Configure SQL Performance Options

Version 2023.3
2024-05-16

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

Configure SQL Performance Options
InterSystems IRIS Data Platform Version 2023.3 2024-05-16

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

1 Configure Parallel Query Processing 1
- 1.1 System-Wide Parallel Query Processing 1
- 1.2 Parallel Query Processing for a Specific Query 2
  - 1.2.1 %PARALLEL in Subqueries 2
- 1.3 Parallel Query Processing Ignored 2
- 1.4 Shared Memory Considerations 3
- 1.5 SQL Statements and Plan State 3

2 Using Runtime Plan Choice 5
- 2.1 Application of RTPC 5
- 2.2 Overriding or Disabling RTPC 6

3 Configure Frozen Plans 7
- 3.1 How to Use Frozen Plans 7
- 3.2 Frozen Plans After Software Version Upgrade 7
- 3.3 Frozen Plans Interface 8
  - 3.3.1 Privileges 8
  - 3.3.2 Frozen Plan Different 9
  - 3.3.3 Frozen Plan in Error 9
- 3.4 %NOFPLAN Keyword 10
- 3.5 Exporting and Importing Frozen Plans 10

4 Using Adaptive Mode to Improve Performance 11
- 4.1 Runtime Plan Choice 11
- 4.2 System-Wide Auto-Parallel 11
- 4.3 Auto-Tune 12
- 4.4 Turning Adaptive Mode Off 12

5 Working with Cached Queries 13
- 5.1 Cached Queries Improve Performance 14
- 5.2 Creating a Cached Query 15
  - 5.2.1 Cached Query Names for Dynamic SQL 15
  - 5.2.2 Cached Query Names for Embedded SQL 16
  - 5.2.3 Separate Cached Queries 16
- 5.3 Literal Substitution 17
  - 5.3.1 Literal Substitution and Performance 18
  - 5.3.2 Suppressing Literal Substitution 18
- 5.4 Cached Query Result Set 18
- 5.5 Listing Cached Queries 18
  - 5.5.1 Counting Cached Queries 18
  - 5.5.2 Listing Cached Queries 19
  - 5.5.3 Listing Tables Referenced by a Cached Query 19
- 5.6 Executing Cached Queries 20
- 5.7 Cached Query Lock 20
- 5.8 Purging Cached Queries 20
  - 5.8.1 Remote Systems 21
- 5.9 SQL Commands That Are Not Cached 21

6 Specify Optimization Hints in Queries 23
- 6.1 FROM Clause Keywords 23

6.1.1 %ALLINDEX 23
6.1.2 %FIRSTTABLE 24
6.1.3 %FULL 24
6.1.4 %IGNOREINDEX 24
6.1.5 %INORDER 25
6.1.6 %NOFLATTEN 25
6.1.7 %NOMERGE 25
6.1.8 %NOREDUCE 26
6.1.9 %NOSVSO 26
6.1.10 %NOTOPOPT 26
6.1.11 %NOUNIONOROPT 26
6.1.12 %PARALLEL 27
6.1.13 %STARTTABLE 27
6.2 Comment Options 28
6.2.1 Syntax 28
6.2.2 Cosharding Comment Option 29
6.2.3 DynamicSQLTypeList Comment Option 29
6.2.4 Display 30

1

Configure Parallel Query Processing

Parallel query hinting directs the system to perform parallel query processing when running on a multi-processor system. This can substantially improve performance of certain types of queries. The SQL optimizer determines whether a specific query could benefit from parallel processing, and performs parallel processing where appropriate. Specifying a parallel query hint does not force parallel processing of every query; it is only applied to those queries that may benefit from parallel processing. If the system is not a multi-processor system, this option has no effect. To determine the number of processors on the current system use the **%SYSTEM.Util.NumberOfCPUs()** method.

Adaptive Mode controls parallel query processing by default. If it has been turned off, you can specify parallel query processing in two ways:

- System-wide, by setting the auto parallel option.
- Per query, by specifying the %PARALLEL keyword in the FROM clause of an individual query.

Parallel query processing is applied to **SELECT** queries. It is not applied to **INSERT**, **UPDATE**, or **DELETE** operations.

Avoid parallel processing for queries that involve process-specific functions such as $job and $tlevel. Also avoid them in queries of process-specific variables such as %ROWID.

1.1 System-Wide Parallel Query Processing

When Adaptive Mode is off, you can still turn on system-wide parallel query processing separately by using either of the following options:

- From the Management Portal choose **System Administration**, then **Configuration**, then **SQL and Object Settings**, then **SQL**. View or change the **Execute queries in a single process** check box. Note that the default for this check box is unselected, which means that parallel processing is activated by default.
- Invoke the **$SYSTEM.SQL.Util.SetOption()** method, as follows: `SET status=$SYSTEM.SQL.Util.SetOption("AutoParallel",1,.oldval)`. The default is 1 (automatic parallel processing activated). To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()** which displays the `Enable auto hinting for %PARALLEL` option.

Note that changing this configuration setting purges all cached queries in all namespaces.

For more information about what system-wide parallel query processing entails, see Auto-Parallel in Using Adaptive Mode to Improve Performance.

1.2 Parallel Query Processing for a Specific Query

The optional %PARALLEL keyword is specified in the FROM clause of a query. It suggests that InterSystems IRIS 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.

Parallel processing can be performed when querying a view. However, parallel processing is never performed on a query that specifies a %VID, even if the %PARALLEL keyword is explicitly specified.

For further details, refer to the FROM clause.

