that a table class definition will be recompiled.
- Fields table with columns for Field Name, Selectivity, Notes, Outlier Selectivity, Outlier Value, and Average Field Size. By clicking on a Fields table heading, you can sort by that column's values. By clicking on a Fields table row, you can manually set values for Selectivity, Outlier Selectivity, Outlier Value, and Average Field Size for that field.

The **Map BlockCount** tab contains the following fields:

- Map Name table with columns for SQL Map Name, BlockCount, and Source of BlockCount. The SQL Map Name for an index is the SQL index name; this may differ from the persistent class index property name.

  By clicking on an individual map name, you can manually set a BlockCount value for that map name.

From the **Selectivity** tab, you can click the **Tune Table** button to run Tune Table on this table.

3.2.2.2 Tune Table using a Method

You can use the **$SYSTEM.SQL.TuneTable()** method to run the Tune Table facility in the current namespace. You can use **TuneTable(tablename)** to run Tune Table on a single table, or **TuneTable(*)** to run Tune Table on all tables is the specified (or default) schema. Running Tune Table on a single table is shown in the following example:

```
ZNSPACE "Samples"
DO $SYSTEM.SQL.TuneTable("Sample.Person",0,1)
```

In this example, the second parameter specifies that the Calculated results from this invocation of Tune Table should not update the table's current *ExtentSize* and *Selectivity* values. The third parameter specifies to display the Tune Table results at the Terminal.

When using the **TuneTable()** method, the following error messages may be generated:

- Non-existent table: `DO $SYSTEM.SQL.TuneTable("NoSuchTable",0,1)`

  `No such table 'SQLUser.NoSuchTable'`
- View: `DO $SYSTEM.SQL.TuneTable("MyView",0,1)`

  `'SQLUser.MyView' is a view, not a table. No tuning will be performed.`

You can also use the **$SYSTEM.SQL.TuneSchema()** method to run the Tune Table facility for all the tables in a schema.

3.2.3 Extent Size and the Row Count

When running the Tune Table facility from the Management Portal, the *ExtentSize* is the actual count of the rows currently in the table. By default, the **TuneTable()** method also uses the actual row count as the ExtentSize. When a table contains a large number of rows, it may be preferable to perform analysis on a smaller number of rows. When running the **TuneTable()** method, you can optionally specify a different ExtentSize, as shown in the following example:

```
ZNSPACE "Samples"
DO $SYSTEM.SQL.TuneTable("Sample.Person",0,1,,,,50)
```

You can use this option to improve **TuneTable()** performance when running against a table with a large number of rows. If you specify an ExtentSize smaller than the actual number of rows in the table, this ExtentSize number should be large enough to sample representative data.

A specified ExtentSize can be smaller or larger than the actual number of rows. However, ExtentSize should not significantly exceed the actual number of rows in the current table data. When you specify an ExtentSize, **TuneTable()** extrapolates row Ids for that number of rows, then performs sampling. If the ExtentSize greatly exceeds the actual number of rows, most of the sampled row Ids will not correspond to actual row data. If this is the case, field selectivities cannot be calculated; instead, **TuneTable()** lists the ExtentSize you specified as the CALCULATED ExtentSize and a smaller number as the SAMPLESIZE; **TuneTable()** returns <Not Specified> for these non-existent Calculated values.

You can specify a SamplePercent for the **TuneTable()** method. This specifies the percentage of the ExtentSize to sample. SamplePercent defaults to "" (the null string). This default samples the data as follows:

- If ExtentSize < 1000: SamplePercent="" uses the entire extent size. For example, an ExtentSize of 600 would use a sample of 600.
- If ExtentSize >= 1000: SamplePercent="" uses 3*sqrt(ExtentSize). For example, an ExtentSize of 10,000 would use a sample of 300.

You can set an ExtentSize of 0. This may be desirable when you have a table that is never intended to be populated with data, but used for other purposes such as query joins. When you set ExtentSize to 0, Caché sets the Selectivity of each field as 100%, and the Average Field Size of each field as 0.

3.2.4 CALCSELECTIVITY Parameter and Not Calculating Selectivity

Under certain circumstances, you may not want the Tune Table facility to calculate the *Selectivity* for a property. To prevent *Selectivity* from being calculated, specify the value of the property's *CALCSELECTIVITY* parameter to 0 (the default is 1). In Studio, you can set *CALCSELECTIVITY* on the **Property Parameters** page of the **New Property Wizard** or in the list of a property's parameters in the Inspector (you may need to contract and re-expand the property parameter list to display it).

One circumstance where you should specify *CALCSELECTIVITY*=0 is a field that is known to contain only one value in all rows (Selectivity=100%), if that field is not indexed.

3.2.5 Selectivity and Outlier Selectivity

Tune Table calculates a *Selectivity* for each property (field) value as a percentage. It does this by sampling the data, so selectivity is always an estimate, not an exact value. Selectivity is based on the assumption that all property values are, or could be, equally likely. This is a reasonable assumption for most data. For example, in a general population table most data values are typical: any given date of birth will appear in around .27% of the data (1 in 365); roughly half will be female and half male (50%). A field that is defined as Unique has a selectivity of 1 (which should not be confused with a selectivity of 1.0000 (1%). A selectivity percentage is sufficient for most properties.

For a few properties, Tune Table also calculates an *Outlier Selectivity*. This is a percentage for a single property value that appears much more frequently in the sample than the other data values. Tune Table only returns an outlier selectivity when there is a substantial difference between the frequency of one data value and the frequency of the other data values. Tune Table returns, at most, one outlier for a table, regardless of the distribution of data values. If an outlier is selected, Tune Table displays this value as the *Outlier Value*. NULL is represented as `<Null>`.

If Tune Table returns an outlier selectivity, the normal selectivity is still the percentage of each non-outlier data value within the whole set of rows. For example, if the outlier selectivity is 80% and the regular selectivity is 1%, then in addition to the outlier value, you can expect to find about 20 ((1-.80)/.01) additional non-outlier values.

The most common example of outlier selectivity is a property that permits NULLs. If the number of records with NULL for a property greatly exceeds the number of records that have any specific data value for that property, NULL is the outlier. The following is the Selectivity and Outlier Selectivity for the FavoriteColors field:

```
SELECTIVITY of FIELD FavoriteColors
  CURRENT =     1.8966%
  CALCULATED =  1.4405%
  CURRENT OUTLIER = 45.0000%, VALUE = <Null>
  CALCULATED OUTLIER = 39.5000%, VALUE = <Null>
```

If a field only contains one distinct value (all rows have the same value), it has a Selectivity of 100%. A value that has a selectivity of 100% is not considered to be an outlier. Normally, Tune Table establishes Selectivity and Outlier Selectivity values by sampling the data and estimating, which is not sufficient for establishing that there is only one value. All rows must be examined. Consequently, Tune Table can only recognize a field of this sort if the field is indexed, the field is the first field of the index, and the field and the index have the same collation type. If the field is not indexed as described, you should manually specify a Selectivity of 100%, delete any outlier selectivity, and set CALCSELECTIVITY=0 to prevent Tune Table attempting to calculate selectivity or specify this value as an outlier.

To modify these *Selectivity*, *Outlier Selectivity*, and *Outlier Value* calculated values, select an individual field from the Tune Table display. This displays these values for that field in the **Details** area to the right of the display. You can modify *Selectivity*, *Outlier Selectivity*, and/or *Outlier Value* to values that better fit the anticipated full data set.

- You can specify *Selectivity* either as a percentage of rows with a percent (%) sign, or as an integer number of rows (no percent sign). If specified as an integer number of rows, Caché uses the extent size to calculate the *Selectivity* percentage.
- You can specify an *Outlier Selectivity* and *Outlier Value* for a field that previously had no outlier. Specify *Outlier Selectivity* as a percentage with a percent (%) sign. If you specify just the *Outlier Selectivity*, Tune Table assumes the

Outlier Value is <Null>. If you specify just the *Outlier Value*, Tune Table will not save this value unless you also specify an *Outlier Selectivity*.

3.2.6 Outlier Optimization

By default, the query optimizer assumes that a query *will not* select for the outlier value. For example, a query commonly selects for specific field value(s) and returns a small number of records from the database, rather than returning the large number of records in which that field value is the outlier. The query optimizer always uses *Selectivity* for construction of the query plan unless you perform some action that requests consideration of the *Outlier Selectivity*.

There are several actions you can perform to adjust query optimization based on selecting for an outlier value:

- If the *Outlier Value* is <null>, specify an IS NULL or IS NOT NULL condition for that field in the query WHERE clause. This causes the query optimizer to use the *Outlier Selectivity* when constructing the query.
- If the *Outlier Value* is a data value, the query optimizer assumes that the field value you are selecting for is not the outlier value. For example, the employee records for a Massachusetts-based company might have an Office_State field outlier of 'MA' (Massachusetts). The optimizer assumes that the query is not going to select for 'MA', because that would return the majority of the records in the database. If, however, you are writing a query that *does* select for the outlier value, you can inform the optimizer of this by enclosing the outlier value in double parentheses. When querying on the field, specify a WHERE clause such as the following: `WHERE Office_State=(('MA'))`. This technique suppresses literal substitution and forces the query optimizer to use the *Outlier Selectivity* when constructing the query plan. This syntax is necessary for Dynamic SQL queries, and for queries written outside of Caché that are supplied using ODBC/JDBC. It is not necessary for class queries, Embedded SQL queries, or queries accessed through a view. For further details, refer to the WHERE clause reference page.
- Do not use the **Bias queries as outlier** configuration option. Leave this check box unselected.

3.2.7 The Notes Column

The Management Portal **Tune Table Information** option displays a Notes column for each field. The values in this field are system-defined and non-modifiable. They include the following:

- `RowID field`: A table has one RowID, which is defined by the system. Its name is commonly ID, but it can have a different system-assigned name. Because all of its values are (by definition) unique, its Selectivity is always 1. If the class definition includes SqlRowIdPrivate, the Notes column value is `RowID field, Hidden field`.
- `Hidden field`: A hidden field is defined as private, and is not displayed by SELECT *. By default, **CREATE TABLE** defines the RowID field as hidden; you can specify the %PUBLICROWID keyword to make the RowID not hidden and public. By default, tables defined by a persistent class definition define the RowID as not hidden; you can specify SqlRowIdPrivate to define the RowID as hidden and private. Container fields are defined as hidden.
- `Stream field`: Indicates a field defined with a stream data type, either character stream (CLOB) or binary stream (BLOB). A stream file has no Average Field Size.
- `Parent reference field`: A field that references a parent table.

An IDENTITY field, ROWVERSION field, SERIAL field, or UNIQUEIDENTIFIER (GUID) field is *not* identified in the Notes column.

3.2.8 Average Field Size

Running Tune Table calculates the average field size (in characters) for all non-Stream fields, based on the current table data set. This is (unless otherwise noted) the same as `AVG($LENGTH(field))`, rounded to two decimal places. You can change this average field size for individual fields to reflect the anticipated average size of the field's data.

- NULL: Because the **$LENGTH** function treats NULL fields as having a length of 0, NULL fields are averaged in, with a length 0. This may result in an Average Field Size of less than one character.
- Empty column: If a column contains no data (no field values for all of the rows), the average field size value is 1, not 0. The `AVG($LENGTH(field))` is 0 for a column that contains no data.
- ExtentSize=0: When you set ExtentSize to 0, Average Field Size for all fields is reset to 0.
- Logical field values: Average Field Size is always calculated based on the field's Logical (internal) value.
- List fields: Caché List fields are calculated based on their Logical (internal) encoded value. This encoded length is longer than the total length of the elements in the list.
- Container fields: A container field for a collection is larger than the total length of its collection objects. For example, in Sample.Person the Home container field Average Field Size is larger than the total of the average field sizes of Home_Street, Home_City, Home_State, and Home_Zip. For further details, refer to "Controlling the SQL Projection of Collection Properties" in *Using Caché Objects*.
- Stream fields: A stream field does not have an average field size.

If the property parameter CALCSELECTIVITY is set to 0 for a property/field, Tune Table does not calculate the Average Field Size for that property/field.

You can modify an *Average Field Size* calculated value by selecting an individual field from the Tune Table display. This displays the values for that field in the **Details** area to the right of the display. You can modify the *Average Field Size* to a value that better fits the anticipated full data set. Because Tune Table performs no validation when you set this value, you should make sure that the field is not a Stream field, and that the value you specify is not larger than the maximum field size (MaxLen).

The Average Field Size is also displayed in the Management Portal **Catalog Details** tab **Fields** option table. Tune Table must have been run for the **Fields** option table to display Average Field Size values. For further details, refer to the Catalog Details Tab section in the "Using the Management Portal SQL Interface" chapter of this guide.

3.2.9 Map BlockCount Tab

The Tune Table **Map BlockCount** tab displays the **SQL Map Name**, **BlockCount** (as a positive integer), and **Source of BlockCount**. The **Source of BlockCount** can be `Defined in class definition`, `Estimated by class compiler`, or `Measured by TuneTable`. Running Tune Table changes `Estimated by class compiler` to `Measured by TuneTable`; it does not affect `Defined in class definition` values.

You can modify a *BlockCount* calculated value by selecting an individual SQL Map Name from the Tune Table display. This displays the block count for that Map Name in the **Details** area to the right of the display. You can modify the *BlockCount* to a value that better fits the anticipated full data set. Because Tune Table performs no validation when you set this value, you should make sure that the block count is a valid value. Modifying *BlockCount* changes the **Source of BlockCount** to `Defined in class definition`. For further details, refer to the BlockCount section in this chapter.

3.2.10 Exporting and Re-importing Tune Table Statistics

You can export Tune Table statistics from a table or group of tables and then import these Tune Table statistics into a table or group of tables. The following are three circumstances in which you might want to perform this export/import. (For simplicity, these describe the export/import of statistics from a single table; in actual use, export/import of statistics from multiple inter-related tables is often performed):

- To model a production system: A production table is fully populated with actual data and optimized using Tune Table. In a test environment you create a table with the same table definition but far less data. By exporting the Tune Table statistics from the production table and importing them into the test table, you can model the production table optimization on the test table.

- To replicate a production system: A production table is fully populated with actual data and optimized using Tune Table. A second production table with the same table definition is created. For example, a production environment and its backup environment, or multiple tables with the same table definition with each table containing, for example, the patient records for a different hospital. By exporting the Tune Table statistics from the first table and importing them into the second table, you can give the second table the same optimization as the first table without the overhead of running Tune Table a second time or waiting for the second table to be populated with representative data.
- To revert to a prior set of statistics: You can create optimization statistics for a table by running Tune Table or by explicitly setting statistics. By exporting these statistics you can preserve them while experimenting with other statistics settings. Once you have determined the optimal set of statistics, you can import them back into the table.

You can export Tune Table statistics to an XML file using the **$SYSTEM.SQL.ExportTuneStats()** method. This method can export the Tune Table statistics for one, more than one, or all tables within a namespace, as shown in the following examples:

```
  DO $SYSTEM.SQL.ExportTuneStats("C:\AllStats.xml")
  /*  Exports TuneTable Statistics for all schemas/tables in the current namespace */

  DO $SYSTEM.SQL.ExportTuneStats("C:\SampleStats.xml","Sample")
  /*  Exports TuneTable Statistics for all tables in the Sample schema */

  DO $SYSTEM.SQL.ExportTuneStats("C:\SamplePStats.xml","Sample","P*")
  /*  Exports TuneTable Statistics for all tables beginning with the letter "P" in the Sample schema
*/

  DO $SYSTEM.SQL.ExportTuneStats("C:\SamplePersonStats.xml","Sample","Person")
  /*  Exports TuneTable Statistics for the Sample.Person table */
```

You can re-import Tune Table statistics that were exported using **ExportTuneStats()** by using the **$SYSTEM.SQL.ImportTuneStats()** method.

ImportTuneStats() has a KeepClassUpToDate boolean option. If TRUE (and update is TRUE), **ImportTuneStats()** will update the class definition with the new EXTENTSIZE and SELECTIVITY values, but the class definition will be kept as up-to-date. In many cases, however, it is desirable to recompile the class after its table has been tuned so that queries in the class definition can be recompiled and the SQL query optimizer can use the updated data statistics. The default is FALSE (0). Note that if the class is deployed the class definition will not be updated.

