InterSystems™
IRIS Data Platform

First Look: Optimizing SQL Performance with InterSystems Products

Version 2019.4
2020-01-28

First Look: Optimizing SQL Performance with InterSystems Products
InterSystems IRIS Data Platform   Version 2019.4   2020-01-28
Copyright © 2020 InterSystems Corporation
All rights reserved.

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

First Look: Optimizing SQL Performance with InterSystems Products

This First Look guide introduces you to InterSystems SQL query optimization, including the use of query analysis tools, several indexing methods, and the ability to review runtime statistics over time.

To browse all of the First Looks, including others that can be performed on a free cloud instance or web instance, see InterSystems First Looks.

1 Query Optimization with InterSystems SQL

InterSystems IRIS® data platform offers a full suite of tools for SQL query performance tuning:

- Graphical displays for query plan analysis

- Indexing strategies such as bitmap and bitslice indexing that are compact and can be processed efficiently by vectorized CPU instructions. Each type of index offers benefits for certain query types, such as logical conditions, counting, and aggregate functions. With indexing, you can achieve query performance results of up to billions of rows per second on one core.

- Metrics on SQL query performance over time

Important: The query performance numbers shown in the demos below are representative of multiple trials of the demos on a single Windows 10 laptop. You may see different query performance numbers depending on your environment.

Want a quick demo of the SQL capabilities of InterSystems IRIS? Check out the SQL QuickStart!

2 Demo: Showing and Interpreting a Query Plan Before Optimization

2.1 Before you Begin

This First Look is best experienced after reading and working through *First Look: InterSystems SQL*. Here you will use the InterSystems IRIS SQL Shell again; the data you will use comes from the million-record table of stock transaction data you created when you worked through the demos in that First Look.

You will also run the `TuneTable` utility, which examines the data in the table and creates statistics used by the *InterSystems SQL query optimizer* (the engine that decides how best to run any query). These statistics include the size of the table (extent size) and the number of unique values per column (selectivity). The optimizer uses table size in scenarios like determining join order, where it's best to start with the smaller table. Selectivity helps the optimizer choose the best index in the case where a table has multiple indices. In a production instance, you normally run `TuneTable` only once: after data is loaded into a table and before you go live.

First Look: InterSystems SQL explains how to take the following steps required to run the demo in that First Look and the one here:

- Select an InterSystems IRIS instance. Your choices include several types of licensed and free evaluation instances; for information on how to deploy each type, see Deploying InterSystems IRIS in *InterSystems IRIS Basics: Connecting an IDE*.

- Open the InterSystems Terminal (Terminal for short) to run the SQL Shell.

- Obtain utility files for this guide from the GitHub repo https://github.com/intersystems/FirstLook-SQLBasics, including

  – stock_table_demo_two.csv, which contains a million rows of stock table data

  – Loader.xml, a class file that contains a utility method to load the data from stock_table_demo_two.csv into an InterSystems IRIS table

2.2 Running the TuneTable Utility

If your InterSystems IRIS instance no longer includes the **StockTableDemoTwo** table, recreate and load it by following the first four steps in Demo: Using Bitmap Indexing To Maximize Query Performance (stop before executing the SELECT DISTINCT query).

In the SQL Shell, run the TuneTable utility on the **FirstLook.StockTableDemoTwo** as follows:

```
OBJ DO $SYSTEM.SQL.TuneTable("FirstLook.StockTableDemoTwo")
```

This command generates no visible output in the SQL Shell.

2.3 Using the EXPLAIN Keyword to Show a Query Plan

This demo assumes that you want to obtain the average price for all "SELL" transactions. Given that the table contains a million rows, the needed query could potentially be very slow.

While you may already want to proceed with creating indices on the Price and TransactionType fields, it will be instructive to see the query plan before you begin optimization work. In the SQL Shell, you can show a plan for a query by prepending the EXPLAIN keyword to it. The query plan shows how the SQL query optimizer will use indices, if any, or whether it will read the table data directly to execute the statements.