1.2.1 %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:

SQL

```
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 FOR SOME and FOR SOME %ELEMENT predicates.

1.3 Parallel Query Processing Ignored

Regardless of the auto parallel option setting or the presence of the %PARALLEL keyword in the FROM clause, some queries may still use linear processing, not parallel processing. InterSystems IRIS makes the decision whether or not to use parallel processing for a query after applying other query optimization options (if specified). InterSystems IRIS 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 InterSystems IRIS 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 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.
- The query references a view and returns a view ID (%VID).
- %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.

1.4 Shared Memory Considerations

For parallel processing, InterSystems IRIS 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.

Failing to allocate adequate gmheap results in errors reported to messages.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 for more information.

1.5 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 SQL Statements for further details.

2

Using Runtime Plan Choice

Runtime Plan Choice (RTPC) is a configuration option that allows the SQL optimizer to take advantage of outlier value information at run time (query execution time). When Adaptive Mode is turned on, RTPC is turned on; however, you may still turn RTPC on even if you have turned Adaptive Mode off.

When RTPC is activated, query preparation includes detecting whether the query contains a condition on a field that has an outlier value. If the prepare detects one or more outlier field conditions, the query is not sent to the optimizer; instead, a Runtime Plan Choice stub is generated. At execution time, the optimizer uses this stub to choose which query plan to execute: a standard query plan that ignores outlier status or an alternative query plan that optimizes for outlier status. If there are multiple outlier value conditions, the optimizer can choose from multiple alternative run time query plans.

Note that the display of the RTPC query plan differs based on the source of the SQL code:

- The **Show Plan** button in Management Portal SQL interface may display an alternative run time query plan because this Show Plan takes its SQL code from the SQL interface text box.
- If the statement has values for all parameters, the plan returned takes those values into account when preparing an optimal plan using RTPC. However, if any parameters have ? as a placeholder, a plan that does not make use of parameter values and contains the text "`Different parameters may use a different plan`" is returned.
- If RTPC is not activated, or the query does not contain appropriate outlier field conditions, the optimizer creates a standard SQL Statement and a corresponding cached query.
- If an RTPC stub is frozen, all associated alternative run time query plans are also frozen. RTPC processing remains active for a frozen query even when the RTPC configuration option is turned off.
- You can manually suppress literal substitution when writing the query by specifying parentheses: `SELECT Name,HaveContactInfo FROM t1 WHERE HaveContactInfo=(('Yes'))`. If you suppress literal substitution of the outlier field in a condition, RTPC is not applied to the query. In this case, the optimizer creates a standard cached query with literal substitution.

2.1 Application of RTPC

For SELECT and CALL statements, the system applies RTPC to any field that Tune Table has determined to have an outlier value, when that field is specified in a condition where it is compared to a literal. This comparison condition can be:

- A WHERE clause condition using an equality (=), non-equality (!=), IN, or %INLIST predicate.
- An ON clause join condition with an equality (=), non-equality (!=), IN, or %INLIST predicate.

If RTPC is applied, the optimizer determines at run time whether to apply the standard query plan or an alternative query plan.

RTPC is not applied to INSERT, UPDATE, or DELETE statements or in the following cases:

- The query contains unresolved ? input parameters.
- The query specifies the literal value surrounded by double parentheses, suppressing literal substitution.
- The literal is supplied to the outlier field condition by a subquery that does not contain an outlier field condition.

2.2 Overriding or Disabling RTPC

You can override RTPC for a specific query by specifying the %NORUNTIME restriction keyword. If the query `SELECT Name,HaveContactInfo FROM t1 WHERE HaveContactInfo=?` would result in RTPC processing, the query `SELECT %NORUNTIME Name,HaveContactInfo FROM t1 WHERE HaveContactInfo=?` would override RTPC, resulting in a standard query plan.

When Adaptive Mode is turned off, you can enable or disable RTPC for all processes system-wide using the **$SYSTEM.SQL.Util.SetOption()** method, as follows: `SET status=$SYSTEM.SQL.Util.SetOption("RTPC",flag,.oldval)`. The *flag* argument is a boolean used to set (1) or unset (0) RTPC. The *oldvalue* argument returns the prior RTPC setting as a boolean value.

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

3

Configure 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.

3.1 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.

3.2 Frozen Plans After Software Version Upgrade

By default, when you upgrade InterSystems IRIS® data platform to a new major version, existing query plans are invalidated and the system generates a new optimized query plan upon the statement's first execution on the new system. This new query plan employs any enhancements made to SQL processing, such as improvements to the query optimizer, Runtime Plan Choice, and code generation. This behavior applies when you upgrade an instance that has Adaptive Mode enabled, which is the default for new installations.

If Adaptive Mode is off when you upgrade to a new major version, query plans are automatically marked as Frozen/Upgrade. Further invocations of the query continue to use this plan. This behavior may be desirable in highly controlled environments, as it ensures query performance remains the same across upgrades, even though it may imply missed opportunities to take advantage of SQL processing enhancements. After such an upgrade, you can manually unfreeze query plans immediately or test their performance using %NOFPLAN. The following steps describe how to compare the performance of a frozen query plan against the performance of a new query plan generated by the new version of InterSystems IRIS:

1. Execute the frozen query plan and monitor its performance.
2. Add the %NOFPLAN keyword to the query, then execute and monitor performance. This keyword 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 and remove the %NOFPLAN keyword.