ImportTuneStats() has a ClearCurrentStats boolean option. If TRUE, **ImportTuneStats()** will clear any prior EXTENTSIZE, SELECTIVITY, BLOCKCOUNT and other Tune Table statistics from the existing table before importing the stats. This can be used if you want to completely clear those table stats that are not specified in the import file, instead of leaving them defined in the persistent class for the table. The default is FALSE (0).

If **ImportTuneStats()** does not find the corresponding table, it skips that table and proceeds to the next table specified in the import file. If a table is found, but some of the fields are not found, those fields will simply be skipped.

The BlockCount for a map in a class storage definition cannot be inherited. The BlockCount can only appear in the storage definition of the class where the map originated. **ImportTuneStats()** only sets the projected table's BlockCount metadata and not the class storage BlockCount metadata if the map originated in a super class.

4

Cached Queries

The system automatically maintains a cache of prepared Dynamic SQL statements (“queries”). This permits the re-execution of an SQL query without repeating the overhead of optimizing the query and developing a Query Plan. A cached query is created when certain SQL statements are prepared using Dynamic SQL, ODBC, JDBC, or the **$SYSTEM.SQL.DDLImport()** method. (The Management Portal execute SQL inteface, the InterSystems SQL Shell, and the **%SYSTEM.SQL.Execute()** method use Dynamic SQL, and thus create cached queries.) A non-cursor Embedded SQL statement does not create a cached query.

SQL statements that are automatically cached are:

- **SELECT**: a **SELECT** cached query is shown in the **Catalog Details** for its table. If the query references more than one table, the same cached query is listed for each referenced table. Purging the cached query from any one of these tables purges it from all tables. From the table’s **Catalog Details** you can select a cached query name to display cached query details, including **Execute** and **Show Plan** options. A **SELECT** cached query created by the **DDLImport("CACHE")** method does not provide **Execute** and **Show Plan** options.

  DECLARE name CURSOR FOR SELECT creates a cached query. However, cached query details do not include **Execute** and **Show Plan** options.
- **INSERT**, **UPDATE**, **INSERT OR UPDATE**, **DELETE**: create a cached query shown in the **Catalog Details** for its table.
- **TRUNCATE TABLE**: issued from Dynamic SQL creates a cached query shown in the **Catalog Details** for its table. Note that **DDLImport("CACHE")** does not support **TRUNCATE TABLE**.
- **CALL**: creates a cached query shown in the **Cached Queries** list for its schema.
- **SET TRANSACTION**, **START TRANSACTION**, **%INTRANSACTION**, **COMMIT**, **ROLLBACK**: create a cached query shown in the **Cached Queries** list for every schema in the namespace.

A cached query is created when you Prepare the query. For this reason, it is important not to put a **%Prepare()** method in a loop structure. A subsequent **%Prepare()** of the same query (differing only in specified literal values) uses the existing cached query rather than creating a new cached query. Note that changes to the query that shouldn’t affect query optimization, such as changing a column name alias or changing the ORDER BY clause, do result in different cached queries.

A cache query is deleted when you purge cached queries. Modifying a table definition automatically purges any queries that reference that table. Issuing a Prepare or Purge automatically requests an exclusive system-wide lock while the query cache metadata is updated. The System Administrator can modify the timeout value for the cached query lock.

The creation of a cached query is not part of a transaction. The creation of a cached query is not journaled.

4.1 Cached Queries Improve Performance

When you first prepare a Dynamic SQL query, the SQL Engine optimizes it and generates a program (a set of one or more Caché routines) that will execute the query. The optimized query text is then stored as a cache query class. If you subsequently attempt to execute the same (or a similar) query, the SQL Engine will find the cached query and directly execute the code for the query, bypassing the need to optimize and code generate.

Cached queries provide the following benefits:

- Subsequent execution of frequently used queries is faster. More importantly, this performance boost is available *automatically* without having to code cumbersome stored procedures. Most relational database products recommend using only stored procedures for database access. This is not necessary with Caché.
- A single cached query is used for similar queries, queries that differ only in their literal values. For example, `SELECT TOP 5 Name FROM Sample.Person WHERE Name %STARTSWITH 'A'` and `SELECT TOP 1000 Name FROM Sample.Person WHERE Name %STARTSWITH 'Mc'` only differ in the literal values for TOP and the %STARTSWITH condition. The cached query prepared for the first query is automatically used for the second query.
- The query cache is shared among all database users; if User 1 prepares a query, then User 1023 can take advantage of it.
- The Query Optimizer is free to use more time to find the best solution for a given query as this price only has to be paid the first time a query is prepared.

Caché SQL stores all cached queries in a single location, the CACHE database. However, cached queries are namespace specific. Each cached query is identified with the namespace from which it was prepared (generated). You can only view or execute a cached query from within the namespace in which it was prepared. You can purge cached queries either for the current namespace or for all namespaces.

A cached query does not include comments.

There is no need for a cache for Embedded SQL, because Embedded SQL statements are replaced with inline code at compilation time.

For use of cached queries when changing a table definition, refer to the "SQL Statements and Frozen Plans" chapter in this manual.

4.1.1 Creating a Cached Query

When Caché Prepares a query it determines:

1. If the query matches a query already in the query cache. If not, it assigns an increment count to the query.
2. If the query prepares successfully. If not, it does not assign the increment count to a cached query name.
3. Otherwise, the increment count is assigned to a cached query name and the query is cached.

4.1.1.1 Cached Query Names

The SQL Engine assigns a unique class name to each cached query, with the following format:

```
%sqlcq.namespace.clsnnn
```

Where *namespace* is the current namespace, in capital letters, and *nnn* is a sequential integer. For example, `%sqlcq.USER.cls16`.

Cached queries are numbered sequentially on a per-namespace basis, starting with 1. The next available *nnn* sequential number depends on what numbers have been reserved or released:

- A number is reserved when you begin to prepare a query if that query does not match an existing cached query. A query matches an existing cached query if they differ only in their literal values — subject to certain additional considerations: suppressed literal substitution or the situations described in "Separate Cached Queries".
- A number is reserved but not assigned if the query does not prepare successfully. Only queries that Prepare successfully are cached.
- A number is reserved and assigned to a cached query if the query prepares successfully. This cached query is listed for every table referred to in the query, regardless of whether any data is accessed from that table. If a query does not refer to any tables, a cached query is created but cannot be listed or purged by table.
- A number is released when a cached query is purged. This number becomes available as the next *nnn* sequential number. Purging individual cached queries associated with a table or purging all of the cached queries for a table releases the numbers assigned to those cached queries. Purging all cached queries in the namespace releases all of the numbers assigned to cached queries, including cached queries that do not reference a table, and numbers reserved but not assigned.

  Purging cached queries resets the *nnn* integer. Integers are reused, but remaining cached queries are not renumbered. For example, a partial purge of cached queries might leave `cls1`, `cls3`, `cls4`, and `cls7`. Subsequent cached queries would be numbered `cls2`, `cls5`, `cls6`, and `cls8`.

A **CALL** statement may result in multiple cached queries. For example, the SQL statement `CALL Sample.PersonSets('A','MA')` results in the following cached queries:

```
%sqlcq.USER.cls1: CALL Sample . PersonSets ( ? , ? )
%sqlcq.USER.cls2: SELECT name , dob , spouse FROM sample . person
                     WHERE name %STARTSWITH ? ORDER BY 1
%sqlcq.USER.cls3: SELECT name , age , home_city , home_state
                     FROM sample . person WHERE home_state = ? ORDER BY 4 , 1
```

In Dynamic SQL, after preparing an SQL query (using the %Prepare() or %PrepareClassQuery() instance method) you can return the cached query name using the **%Display()** instance method or the **%GetImplementationDetails()** instance method. See Results of a Successful Prepare.

The cached query name is also a component of the result set OREF returned by the **%Execute()** instance method of the %SQL.Statement class (and the *%CurrentResult* property). Both of these methods of determining the cached query name are shown in the following example:

```
SET randtop=$RANDOM(10)+1
SET randage=$RANDOM(40)+1
SET myquery = "SELECT TOP ? Name,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 x = tStatement.%GetImplementationDetails(.class,.text,.args)
IF x=1 { WRITE "cached query name is: ",class,! }
SET rset = tStatement.%Execute(randtop,randage)
WRITE "result set OREF: ",rset.%CurrentResult,!
DO rset.%Display()
WRITE !,"A sample of ",randtop," rows, with age < ",randage
```

In this example, the number of rows selected (TOP clause) and the WHERE clause predicate value change with each query invocation, but the cached query name does not change.

4.1.1.2 Separate Cached Queries

Differences between two queries that shouldn't affect query optimization nevertheless generate separate cached queries:

- Different syntactic forms of the same function generate separate cached queries. Thus `ASCII('x')` and `{fn ASCII('x')}` generate separate cached queries, and `{fn CURDATE()}` and `{fn CURDATE}` generate separate cached queries.

- A case-sensitive table alias or column alias value, and the presence or absence of the optional AS keyword generate separate cached queries. Thus `ASCII('x')`, `ASCII('x') AChar`, and `ASCII('x') AS AChar` generate separate cached queries.
- Using a different ORDER BY clause.
- Using TOP ALL instead of TOP with an integer value.

4.2 Literal Substitution

When the SQL Engine caches a Dynamic SQL query, it performs literal substitution. The query in the query cache represents each literal with a "?" character, representing an input parameter. This means that queries that differ only in their literal values are represented by a single cached query. For example, the two queries:

```
SELECT TOP 11 Name FROM Sample.Person WHERE Name %STARTSWITH 'A'

SELECT TOP 5 Name FROM Sample.Person WHERE Name %STARTSWITH 'Mc'
```

Are both represented by a single cached query:

```
SELECT TOP ? Name FROM Sample.Person WHERE Name %STARTSWITH ?
```

This minimizes the size of the query cache, and means that query optimization does not need to be performed on queries that differ only in their literal values.

Literal values supplied using input host variables (for example, `:myvar`) and ? input parameters are also represented in the corresponding cached query with a "?" character. Therefore, the queries `SELECT Name FROM t1 WHERE Name='Adam'`, `SELECT Name FROM t1 WHERE Name=?`, and `SELECT Name FROM t1 WHERE Name=:namevar` are all matching queries and generate a single cached query.

You can use the %GetImplementationDetails() method to determine which of these entities is represented by each "?" character for a specific prepare.

The following considerations apply to literal substitution:

- Plus and minus signs specified as part of a literal generate separate cached queries. Thus `ABS(7)`, `ABS(-7)`, and `ABS(+7)` each generate a separate cached query. Multiple signs also generate separate cached queries: `ABS(+?)` and `ABS(++?)`. For this reason, it is preferable to use an unsigned variable `ABS(?)` or `ABS(:num)`, for which signed or unsigned numbers can be supplied without generating a separate cached query.
- Precision and scale values usually do not take literal substitution. Thus `ROUND(567.89,2)` is cached as `ROUND(?,2)`. However, the optional precision value in `CURRENT_TIME(n)`, `CURRENT_TIMESTAMP(n)`, `GETDATE(n)`, and `GETUTCDATE(n)` does take literal substitution.
- A boolean flag does not take literal substitution. Thus `ROUND(567.89,2,0)` is cached as `ROUND(?,2,0)` and `ROUND(567.89,2,1)` is cached as `ROUND(?,2,1)`.
- A literal used in an IS NULL or IS NOT NULL condition does not take literal substitution.
- Any literal used in an ORDER BY clause does not take literal substitution. This is because ORDER BY can use an integer to specify a column position. Changing this integer would result in a fundamentally different query.
- An alphabetic literal must be enclosed in single quotes. Some functions permit you to specify an alphabetic format code with or without quotes; only a quoted alphabetic format code takes literal substitution. Thus `DATENAME(MONTH,63120)` and `DATENAME('MONTH',63120)` are functionally identical, but the corresponding cached queries are `DATENAME(MONTH,?)` and `DATENAME(?,?)`.

- Functions that take a variable number of arguments generate separate cached queries for each argument count. Thus `COALESCE(1,2)` and `COALESCE(1,2,3)` generate separate cached queries.

4.2.1 Literal Substitution and Performance

The SQL Engine performs literal substitution for each value of an IN predicate. A large number of **IN** predicate values can have a negative effect on cached query performance. A variable number of **IN** predicate values can result in multiple cached queries. Converting an **IN** predicate to an %INLIST predicate results in a predicate with only one literal substitution, regardless of the number of listed values. **%INLIST** also provides an order-of-magnitude SIZE argument, which SQL uses to optimize performance.

4.2.2 Suppressing Literal Substitution

This literal substitution can be suppressed. There are circumstances where you may wish to optimize on a literal value, and create a separate cached query for queries with that literal value. To suppress literal substitution, enclose the literal value in double parentheses. This is shown in the following example:

```
SELECT TOP 11 Name FROM Sample.Person WHERE Name %STARTSWITH (('A'))
```

Specifying a different %STARTSWITH value would generate a separate cached query. Note that suppression of literal substitution is specified separately for each literal. In the above example, specifying a different TOP value would not generate a separate cached query.

To suppress literal substitution of a signed number, specify syntax such as `ABS(-((7)))`.

Note: Different numbers of enclosing parentheses may also suppress literal substitution in some circumstances. InterSystems recommends always using double parentheses as the clearest and most consistent syntax for this purpose.

4.3 Cached Query Result Set

When you execute a cached query it creates a result set. A cached query result set is an Object instance. This means that the values you specify for literal substitution input parameters are stored as object properties. These object properties are referred to using i%PropName syntax.

4.4 Listing Cached Queries

You can count and list existing cached queries in the current namespace:

- Counting cached queries
- Displaying cached queries using the Caché Management Portal
- Listing cached queries using the ^rINDEXSQL global
- Exporting cached queries to a file using the ExportSQL^%qarDDLExport utility

4.4.1 Counting Cached Queries

You can determine the current number of cached queries for a table by invoking the **GetCachedQueryTableCount()** method of the %Library.SQLCatalog class. This is shown in the following example:

```
SET tbl="Sample.Person"
SET num=##class(%Library.SQLCatalog).GetCachedQueryTableCount(tbl)
IF num=0 {WRITE "There are no cached queries for ",tbl }
ELSE {WRITE tbl," is associated with ",num," cached queries" }
```

Note that a query that references more than one table creates a single cached query. However, each of these tables counts this cached query separately. Therefore, the number of cached queries counted by table may be larger than the number of actual cached queries.

4.4.2 Displaying Cached Queries

You can view (and manage) the contents of the query cache using the Caché Management Portal. From **System Explorer**, select **SQL**. Select a namespace with the **Switch** option at the top of the page; this displays the list of available namespaces. On the left side of the screen open the **Cached Queries** folder. Selecting one of these cached queries displays the details.

The **Query Type** can be one of the following values:

- `%SQL.Statement or %ResultSet.SQL Dynamic SQL`: a Dynamic SQL query using %SQL.Statement or %ResultSet.SQL.
- `%Library.ResultSet Dynamic SQL`: a Dynamic SQL query using %Library.ResultSet.
- `ODBC/JDBC Statement`: a dynamic query from either ODBC or JDBC.

When you successfully prepare an SQL statement, the system generates a new class that implements the statement. If you have set the **Cached Query - Save Source** system-wide configuration option, the source code for this generated class is retained and can be opened for inspection using Studio. To do this, go to the Caché 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 set the **Cached Query - Save Source** option. If this option is not set (the default), the system generates and deploys the class and does not save the source code.

You can also set this system-wide option using the **SetCachedQuerySaveSource()** method. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**.

4.4.3 Listing Cached Queries Using ^rINDEXSQL

You can use the ^rINDEXSQL global to list all of the cached queries and all of the SQL Statements for the current namespace:

```
ZWRITE ^rINDEXSQL("sqlidx",2)
```

The third subscript is the location. For example, "%sqlcq.USER.cls4.1" is a cached query; "Sample.MyTable.1" is an SQL Statement.

The fourth subscript is the Statement hash.