To use the EXPLAIN keyword to show a query plan, execute the following statement in the SQL Shell:

```
EXPLAIN SELECT AVG(Price) As AveragePrice FROM FirstLook.StockTableDemoTwo
    WHERE TransactionType = 'SELL'
```

This will return the query plan, formatted as XML:

```
Plan
"<plans>
 <plan>
 <sql>
  SELECT AVG ( Price ) AS AveragePrice FROM FirstLook . StockTableDemoTwo WHERE TransactionType = ?
  /*#OPTIONS {""DynamicSQLTypeList"":""1""} */
 </sql>
 <cost value=""1827000""/>
 Call module B.
 Output the row.
 <module name=""B"" top=""1"">
 Process query in parallel, partitioning master map FirstLook.StockTableDemoTwo.IDKEY into
    subranges of T1.ID values, piping results to temp-file A:
    SELECT count(T1.Price),sum(T1.Price) FROM  %NOPARALLEL FirstLook.StockTableDemoTwo T1
    where ((%SQLUPPER(T1.TransactionType) = %SQLUPPER(?)))
 Read temp-file A, looping on a counter.
 For each row:
    Accumulate the count([value]).
    Accumulate the sum([value]).
 </module>
```

```
</plan>
<plan>
<sql>
 SELECT COUNT ( T1 . Price ) , SUM ( T1 . Price ) FROM %NOPARALLEL FirstLook . StockTableDemoTwo T1
 WHERE ( ( %SQLUPPER ( T1 . TransactionType ) = %SQLUPPER ( ? ) ) ) %PARTITION BY T1 . ID > ? AND T1
 . ID <= ?
</sql>
<cost value=""1827000""/>
Call module B.
Output the row.
<module name=""B"" top=""1"">
Read master map FirstLook.StockTableDemoTwo.IDKEY, looping on ID (with a range condition).
For each row:
    Accumulate the count(Price).
    Accumulate the sum(Price).
</module>
</plan>
</plans>"
```

You'll see that a query plan generated to execute a SQL query can be divided into *modules*, each of which performs a distinct part of the execution plan, such as evaluating a subquery.