4. After testing your performance-critical queries, you can unfreeze all remaining frozen plans.

Query plans that had been marked as Frozen/Upgrade can be promoted to Frozen/Explicit by using any of the methods described in Frozen Plans Interface. Commonly, you would perform this upgrade to selectively promote Frozen/Upgrade plans that you want to retain, then unfreeze all remaining Frozen/Upgrade plans.

Note: The behavior described in this section applies to upgrades to InterSystems 2023.3 or above, where the instance being upgraded from has Adaptive Mode enabled, as introduced in InterSystems IRIS 2022.2. When upgrading to a release below 2023.3 or upgrading from a release below 2022.2, Adaptive Mode is considered to be off and the corresponding automatic freezing of query plans applies as described above.

3.3 Frozen Plans Interface

You can use the FREEZE PLANS and UNFREEZE PLANS commands to freeze and unfreeze query plans individually, by table, by schema, or by namespace. To freeze or unfreeze an individual plan, find the Hash for the desired Statement by querying INFORMATION_SCHEMA.STATEMENTS. You can then use FREEZE PLANS or UNFREEZE PLANS to change the plan state of a specific statement by providing the Statement's Hash.

You can list the plan state for all SQL Statements in the current namespace by querying the INFORMATION_SCHEMA.STATEMENTS table for the Frozen property. The values in the Frozen column can be: Unfrozen (0), Frozen/Explicit (1), Frozen/Upgrade (2), or Unfrozen/Parallel (3). You can also use EXPLAIN on a specific query to determine whether it is frozen or not.

You can also freeze or unfreeze one or more plans using the $SYSTEM.SQL.Statement Freeze and Unfreeze methods. You can specify the scope of the freeze or unfreeze operation by specifying the appropriate method: **FreezeStatement()** for a single plan; **FreezeRelation()** for all plans for a relation; **FreezeSchema()** for all plans for a schema; **FreezeAll()** for all plans in the current namespace. There are corresponding Unfreeze methods.

3.3.1 Privileges

A user can view only those SQL Statements for which they have execute privileges, including for INFORMATION.SCHEMA.STATEMENTS class queries. 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 $SYSTEM.SQL.Statement Freeze or Unfreeze method calls, you must have “U” privilege on the %Developer resource.

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.

3.3.2 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 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. You can view whether or not an index is active by querying the INFORMATION_SCHEMA.INDEXES catalog table and viewing the value of the MAP_SELECTABLE column.
- Running TuneTable on a table.
- Upgrading the InterSystems software 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. You can scan all frozen plans to determine whether a new query plan will be generated.

3.3.2.1 Automatic Daily Frozen Plan Check

InterSystems 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, the system notes where it left off, and continues from that point on the next daily scan.

Additionally, you can use the Management Portal to force the scan to occur: select **System Operation**, **Task Manager**, **Task Schedule**, then select the `Scan frozen plans` task.

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:

SQL

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

3.3.3 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-executed while the frozen plan is in an error state, InterSystems IRIS does not use the frozen plan. Instead, the system creates a new Query Plan that will work given the current definitions and executes the query. This Query Plan is assigned the same cached query class name as the prior Query Plan.

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 SQL 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 InterSystems IRIS 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.

3.4 %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.

3.5 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. For details, refer to Exporting and Importing SQL Statements.

4

Using Adaptive Mode to Improve Performance

InterSystems SQL packages multiple options that define query planning and execution behavior in Adaptive Mode, an on by default setting that ensures the best out of the box performance for a wide set of use cases. Specifically, Adaptive Mode controls Runtime Plan Choice (RTPC), parallel processing, and automatically runs TUNE TABLE to optimize the efficiency of query execution. The individual features that Adaptive Mode governs cannot be controlled independently without turning Adaptive Mode off.

4.1 Runtime Plan Choice

RTPC is turned on when Adaptive Mode is active, enabling the optimizer to take advantage of runtime parameter values supplied within a query to determine the most optimal plan. For more information about the how RTPC functions, see Configure Runtime Plan Choice

Note that RTPC can only be effective when it has accurate information about the contents of the table. As a result, the efficacy of RTPC relies on the table statistics collected by calling TUNE TABLE.

4.2 System-Wide Auto-Parallel

When Adaptive Mode is on, system-wide automatic parallel query processing is turned on. Consequently, all SELECT queries are automatically hinted with %PARALLEL so that parallel processing is applied to any query that may benefit from it.

However, the automatic hinting does not mean that all queries are executed with parallel processing. The SQL Optimizer may decide that a query is not a good candidate for parallel processing and ignore the hint. For a list of some examples where parallel processing is not applied, even when Adaptive Mode is applied, see Parallel Query Processing Ignored.

The auto parallel threshold configuration parameter will also determine whether a query is executed in parallel or not. The higher the threshold value is, the lower the chance that parallel processing will be used. This threshold is used in complex optimization calculations, but you can think about this value as the minimal number of tuples that must reside in the visited map before the optimizer considers parallel query processing worthwhile. By default, this value is 3200. The minimum is 0. You may change the value of this setting by using

`$SYSTEM.SQL.Util.SetOption("AutoParallelThreshold",n,.oldval)`, where `n` is the value you would like to set for the auto parallel threshold (`.oldval` is a return variable that will hold the overwritten value).

In a sharded environment, all queries will be executed with parallel processing, regardless of the parallel threshold value, when Adaptive Mode is turned on.