4.4.4 Exporting Cached Queries to a File

The following utility lists all of the cached queries for the current namespace to a text file.

```
ExportSQL^%qarDDLExport(file,fileOpenParam,eos,cachedQueries,classQueries,classMethods,routines,display)
```

file	A file pathname where cached queries are to be listed. Specified as a quoted string. If the file does not exist, the system creates it. If the file already exists, Caché overwrites it.
fileOpenParam	*Optional* — The OPEN mode parameters for the file. Specified as a quoted string. The default is “WNS”. “W” specifies that the file is being opened for writing. “N” specifies that if the file does not exist, create a new sequential file with this name. “S” specifies stream format with carriage return, line feed, or form feed as default terminators. For a full list of OPEN mode parameters refer to the “Sequential File I/O” chapter of the *Caché I/O Device Guide*.
eos	*Optional* — The end-of-statement delimiter used to separate the individual cached queries in the listing. Specified as a quoted string. The default is “GO”.
cachedQueries	*Optional* — Export all SQL queries from the query cache to *file*. A boolean flag. The default is 1.
classQueries	*Optional* — Export all SQL queries from SQL class queries to *file*. A boolean flag. The default is 1.
classMethods	*Optional* — Export embedded SQL queries from class methods to *file*. A boolean flag. The default is 1.
routines	*Optional* — Export embedded SQL queries from MAC routines to *file*. This listing does not include system routines, cached queries, or generated routines. A boolean flag. The default is 1.
display	*Optional* — Display export progress on the Terminal screen. A boolean flag. The default is 0.

The following is an example of evoking this cached queries export utility:

```
DO ExportSQL^%qarDDLExport("C:\temp\test\qcache.txt","WNS","GO",1,1,1,1,1)
```

When executed from the Terminal command line with *display*=1, export progress is displayed to the terminal screen, such as the following example:

```
Export SQL Text for Cached Query: %sqlcq.SAMPLES.cls14..              Done
Export SQL Text for Cached Query: %sqlcq.SAMPLES.cls16..              Done
Export SQL Text for Cached Query: %sqlcq.SAMPLES.cls17..              Done
Export SQL Text for Cached Query: %sqlcq.SAMPLES.cls18..              Done
Export SQL Text for Cached Query: %sqlcq.SAMPLES.cls19..              Done
Export SQL statement for Class Query: Cinema.Film.TopCategory...      Done
Export SQL statement for Class Query: Cinema.Film.TopFilms...         Done
Export SQL statement for Class Query: Cinema.FilmCategory.CategoryName...Done
Export SQL statement for Class Query: Cinema.Show.ShowTimes...        Done
Export SQL statement for Class Query: Cinema.TicketItem.ShowItem...   Done
Export SQL statement from Class Method: Aviation.EventCube.Fact.%BuildAllFacts...Done
```

```
Export SQL statement from Class Method: Aviation.EventCube.Fact.%BuildTempFile...Done
Export SQL statement from Class Method: Aviation.EventCube.Fact.%Count...Done
Export SQL statement from Class Method: Aviation.EventCube.Fact.%DeleteFact...Done
Export SQL statement from Class Method: Aviation.EventCube.Fact.%ProcessFact...Done
Export SQL statement from Class Method: Aviation.EventCube.Fact.%UpdateFacts...Done
Export SQL statement from Class Method: Aviation.EventCube.Star1032357136.%Count...Done
Export SQL statement from Class Method: Aviation.EventCube.Star1032357136.%GetDimensionProperty...Done
Export SQL statement from Class Method: Aviation.EventCube.Star1035531339.%Count...Done
Export SQL statement from Class Method: Aviation.EventCube.Star1035531339.%GetDimensionProperty...Done

20 SQL statements exported to script file C:\temp\test\qcache.txt
```

The created export file contains entries such as the following:

```
  -- SQL statement from Cached Query %sqlcq.SAMPLES.cls30
  SELECT TOP ? Name , Home_State , Age , AVG ( Age ) AS AvgAge FROM Sample . Person ORDER BY Home_State
GO

  -- SQL statement from Class Query Cinema.Film.TopCategory
#import Cinema
SELECT TOP 3 ID, Description, Length, Rating, Title, Category->CategoryName
  FROM Film
  WHERE (PlayingNow = 1) AND (Category = :P1)
  ORDER BY TicketsSold DESC
GO

  -- SQL statement(s) from Class Method Aviation.EventCube.Fact.%Count
#import Aviation.EventCube
SELECT COUNT(*) INTO :tCount FROM Aviation_EventCube.Fact
GO
```

This cached queries listing can be used as input to the Query Optimization Plans utility.

4.5 Executing Cached Queries

- From Dynamic SQL: A %SQL.Statement Prepare operation (**%Prepare()**, **%PrepareClassQuery()**, or **%ExecDirect()**) creates a cached query. A Dynamic SQL %Execute() method using the same instance executes the most recently prepared cached query.
- From the Terminal: You can directly execute a cached query using the **ExecuteCachedQuery()** method of the $SYSTEM.SQL class. This method allows you to specify input parameter values and to limit the number of rows to output. You can execute a Dynamic SQL %SQL.Statement cached query or an xDBC cached query from the Terminal command line. This method is primarily useful for testing an existing cached query on a limited subset of the data.
- From the Management Portal SQL Interface: Follow the "Displaying Cached Queries" instructions above. From the selected cached query's **Catalog Details** tab, click the **Execute** link.

4.6 Cached Query Lock

Issuing a Prepare or Purge statement automatically requests an exclusive system-wide lock while the cached query metadata is updated. SQL supports the **SetCachedQueryLockTimeout()** method, which governs lock timeout when attempting to acquire a lock on cached query metadata. The default is 120 seconds. This is significantly longer than the standard SQL lock timeout, which defaults to 10 seconds. A System Administrator may need to modify this cached query lock timeout on systems with large numbers of concurrent Prepare and Purge operations, especially on a system which performs bulk purges involving a large number (several thousand) cached queries.

The **SetCachedQueryLockTimeout()** method sets the timeout value system-wide and returns the previous value:

```
SetCQTimeout
   DO $SYSTEM.SQL.SetCachedQueryLockTimeout(150,.oldval)
   WRITE oldval," prior value cached query seconds",!!
SetCQTimeoutAgain
   DO $SYSTEM.SQL.SetCachedQueryLockTimeout(180,.oldval2)
   WRITE oldval2," prior value cached query seconds",!!
ResetCQTimeoutToDefault
   DO $SYSTEM.SQL.SetCachedQueryLockTimeout(,.oldval3)
   WRITE oldval3," prior value cached query seconds"
```

SetCachedQueryLockTimeout() sets the cached query lock timeout for all new processes system-wide. It does not change the cached query lock timeout for existing processes.

4.7 Purging Cached Queries

Whenever you modify (alter or delete) a table definition, any queries based on that table are automatically purged from the query cache on the local system. If you recompile a persistent class, any queries that use that class are automatically purged from the query cache on the local system.

You can use the **$SYSTEM.SQL.Purge(n)** method to explicitly purge cached queries that have not been recently used. Specifying *n* number of days purges all cached queries in the current namespace that have not been used (prepared) within the last *n* days. Specifying an *n* value of 0 or "" purges all cached queries in the current namespace. For example, if you issue a $SYSTEM.SQL.Purge(30) method on May 11, 2014, it will purge only the cached queries that were last prepared before April 11, 2014. A cached query that was last prepared exactly 30 days ago (April 11, in this example) would not be purged.

You can also purge cached queries using the following methods:

- **$SYSTEM.SQL.PurgeCQClass()** purges one or more cached queries by name in the current namespace. You can specify cached query names as a comma-separated list. The specified cached query name or list of cached query names must be enclosed with quotation marks.
- **$SYSTEM.SQL.PurgeForTable()** purges all cached queries in the current namespace that reference the specified table.
- **$SYSTEM.SQL.PurgeAllNamespaces()** purges all cached queries in all namespaces on the current system. Note that when you delete a namespace, its associated cached queries are not purged. Executing **PurgeAllNamespaces()** checks if there are any cached queries associated with namespaces that no longer exist; if so, these cached queries are purged.

Purging a cached query also purges related query performance statistics (SQLStats).

CAUTION: When you change the systemwide default schema name, the system automatically purges all cached queries in all namespaces on the system.

4.7.1 Remote Systems

Purging a cached query on a local system does not purge copies of that cached query on shadow or mirror systems. Copies of a purged cached query on a remote system must be manually purged.

When a persistent class is modified and recompiled, the local cached queries based on that class are automatically purged. Caché does not automatically purge copies of those cached queries on remote systems. This could mean that some cached queries on a remote system are “stale” (no longer valid). However, when a remote system attempts to use a cached query, the remote system checks whether any of the persistent classes that the query references have been recompiled. If a persistent class on the local system has been recompiled, the remote system automatically purges and recreates the stale cached query before attempting to use it.

4.8 SQL Commands That Are Not Cached

The following non-query SQL commands are not cached; they are purged immediately after use:

- Data Definition Language (DDL): CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE VIEW, ALTER VIEW, DROP VIEW, CREATE INDEX, DROP INDEX, CREATE FUNCTION, CREATE METHOD, CREATE PROCEDURE, CREATE QUERY, DROP FUNCTION, DROP METHOD, DROP PROCEDURE, DROP QUERY, CREATE TRIGGER, DROP TRIGGER, CREATE DATABASE, USE DATABASE, DROP DATABASE
- User, Role, and Privilege: CREATE USER, ALTER USER, DROP USER, CREATE ROLE, DROP ROLE, GRANT, REVOKE, %CHECKPRIV
- Locking: LOCK TABLE, UNLOCK TABLE
- Miscellaneous: SAVEPOINT, SET OPTION

Note that if you issue one of these SQL commands from the Management Portal Execute Query interface, the Performance information includes text such as the following: `Cached Query: %sqlcq.USER.cls16`. This appears in indicate that a cached query name was assigned. However, this cached query name is not a link. No cached query was created, and the incremental cached query number `.cls16` was not set aside. InterSystems SQL assigns this cached query number to the next SQL command issued.

5

Optimizing Query Performance

Caché SQL automatically uses a Query Optimizer to create a query plan that provides optimal query performance in most circumstances. This Optimizer improves query performance in many ways, including determining which indices to use, determining the order of evaluation of multiple AND conditions, determining the sequence of tables when performing multiple joins, and many other optimization operations. You can supply "hints" to this Optimizer in the **FROM** clause of the query. This chapter describes tools that you can use to evaluate a query plan and to modify how Caché SQL will optimize a specific query.

Caché supports the following tools for optimizing SQL queries:

- SQL Runtime Statistics to generate performance statistics on query execution.
- Index Analyzer to display various index analyzer reports for all queries in the current namespace. This shows how Caché SQL is going to execute the query, giving you an overall view of how indices are being used. This index analysis may indicate that you should add one or more indices to improve performance.
- Show Plan to display the optimal (default) execution plan for an SQL query.
- Alternate Show Plans to display available alternate execution plans for an SQL query, with statistics.
- Index Optimization Options available FROM clause options governing all conditions, or %NOINDEX prefacing an individual condition.
- Parallel Query Processing available %PARALLEL keyword FROM clause option allows multi-processor systems to divide query execution amongst the processors.

The following SQL query performance tools are described in other chapters of this manual:

- Cached Queries to enable Dynamic SQL queries to be rerun without the overhead of preparing the query each time it is executed.
- SQL Statements to preserve the most-recently compiled Embedded SQL query. In the "SQL Statements and Frozen Plans" chapter.
- Frozen Plans to preserve a specific compile of an Embedded SQL query. This compile is used rather than a more recent compile. In the "SQL Statements and Frozen Plans" chapter.

The following tools are used to optimize table data, and thus can have a significant effect on all queries run against that table:

- Defining Indices can significantly speed access to data in specific indexed fields.
- ExtentSize, Selectivity, and BlockCount to specify table data estimates before populating the table with data; this metadata is used to optimize future queries.

- Tune Table to analyze representative table data in a populated table; this generated metadata is used to optimize future queries.

This chapter also describes how to Write Query Optimization Plans to a File, and how to generate an SQL Troubleshooting Report to submit to InterSystems WRC.

5.1 Management Portal SQL Performance Tools

The Caché Management Portal provides access to the following SQL performance tools. From the Management Portal, select the **System Explorer** option. From there you select **Tools**, then select **SQL Performance Tools**, then one of the following SQL performance tools:

- SQL Runtime Statistics to generate performance statistics on query execution.
- Index Analyzer to display various index analyzer reports for all queries in the current namespace. This shows how Caché SQL is going to execute the query, giving you an overall view of how indices are being used. This index analysis may indicate that you should add one or more indices to improve performance.
- Alternate Show Plans to display available alternate execution plans for an SQL query, with statistics.
- **Generate Report** to submit an SQL performance report to InterSystems WRC (Worldwide Response Center customer support). To use this reporting tool you must first get a WRC tracking number from the WRC.
- **Import Report** allows you to view SQL query performance reports.

5.2 SQL Runtime Statistics

You can use SQL Runtime Statistics to measure the performance of query execution on your system. SQL Runtime Statistics measures the performance of **SELECT**, **INSERT**, **UPDATE**, and **DELETE** operations (collectively known as query operations). This feature is off by default. After activating it, you must recompile SQL queries.

You can use the Caché Management Portal or the %SYS.PTools.SQLStats class to collect performance statistics on an SQL query. By using this class you can determine for each SQL query: the compile time, the number of global references, the number of lines of code executed, the number of times a module is called, the total execution time, the time to first row, disk wait (the disk read access time, in milliseconds), and the number of rows processed.

Note: A system task is automatically run once per hour in all namespaces to aggregate process-specific SQL query statistics into global statistics. Therefore, the global statistics may not reflect statistics gathered within the hour. You can use the Management Portal to monitor this hourly aggregation or to force it to occur immediately. To view when this task was last finished and next scheduled, select **System Operation**, **Task Manager**, **Task Schedule** and view the `Update SQL query statistics` task. You can click on the task name for task details. From the **Task Details** display you can use the **Run** button to force the task to be performed immediately.

5.2.1 Gather Query Performance Statistics

You must activate statistics (Stats) code generation to collect performance statistics, using either of the following:

- The Management Portal **SQL Runtime Statistics** tab. (From the Management Portal, select **System Explorer**, then **Tools**, then **SQL Performance Tools**, then **SQL Runtime Statistics**).
- The **SetSQLStats()** or **SetSQLStatsJob()** method.

For either of these interfaces, you specify one of the following options: 0 turn off statistics code generation; 1 turn on statistics code generation for all queries, but do not gather statistics (the default); 2 record statistics for just the outer loop of the query (gather statistics at the open and close of the query); 3 record statistics for all module levels of the query. Modules can be nested. If so, the MAIN module statistics are inclusive numbers, the overall results for the full query.

For **SetSQLStatsJob()** the options differ slightly. They include: -1 turn off statistics for this job; 0 use the system setting value. The 1, 2, and 3 options are the same as **SetSQLStats()** and override the system setting. The default is 0.

To gather SQL Stats data, queries need to be compiled with statistics code generation turned on (option 1, the default):

- To go from 0 to 1: after changing the SQL Stats option, runtime Routines and Classes that contain SQL will need to be compiled to perform statistics code generation. For xDBC and Dynamic SQL, you must purge cached queries to force code regeneration.
- To go from 1 to 2: you simply change the SQL Stats option to begin gathering statistics. This allows you to enable SQL performance analysis on a running production environment with minimal disruption.
- To go from 1 to 3 (or 2 to 3): after changing the SQL Stats option, runtime Routines and Classes that contain SQL will need to be compiled to record statistics for all module levels. For xDBC and Dynamic SQL, you must purge cached queries to force code regeneration. Option 3 is commonly only used on an identified poorly-performing query in a non-production environment.
- To go from 1, 2, or 3 to 0: to turn off statistics code generation you do not need to purge cached queries.

This information is stored in %SYS.PTools.SQLQuery and %SYS.PTools.SQLStats.

Purging a cached query purges any related SQL Stats data. Dropping a table or view purges any related SQL Stats data.

5.2.2 Display Query Performance Statistics

You can display performance statistics for an SQL query as follows:

- From the Management Portal select **System Explorer**, then **Tools**, then select **SQL Performance Tools**, then **SQL Runtime Statistics** and click the **View Stats** tab. This gives you an overall view of the runtime statistics that have been gathered on this system.

  You can click on a **View Stats** column to sort the query statistics. You can then click Show Plan for a specific query.

