How InterSystems IRIS Processes SQL Statements

Version 2024.1
2024-07-02

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

How InterSystems IRIS Processes SQL Statements
InterSystems IRIS Data Platform Version 2024.1 2024-07-02

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

How InterSystems IRIS Processes SQL Statements

With every SQL statement you submit, InterSystems IRIS® performs several processing steps to make the statement run as quickly and efficiently as possible. These steps enable you to write statements without worrying about how to structure them to maximize performance. This topic describes how InterSystems processes SQL statements, from preparation to execution. It specifically focuses on query statements and statements that are part of the Data Manipulation Language (DML), including:

- SELECT statements that query the data and retrieve results
- INSERT, UPDATE, and DELETE statements that modify the data

Data Definition Language (DDL) statements that modify or define table schema, such as CREATE TABLE or ALTER TABLE, are not covered.

All the steps described below are executed by InterSystems IRIS automatically and are transparent to user and any application code.

1 Preparse Statement

When you first submit a statement, InterSystems IRIS runs it through a preparser to normalize it. These normalization steps include:

- Remove irrelevant whitespace. For example, the query

  SQL

  ```
  SELECT    Name     FROM     Customer
  ```

  becomes

  SQL

  ```
  SELECT Name FROM Customer
  ```

- Uppercase keywords that are part of the SQL standard. For example, the query

  SQL

  ```
  Select Name From Customer
  ```

  becomes

  SQL

  ```
  SELECT Name FROM Customer
  ```

- Perform *literal substitution*, where literal values passed into the statement as input parameters are replaced with question marks and stored separately. For example, the query

SQL

```
SELECT Name FROM Customer WHERE Zip = ‘00001’
```

becomes

SQL

```
SELECT Name FROM Customer WHERE Zip = ?
```

This normalized version of the query is then passed to the SQL server, and the SQL Engine generates a unique hash value based on the normalized form of the statement and environment variables such as the SQL dialect used. Using this hash value, the SQL Engine can look up whether this statement is already stored in the Universal Query Cache (UQC).

If a statement is in the cache, then the SQL Engine uses the hash to look up the ObjectScript code that executes the normalized query. It then replaces the question marks with the literal values that were stored during literal substitution and executes the query. The remaining statement preparation steps are skipped. This enables faster re-execution of statements and faster execution of statements that differ only in the parameters passed to them.

For example, suppose you execute a query on the Customer table for the first time:

SQL

```
SELECT Name FROM Customer WHERE Zip = '00001'
```

The SQL Engine stores a normalized form of this query in the cache:

SQL

```
SELECT Name FROM Customer WHERE Zip = ?
```

If you run this query, which has the same normalized form as the previous query, then the SQL Engine executes the stored query, changing only the input parameter passed in.

SQL

```
SELECT Name FROM Customer WHERE Zip = '00002'
```

If a statement is not in the cache, the SQL Engine stores the normalized query in the cache, passes the statement to the SQL compiler, and statement preparation continues.

When using a JDBC or ODBC client, the preparsing happens on the client side, off-loading some work from the server. In some specific cases, this can leverage a client-side cache, further reducing the load on the server.

2 Compile Statement

Upon receiving a normalized statement from the preparser, the SQL compiler does a full parsing of the statement. These parsing steps include:

- Check that the statement is syntactically correct and adheres to the SQL standard.
- Check what tables the statement touches. For those tables, it then:
  - Retrieves metadata for those tables from the data dictionary that is stored on the SQL server. The metadata contains information, such as the number of rows in the table, that informs the plan used to generate the statement, as described in the Generate Query Plan section.

  - Checks that the information requested in the statements matches the data in the table. For example, if the statements requests data from a column that does not exist in the table, the compiler issues an error.

The SQL compiler than passes the statement to the SQL optimizer, which generates an optimized plan for executing the statement.

The SQL compiler may also add additional option flags to the normalized statement for use by the SQL optimizer. These are typically enclosed in `/*#OPTIONS */` tags and may be ignored by the user.

3 Generate Query Plan

A *query plan* is a formal strategy that the SQL optimizer uses to perform the statement operation. The optimizer generates a range of different plans, estimates the execution cost of each one, and chooses the plan that has the lowest cost.