For more information about memory usage with parallel processing, see Shared Memory Considerations in Configure Parallel Processing.

4.3 Auto-Tune

For the query optimizer to choose the most efficient query plans, your tables will need to have an up-to-date set of statistics that accurately describe the data contained within the table. These statistics can be gathered through an efficient sampling using the TUNE TABLE command or by hard-coding them in the table definition. New tables will typically not have these statistics until a call to TUNE TABLE has been made. Adaptive Mode ensures that TUNE TABLE is run automatically on tables that do not have any statistics and are eligible for fast block sampling before the first query issued against them is executed. This facility ensures that no query misses out on the substantial performance benefits that come with gathering a set of table statistics.

For more information about some of the statistics that TUNE TABLE collects, see Table Statistics for Query Optimizer.

Note: It is still appropriate to run TUNE TABLE after a table has been loaded with representative data or making significant changes to its contents. The auto-tune facility only operates once, when the table is queried for the first time, which may occur before the table is populated with representative data.

4.4 Turning Adaptive Mode Off

In the uncommon case that you want to turn Adaptive Mode off, you can do so by navigating to **System Administration** > **Configuration** > **SQL and Object Settings** > **SQL** and selecting the check box labelled “Turn off Adaptive Mode to disable run time plan choice and automatic tuning.”

When Adaptive Mode is off, InterSystems recommends that you consider manually configuring Runtime Plan Choice and parallel processing based on the needs of your application, and that you run TUNE TABLE manually when the distribution of your data changes significantly.

5

Working with Cached Queries

The system automatically maintains a cache of prepared 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. Preparing a query occurs at runtime, not when the routine containing the SQL query code is compiled. Commonly, a prepare is immediately followed by the first execution of the SQL statement, though in Dynamic SQL it is possible to prepare a query without executing it. Subsequent executions ignore the prepare statement and instead access the cached query. To force a new prepare of an existing query it is necessary to purge the cached query.

All invocations of SQL create cached queries, whether invoked in an ObjectScript routine or a class method.

- Dynamic SQL, ODBC, JDBC, and the **$SYSTEM.SQL.DDLImport()** method create a cached query when the query is prepared. The Management Portal execute SQL interface, the InterSystems SQL Shell, and the **%SYSTEM.SQL.Execute()** method use Dynamic SQL, and thus use a prepare operation to create cached queries.

  They are listed in the Management Portal general **Cached Queries** listing for the namespace (or specified schema), the Management Portal Catalog Details **Cached Queries** listings for each table being accessed, and the SQL Statements listings. Dynamic SQL follows the cached query naming conventions described in this chapter.

- Class Queries create a cached query upon prepare (%PrepareClassQuery() method) or first execution (CALL).

  They are listed in the Management Portal general **Cached Queries** listing for the namespace. If the class query is defined in a Persistent class, the cached query is also listed in the Catalog Details **Cached Queries** for that class. It is not listed in the Catalog Details for the table(s) being accessed. It is not listed in the SQL Statements listings. Class queries follow the cached query naming conventions described in this chapter.

- Embedded SQL creates a cached query upon first execution of the SQL code, or the initiation of code execution by invoking the OPEN command for a declared cursor. Embedded SQL cached queries are listed in the Management Portal **Cached Queries** listings with a **Query Type** of `Embedded cached SQL`, and the SQL Statements listings. Embedded SQL cached queries follow a different cached query naming convention.

Cached queries of all types are deleted by all purge cached queries operations.

SQL query statements that generate a cached query 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 **$SYSTEM.SQL.Schema.ImportDDL("IRIS")** 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.

- **CALL**: creates a cached query shown in the **Cached Queries** list for its schema.

- **INSERT**, **UPDATE**, **INSERT OR UPDATE**, **DELETE**: create a cached query shown in the **Catalog Details** for its table.
- **TRUNCATE TABLE**: creates a cached query shown in the **Catalog Details** for its table. Note that **$SYSTEM.SQL.Schema.ImportDDL("IRIS")** does not support **TRUNCATE TABLE**.
- **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.

Changing the **SetMapSelectability()** value for a table invalidates all existing cached queries that reference that table. A subsequent Prepare of an existing query creates a new cached query and removes the old cached query from the listing.

A cached 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.

5.1 Cached Queries Improve Performance

When you first prepare a query, the SQL Engine optimizes it and generates a program (a set of one or more InterSystems IRIS® data platform routines) that will execute the query. The optimized query text is then stored as a cached 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 InterSystems IRIS.
- 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. For other considerations that result in two “identical” queries resulting in separate cached queries, see below.
- 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.

InterSystems SQL stores all cached queries in a single location, the IRISLOCALDATA 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. However, it can include comment options following the query text, such as `/*#OPTIONS {"optionName":value} */`.

Since a cached query uses an existing query plan, it provides continuity of operation for existing queries. Changes to the underlying tables such as adding indexes or redefining the table optimization statistics have no effect on an existing cached query if the plan is frozen. For use of cached queries when changing a table definition, refer to the "SQL Statements and Frozen Plans" chapter in this manual.

5.2 Creating a Cached Query

When InterSystems IRIS 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.

5.2.1 Cached Query Names for Dynamic SQL

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

`%sqlcq.`*`namespace`*`.cls`*`nnn`*

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, different comment options, 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:

ObjectScript