- By using the **GetLastSQLStats()** method of the %SYS.PTools.SQLStats class, as shown in the following example:

```
ZNSPACE "Samples"
DO $SYSTEM.SQL.SetSQLStatsJob(2)
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 query result set",!!
KILL rset
DO ##class(%SYS.PTools.SQLStats).GetLastSQLStats()
DO %sqlcontext.DumpResults()
```

- By calling a stored procedure, as shown in the following example:

```
ZNSPACE "Samples"
DO $SYSTEM.SQL.SetSQLStatsJob(2)
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 query result set",!!
KILL rset
&sql(CALL %SYS_PTools.GetLastSQLStats())
DO %sqlcontext.DumpResults()
```

The following examples collect runtime statistics from an **INSERT** statement:

```
CreateATable
  &sql(CREATE TABLE sample.sqltest (FullName VARCHAR(25),MyDate DATE DEFAULT CURRENT_DATE))
    IF SQLCODE=0 { WRITE "sqltest table created",! }
    ELSE {WRITE "table create failed SQLCODE=",SQLCODE,! }

InsertData
  SET oldstat=$SYSTEM.SQL.SetSQLStatsJob(2)
  &sql(INSERT INTO sample.sqltest(FullName) SELECT Name FROM Sample.Person WHERE Name BETWEEN 'A' AND
 'J')
  WRITE "Inserted ",%ROWCOUNT," rows in table SQLCODE=",SQLCODE,!
  DO ##class(%SYS.PTools.SQLStats).GetLastSQLStats()
  DO %sqlcontext.DumpResults()
Cleanup
   DO $SYSTEM.SQL.SetSQLStatsJob(oldstat)
   &sql(DROP TABLE sample.sqltest)
```

You can use the SQLStatsView query to display these statistics, as shown in the following example:

```
  ZNSPACE "Samples"
  DO ##class(%SYS.PTools.SQLStats).Purge("Samples")
  DO ##class(%SYSTEM.SQL).SetSQLStatsJob(2)
  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 query result set",!!
  KILL rset
  DO ##class(%ResultSet).RunQuery("%SYS.PTools.SQLStats","SQLStatsView")
  WRITE !!,"End of SQL Statistics"
```

5.2.2.1 Exporting Query Performance Statistics

You can export query performance statistics to a text file. By default, columns in this text file are delimited by tabs. If you don't specify a filename argument, these methods create a .psql file in the `Mgr` directory, using your system ID, Caché installation directory, and Caché version to generate a file name. If you specify a filename argument, these methods create a file in the `Mgr` subdirectory for the current namespace, or in the path location you specify. This export is limited to data in the current namespace.

- The **Export()** method of %SYS.PTools.SQLStats: This method is used to export statistics data from %SYS.PTools.SQLStats classes to a delimited text file.
- The **ExportAll()** method of %SYS.PTools.SQLStats: This method is used to export from %SYS.PTools.SQLQuery and %SYS.PTools.SQLStats classes to a delimited text file. It exports the SQL statement text, the statistics data, and, optionally, the SQL Show Plan.

5.2.3 Runtime Statistics and Show Plan

The **SQL Runtime Statistics** tool can be used to display the Show Plan for a query with runtime statistics.

The **Alternate Show Plans** tool can be used to compare show plans with stats, displaying runtime statistics for a query. The Alternate Show Plans tool in its **Show Plan Options** displays estimated statistics for a query. If gathering runtime statistics is activated, its **Compare Show Plans with Stats** option displays actual runtime statistics; if runtime statistics are not active, this option displays estimate statistics.

5.3 Using Indices

Indexing provides a mechanism for optimizing queries by maintaining a sorted subset of commonly requested data. Determining which fields should be indexed requires some thought: too few or the wrong indices and key queries will run

too slowly; too many indices can slow down **INSERT** and **UPDATE** performance (as the index values must be set or updated).

5.3.1 What to Index

To determine if adding an index improves query performance, run the query from the Management Portal SQL interface and note in Performance the number of global references. Add the index and then rerun the query, noting the number of global references. A useful index should reduce the number of global references. You can prevent use of an index by using the %NOINDEX keyword as preface to a WHERE clause or ON clause condition.

You should index fields (properties) that are specified in a JOIN. A LEFT OUTER JOIN starts with the left table, and then looks into the right table; therefore, you should index the field from the right table. In the following example, you should index T2.f2:

```
FROM Table1 AS T1 LEFT OUTER JOIN Table2 AS T2 ON T1.f1 = T2.f2
```

An INNER JOIN should have indices on both ON clause fields.

Run Show Plan and follow to the first map. If the first bullet item in the Query Plan is “Read master map”, or the Query Plan calls a module whose first bullet item is “Read master map”, the query first map is the master map rather than an index map. Unless the table is relatively small, you should create an index so that when you rerun this query the Query Plan first map says “Read index map.”

You should index fields that are specified in a WHERE clause equal condition.

You may wish to index fields that are specified in a **WHERE** clause range condition, and fields specified in **GROUP BY** and **ORDER BY** clauses.

Under certain circumstances, an index based on a range condition could make a query slower. This can occur if the vast majority of the rows meet the specified range condition. For example, if the query clause `WHERE Date < CURRENT_DATE` is used with a database in which most of the records are from prior dates, indexing on Date may actually slow down the query. This is because the Query Optimizer assumes range conditions will return a relatively small number of rows, and optimizes for this situation. You can determine if this is occurring by prefacing the range condition with %NOINDEX and then run the query again.

If you are performing a comparison using an indexed field, the field as specified in the comparison should have the same collation type as it has in the corresponding index. For example, the Name field 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 field. If there is a mismatch between the field 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 this manual.

For details on how to create an index and the available index types and options, refer to the CREATE INDEX command in the *Caché SQL Reference*, and the “Defining and Building Indices” chapter of this manual.

5.3.2 Index Configuration Options

The following system-wide configuration methods can be used to optimize use of indices in queries:

- **$SYSTEM.SQL.SetDDLPKeyNotIDKey()** to use the PRIMARY KEY as the IDKey index.
- **$SYSTEM.SQL.SetFastDistinct()** to use indices for **SELECT DISTINCT** queries.

For further details, refer to SQL configuration settings described in *Caché Advanced Configuration Settings Reference*.

5.3.3 Index Usage Analysis

You can analyze index usage by SQL cached queries using either of the following:

- The Management Portal Index Analyzer SQL performance tool.
- The %SYS.PTools.SQLUtilities methods **IndexUsage()**, **TableScans()**, **TempIndices()**, and **JoinIndices()**.

5.3.4 Index Analyzer

From the Management Portal **Tools** interface, select **System Explorer**, then **Tools**, then select **SQL Performance Tools**, then **Index Analyzer**. It provides an SQL Statement Count display for the current namespace, and four index analysis report options.

5.3.4.1 SQL Statement Count

At the top of the **SQL Index Analyzer** there is an option to count all SQL statements in the namespace. Press the **Gather SQL Statements** button. The **SQL Index Analyzer** displays "Gathering SQL statements ...." while the count is in progress, then "Done" when the count is complete. SQL statements are counted in three categories: a Cached Query count, a Class Method count, and a Class Query count. These counts are for the entire current namespace, and are not affected by the **Include System Queries?** option or the **Schema Selection** option.

However, note that running an **SQL Index Analyzer Report Option** with a **Schema Selection** generates 1 cached query. Running the **Index usage** option generates an additional 3 cached queries (a total of 4 if **Schema Selection** is specified). These generated cached queries will be counted in subsequent use of **Gather SQL Statements**. Repeated use of the different **Report Option** choices with different schema selections does not generate additional cached queries.

The corresponding method is **GetSQLStatements()** in the %SYS.PTools.SQLUtilities class.

5.3.4.2 Report Options

You can either examine reports for the cached queries for a selected schema in the current namespace, or (by not selecting a schema) examine reports for all cached queries in the current namespace. You can include or exclude system queries in this analysis. The index analysis report options are:

- **Index usage**: This option takes all of the cached queries in the current namespace, generates a Show Plan for each and keeps a count of how many times each index is used by each query and the total usage for each index by all queries in the namespace. This can be used to reveal indices that are not being used so they can either be removed or modified to make them more useful. The result set is ordered from least used index to most used index.

  The corresponding method is **IndexUsage()** in the %SYS.PTools.SQLUtilities class.

- **Queries with table scans**: This option identifies all queries in the current namespace that do table scans. Table scans should be avoided if possible. A table scan can't always be avoided, but if a table has a large number of table scans, the indices defined for that table should be reviewed. Often the list of table scans and the list of temp indices will overlap; fixing one will remove the other. The result set lists the tables from largest Block Count to smallest Block Count. A Show Plan link is provided to display the Statement Text and Query Plan.

  The corresponding method is **TableScans()** in the %SYS.PTools.SQLUtilities class.

- **Queries with temp indices**: This option identifies all queries in the current namespace that build temporary indices to resolve the SQL. Sometimes the use of a temp index is helpful and improves performance, for example building a small index based on a range condition that Caché can then use to read the master map in order. Sometimes a temp index is simply a subset of a different index and might be very efficient. Other times a temporary index degrades performance, for example scanning the master map to build a temporary index on a property that has a condition. This situation indicates that a needed index is missing; you should add an index to the class that matches the temporary index. The result set lists the tables from largest Block Count to smallest Block Count. A Show Plan link is provided to display the Statement Text and Query Plan.

  The corresponding method is **TempIndices()** in the %SYS.PTools.SQLUtilities class.

- **Queries with missing JOIN Indices**: This option examines all queries in the current namespace that have joins, and determines if there is an index defined to support that join. It ranks the indices available to support the joins from 0 (no index present) to 4 (index fully supports the join). Outer joins require an index in one direction. Inner joins require an index in both directions. The result set only contains rows that have a JoinIndexFlag < 4. JoinIndexFlag=4 means there is an index that fully supports the join; these are not listed.

  The corresponding method is **JoinIndices()** in the %SYS.PTools.SQLUtilities class, which provides descriptions of the JoinIndexFlag values.

When you select one of these options, the system automatically performs the operation and displays the results. The first time you select an option or invoke the corresponding method, the system generates the results data; if you select that option or invoke that method again, Caché redisplays the same results. To generate new results data you must use the **Gather SQL Statements** button to reinitialize the Index Analyzer results tables. To generate new results data for the %SYS.PTools.SQLUtilities methods, you must invoke **GetSQLStatements()** to reinitialize the Index Analyzer results tables. Changing the **Include System Queries?** check box option also reinitializes the Index Analyzer results tables.

5.3.5 IndexUsage() Method

The following example demonstrates the use of the **IndexUsage()** method:

```
ZNSPACE "Samples"
DO ##class(%SYS.PTools.SQLUtilities).IndexUsage(1)
SET utils = "SELECT %EXACT(Type), Count(*) As QueryCount "_
            "FROM %SYS_PTools.SQLUtilities GROUP BY Type"
SET utilresults = "SELECT SchemaName, Tablename, IndexName, UsageCount "_
                  "FROM %SYS_PTools.SQLUtilResults ORDER BY UsageCount"
SET tStatement = ##class(%SQL.Statement).%New()
SET qStatus = tStatement.%Prepare(utils)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of utilities data",!!
SET qStatus = tStatement.%Prepare(utilresults)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of results data"
```

Note that because results are ordered by UsageCount, indices with UsageCount > 0 are listed at the end of the result set.

5.3.6 Index Optimization Options

By default, the Caché SQL query optimizer uses sophisticated and flexible algorithms to optimize the performance of complex queries involving multiple indices. In most cases, these defaults provide optimal performance. However, in infrequent cases, you may wish to give “hints” to the query optimizer by specifying *optimize-option* keywords.

The **FROM** clause supports the %ALLINDEX and %IGNOREINDEX *optimize-option* keywords. These *optimize-option* keywords govern all index use in the query. They are described in detail in the FROM clause reference page of the *Caché SQL Reference*.

You can use the %NOINDEX condition-level hint to specify exceptions to the use of an index for a specific condition. The %NOINDEX hint is placed in front of each condition for which no index should be used. For example, `WHERE %NOINDEX hiredate < ?`. This is most commonly used when the overwhelming majority of the data is selected (or not selected) by the condition. With a less-than (<) or greater-than (>) condition, use of the %NOINDEX condition-level hint is often beneficial. With an equality condition, use of the %NOINDEX condition-level hint provides no benefit. With a join condition, %NOINDEX is not supported for =* and *= WHERE clause outer joins; %NOINDEX is supported for ON clause joins.

The %NOINDEX keyword can be used to override indexing optimization established in the **FROM** clause. In the following example, the %ALLINDEX optimization keyword applies to all condition tests except the E.Age condition:

```
SELECT P.Name,P.Age,E.Name,E.Age
FROM %ALLINDEX Sample.Person AS P LEFT OUTER JOIN Sample.Employee AS E
     ON P.Name=E.Name
WHERE P.Age > 21 AND %NOINDEX E.Age < 65
```

5.4 Show Plan

Show Plan displays the execution plan for **SELECT**, **UPDATE**, **DELETE**, **TRUNCATE TABLE**, and some **INSERT** operations. These are collectively known as query operations because they use a **SELECT** query as part of their execution. Show Plan is performed when a query operation is prepared; you do not have to actually execute the query operation to generate an execution plan.

Show Plan displays what Caché considers to be the optimal execution plan. However, for most queries there is more than one possible execution plan. You can also display alternate show plans.

5.4.1 Displaying an Execution Plan

You can use Show Plan to display the execution plan for a query in any of the following ways:

- From the Management Portal **SQL** interface. Select **System Explorer**, then **SQL**. Select a namespace with the **Switch** option at the top of the page. (You can set the Management Portal default namespace for each user.) Write a query, then press the **Show Plan** button. (You can also invoke Show Plan from the **Show History** listing by clicking the plan option for a listed query.) See Executing SQL Statements in the "Using the Management Portal SQL Interface" chapter of this manual.
- From the Management Portal **Tools** interface: Select **System Explorer**, then **Tools**, then select **SQL Performance Tools**, then **SQL Runtime Statistics**:
  - From the **Query Test** tab: Select a namespace with the **Switch** option at the top of the page. Write a query in the text box. Then press the **Show Plan with SQL Stats** button. This generates a Show Plan without executing the query.
  - From the **View Stats** tab: Press the **Show Plan** button for one of the listed queries. The listed queries include both those written at **Execute Query**, and those written at **Query Test**.
- By running the **ShowPlan()** method, as shown in the following example:

```
ZNSPACE "Samples"
SET oldstat=$SYSTEM.SQL.SetSQLStatsJob(3)
SET mysql=2
SET mysql(1)="SELECT TOP 10 Name,DOB FROM Sample.Person "
SET mysql(2)="WHERE Name [ 'A' ORDER BY Age"
DO $SYSTEM.SQL.ShowPlan(.mysql,0,1)
DO $SYSTEM.SQL.SetSQLStatsJob(oldstat)
```

- By running Show Plan against a cached query result set, using :i%Prop syntax for literal substitution values stored as properties:

```
ZNSPACE "Samples"
SET cqsql=2
SET cqsql(1)="SELECT TOP :i%PropTopNum Name,DOB FROM Sample.Person "
SET cqsql(2)="WHERE Name [ :i%PropPersonName ORDER BY Age"
DO ShowPlan^%apiSQL(.cqsql,0,"",0,$LB("Sample"),"",1)
```

Show Plan by default returns values in Logical mode. However, when invoking Show Plan from the Management Portal or the SQL Shell, Show Plan uses Runtime mode.

5.4.2 Execution Plan: Statement Text and Query Plan

The Show Plan execution plan consists of two components, Statement Text and Query Plan:

Statement Text replicates the original SQL statement, with the following modifications: The **Show Plan** button from the Management Portal **SQL** interface displays a `SELECT` query prefaced with `DECLARE QRS CURSOR FOR` (QRS is Query Result Set). This is done to allow Show Plan to use a frozen plan. For all SQL statements, whitespace is standardized, comments are removed. The **Show Plan** button display also performs literal substitution, replacing each literal with a ?, unless you have suppressed literal substitution by enclosing the literal value in double parentheses. These modifications are not done when displaying a show plan using the **ShowPlan()** method, or when displaying alternate show plans.