To determine query costs, InterSystems SQL relies on table statistics. InterSystems SQL collects statistics by running the TUNE TABLE utility. On large data sets, TUNE TABLE uses sampling and does not examine every row. The sampling for tables with standard storage layout is based on low-level scanning of database blocks and can yield accurate statistics in seconds, even for tables with gigabytes of data. For tables where block scanning is available, InterSystems SQL will automatically collect statistics when the table is queried for the first time and no earlier statistics were found.

For example, suppose you want to return all customers that live in a certain zip code. One possible query plan is to do a full table scan: Go through all rows and keep the ones where the zip code column value is the value you specified. The cost for this plan is easy to estimate. Multiply the total number of rows by the time that it takes to retrieve a single row from storage. The total number of rows is part of the table statistics and is part of the data retrieved from the metadata.

If you have an index on the `ZipCode` column, another query plan might take advantage of this index and generate a plan with a lower cost. Selectivity is a major determinant of which query plan is chosen. *Selectivity* is the percentage of the total number of table rows that are matched for a single supplied value. For example, suppose a table with 1,000,000 rows has 1% selectivity on the `ZipCode` column. If the query plan uses an index to retrieve all the rows with a given value, then on average that index returns 1,000,000 * 1% = 10,000 rows. In the `ZipCode` index example, the query plan that uses the index has this cost: selectivity * total number of rows * (cost to read a single index entry + cost to read a single row).

If a table has indexes on two different columns that can be used to satisfy filter criteria, then the one with the lower selectivity is chosen first, as it will more quickly narrow down the matching rows and therefore yields the plan with the lower cost.

In some cases, the optimal query plan depends on what parameters are passed in at runtime. For example, consider a Customer table for a nationwide United States retailer. Suppose you want to select all customers in Wyoming using this query:

SQL

```
SELECT Name FROM Customer WHERE State = 'WY'
```

The `State` column has a default selectivity of 2%, because selecting one of the 50 states results in selecting 2% of the data. If there is an index on the `State` column, then with such low selectivity, a query plan that uses this index might be worthwhile.

Consider a separate `Customer` table for a local store in Wyoming. Suppose you run the same query as before:

SQL

```
SELECT Name FROM Customer WHERE State = 'WY'
```

For this particular dataset, where data is not evenly spread across column values, the query plan for the index might not be worth the cost because the selectivity is very high: 90% of more of the rows might be for Wyoming. Now suppose the query was for another state instead:

SQL

```
SELECT Name FROM Customer WHERE State = 'CO'
```

The selectivity for states other than Wyoming is much lower. For some states, this value could be lower than 0.1%. InterSystems SQL uses Runtime Plan Choice (RTPC) to choose an appropriate query plan based on the parameters passed in. RTPC is a system-wide setting that is on by default. Here is one way that RTPC might choose a query plan based on a parameter:

- If a parameter value has high selectivity (that is, it appears many times in a particular column), do not use an index.
- If a parameter value has low selectivity, use the index.

The optimizer might even create its own indexes on the fly and generate a plan to use those, further optimizing queries.

The best query plan, containing everything needed for subsequent code generation, is then stored in a registry of all statements called the Statement Index. The Statement Index includes:

- The selected query plan
- The locations of cached queries
- The generated execution code

The Statement Index also includes the runtime statistics of a statement, such as how many times it was run, the average time it takes to run it, and so on. It is recorded daily and hourly, so you can look at a historical view of data to see how a query performs over time. The Management Portal displays all the metadata and runtime statistics for a query.

To lock in use of a specific plan for a given query, you can designate a frozen plan. If a frozen plan exists for a cached query, after preparsing of the statement, query plan generation is skipped and InterSystems SQL uses the frozen plan. You can check whether a plan is frozen or not from the Statement Index.

4 Generate Statement Execution Code

With the query plan generated, InterSystems IRIS, looks at the plan and the storage definition of the table (physical locations of the table data and index data) and generates ObjectScript code to execute the statement. The code is a `Query` class that has:

- A means to instantiate it, using the parameter values that were stored during the preparsing step
- A `Next` method, which is used to retrieve row after row of data, like in a result set returned by a query.

Once the query code has been generated, InterSystems SQL executes the statement and returns the results.