```
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.

5.2.2 Cached Query Names for Embedded SQL

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

`%sqlcq.`*`namespace.hash`*

Where *namespace* is the current namespace, in capital letters, and *hash* is a unique hash value. For example, `%sqlcq.USER.xEM1h5QIeF4l3jhLZrXlnThVJZDh`.

Embedded SQL cached queries are listed in the Management Portal for each table in the **Catalog Details Cached Queries** listing for each table with this **Class Name** and a **Query Type** of `Embedded cached SQL`.

5.2.3 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.

5.3 Literal Substitution

When the SQL Engine caches an 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:

SQL

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

SQL

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

Are both represented by a single cached query:

SQL

```
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,64701)` and `DATENAME('MONTH',64701)` 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.

5.3.1 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.

5.3.2 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:

SQL

```
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.

5.4 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.

5.5 Listing Cached Queries

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

- Counting cached queries
- Listing cached queries using the InterSystems IRIS Management Portal
- Listing tables referenced by a cached query or a stored procedure using the InterSystems IRIS Management Portal

5.5.1 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:

ObjectScript

```
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.

5.5.2 Listing Cached Queries

You can list (and manage) the contents of the query cache using the InterSystems IRIS 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 Dynamic SQL`: a Dynamic SQL query using %SQL.Statement.
- `Embedded cached SQL`: an Embedded SQL query.
- `ODBC/JDBC Statement`: a dynamic query from either ODBC or JDBC.

The **Statement** is displayed for a Dynamic SQL cached query. This consists of the statement hash, which is a selectable link that takes you to the SQL Statement Details, and the plan state, such as `(Unfrozen)` or `(Frozen/Explicit)`.

When you successfully prepare an SQL statement, the system generates a new class that implements the statement. If you have set the **Retain cached query 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 InterSystems IRIS Management Portal. From **System Administration**, select **Configuration**, then **SQL and Object Settings**, then **SQL**. On this screen you can set the **Retain cached query 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 **$SYSTEM.SQL.Util.SetOption()** method, as follows: `SET status=$SYSTEM.SQL.Util.SetOption("CachedQuerySaveSource",flag,.oldval)`. The *flag* argument is a boolean used to retain (1) or not retain (0) query source code after a cached query is compiled; the default is 0. To determine the current setting, call **$SYSTEM.SQL.CurrentSettings()**.

5.5.3 Listing Tables Referenced by a Cached Query

A cached query can reference multiple tables, views, and procedures.

- For tables referenced by a given cached query or stored procedure, use Management Portal **SQL Statements** tab to list SQL Statements. You can use the **Filter** option provided with this tab to filter by a **Location(s)** column value:
  - Cached Query: For example, the **Location(s)** column value `%sqlcq.USER.cls2.1` lists all the tables referenced by that cached query in the **Table/View/Procedure Name(s)** column.
  - Stored Procedure: For example, the **Location(s)** column value `Sample.procNamesJoinSP.1` lists all the tables referenced by that stored procedure in the **Table/View/Procedure Name(s)** column.
- For cached queries for a given table, use the Management Portal display of cached queries.

5.6 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 Management Portal SQL Interface: Follow the "Listing Cached Queries" instructions above. From the selected cached query's **Catalog Details** tab, click the **Execute** link.

5.7 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 system-wide CachedQueryLockTimeout option of the **$SYSTEM.SQL.Util.SetOption()** method. This option 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.

`SET status=$SYSTEM.SQL.Util.SetOption("CachedQueryLockTimeout",seconds,.oldval)` method sets the timeout value system-wide:

ObjectScript

```
SetCQTimeout
   SET status=$SYSTEM.SQL.Util.SetOption("CachedQueryLockTimeout",150,.oldval)
   WRITE oldval," initial value cached query seconds",!!
SetCQTimeoutAgain
   SET status=$SYSTEM.SQL.Util.SetOption("CachedQueryLockTimeout",180,.oldval2)
   WRITE oldval2," prior value cached query seconds",!!
ResetCQTimeoutToDefault
   SET status=$SYSTEM.SQL.Util.SetOption("CachedQueryLockTimeout",oldval,.oldval3)
```

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

5.8 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 explicitly purge cached queries via the Management Portal using one of the Purge Cached Queries options. You can explicitly purge cached queries using the SQL command PURGE CACHED QUERIES. You can explicitly purge cached queries using the SQL Shell PURGE command.

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, 2018, it will purge only the cached queries that were last prepared before April 11, 2018. 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:

- **$SYSTEM.SQL.PurgeCQClass()** purges one or more cached queries by name in the current namespace. You can specify cached query names as a comma-separated list. Cached query names are case sensitive; the namespace name must be specified in all-capital letters. The specified cached query name or list of cached query names must be enclosed with quotation marks.
- **$SYSTEM.SQL.PurgeForTable()** purges all cached queries in the current namespace that reference the specified table. The schema and table name are not case-sensitive.
- **$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.

  To purge all cached queries in the current namespace, use the Management Portal Purge ALL queries for this namespace option.

Purging a cached query also purges related query performance statistics.

Purging a cached query also purges related SQL Statement list entries. SQL Statements listed in the Management Portal may not be immediately purged, you may have to press the **Clean stale** button to purge these entries from the SQL Statements list.

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

5.8.1 Remote Systems

Purging a cached query on a local system does not purge copies of that cached query on 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. InterSystems IRIS 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.

5.9 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.

6

Specify Optimization Hints in Queries

By default, the InterSystems SQL query optimizer uses sophisticated and flexible algorithms to optimize the performance of complex queries involving multiple indexes. In nearly all cases, these defaults provide optimal performance. However, InterSystems SQL provides hints that can be used to manually modify the execution plan. You will most typically use these hints at the instruction of the InterSystems Worldwide Resource Center (WRC) to configure an optimal query execution plan. To get help from the WRC with SQL performance, refer to Generate Report.

There are two ways to provide such hints to the query optimizer to force it to employ certain optimizations or avoid others. The first is employing keywords in a **FROM** clause of a **SELECT** statement; the second is specifying comment options.

6.1 FROM Clause Keywords

SELECT statements can make use of the **FROM** clause, to which you may supply a keyword that will specify certain query optimization behaviors. Multiple keywords may be provided in any order, separated by blank spaces.

6.1.1 %ALLINDEX

This optional keyword specifies that all indexes that provide any benefit are used for the first table in the query join order. This keyword should only be used when there are multiple defined indexes. The optimizer default is to use only those indexes that the optimizer judges to be most beneficial. By default, this includes all efficient equality indexes, and selected indexes of other types. %ALLINDEX uses all possibly beneficial indexes of all types. Testing all indexes has a larger overhead, but under some circumstances it may provide better performance than the default optimization. This option is especially helpful when using multiple range condition indexes and inefficient equality condition indexes. In these circumstances, accurate index selectivity may not be available to the query optimizer. %ALLINDEX can be used with %IGNOREINDEX to include/exclude specific indexes. Generally, %ALLINDEX should not be used with a TOP clause query.

You can use %STARTTABLE with %ALLINDEX to specify which table the %ALLINDEX applies to.

You can specify exceptions to %ALLINDEX for specific conditions with the %NOINDEX condition-level hint. The %NOINDEX hint is placed in front of each query selection condition for which no index should be used. For example, `WHERE %NOINDEX hiredate < ?`. This is most commonly used when the overwhelming majority of the data is not excluded by the condition. With a less-than (<) or greater-than (>) condition, use of the %NOINDEX condition-level hint is often beneficial. With an equality condition, use of the %NOINDEX condition-level hint provides no benefit. With a join condition, %NOINDEX is supported for ON clause joins. For further details, refer to Using Indexes in Query Processing.

6.1.2 %FIRSTTABLE