Query Plan shows the plan that would be used to execute the query. A Query Plan can consist of the following:

- "Frozen Plan" is the first line of **Query Plan** if the query plan has been frozen; otherwise, the first line is blank.
- "Relative cost" is an integer value which is computed from many factors as an abstract number for comparing the efficiency of different execution plans for the same query. This calculation takes into account (among other factors) the complexity of the query, the presence of indices, and the size of the table(s). Relative cost is not useful for comparing two different queries. "Relative cost not available" is returned by certain aggregate queries, such as COUNT(*) or MAX(%ID) without a WHERE clause.
- The **Query Plan** consists of a main module, and (when needed) one or more subcomponents. One or more module subcomponents may be shown, named alphabetically, starting with B: `Module B`, `Module C`, etc.), and listed in the order of execution (not necessarily alphabetically). When the end of the alphabet is reached, additional modules are numbered, parsing Z=26, so the next module after Module Z is Module 27. A module performs processing and populates an internal temp-file (internal temporary table) with its results.

  One or more subquery subcomponents may be shown; each subquery is shown as a separate subquery module in the order specified in the query. Subquery modules are not named. If a subquery calls a module, the module is placed after the subquery and given an appropriate non-sequential alphabetical name. Therefore, a query plan could contain a main module that calls `Module B` and a `Subquery` that calls `Module H`.
- "Read master map" as the first bullet item in the main module indicates an inefficient Query Plan. The Query Plan begins execution with one of the following map type statements `Read master map...` (no available index), `Read index map...` (use available index), or `Generate a stream of idkey values using the multi-index combination...` (Multi Index, use multiple indices). Because the master map reads the data itself, rather than an index to the data, `Read master map...` almost always indicates an inefficient Query Plan. Unless the table is relatively small, you should define an index so that when you regenerate the Query Plan the first map says `Read index map...`. For information on interpreting a Query Plan, refer to "Interpreting an SQL Query Plan."

Some operations create a Show Plan that indicates no Query Plan could be generated:

- Non-query INSERT: An **INSERT... VALUES()** command does not perform a query, and therefore does not generate a Query Plan.
- Query always FALSE: In a few cases, Caché can determine when preparing a query that a query condition will always be false, and thus cannot return data. The Show Plan informs you of this situation in the Query Plan component. For example, a query containing the condition `WHERE %ID IS NULL` or the condition `WHERE Name %STARTSWITH('A') AND Name IS NULL` cannot return data, and therefore Caché generates no execution plan. Rather than generating an execution plan, the Query Plan says "`Output no rows`". If a query contains a subquery with one of these conditions, the subquery module of the Query Plan says "`Subquery result NULL, found no rows`". This condition check is limited to a few situations involving NULL, and is not intended to catch all self-contradictory query conditions.
- Invalid query: Show Plan displays an SQLCODE error message for most invalid queries. However, in a few cases, Show Plan displays as empty. For example, `WHERE Name = $$$$$` or `WHERE Name %STARTSWITH('A")` (note single-quote and double-quote). In these cases, Show Plan displays no **Statement Text**, and **Query Plan** says `[No plan created for this statement]`. This commonly occurs when quotation marks delimiting a literal are imbalanced.

It also occurs when you specify two or more leading dollar signs without specifying the correct syntax for a user-defined ("extrinsic") function.

5.5 Alternate Show Plans

You can display alternate execution plans for a query using the Management Portal or the **ShowPlanAlt()** method.

From the Management Portal **System Explorer**, select **Tools**, **SQL Performance Tools**, **Alternate Show Plans**. Using this tool, you input a query then press the **Show Plan Options** button to display multiple alternate show plans. Select the plans that you wish to compare, then press the **Compare Show Plans with Stats** button to run them and display their SQL statistics.

The **ShowPlanAlt()** method shows all of the execution plans for a query. It first shows the plan the Caché considers optimal (lowest cost), the same Show Plan display as the **ShowPlan()** method. **ShowPlanAlt()** then allows you to select an alternate plan to display. Alternate plans are listed in ascending order of cost. Specify the ID number of an alternate plan at the prompt to display its execution plan. **ShowPlanAlt()** then prompts you for the ID of another alternate plan. To exit this utility, press the return key at the prompt.

The following example displays the same execution plan as the **ShowPlan()** example, then lists alternate plans and prompts you to specify an alternate plan for display:

```
ZNSPACE "Samples"
DO $SYSTEM.SQL.SetSQLStatsJob(3)
SET mysql=1
SET mysql(1)="SELECT TOP 4 Name,DOB FROM Sample.Person ORDER BY Age"
DO $SYSTEM.SQL.ShowPlanAlt(.mysql,0,1)
```

To display an alternate plan, specify the plan's ID number from the displayed list and press Return. To exit **ShowPlanAlt()**, just press Return.

Also refer to the **PossiblePlans** methods in the %SYS.PTools.SQLUtilities class.

5.5.1 Stats

The Show Plans Options lists assigns each alternate show plan a **Cost** value, which enables you to make relative comparisons between the execution plans.

The Alternate Show Plan details provides for each Query Plan a set of stats (statistics) for the Query Totals, and (where applicable) for each Query plan module. The stats for each module include Time (overall performance, in seconds), Globals (number of global references), Commands (number of commands executed), and Disk Wait (disk read latency, in milliseconds). The Query Totals stats also includes Rows (the number of rows returned).

5.6 Writing Query Optimization Plans to a File

The following utility lists the query optimization plan(s) for one or more queries to a text file.

```
QOPlanner^%apiSQL(infile,outfile,eos,schemapath)
```

infile	A file pathname to a text file containing a listing of cached queries. Specified as a quoted string.
outfile	A file pathname where query optimization plans are to be listed. Specified as a quoted string. If the file does not exist, the system creates it. If the file already exists, Caché overwrites it.
eos	*Optional* — The end-of-statement delimiter used to separate the individual cached queries in the *infile* listing. Specified as a quoted string. The default is "GO". If this *eos* string does not match the cached query separator, no *outfile* is generated.
schemapath	*Optional* — A comma-separated list of schema names that specifies a schema search path for unqualified table names, view names, or stored procedure names. Can include DEFAULT_SCHEMA, the current system-wide default schema. If *infile* contains #Import directives, QOPlanner adds these #Import package/schema names to the end of *schemapath*.

The following is an example of evoking this query optimization plans listing utility. This utility takes as input the file generated by the **ExportSQL^%qarDDLExport()** utility, as described in "Listing Cached Queries to a File" section of the "Cached Queries" chapter. You can either generate this query listing file, or write a query (or queries) to a text file.

```
DO QOPlanner^%apiSQL("C:\temp\test\qcache.txt","C:\temp\test\qoplans.txt","GO")
```

When executed from the Terminal command line progress is displayed to the terminal screen, such as the following example:

```
Importing SQL Statements from file: C:\temp\test\qcache.txt

Recording any errors to principal device and log file: C:\temp\test\qoplans.txt

 SQL statement to process (number 1):
    SELECT TOP ? P . Name , E . Name FROM Sample . Person AS P ,
    Sample . Employee AS E ORDER BY E . Name
 Generating query plan...Done

 SQL statement to process (number 2):
    SELECT TOP ? P . Name , E . Name FROM %INORDER Sample . Person AS P
    NATURAL LEFT OUTER JOIN Sample . Employee AS E ORDER BY E . Name
 Generating query plan...Done

Elapsed time: .16532 seconds
```

The created query optimization plans file contains entries such as the following:

```
<pln>
<sql>
 SELECT TOP ? P . Name , E . Name FROM Sample . Person AS P , Sample . Employee AS E ORDER BY E . Name
</sql>
Read index map Sample.Employee.NameIDX.
Read index map Sample.Person.NameIDX.
```

```
</pln>
######
<pln>
<sql>
 SELECT TOP ? P . Name , E . Name FROM %INORDER Sample . Person AS P
    NATURAL LEFT OUTER JOIN Sample . Employee AS E ORDER BY E . Name
</sql>
Read master map Sample.Person.IDKEY.
Read extent bitmap Sample.Employee.$Employee.
Read master map Sample.Employee.IDKEY.
Update the temp-file.
Read the temp-file.
Read master map Sample.Employee.IDKEY.
Update the temp-file.
Read the temp-file.
</pln>
######
```

You can use the query optimization plan text files to compare generated optimization plans using different variants of a query, or compare optimization plans between different versions of Caché.

When exporting the SQL queries to the text file, a query that comes from a class method or class query will be preceded by the code line:

```
#import <package name>
```

This #Import statement tells the QOPlanner utility what default package/schema to use for the plan generation of the query. When exporting the SQL queries from a routine, any #import lines in the routine code prior to the SQL statement will also precede the SQL text in the export file. Queries exported to the text file from cached queries are assumed to contain fully qualified table references; if a table reference in a text file is not fully qualified, the QOPlanner utility uses the system-wide default schema that is defined on the system when QOPlanner is run.

5.7 Parallel Query Processing

The optional %PARALLEL keyword is specified in the FROM clause of a query. It suggests that Caché perform parallel processing of the query, using multiple processors (if applicable). This can significantly improve performance of some queries that uses one or more COUNT, SUM, AVG, MAX, or MIN aggregate functions, and/or a GROUP BY clause, as well as many other types of queries. These are commonly queries that process a large quantity of data and return a small result set. For example, `SELECT AVG(SaleAmt) FROM %PARALLEL User.AllSales GROUP BY Region` would likely use parallel processing.

A “one row” query that specifies only aggregate functions, expressions, and subqueries performs parallel processing, with or without a GROUP BY clause. However, a “multi-row” query that specifies both individual fields and one or more aggregate functions does not perform parallel processing unless it includes a GROUP BY clause. For example, `SELECT Name,AVG(Age) FROM %PARALLEL Sample.Person` does not perform parallel processing, but `SELECT Name,AVG(Age) FROM %PARALLEL Sample.Person GROUP BY Home_State` does perform parallel processing.

If a query that specifies %PARALLEL is compiled in Runtime mode, all constants are interpreted as being in ODBC format.

Specifying %PARALLEL may degrade performance for some queries. Running a query with %PARALLEL on a system with multiple concurrent users may result in degraded overall performance.

For further details, refer to the FROM clause in the *Caché SQL Reference*.

5.7.1 %PARALLEL Keyword Ignored

Regardless of the presence of the %PARALLEL keyword in the FROM clause, some queries may use linear processing, not parallel processing. Caché makes the decision whether or not to use parallel processing for a query after optimizing that query, applying other query optimization options (if specified). Caché may determine that the optimized form of the

query is not suitable for parallel processing, even if the user-specified form of the query would appear to benefit from parallel processing. You can determine if and how Caché has partitioned a query for parallel processing using Show Plan.

In the following circumstances specifying %PARALLEL does not perform parallel processing. The query executes successfully and no error is issued, but parallelization is not performed:

- The query contains the FOR SOME predicate.
- The query contains both a TOP clause and an ORDER BY clause. This combination of clauses optimizes for fastest time-to-first-row which does not use parallel processing. Adding the FROM clause %NOTOPOPT *optimize-option* keyword optimizes for fastest retrieval of the complete result set. If the query does not contain an aggregate function, this combination of %PARALLEL and %NOTOPOPT performs parallel processing of the query.
- A query containing a LEFT OUTER JOIN or INNER JOIN in which the ON clause is not an equality condition. For example, `FROM %PARALLEL Sample.Person p LEFT OUTER JOIN Sample.Employee e ON p.dob > e.dob`. This occurs because SQL optimization transforms this type of join to a FULL OUTER JOIN. %PARALLEL is ignored for a FULL OUTER JOIN.
- The %PARALLEL and %INORDER optimizations cannot be used together; if both are specified, %PARALLEL is ignored.
- COUNT(*) does not use parallel processing if the table has a BITMAPEXTENT index.
- %PARALLEL is intended for tables using standard data storage definitions. Its use with customized storage formats may not be supported. %PARALLEL is not supported for GLOBAL TEMPORARY tables or tables with extended global reference storage.
- %PARALLEL is intended for a query that can access all rows of a table, a table defined with row-level security (*ROWLEVELSECURITY*) cannot perform parallel processing.
- %PARALLEL is intended for use with data stored in the local database. It does not support global nodes mapped to a remote database.

5.7.2 %PARALLEL in Subqueries

%PARALLEL is intended for **SELECT** queries and their subqueries. An INSERT command subquery cannot use %PARALLEL.

%PARALLEL is ignored when applied to a subquery that is correlated with an enclosing query. For example:

```
SELECT name,age FROM Sample.Person AS p
WHERE 30<(SELECT AVG(age) FROM %PARALLEL Sample.Employee where Name = p.Name)
```

%PARALLEL is ignored when applied to a subquery that includes a complex predicate, or a predicate that optimizes to a complex predicate. Predicates that are considered complex include the %CONTAINS, %CONTAINSTERM, FOR SOME, and FOR SOME %ELEMENT predicates.

5.7.3 Shared Memory Considerations

For parallel processing, Caché supports multiple InterProcess Queues (IPQ). Each IPQ handles a single parallel query. It allows parallel work unit subprocesses to send rows of data back to the main process so the main process does not have to wait for a work unit to complete. This enables parallel queries to return their first row of data as quickly as possible, without waiting for the entire query to complete. It also improves performance of aggregate functions.

Parallel query execution uses shared memory from the generic memory heap (gmheap). Users may need to increase gmheap size if they are using parallel SQL query execution. As a general rule, the memory requirement for each IPQ is 4 x 64k = 256k. Caché splits a parallel SQL query into the number of available CPU cores. Therefore, users need to allocate this much extra gmheap:

<Number of concurrent parallel SQL requests> x <Number cores> x 256 = <required size increase (in kilobytes) of gmheap>

Note that this formula is not 100% accurate, because a parallel query can spawn sub queries which are also parallel. Therefore it is prudent to allocate more extra gmheap than is specified by this formula.

Failing to allocate adequate gmheap results in errors reported to cconsole.log. SQL queries may fail. Other errors may also occur as other subsystems try to allocate gmheap.

To review gmheap usage by an instance, including IPQ usage in particular, from the home page of the Management Portal choose **System Operation** then **System Usage**, and click the **Shared Memory Heap Usage** link; see Generic (Shared) Memory Heap Usage in the "Monitoring Caché Using the Management Portal" chapter of the *Caché Monitoring Guide* for more information.

To change the size of the generic memory heap or gmheap (sometimes known as the shared memory heap or SMH), from the home page of the Management Portal choose **System Administration** then **Configuration** then **Additional Settings** then **Advanced Memory**; see Advanced Memory Settings in the "Caché Additional Configuration Settings" chapter of the *Caché Additional Configuration Settings Reference* for more information.

5.7.4 Cached Query Considerations

If you are running a cached SQL query which uses %PARALLEL and while this query is being initialized you do something that purges cached queries, then this query could get a <NOROUTINE> error reported from one of the worker jobs. Typical things that causes cached queries to be purged are calling **$SYSTEM.SQL.Purge()** or recompiling a class which this query references. Recompiling a class automatically purges any cached queries relating to that class.

If this error occurs, running the query again will probably execute successfully. Removing %PARALLEL from the query will avoid any chance of getting this error.

5.7.5 SQL Statements and Plan State

An SQL query which uses %PARALLEL can result in multiple SQL Statements. The **Plan State** for these SQL Statements is Unfrozen/Parallel. A query with a plan state of Unfrozen/Parallel cannot be frozen by user action. Refer to the "SQL Statements" chapter for further details.

6

Interpreting an SQL Query Plan

This chapter explains the language and terms used in an SQL query access plan generated by ShowPlan.

6.1 Tables Stored In Maps

An SQL table is stored as a set of `maps`. Each table has a master map that contains all the data in the table; the table may also have other maps such as index maps and bitmaps. Each map can be pictured as a multidimensional global, with the data for some fields in one or more `subscripts`, and with the remaining fields stored in the node value. The subscripts control what data is being accessed.

- For the master map, the rowid or the *IDKEY* fields are normally used as the map subscripts.
- For an index map, normally other fields are used as the leading subscript(s), with the rowid/*IDKEY* fields as additional lower-level subscripts.
- For a bitmap, the bitmap layer can be thought of as an additional rowid subscript level. However, bitmaps can only be used for rowids that are positive integers.

6.2 Developing the Plan

Compiling an SQL query produces a set of instructions to access and return the data specified by the query. These instructions are expressed as ObjectScript code in an .INT routine.

The instructions and the sequence in which they are executed are influenced by the data the SQL compiler has about the structure and content of the tables involved in the query. The compiler attempts to use information such as table sizes and available indices to make the set of instructions as efficient as possible.

The query access plan (ShowPlan) is a human-readable translation of that resulting set of instructions. The author of the query can use this query access plan to see how the data will be accessed. While the SQL compiler tries to make the most efficient use of data as specified by the query, sometimes the author of the query knows more about some aspect of the stored data than is evident to the compiler. In this case, the author can make use of the query plan to modify the original query to provide more information or more guidance to the query compiler.

6.3 Reading the Plan

The result of "ShowPlan" is a series of statements about what processing will be done to access and present the data specified in the original query. The following provides information on how to interpret ShowPlan statements.

6.3.1 Accessing Maps

The plan for a query could access several tables. When accessing a table, the plan may access a single map (index or master map), two maps (an index map followed by the master map), or, in the case of a `multi-index plan`, several maps.

In accessing the data via a map, the plan indicates the subscripts used. It also indicates what the actual subscript values will be: a single given value, a set of given values, a range of values, or all values present in the table for that subscript. Which one is chosen depends on the conditions specified in the query. Obviously, accessing a single subscript value or only a few subscript values is faster than accessing all the values at that subscript level.

6.3.2 Conditions and Expressions

When the query is run, various conditions specified by the query are tested. Except for certain subscript-limiting conditions as just mentioned, The ShowPlan output does not explicitly indicate the testing of conditions. It is always best to test conditions as early as possible. The optimal place for testing the various conditions can be inferred from the plan details.

Similarly, ShowPlan does not detail the computation of expressions and sub-expressions. Besides simplicity, the main reason for this is that in most database environments, table and index access constitute the more important aspect of processing; the cost of retrieving the table data dominates the overall query cost, as disk access speed is still orders of magnitude slower than CPU processing.

6.3.3 Loops

When accessing data from a table, it is often necessary to examine multiple rows iteratively. Such access is indicated by a `loop`. The instructions to be executed for each pass are referred to as the `body` of the loop. They are visually indicated by being indented. It is common for database access involving multiple tables to require loops within loops. In this case, each loop level is indicated by a further indentation when compared to the previous level.

6.3.4 Temporary Files

6.3.4.1 Definition

A query plan might also indicate the need to build and use an intermediate temporary file (`temp-file`). This is a "scratch" area in a local array. It is used to save temporary results for various purposes, such as sorting. Just like a map, a temp-file has one or more subscripts, and possibly also node data.

6.3.4.2 Use

Some temp-files contain data from processing a single table. In this instance, building the temp-file could be considered `pre-processing` for the data in that table. Reading such a temp-file may or may not be followed by accessing the master map of the source table. In other cases, temp-files could contain the results of processing multiple tables. In still other situations, temp-files are used to store grouped aggregate values, to check DISTINCT, etc.

6.3.5 Modules

The building of temp-files, as well as other processing, may be delegated to a separate unit of work called a `module`. Each module is named. When separate modules are listed, the plan indicates where each module is invoked. When execution of the module finishes, processing resumes at the next statement following the module invocation.

6.3.6 Queries Sent for Processing

For external tables linked through an ODBC or JDBC gateway connection, the plan shows the text of the query being sent to the remote SQL Gateway Connection to retrieve the requested data from the remote tables.

For parallel query processing, the plan shows the various queries being sent to be processed in parallel.

6.3.7 Sub-Queries, JOINs and UNIONs

Some subqueries (and views) within the given query might also be processed separately. Their plans are specified in separate `subquery` sections. The precise place where a subquery section is called from is not indicated in the plan. This is because they are often invoked as part of the processing of conditions or expressions.

For queries that specify OUTER JOIN, the plan may indicate the possible generation of a row of NULLs if no matching rows were found, in order to satisfy the requirements of the outer join semantics.

For UNION, the plan might indicate the combining of the result rows from the various union subqueries in a separate module, where further processing of these result rows may be done.

6.4 Plan Analysis

When analyzing the plan for a given query, application developers might sometimes feel that a different plan would be more efficient. The application developer has available a variety of ways to affect the plan.

First and foremost, plans would be affected by properly running **TuneTable()** in an environment that includes actual application data. Manually defining in the class source definition some of the values that **TuneTable()** normally computes — such as table EXTENTSIZE, field SELECTIVITY, and map BlockCount — can also be used to achieve the desired plan. Refer to Use the Tune Table Facility in the "Optimizing Tables" chapter of this manual.

In addition, analyzing the plan may indicate that certain changes to the class definition could result in plans that are more efficient, for example:

6.4.1 Adding an Index

In some cases (though not always), the use of a temp-file for preprocessing can indicate that adding to the original table an index with the same or similar structure as the temp-file eliminate the need to build the temp-file. Removing this processing step from the query plan obviously could make the query run faster, but this must be balanced against the amount of work needed to maintain the index when updating the table. For further details on creating an index, refer to the Defining and Building Indices chapter of this manual.

6.4.2 Adding Fields to Index Data

When the plan shows an index being used, followed by access to the master map, this suggests that adding the master map fields being used in the query to the index node data might result in a faster plan for this query. Again, this must be balanced

against the additional update time, as well as the extra time added to the processing of other queries that use this index, since the index would be larger and thus require somewhat more read time.

6.4.3 Adding a Join Index

When the plan shows two tables being joined in a particular order (for example t1 being retrieved first, and then joined to t2 using the join condition t1.a=t2.b), it may be the case that the reverse table order would result in a faster plan. For example, if t2 has additional conditions that would significantly limit the number of qualifying rows. In that case, adding a t1 index on t1.a would enable such a join order to be considered. For further details on creating an index, refer to the Defining and Building Indices chapter of this manual.

7

SQL Statements

This list of SQL statements provide a record of SQL queries and other operations for each table, including table and index definition, insert, update, and delete. These SQL statements are linked to a query plan, and this link provides the option to freeze this query plan.

The system creates an SQL Statement for each compiled SQL operation. This provides a list of SQL DDL and DML operations listed by table, view, or procedure name. If you change the table definition, you can use this SQL Statements list to determine whether the query plan for each SQL operation will be affected by this DDL change and/or whether an SQL operation may need to be modified. You can then:

- Determine which query plan to use for each SQL operation. You can decide to use a revised query plan that reflects changes made to the table definition. Or you can freeze the current query plan, retaining the query plan generated prior to making changes to the table definition.
- Determine whether to make code changes to routines that perform SQL operations against that table, based on changes made to the table definition.

Note: SQL Statements is a listing of SQL routines that may be affected by a change to a table definition. It *should not* be used as a history of changes to either the table definition or table data.

7.1 Operations that Create SQL Statements

The following SQL operations create corresponding SQL Statements:

Data definition (DDL) operations that compile an underlying Persistent Class always create (or update and re-compile) one or more SQL Statements. Multiple statements are defined if constraints such as UNIQUE and PRIMARY KEY are defined.

Data management (DML) operations include queries against the table, and insert, update, and delete operations. Each Embedded SQL data management (DML) operation creates an SQL Statement because Embedded SQL is compiled (by default) when the routine containing it is compiled. Dynamic SQL SELECT commands create an SQL Statement when the query is prepared and is preserved as a cached query. This creates a list of the most-recently compiled versions of all SQL commands. If a query references more than one table, a single SQL Statement is created that lists all of the referenced tables in the **Table/View/Procedure Name(s)** column.

Most SQL Statements have an associated Query Plan. When created, this Query Plan is unfrozen; you can subsequently designate this Query Plan as a frozen plan. SQL Statements with a Query Plan include DDL commands and DML commands that involve a SELECT operation. SQL Statements without a Query Plan are listed in the "Plan State" section below.

Note: SQL Statements only lists the most recently compiled version of an SQL operation. Unless you freeze the SQL Statement, Caché replaces it with the next version. Thus rewriting the SQL code in a routine causes the old SQL code to disappear from SQL Statements.

7.1.1 Other SQL Statement Operations

The following SQL commands perform more complex SQL Statement operations:

- **ALTER TABLE**: Adding or removing a column does not create an SQL Statement. Instead, the existing DDL SQL Statements are recompiled, incorporating the changes to the table definition. Adding a column constraint, such as UNIQUE, does create additional SQL Statements, and recompiles existing DDL statements.
- **CREATE TRIGGER**: No SQL Statement is created in the table in which the trigger is defined, either when the trigger is defined or when it is pulled. However, if the trigger performs a DML operation on another table, defining a trigger creates an SQL Statement in the table modified by the trigger code. The **Location** specifies the table in which the trigger is defined. The SQL Statement is defined when the trigger is defined; dropping a trigger deletes the SQL Statement. Pulling a trigger does not create an SQL Statement.
- **CREATE VIEW** does not create an SQL Statement, because nothing is compiled. It also does not change the Plan Timestamp of the SQL Statements of its source table. However, compiling a DML command for a view creates an SQL Statement for that view.

7.2 Listing SQL Statements

This section describes listing SQL Statements in detail using the Management Portal interface. You can also return an index list of SQL Statements using the ^rINDEXSQL global.

Note: A system task is automatically run once per hour in all namespaces to clean up any SQL Statement Index entries that might be stale or have stale routine references. Therefore, the SQL Statements listings may not reflect all changes made within the hour. You can use the Management Portal to monitor this hourly cleanup or to force it to occur immediately. To view when this task was last finished and next scheduled, select **System Operation**, **Task Manager**, **Task Schedule** and view the `Cleanup SQL Statement Index` task. You can click on the task name for task details. From the **Task Details** display you can use the **Run** button to force the task to be performed immediately.

From the Management Portal SQL interface you can list SQL Statements as follows:

- **SQL Statements** tab: this lists all SQL Statements in the namespace, in collation sequence by schema then by table/view name within each schema. This listing only includes those tables/views for which the current user has privileges. If a SQL Statement references more than one table, the **Table/View/Procedure Name(s)** column lists all the referenced tables in alphabetical order.

  By clicking a column heading you can sort the list of SQL Statements by **Table/View/Procedure Name(s)**, **Plan State**, **Location(s)**, **SQL Statement Text**, or any other column of the list. These sortable columns enable you to quickly find, for example, all frozen plans (**Plan State**), all cached queries (**Location(s)**), or the slowest queries (**Average time**).

  You can use the **Filter** option provided with this tab to narrow the listed SQL Statements to a specified subset. A specified filter string filters on all data in the SQL Statements listing, most usefully on schema or schema.table name, routine location, or a substring found in the SQL Statement texts. A filter string is not case-sensitive. If a query references more than one table, the **Filter** includes the SQL Statement if it selects for any referenced table in the **Table/View/Procedure Name(s)** column.

- **Catalog Details** tab: select a table and display its catalog details. This tab provides an **Table's SQL Statements** button to display the SQL Statements associated with that table. Note that if a SQL Statement references more than one table, it will be listed in the **Table's SQL Statements** listing for each referenced table, but only the currently selected table is listed in the **Table Name** column.

  By clicking a column heading you can sort the list of the table's SQL Statements by any column of the list.

Both listing interfaces specify the qualified table (or view) name, the plan state, the location of the routine that defined this statement (truncated to 128 characters), and the SQL Statement text. The SQL Statement text is in normalized format, which may differ from the command text, as specified below.

You can use the **SQLTableStatements()** catalog query or INFORMATION_SCHEMA.STATEMENTS to list SQL Statements selected by various criteria, as described in Querying the SQL Statements, below.

7.2.1 Listing Columns

The **SQL Statements** tab lists all SQL statements in the namespace. The **Catalog Details** tab **Table's SQL Statements** button lists the SQL Statements for the selected table. Both listing contain the following column headings:

- **#**: a sequential numbering of the list rows. These numbers are not associated with specific SQL Statements.
- **Table/View/Procedure Name(s)**: the qualified SQL table (or view or procedure) name: schema.name. If an SQL Statement query references multiple tables or views, all of them are listed here.
- **Plan State**: see Plan State below.
- **New plan**: see New Plan Different in "Frozen Plans" chapter.
- **Natural Query**: see Statement Details Section below.
- **Count**: see Performance Statistics below.
- **Average Count**: see Performance Statistics below.
- **Total Time**: see Performance Statistics below.
- **Average Time**: see Performance Statistics below.
- **Std Dev**: see Performance Statistics below.
- **Location(s)**: the location of the compiled query, either a routine name (for Embedded SQL) or a cached query name (for Dynamic SQL). If the package name is %sqlcq, the SQL Statement is a cached query. For DDL, the persistent class name: package.class.
- **SQL Statement Text**: the SQL Statement text (truncated to 128 characters) in normalized format, which may differ from the command text, as specified in SQL Statement text below.

7.2.2 Plan State

The **Plan State** lists one of the following:

- Unfrozen: not frozen, can be frozen.
- Unfrozen/Parallel: not frozen, cannot be frozen.
- Frozen/Explicit: frozen by user action, can be unfrozen.
- Frozen/Upgrade: frozen by Caché version upgrade, can be unfrozen.
- blank: no associated Query Plan:

- An INSERT... VALUES() command creates an SQL Statement that does not have an associated Query Plan, and therefore cannot be unfrozen or frozen (the **Plan State** column is blank). Even though this SQL command does not produce a Query Plan, its listing in SQL Statements still is useful, because it allows you to quickly locate all the SQL operations against this table. For example, if you add a column to a table, you may want to find out where all of the SQL INSERTs are for that table so you can update these commands to include this new column.
- A cursor-based UPDATE or DELETE command does not have an associated Query Plan, and therefore cannot be unfrozen or frozen (the **Plan State** column is blank). The DECLARE CURSOR command does generate an SQL Statement with an associated Query Plan. Embedded SQL statements that use that cursor (**OPEN cursor**, **FETCH cursor**, **CLOSE cursor**) do not generate separate SQL Statements. Even though a cursor-based UPDATE or DELETE does not produce a Query Plan, its listing in SQL Statements is still useful, because it allows you to quickly locate all the SQL operations against this table.

7.2.3 SQL Statement Text

The SQL Statement text commonly differs from the SQL command because Caché normalizes lettercase and whitespace. Other differences are as follows:

If you issue a query from the Management Portal interface or the SQL Shell interface, the resulting SQL Statement differs from the query by preceding the SELECT statement with `DECLARE Q1 CURSOR FOR` (where "Q1" can be a variety of generated cursor names). This allows the statement text to match that of the Dynamic SQL cached query.

If the SQL command specifies an unqualified table or view name, the resulting SQL Statement provides the schema by using either a schema search path (for DML, if provided) or the system-wide default schema name (for DDL and DML).

SQL Statement Text is truncated after 1024 characters. To view the complete SQL Statement Text, display the SQL Statement Details.

A single SQL command may result in more than one SQL Statement. For example, if a query references a view, **SQL Statements** displays two statement texts, one listed under the view name, the other listed under the underlying table name. Freezing either statement results in **Plan State** of Frozen for both statements.

7.3 Data Management (DML) SQL Statements

The Data Management Language (DML) commands that create an SQL Statement are: INSERT, UPDATE, INSERT OR UPDATE, DELETE, TRUNCATE TABLE, SELECT, and DECLARE CURSOR FOR SELECT. You can use Dynamic SQL or Embedded SQL to compile (or prepare) a DML command. A DML command can be compiled for a table or a view, and Caché creates a corresponding SQL Statement.

Note: The system creates an SQL Statement when Dynamic SQL is prepared or when Embedded SQL is compiled, *not* when the SQL is executed. The SQL Statement timestamp records when the SQL code was prepared or compiled, not when (or if) it was executed. Thus an SQL Statement may represent a change to the table that was never actually performed.

Preparing a Dynamic SQL DML command creates a corresponding SQL Statement. The **Location** associated with this SQL Statement is a cached query. Dynamic SQL is prepared when SQL is executed from the Management Portal SQL interface, from the SQL Shell interface, or imported from a .txt file. Purging an unfrozen cached query deletes the corresponding SQL Statement. Purging a frozen cached query removes the **Location** value for the corresponding SQL Statement; the SQL Statement is deleted when it is unfrozen.

Compiling a non-cursor Embedded SQL Data Management Language (DML) command creates a corresponding SQL Statement. Each Embedded SQL DML command creates a corresponding SQL Statement. If a routine contains multiple

Embedded SQL commands, each Embedded SQL command creates a separate SQL Statement. (Some Embedded SQL commands create multiple SQL Statements.) The **Location** column of the SQL Statement listing specifies the routine that contains the Embedded SQL. In this way, SQL Statements maintains a record of each Embedded SQL DML command.