```
%FIRSTTABLE tablename
```

This optional keyword specifies that the query optimizer should start to performs joins with the specified *tablename*. The *tablename* names a table that is specified later in the join sequence. The join order for the remaining tables is left to the query optimizer. This hint is functionally identical to %STARTTABLE, but provides you with the flexibility to specify the join table sequence in any order.

The *tablename* must be a simple identifier, either a table alias or an unqualified table name. A qualified table name (schema.table) cannot be used. If the query specifies a table alias, the table alias must be used as *tablename*. For example:

```
FROM %FIRSTTABLE P Sample.Employee AS E JOIN Sample.Person AS P ON E.Name = P.Name
```

%FIRSTTABLE and %STARTTABLE both enable you to specify the initial table to use for join operations. %INORDER enables you to specify the order of all tables used for join operations. These three keywords are mutually exclusive; specify one and one only. If these keywords are not used the query optimizer performs joins on tables in the sequence it considers optimal, regardless of the sequence in which the tables are listed.

You cannot use %FIRSTTABLE or %STARTTABLE to begin the join order with the right-hand side of a LEFT OUTER JOIN (or the left-hand side of a RIGHT OUTER JOIN). Attempting to do so results in an SQLCODE -34 error: “Optimizer failed to find a usable join order”.

For further details, refer to the %STARTTABLE query optimization option.

6.1.3 %FULL

This optional keyword specifies that the compiler optimizer examines all alternative join sequences to maximize access performance. For example, when creating a stored procedure, the increased compile time may be worthwhile to provide for more optimized access. The default optimization is to not examine less likely join sequences when there are many tables in the FROM clause. %FULL overrides this default behavior.

You might specify both the %INORDER and the %FULL keywords when the FROM clause includes tables accessed with arrow syntax, which lead to tables whose order is unconstrained.

6.1.4 %IGNOREINDEX

This optional keyword specifies that the query optimizer ignore the specified index or list of indexes. (The deprecated synonym %IGNOREINDICES is supported for backwards compatibility.)

Following this keyword, you specify one or more index names. Multiple index names must be separated by commas. Indexes that are ignored can be specified in a few ways:

- %IGNOREINDEX * — Ignores all indexes used by the query on tables specified in the FROM clause.
- %IGNOREINDEX *SchemaName*.* — Ignores all indexes in the SchemaName.
- %IGNOREINDEX *SchemaName.TableName*.* — Ignores all indexes in the particular table.
- %IGNOREINDEX *SchemaName.TableName.IndexName* — Ignores a particular index in the table.

The following example query ignores all indexes that would otherwise be used in the query:

SQL

```
SELECT ID FROM %IGNOREINDEX * Opus.Fact WHERE Date > '2023-01-22'
```

By using this optimization constraint, you can cause the query optimizer to not use an index that is not optimal for a specific query. By specifying all index names but one, you can, in effect, force the query optimizer to use the remaining index.

You can also ignore a specific index for a specific condition expression by prefacing the condition with the %NOINDEX keyword. For further details, refer to Using Indexes in Query Processing.

6.1.5 %INORDER

This optional keyword specifies that the query optimizer performs joins in the order that the tables are listed in the FROM clause. This minimizes compile time. The join order of tables referenced with arrow syntax is unrestricted (for information on using arrow syntax, refer to Implicit Joins). Flattening of subqueries and index usage are unaffected.

%INORDER cannot be used with a CROSS JOIN or a RIGHT OUTER JOIN. If the table order specified is inconsistent with the requirements of an outer join, an SQLCODE -34 error is generated: “Optimizer failed to find a usable join order.” To avoid this, it is recommended that %INORDER, when used with outer joins, only be used with ANSI-style left outer joins or full outer joins.

%INORDER cannot be used when querying a sharded table. See Querying the Sharded Cluster.

Views and table subqueries are processed in the order that they are specified in the FROM clause.

- Streamed View: %INORDER has no effect on the order of processing of tables within the view.
- Merged View: %INORDER causes the view tables to be processed in the view’s FROM clause order, at the point of reference to the view.

Compare this keyword with %FIRSTTABLE and %STARTTABLE, both of which specify only the initial join table, rather than the full join order. See %STARTTABLE for a table of merge behaviors with different join order optimizations.

The %INORDER and %PARALLEL optimizations cannot be used together; if both are specified, %PARALLEL is ignored.

6.1.6 %NOFLATTEN

This optional keyword is specified in the FROM clause of a quantified subquery — a subquery that returns a boolean value. It specifies that the compiler optimizer should inhibit subquery flattening. This optimization option disables “flattening” (the default), which optimizes a query containing a quantified subquery by effectively integrating the subquery into the query: adding the tables of the subquery to the FROM clause of the query and converting conditions in the subquery to joins or restrictions in the query's WHERE clause.

The following are examples of quantified subqueries using %NOFLATTEN:

SQL

```
SELECT Name,Home_Zip FROM Sample.Person WHERE Home_Zip IN
      (SELECT Office_Zip FROM %NOFLATTEN Sample.Employee)
```

SQL