Compiling a cursor-based Embedded SQL Data Management Language (DML) command creates an SQL Statement for **DECLARE CURSOR** with a Query Plan. Associated Embedded SQL statements (**OPEN cursor**, **FETCH cursor**, **CLOSE cursor**) do not generate separate SQL Statements. Following a **FETCH cursor**, an associated **UPDATE table WHERE CURRENT OF cursor** or **DELETE FROM table WHERE CURRENT OF cursor** does generate a separate SQL Statement, but no separate Query Plan.

An **INSERT** command that inserts literal values creates a SQL Statement with the **Plan State** column blank. Because this command does not create a Query Plan, the SQL Statement cannot be frozen.

7.3.1 Modifying a Routine Containing Embedded SQL

When you compile a routine containing Embedded SQL, each SQL command is recorded as an entry in a statement dictionary for that routine. If you change and re-compile the routine, Caché removes all of the previous statement dictionary entries for that routine from all tables, then creates new entries corresponding to the current Embedded SQL content. If the re-compiled routine contains no Embedded SQL, the prior SQL Statements are removed and no new SQL Statements are added. Thus, by default, only the most recently compiled version of the Embedded SQL in a routine is preserved as an SQL Statement.

You can prevent a re-compile from deleting/replacing an SQL Statement by designating it as a Frozen Plan. This allows you to retain the query plan for that statement. Subsequent changes to the routine's SQL code have no effect on frozen SQL Statements.

7.3.2 SELECT Commands

Compiling (or Preparing) a query creates a corresponding SQL Statement. It can be a simple SELECT, or a CURSOR-based SELECT/FETCH operation. The query can be issued against a table or a view.

- A query containing a JOIN creates an identical SQL Statement for each table. The **Location** is the same stored query in the listing for each table. The **Statement uses the following relations** lists all of the tables, as described in the SQL Statement Details Routines and Relations Sections.
- A query containing a select-item subquery creates an identical SQL Statement for each table. The **Location** is the same stored query in the listing for each table. The **Statement uses the following relations** lists all of the tables, as described in the SQL Statement Details Routines and Relations Sections.
- A query that references an external (linked) table cannot be frozen.
- A query containing the FROM clause %PARALLEL keyword may create more than one SQL Statement. You can display these generated SQL Statements by invoking:

  ```
  SELECT * FROM INFORMATION_SCHEMA.STATEMENT_CHILDREN
  ```

  This displays the Statement column containing the statement hash of the original query and the ParentHash column containing the statement hash of a generated version of the query.

  SQL Statements for a %PARALLEL query have a Plan State of Unfrozen/Parallel, and cannot be frozen.
- A query containing no FROM clause, and therefore not referencing any table, still creates an SQL Statement. For example: `SELECT $LENGTH('this string')` creates a SQL Statement with the **Table** column value `%TSQL_sys.snf`.

7.4 SQL Statement Details

From the table's **Catalog Details** tab (or the **SQL Statements** tab), select an SQL Statement by clicking the **Statement Text** link in the right-hand column. This displays the **SQL Statement Details** in a pop-up window. You can use this **SQL Statement Details** display to view the Query Plan and to freeze or unfreeze the query plan.

In addition to the buttons to Freeze or Unfreeze the query plan, this box contains the following sections:

- Statement Details, which includes Performance Statistics
- Compile Settings
- Statement is Defined in the Following Routines
- Statement Uses the Following Relations
- Statement Text and Query Plan (described elsewhere)

7.4.1 Statement Details Section

Statement Details section:

- **Statement Hash**: an internal hash representation of the statement definition that is used as the key of the SQL Statement Index (for internal use only).
- **Timestamp**: Initially, the timestamp when the plan was created. This timestamp is updated following a freeze / unfreeze to record the time the plan was unfrozen, not the time the plan was re-compiled. You may have to click the **Refresh Page** button to display the unfreeze timestamp. Comparing the **Plan Timestamp** with the datetime value of the routine/class that contain the statement will let you know if the routine/class is not using the same query plan if it was recompiled again.
- **Version**: the Caché version under which the plan was created. If the **Plan State** is Frozen/Upgrade, this is an earlier version of Caché. When you unfreeze a query plan, the **Plan State** is changed to Unfrozen and the **Plan Version** is changed to the current Caché version.
- **Plan State**: Frozen/Explicit, Frozen/Upgrade, Unfrozen, or Unfrozen/Parallel. Frozen/Explicit means that this statement's plan has been frozen by an explicit user action and this frozen plan is the query plan that will be used, regardless of changes to the code that generated this SQL Statement. Frozen/Upgrade means that this statement's plan has been automatically frozen by a Caché version upgrade. Unfrozen means that the plan is currently unfrozen and may be frozen. Unfrozen/Parallel mean that the plan is unfrozen and uses %PARALLEL processing, and therefore cannot be frozen. A NULL (blank) plan state means that there is no associated query plan.
- **Natural Query**: a boolean flag indicating whether the query is a "natural query." If checked, the query is a natural query, and no query performance statistics are recorded. If not checked, performance statistics may be recorded; other factors determine whether statistics actually are recorded. A natural query is defined as an Embedded SQL query that is so simple that the overhead of recording statistics would affect query performance. There is no advantage to keeping statistics on a natural query, as the query is already very simple. A good example of a natural query is `SELECT Name INTO :n FROM Table WHERE %ID=?`. The WHERE clause of this query is an equality condition. This query does not involve any looping or any index references. A Dynamic SQL query (cached query) is never flagged as a natural query; statistics may or may not be recorded for a cached query.
- **Unfrozen plan different** (not displayed unless the plan is frozen): if you freeze the query plan, a **Check Frozen** button is displayed. If you select this button, this additional field is displayed, displaying whether the frozen plan is different from the unfrozen plan.
- This section also includes six query performance statistics fields which are described in the following section.

7.4.2 Performance Statistics

Executing a query adds performance statistics to the corresponding SQL Statement. This information can be used to determine which queries are the slowest and which queries are executed the most. By using this information you can determine which queries would provide significant benefits by being optimized.

In addition to the SQL Statement name, Plan state, location, and text, the following additional information is provided for cached queries:

- **Count**: an integer count of the number of times this query has been run. A change that results in a different Query Plan for this query (such as adding an index to a table) will reset this count.
- **Average Count**: the average number of times this query is run per day.
- **Total time**: the amount of time (in seconds) that running this query has taken.
- **Average time**: the average amount of time (in seconds) that running this query has taken. If the query is a cached query, the first execution of the query likely took significantly more time than subsequent executions of the optimized query from the query cache.
- **Standard Deviation**: the standard deviation of the total time and the average time. A query that is only run once has a standard deviation of 0. Queries that are run many times commonly have a lower standard deviation than those that are run only a few times.
- **Date first seen**: the date the query was first run (executed). This may differ from the **Last Compile Time**, which is when the query was prepared.

Query performance statistics are periodically updated for completed query executions by the UpdateSQLStats task. This minimizes the overhead involved in maintaining these statistics. As a consequence, currently running queries do not appear in the query performance statistics. Recently-completed queries (roughly, within the last hour) may not immediately appear in the query performance statistics.

You can use the **Clear SQL Statistics** button to clear the values of these six fields.

Caché does not separately record performance statistics for %PARALLEL subqueries. %PARALLEL subquery statistics are summed with the statistics for the outer query. Queries generated by the implementation to run in parallel do not have their performance statistics tracked individually.

Caché does not record performance statistics for "natural" queries. If the system collected statistics it would slow the query performance, and a natural query is already optimal, so there is no potential for optimization.

You can view these query performance statistics for multiple SQL statements in the SQL Statements tab display. You can sort the **SQL Statements** tab listing by any column. This makes it easy to determine, for example, which queries have the largest average time.

You can also access these query performance statistics by querying the INFORMATION.SCHEMA.STATEMENTS class properties, as described in Querying the SQL Statements.

7.4.3 Compile Settings Section

Compile Settings section:

- **Select Mode**: the SelectMode the statement was compiled with. For DDL commands, this is always Runtime. For DML commands this can be set using #SQLCompile Select; the default is Logical. If #SQLCompile Select=Runtime, a call to $SYSTEM.SQL.SetSelectMode() can change the query result set display, but does not change the **Select Mode** value, which remains Runtime.
- **Default Schema(s)**: the system-wide default schema name that were set when the statement was compiled. For DDL commands this is the schema actually used to resolve unqualified names. For DML commands this is the system-wide

default schema in effect when the command was issued, though SQL may have resolved the schema for unqualified names using a schema search path (if provided) rather than this system-wide default schema. However, if the statement is a DML command in Embedded SQL using one or more #Import macro directives, the schemas specified by **#Import** directives are listed here.

- **Schema path**: the schema path defined when the statement was compiled. For DDL commands this is always blank. For DML commands this is the schema search path, if specified. If no schema search path is specified, this setting is blank. However, for a DML Embedded SQL command with a search path specified in an #Import macro directive, the **#Import** search path is shown in the **Default schema(s)** setting and this **Schema path** setting is blank.

7.4.4 Routines and Relations Sections

Statement is defined in the following routines section:

- **Routine**: the class name associated with the table (for DDL), the cached query (for Dynamic SQL DML), or the routine name (for Embedded SQL DML).
- **Type**: Class Method or MAC Routine (for Embedded SQL DML).
- **Last Compile Time**: the last compile time or prepare time for the routine. If the SQL Statement is Unfrozen, recompiling a MAC routine updates both this timestamp and the **Plan Timestamp**. If the SQL Statement is Frozen, recompiling a MAC routine updates only this timestamp; the **Plan Timestamp** is unchanged until you unfreeze the plan; the **Plan Timestamp** then shows the time the plan was unfrozen.

Statement uses the following relations section lists one or more defined tables used to create the query plan. For an **INSERT** that uses a query to extract values from another table, or an **UPDATE** or **DELETE** that uses a FROM clause to reference another table, both tables are listed here. For each table the following values are listed:

- **Table or View Name**: the qualified name of the table or view.
- **Type**: Table or View.
- **Last Compile Time**: The time the table DDL was last compiled.
- **Classname**: the classname associated with the table.

This section includes an option to re-compile the class. If you re-compile an unfrozen plan, all three time fields are updated. If you re-compile a frozen plan, the two **Last Compile Time** fields are updated, but the **Plan Timestamp** is not. When you unfreeze the plan and click the **Refresh Page** button, the **Plan Timestamp** updates to the time the plan was unfrozen.

7.5 Data Definition (DDL) SQL Statements

Creating a table compiles a corresponding Persistent Class, and therefore creates one or more SQL Statements. Creating a view does not create a persistent class, so no SQL Statements are created.

You can create an SQL table by defining it as a persistent class, or defining it using the SQL **CREATE TABLE** command from either Embedded SQL or Dynamic SQL. Regardless of how you create a table, the system creates one or more corresponding SQL Statements. The **Location** specifies the class name associated with the table definition.

When you create a table, Caché SQL defines a bitmap extent index. In doing so, it creates a corresponding SQL Statement, such as the following:

```
DECLARE QEXTENT CURSOR FOR SELECT ID FROM SAMPLE . SQLTEST
```

Altering a table definition recompiles the existing create table SQL Statements.

When you create an index, either by modifying the table class definition or by issuing an SQL **CREATE INDEX** command, Caché stores a corresponding SQL Statement, such as the following, for each created index. The SQL Statement is the same regardless of the type of index created. Creating more than one type of index for a field does not store additional SQL statements:

```
SELECT %ID INTO :id FROM SAMPLE . SQLTEST WHERE ( :K1 IS NOT NULL AND LASTNAME = :K1
 ) OR ( :K1 IS NULL AND LASTNAME IS NULL )
```

Adding an index also causes all of the create table SQL Statements to be re-compiled, updating the Plan Timestamp for all of them, including SQL Statements for other indices. A **DROP INDEX** removes this SQL Statement and causes all of the remaining DDL SQL Statements to be re-compiled, updating the Plan Timestamp.

If you define a Primary Key or Unique constraint, Cache defines a primary key index, and therefore creates an SQL Statement such as the one above. In addition to the two SQL Statements described above, Caché adds the following four statements (in this example the LastName field):

```
SELECT 1 AS _PASSFAIL FROM SAMPLE . SQLTEST WHERE LASTNAME = :pValue(1) AND %ID <>
:id
SELECT LASTNAME INTO :tCol1 FROM SAMPLE . SQLTEST WHERE %ID = :pID
DECLARE EXT CURSOR FOR SELECT %ID INTO :tID FROM SAMPLE . SQLTEST
SELECT %ID INTO :id FROM SAMPLE . SQLTEST WHERE LASTNAME = :%d(2)
```

Each additional UNIQUE field adds three more SQL statements like these: specifying in the WHERE clause the UNIQUE field (In this example the SSN field).

```
SELECT %ID INTO :id FROM SAMPLE . SQLTEST WHERE ( :K1 IS NOT NULL AND SSN = :K1 ) OR
 ( :K1 IS NULL AND SSN IS NULL )
SELECT 1 AS _PASSFAIL FROM SAMPLE . SQLTEST WHERE SSN = :pValue(1) AND %ID <> :id
SELECT %ID INTO :id FROM SAMPLE . SQLTEST WHERE SSN = :%d(3)
```

It also modifies the following statement to include the additional UNIQUE field, such as the following:

```
SELECT SSN , LASTNAME INTO :tCol2 , :tCol1 FROM SAMPLE . SQLTEST WHERE %ID = :pID
```

7.6 Delete Table and SQL Statements

When a table is deleted, all non-frozen SQL Statements (**Plan State** Unfrozen) are deleted. Frozen statements (**Plan State** Frozen/Explicit) are not deleted, but the **Table/View/Procedure Name(s)** column is modified, as in the following example: `SAMPLE.MYTESTTABLE - Deleted?? Sample.Person`; the name of the deleted table is converted to all uppercase letters and is flagged as “Deleted??”. The **Location** column is blank for DDL statements because the table has been deleted. The **Location** column is blank for Dynamic SQL DML statements because all cached queries associated with the table have been automatically purged.

7.7 Querying the SQL Statements

You can use the **SQLTableStatements()** stored query to return the SQL Statements for a specified table. This is shown in the following example:

```
SET mycall = "CALL %Library.SQLTableStatements('Sample','Person')"
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()
IF rset.%SQLCODE '= 0 {WRITE "SQL error=",rset.%SQLCODE QUIT}
DO rset.%Display()
```

You can use the INFORMATION_SCHEMA package tables to query the list of SQL Statements. Caché supports the following classes:

- INFORMATION_SCHEMA.STATEMENTS: Contains SQL Statement Index entries that can be accessed by the current user in the current namespace.
- INFORMATION_SCHEMA.STATEMENT_LOCATIONS: Contains each routine location from which an SQL statement is invoked: the persistent class name or the cached query name.
- INFORMATION_SCHEMA.STATEMENT_RELATIONS: Contains each table or view entry use by an SQL statement.

The following are some example queries using these classes:

The following example returns all of the SQL Statements in the namespace, listing the hash value (a computed Id that uniquely identifies the normalized SQL statement), the frozen status flag (values 0 through 3), the local timestamp when the statement was prepared and the plan saved, and the statement text itself:

```
SELECT Hash,Frozen,Timestamp,Statement FROM INFORMATION_SCHEMA.STATEMENTS
```

The following example returns the SQL Statements for all frozen plans, indicating whether the frozen plan is different from what the plan would be if not frozen. Note that an unfrozen statement may be Frozen=0 or Frozen=3. A statement such as a single row INSERT, that cannot be frozen, displays NULL in the Frozen column:

```
SELECT Frozen,FrozenDifferent,Timestamp,Statement FROM INFORMATION_SCHEMA.STATEMENTS
WHERE Frozen=1 OR Frozen=2
```

The following example returns all the SQL Statements and the routines the statements are located in for a given SQL table. (Note that the table name (SAMPLE.PERSON) must be specified with the same letter case used in the SQL Statement text: all uppercase letters):

```
SELECT Statement,Frozen,STATEMENT_LOCATIONS->Location AS Routine,STATEMENT_LOCATIONS->Type AS RoutineType

       FROM INFORMATION_SCHEMA.STATEMENTS
       WHERE STATEMENT_RELATIONS->Relation='SAMPLE.PERSON'
```

The following example returns all the SQL Statements in the current namespace that have frozen plans:

```
SELECT Statement,Frozen,Frozen_Different,STATEMENT_LOCATIONS->Location AS
Routine,STATEMENT_LOCATIONS->Type AS RoutineType
       FROM INFORMATION_SCHEMA.STATEMENTS
       WHERE Frozen=1 OR Frozen=2
```

The following example returns all the SQL Statements in the current namespace that contain a **COUNT(*)** aggregate function. (Note that the statement text (COUNT (*)) must be specified with the same whitespace used in the SQL Statement text):

```
SELECT Statement,Frozen,STATEMENT_LOCATIONS->Location AS Routine,STATEMENT_LOCATIONS->Type AS RoutineType

       FROM INFORMATION_SCHEMA.STATEMENTS
       WHERE Statement [ ' COUNT ( * ) '
```

7.8 Exporting and Importing SQL Statements

You can export or import SQL Statements as an XML-formatted text file. This enables you to move a frozen plan from one location to another. SQL Statement exports and imports include the associated query plan.