```
SELECT Name,(SELECT Name FROM Sample.Company WHERE EXISTS
             (SELECT * FROM %NOFLATTEN Sample.Company WHERE Revenue > 500000000))
 FROM Sample.Person
```

The %INORDER and %STARTTABLE optimizations implicitly specify %NOFLATTEN.

6.1.7 %NOMERGE

This optional keyword is specified in the FROM clause of a subquery. It specifies that the compiler optimizer should inhibit the conversion of a subquery to a view. This optimization option disables the optimizing of a query containing a subquery

by adding the subquery to the FROM clause of the query as an in-line view; comparisons from the subquery to fields of the query are moved to the query's WHERE clause as joins.

6.1.8 %NOREDUCE

This optional keyword is specified in the FROM clause of a streamed subquery — a subquery that returns a result set of rows, a subquery in the enclosing query's FROM clause. It specifies that the compiler optimizer should inhibit the merging of the subquery (or view) into the containing query.

In the following example, the query optimizer would normally "reduce" this query by performing a Cartesian product join of Sample.Person with the subquery. The %NOREDUCE optimization option prevents this. InterSystems IRIS instead builds a temporary index on gname and performs the join on this temporary index:

SQL

```
SELECT * FROM Sample.Person AS p,
   (SELECT Name||'goo' AS gname FROM %NOREDUCE Sample.Employee) AS e
    WHERE p.name||'goo' = e.gname
```

6.1.9 %NOSVSO

This optional keyword is specified in the FROM clause of a quantified subquery — a subquery that returns a boolean value. It specifies that the compiler optimizer should inhibit Set-Valued Subquery Optimization (SVSO).

In most cases, Set-Valued Subquery Optimization improves the performance of [NOT] EXISTS and [NOT] IN subqueries, especially with subqueries with only one, separable correlating condition. It does this by populating a temporary index with the data values that fulfill the condition. Rather than repeatedly executing the subquery, InterSystems IRIS looks up these values in the temporary index. For example, SVSO optimizes `NOT EXISTS (SELECT P.num FROM Products P WHERE S.num=P.num AND P.color='Pink')` by creating a temporary index for `P.num`.

SVSO optimizes subqueries where the ALL or ANY keyword is used with a relative operator (>, >=, <, or <=) and a subquery, such as `...WHERE S.num > ALL (SELECT P.num ...)`. It does this by replacing the subquery expression *sqbExpr* (`P.num` in this example) with MIN(*sqbExpr*) or MAX(*sqbExpr*), as appropriate. This supports fast computation when there is an index on *sqbExpr*.

The %INORDER and %STARTTABLE optimizations do not inhibit Set-Valued Subquery Optimization.

6.1.10 %NOTOPOPT

This optional keyword is specified when using a TOP clause with an ORDER BY clause. By default, TOP with ORDER BY optimizes for fastest time-to-first-row. Specifying %NOTOPOPT (no TOP optimization) instead optimizes the query for fastest retrieval of the complete result set.

6.1.11 %NOUNIONOROPT

This optional keyword is specified in the FROM clause of a query or subquery. It disables the automatic optimizations provided for multiple OR conditions and for subqueries against a UNION query expression. These automatic optimizations transform multiple OR conditions to UNION subqueries, or UNION subqueries to OR conditions, where deemed appropriate. These UNION/OR transformations allow EXISTS and other low-level predicates to migrate to top-level conditions where they are available to InterSystems IRIS query optimizer indexing. These default transformations are desirable in most situations.

However, in some situations these UNION/OR transformations impose a significant overhead burden. %NOUNIONOROPT disables these automatic UNION/OR transformations for all conditions in the WHERE clause associated with this FROM

clause. Thus, in a complex query, you can disable these automatic UNION/OR optimizations for one subquery while allowing them in other subqueries.

The UNION %PARALLEL keyword disables automatic UNION-to-OR optimizations.

The %INORDER and %STARTTABLE optimizations inhibit OR-to-UNION optimizations. The %INORDER and %STARTTABLE optimizations do not inhibit UNION-to-OR optimizations.

6.1.12 %PARALLEL

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

A query that specifies both individual fields and an aggregate function and does not include a GROUP BY clause cannot perform parallel processing. For example, `SELECT Name,AVG(Age) FROM %PARALLEL Sample.Person` does not perform parallel processing, but `SELECT Name,AVG(Age) FROM %PARALLEL Sample.Person GROUP BY Home_State` does perform parallel processing.

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

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

Note: A query that specifies %PARALLEL must be run in a database that is read/write, not readonly. Otherwise, a <PROTECT> error may occur.

Regardless of the presence of the %PARALLEL keyword in the FROM clause, some queries may use linear processing, not parallel processing. In particular, on columnar tables, queries that use DISTINCT, ORDER BY, TOP, or UNION will not perform parallel processing. In other cases queries may be found to not benefit from parallel processing, when optimized. You can determine if and how InterSystems IRIS has partitioned a query for parallel processing using Show Plan. To determine the number of processors on the current system use the **%SYSTEM.Util.NumberOfCPUs()** method.

For further details, refer to Parallel Query Processing.

6.1.13 %STARTTABLE

This optional keyword specifies that the query optimizer should start to performs joins with the first table listed in the FROM clause. The join order for the remaining tables is left to the query optimizer. Compare this keyword with %INORDER, which specifies the complete join order.

%STARTTABLE cannot be used with a CROSS JOIN or a RIGHT OUTER JOIN. You cannot use %STARTTABLE (or %FIRSTTABLE) to begin the join order with the right-hand side of a LEFT OUTER JOIN (or the left-hand side of a RIGHT OUTER JOIN). If the start table specified is inconsistent with the requirements of an outer join, an SQLCODE -34 error is generated: “Optimizer failed to find a usable join order.” To avoid this, it is recommended that %STARTTABLE, when used with outer joins, only be used with ANSI-style left outer joins or full outer joins.

The following table shows the merge behavior when combining a superquery parent and an in-line view with %INORDER and %STARTTABLE optimizations:

	Superquery with no join optimizer	*Superquery with %STARTTABLE*	*Superquery with %INORDER*
View with no join optimizer	merge view if possible	If the view is the super-query start: don't merge. Otherwise, merge view if possible.	merge if possible; view's underlying tables are unordered.
View with %STARTTABLE	don't merge	If the view is the super-query start: merge, if possible. View's start table becomes super-query's start table. Otherwise, don't merge.	don't merge
View with %INORDER	don't merge	don't merge	If the view is not controlled by the %INORDER: don't merge. Otherwise, merge view if possible; view's order becomes substituted into superquery join order.

The %FIRSTTABLE hint is functionally identical to %STARTTABLE, but provides you with the flexibility to specify the join table sequence in any order.

6.2 Comment Options

You can specify one or more comment options to the Query Optimizer within a **SELECT**, **INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE TABLE** command. A comment option specifies a option that the query optimizer uses during the compile of the SQL query. Often a comment option is used to override a system-wide configuration default for a specific query.

6.2.1 Syntax

The syntax `/*#OPTIONS */`, with no space between the /* and the #, specifies a comment option. A comment option is not a comment; it specifies a value to the query optimizer. A comment option is specified using JSON syntax, commonly a key:value pair such as the following: `/*#OPTIONS {"optionName":value} */`. More complex JSON syntax, such as nested values, is supported.

A comment option is not a comment; it may not contain any text other than JSON syntax. Including non-JSON text within the /* ... */ delimiters results in an SQLCODE -153 error. InterSystems SQL does not validate the contents of the JSON string.

The #OPTIONS keyword must be specified in uppercase letters. No spaces should be used within the curly brace JSON syntax. If the SQL code is enclosed with quote marks, such as a Dynamic SQL statement, quote marks in the JSON syntax should be doubled. For example: `myquery="SELECT Name FROM Sample.MyTest /*#OPTIONS {""optName"":""optValue""} */"`.

You can specify a `/*#OPTIONS */` comment option anywhere in SQL code where a comment can be specified. In displayed statement text, the comment options are always shown as comments at the end of the statement text.

You can specify `/*#OPTIONS */` comment options in SQL code. They are shown in returned Statement Text in the order specified. If multiple comment options are specified for the same option, the last-specified option value is used.

The following comment options are documented:

- **/*#OPTIONS {"NoTempFile":1} */**: By default, a module performs processing and populates an internal temp-file (internal temporary table) with its results. You can force the query optimizer to create a query plan that does not generate internal temp-files by specifying 1 in the **NoTempFile** comment option.

6.2.2 Cosharding Comment Option

If an SQL query specifies multiple sharded tables, the SQL preprocessor generates a Cosharding comment option, which it appends to the end of the cached query text. This Cosharding option shows whether or not the specified tables are cosharded.

Note: The Cosharding option is applied automatically. Users should not manually supply this option.

In the following example, all three specified tables are cosharded:

```
/*#OPTIONS {"Cosharding":[["T1","T2","T3"]]} */
```

In the following example, none of the three specified tables are cosharded:

```
/*#OPTIONS {"Cosharding":[["T1"],["T2"],["T3"]]} */
```

In the following example, table T1 is not cosharded, but tables T2 and T3 are cosharded:

```
/*#OPTIONS {"Cosharding":[["T1"],["T2","T3"]]} */
```

6.2.3 DynamicSQLTypeList Comment Option

If an SQL query contains literal values, the SQL preprocessor generates a DynamicSQLTypeList comment option, which it appends to the end of the cached query text. This comment option assigns a data type to each literal. Data types are listed in the order that the literals appear in the query. Only actual literals are listed, not input host variables or ? input parameters. The following is a typical example:

SQL

```
SELECT TOP 2 Name,Age FROM Sample.MyTest WHERE Name %STARTSWITH 'B' AND Age > 21.5
```

generates the cached query text:

```
SELECT TOP ? Name , Age FROM Sample . MyTest WHERE Name %STARTSWITH ? AND Age > ? /*#OPTIONS
{"DynamicSQLTypeList":"10,1,11"} */
```

In this example, the literal 2 is listed as type 10 (integer), the literal “B” is listed as type 1 (string), and the literal 21.5 is listed as type 11 (numeric).

Note that the data type assignment is based solely on the literal value itself, not the data type of the associated field. For instance, in the above example Age is defined as data type integer, but the literal value 21.5 is listed as numeric. Because InterSystems IRIS converts numbers to canonical form, a literal value of 21.0 would be listed as integer, not numeric.

DynamicSQLTypeList returns the following data type values:

Data Type Value	Meaning
1	String of length 1 to 32 (inclusive)
2	String of length 33 to 128 (inclusive)
3	String of length 129 to 512 (inclusive)
4	String of length > 512
10	Integer
11	Numeric

Because the DynamicSQLTypeList comment option is part of the query text, changing a literal so that it results in a different data type results in creating a separate cached query. For example, increasing or decreasing the length of a literal string so that it falls into a different range.

6.2.4 Display

The `/*#OPTIONS */` comment options display at the end of the SQL statement text, regardless of where they were specified in the SQL command. Some displayed `/*#OPTIONS */` comment options are not specified in the SQL command, but are generated by the compiler pre-processor.

The `/*#OPTIONS */` comment options display in the Show Plan **Statement Text**, in the Cached Query **Query Text**, and in the SQL Statement **Statement Text**.

A separate cached query is created for queries that differ only in the `/*#OPTIONS */` comment options.