You can export a single SQL Statement or export all of the SQL Statements in the namespace.

You can import a previously-exported XML file containing one or more SQL Statements.

Note: This import of SQL Statements as XML should not be confused with the import and execution of SQL DDL code from a text file.

7.8.1 Exporting SQL Statements

- Export a single SQL Statement: Use the **ExportSQLStatement()** method.
- Export all SQL Statements in the namespace: Use the **ExportAllSQLStatements()** method.

7.8.2 Importing SQL Statements

Import an SQL Statement or multiple SQL Statements from a previously-exported file: Use the **ImportSQLStatement()** method.

8

Frozen Plans

Most SQL statements have an associated Query Plan. A query plan is created when an SQL statement is prepared. By default, operations such as adding an index and recompiling the class purge this Query Plan. The next time the query is invoked it is re-prepared and a new Query Plan is created. Frozen plans enable you to retain (freeze) a existing Query Plan across compiles. Query execution uses the frozen plan, rather than performing a new optimization and generating a new query plan.

Changes to system software may also result in a different Query Plan. Usually, these upgrades result in better query performance, but it is possible that a software upgrade may worsen the performance of a specific query. Frozen plans enable you to retain (freeze) a Query Plan so that query performance is not changed (degraded or improved) by a system software upgrade.

8.1 How to Use Frozen Plans

There are two strategies for using frozen plans — the optimistic strategy and the pessimistic strategy:

- Optimistic: use this strategy if your assumption is that a change to the system software or to a class definition will improve performance. Run the query and freeze the plan. Export (backup) the frozen plan. Unfreeze the plan. Make the software change. Re-run the query. This generates a new plan. Compare the performance of the two queries. If the new plan did not improve performance, you can import the prior frozen plan from the backup file.
- Pessimistic: use this strategy if your assumption is that a change to the system software or to a class definition will probably not improve performance of a specific query. Run the query and freeze the plan. Make the software change. Re-run the query with the %NOFPLAN keyword (which causes the frozen plan to be ignored). Compare the performance of the two queries. If ignoring the frozen plan did not improve performance, keep the plan frozen and remove %NOFPLAN from the query.

8.1.1 Caché Version Upgrade Automatically Freezes Plans

When you upgrade Caché to a new major version, existing Query Plans are automatically frozen. This ensures that a major software upgrade will never degrade the performance of an existing query. After a software version upgrade, perform the following steps for performance-critical queries:

1. Execute the query with the plan state as Frozen/Upgrade and monitor performance. This is the optimized Query Plan that was created prior to the software upgrade.
2. Add the %NOFPLAN keyword to the query, then execute and monitor performance. This optimizes the Query Plan using the SQL optimizer provided with the software upgrade. It does not unfreeze the existing Query Plan.

3. Compare the performance metrics.
    - If the %NOFPLAN performance is better, the software upgrade improved the Query Plan. Unfreeze the Query Plan. Remove the %NOFPLAN keyword.
    - If the %NOFPLAN performance is worse, the software upgrade degraded the Query Plan. Keep the Query Plan frozen; promote it from Frozen/Upgrade to Frozen/Explicit. Remove the %NOFPLAN keyword.
4. After testing your performance-critical queries, you can unfreeze all remaining Frozen/Upgrade plans.

This automatic freeze occurs when you prepare/compile a query under a Caché version newer than the version under which the plan was originally created. For example, consider an SQL statement that was prepared/compiled under Caché version 2016.2. You then upgrade Caché to version 2017.1, and the SQL statement is prepared/compiled again. The system will detect this is the first prepare/compile of the SQL statement on the new version, and automatically marks the plan state as Frozen/Upgrade, and uses the existing plan for the new prepare/compile. This ensures the query plan used is no worse than the query plan of the previous version.

Only major version Caché upgrades automatically freeze existing query plans. The earliest upgrade that performs this operation is an upgrade from 2016.2.0. For example, a major version upgrade, such as from 2016.2 to 2017.1, or from 2017.1 to 2017.2 performs this operation. A maintenance release version upgrade, such as 2017.1.0 to 2017.1.1 does not perform this operation.

In the Management Portal SQL interface the **SQL Statements Plan State** column indicates these automatically frozen plans as Frozen/Upgrade and the **Plan Version** indicates the Caché version of the original plan. Refer to SQL Statement Details for further information. You can unfreeze individual plans using this interface.

You can list all Frozen/Upgrade plans in the current namespace using the INFORMATION.SCHEMA.STATEMENTS Frozen=2 property.

You can use the **FreezePlans()** method to freeze or unfreeze a single plan or multiple plans:

- **FreezePlans()** can unfreeze Frozen/Upgrade query plans within the specified scope: namespace, schema, relation (table), or individual query.
- **FreezePlans()** can promote (“freeze”) query plans flagged as Frozen/Upgrade to Frozen/Explicit. Commonly, you would use this method to selectively promote appropriate Frozen/Upgrade plans to Frozen/Explicit, then unfreeze all remaining Frozen/Upgrade plans.

Note: An upgrade from Caché version 2016.2 to a later version does not change INSERT ... SELECT query plans to Frozen/Upgrade. If the plan state was Unfrozen, the plan state for these statements will remain Unfrozen. If the plan state was Frozen, the plan will be put in an error state and will be unusable; you must explicitly unfreeze the plan. Following upgrade, you should recompile the container code (routine, class) that contains the **INSERT ... SELECT** statement, or purge and re-prepare cached queries. You can then re-freeze formerly frozen plans, if desired.

8.2 Frozen Plans Interface

There are two frozen plan interfaces, used for different purposes:

- Management Portal **SQL Statements** interface, used to freeze (or unfreeze) the plan for an individual query.
- **$SYSTEM.SQL.FreezePlans()** method interface, used to freeze or unfreeze all plans for a namespace, a schema, a table, or an individual query.

In the Management Portal SQL interface select the **Execute Query** tab. Write a query, then click the **Show Plan** button to display the current query execution plan. If the plan is frozen, the first line in the Query Plan section is "`Frozen Plan`".

In the Management Portal SQL interface select the **SQL Statements** tab. This displays a list of SQL Statements. The **Plan State** column of this list specifies Unfrozen, Unfrozen/Parallel, Frozen/Explicit, or Frozen/Upgrade. (The **Plan State** column is blank if the statement has no associated Query Plan.)

You can list the plan state for all SQL Statements in the current namespace using the INFORMATION.SCHEMA.STATEMENTS Frozen property values: Unfrozen (0), Frozen/Explicit (1), Frozen/Upgrade (2), or Unfrozen/Parallel (3).

To freeze or unfreeze a plan, choose an SQL statement in the **SQL Statement Text** column. This displays the **SQL Statement Details** box. At the bottom of this box it displays the **Statement Text and Query Plan**. The background color for these sections is green if the plan is not frozen, and blue if the plan is frozen. Just above that, under **Statement Actions**, you can select the **Freeze Plan** or **Un-Freeze Plan** button, as appropriate. You then select **Close**.

- **Freeze Plan** button: Clicking this button will cause the query optimization plan for this statement to be frozen. When a plan is frozen, and that SQL statement is compiled, the SQL compilation will use the frozen plan information and skip the query optimization phase.
- **Un-Freeze Plan** button: Clicking this button will delete the frozen plan for this statement and new compilations of this statement will go through query optimization phase to determine the best plan to use.

You can also freeze or unfreeze one or more plans using the **FreezePlans()** method. You can specify the scope of the freeze or unfreeze operation by specifying the namespace, the SQL schema name, the SQL schema.table name, or the query plan statement specified by the SQL Statement hash value.

The meaning and use of the other fields in the SQL Statement Details box are described in the "SQL Statements" chapter of this guide.

8.2.1 Privileges

A user can view only those SQL Statements for which they have execute privileges. This applies both to Management Portal **SQL Statements** listings and to INFORMATION.SCHEMA.STATEMENTS class queries.

Management Portal **SQL Statements** access requires "USE" privilege on the %Development resource. Any user that can see an SQL Statement in the Management Portal can freeze or unfreeze it.

For catalog access to SQL Statements, you can see the statements if you are privileged to execute the statement or you have "USE" privilege on the %Development resource.

For the $SYSTEM.SQL.FreezePlan() method call, you must have "U" privilege on the %Developer resource.

8.2.2 Frozen Plan Different

If a plan is frozen, you can determine if unfreezing the plan would result in a different plan without actually unfreezing the plan. This information can assist you in determining which SQL statements are worth testing using %NOFPLAN to determine if unfreezing the plan would result in better performance.

You can list all frozen plans of this type in the current namespace using the INFORMATION.SCHEMA.STATEMENTS FrozenDifferent property.

A frozen plan may be different from the current query plan due to any of the following operations:

- Recompiling the table or a table referenced by the table
- Using **SetMapSelectability()** to activate or deactivate an index
- Running TuneTable on a table
- Upgrading the Caché version

Recompiling automatically purges existing cached queries. For other operations, you must manually purge existing cached queries for a new query plan to take effect.

These operations may or may not result in a different query plan. There are two ways to determine if they do:

- Manually checking individual frozen plans
- Automatically scanning all frozen plans on a daily basis

If the plan has not yet been checked by either of these operations, or a plan is not frozen, the SQL Statements listing **New Plan** column is blank. Unfreezing a checked frozen plan resets the **New Plan** column to blank.

8.2.2.1 Manual Frozen Plan Check

At the top of the SQL Statement Details page for a frozen plan there is a **Check frozen** button. Pressing this button displays the **Unfrozen plan different** check box. If this box is checked, unfreezing the plan would result in a different query plan.

When you have performed this **Check frozen** test on a frozen plan:

- If the **Unfrozen plan different** box is checked, the SQL Statements listing **New Plan** column contains a "1". This indicates that unfreezing the plan would result in a different plan.
- If the **Unfrozen plan different** box is not checked, the SQL Statements listing **New Plan** column contains a "0". This indicates that unfreezing the plan would not result in a different plan.
  - A cached query that has been frozen has a **New Plan** of "0"; purging the cached query and then unfreezing the plan causes the SQL statement to disappear.
  - A Natural Query that has been frozen has a blank in the **New Plan** column.

After performing this test, the **Check frozen** button disappears. If you wish to re-test a frozen plan, select the **Refresh Page** button. This re-displays the **Check frozen** button.

8.2.2.2 Automatic Daily Frozen Plan Check

Caché SQL automatically scans all frozen statements in the SQL Statement listing every night at 2:00am. This scan lasts for, at most, one hour. If the scan is not completed in one hour, Caché notes where it left off, and continues from that point on the next daily scan. You can use the Management Portal to monitor this daily scan or to force it to scan immediately: select **System Operation**, **Task Manager**, **Task Schedule**, then select the `Scan frozen plans` task.

This scan examines all frozen plans:

- If the frozen plan has the same Caché version as the current version, Caché compute a hash on referenced tables and timestamps of the two plans to create an internal list of plans that may have changed. For this subset it then performs a string-for-string comparison of the two plans to determine which plans actually differ. If there is any difference between the two plans (however minor), it flags the SQL statement in the SQL Statements listing **New Plan** column with a "1". This indicates that unfreezing the plan would result in a different query plan.
- If the frozen plan has the same Caché version as the current version and string-for-string comparison of the two plans is an exact match, it flags the SQL statement in the SQL Statements listing **New Plan** column with a "0". This indicates that unfreezing the plan would not result in a different query plan.
- If the frozen plan has a different Caché version from the current version (Frozen/Update), Caché determines if a change to the SQL optimizer logic would result in a different query plan. If so, it flags the SQL statement in the SQL Statements listing **New Plan** column with a "1". Otherwise, it flags the SQL statement **New Plan** column with a "0".

You can check the results of this scan by invoking INFORMATION.SCHEMA.STATEMENTS. The following example returns the SQL Statements for all frozen plans, indicating whether the frozen plan is different from what the plan would be if not frozen. Note that an unfrozen statement may be Frozen=0 or Frozen=3:

```
SELECT Frozen,FrozenDifferent,Timestamp,Statement FROM INFORMATION_SCHEMA.STATEMENTS
WHERE Frozen=1 OR Frozen=2
```

8.2.3 Frozen Plan in Error

If a statement's plan is frozen, and something changes to a definition used by the plan to cause the plan to be invalid, an error occurs. For example, if an index was deleted from the class that was used by the statement plan:

- The statement's plan remains frozen.
- On the SQL Statement Details page the **Compile Settings** area displays a **Plan Error** field. For example, if a query plan used an index name indxdob and then you modified the class definition to drop index indxdob, a message such as the following displays: `Map 'indxdob' not defined in table 'Sample.Mytable', but it was specified in the frozen plan for the query.`
- On the SQL Statement Details page the **Query Plan** area displays `Plan could not be determined due to an error in the frozen plan`.

If the query is [re]compiled and the frozen plan is in an error state, Caché does not use the frozen plan. Instead, the system creates a new Query Plan that will work given the current definitions. However, this Query Plan is not preserved in a cached query or an SQL Statement if a frozen plan is in effect.

The plan in error remains in error until either the plan is unfrozen, or the definitions are modified to bring the plan back to a valid state.

If you modify the definitions to bring the plan back to a valid state, go to the SQL Statement Details page and press the **Clear Error** button to determine if you have corrected the error. If corrected, the **Plan Error** field disappears; otherwise the **Plan Error** message re-displays. If you have corrected the definition, you do not have to explicitly clear the plan error for Caché to begin using the frozen plan. If you have corrected the definition, the **Clear Error** button causes the SQL Statement Details page **Frozen Query Plan** area to again display the execution plan.

A **Plan Error** may be a “soft error.” This can occur when the plan uses an index, but that index is currently not selectable by the query optimizer because its selectability has been set to 0 by **SetMapSelectability()**. This was probably done so the index could be [re]built. When Caché encounters a soft error for a statement with a frozen plan, the query processor attempts to clear the error automatically and use the frozen plan. If the plan is still in error, the plan is again marked in error and query execution uses the best plan it can.

8.3 %NOFPLAN Keyword

You can use the %NOFPLAN keyword to override a frozen plan. An SQL statement containing the %NOFPLAN keyword generates a new query plan. The frozen plan is retained but not used. This allows you to test generated plan behavior without losing the frozen plan.

The syntax of %NOFPLAN is as follows:

```
DECLARE <cursor name> CURSOR FOR SELECT %NOFPLAN ...
SELECT %NOFPLAN ....
INSERT [OR UPDATE] %NOFPLAN ...
DELETE %NOFPLAN ...
UPDATE %NOFPLAN
```

In a **SELECT** statement the %NOFPLAN keyword can only be used immediately after the first **SELECT** in the query: it can only be used with the first leg of a **UNION** query, and cannot be used in a subquery. The %NOFPLAN keyword must immediately follow the **SELECT** keyword, preceding other keywords such as DISTINCT or TOP.

8.4 Exporting and Importing Frozen Plans

You can export or import SQL Statements as an XML-formatted text file. This enables you to move a frozen plan from one location to another. SQL Statement exports and imports include an encoded version of the associated query plan and a flag indicating whether the plan is frozen.

To export an SQL Statement to a file, use the **ExportSQLStatement()** method.

To export all of the SQL Statement entries in this namespace to a file, use the **ExportAllSQLStatements()** method.

To import an SQL Statement or multiple SQL Statements from a file, use the **ImportSQLStatement()** method.