Actually, this query plan is divided into two separate plans. The top plan is for the initial query. It calls a module B, in which the "master map" is partitioned and a subquery is executed in parallel over each partition. A plan for the subquery follows the plan for the initial query.
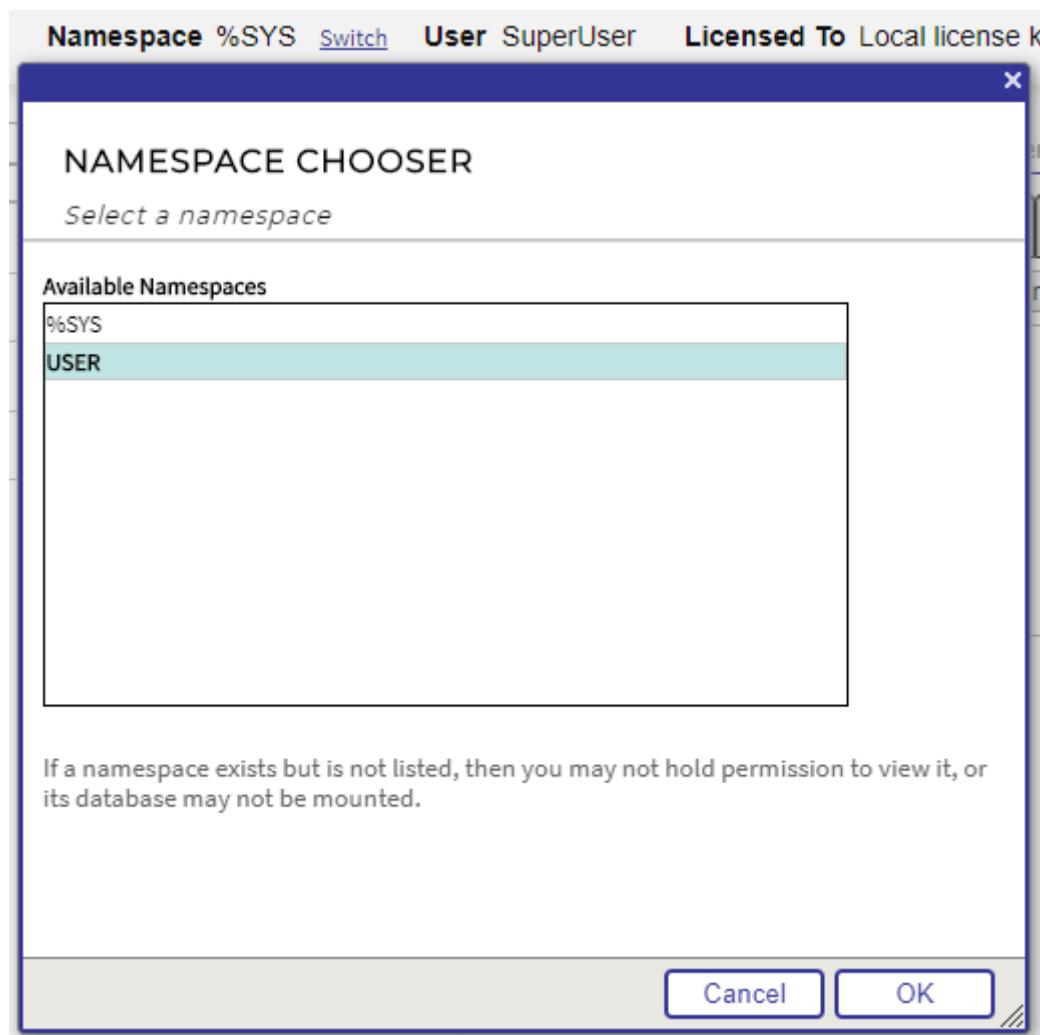
In "Spotting Potential Performance Issues in Query Plan Results", you'll learn to recognize the problems with this query.

2.4 Using the SQL Query Interface in the Management Portal to Show a Query Plan

InterSystems IRIS offers a web-based interface in the Management Portal for SQL query execution and plan analysis.

To show a query plan using the SQL query interface in the Management Portal:

1. Open the Management Portal for your instance in your browser, using the URL described for your instance in *InterSystems IRIS Basics: Connecting an IDE*.

2. Make sure you are in the **USER** namespace. If you are not already there:

    •   In the top panel of the screen, click **SWITCH** to the right of the name of the current namespace.

    •   In the popup, choose **USER** and click **OK**.

3. Navigate to SQL page (**System Explorer** > **SQL**).

4. Omitting the EXPLAIN keyword, paste the query from "Using the EXPLAIN Keyword to Show a Query Plan" into the text field in the **Execute Query** tab.

5. Click **Show Plan** to display a query plan. The results will look much like this:

Statement Text

```
SELECT AVG ( Price ) AS AveragePrice FROM FirstLook . StockTableDemoTwo WHERE TransactionType = ?
/*#OPTIONS {"DynamicSQLTypeList":"1"} */
```

Query Plan

Relative Cost = 1827000

- Call module B.
- Output the row.

Module: B

- Process query in parallel, partitioning master map FirstLook.StockTableDemoTwo.IDKEY into
  subranges of T1.ID values, piping results to temp-file A:
  - SELECT count(T1.Price),sum(T1.Price) FROM %NOPARALLEL FirstLook.StockTableDemoTwo T1 where
((%SQLUPPER(T1.TransactionType) = %SQLUPPER(?)))
- Read temp-file A, looping on a counter.
- For each row:
  - Accumulate the count([value]).
  - Accumulate the sum([value]).

Statement Text

```
SELECT COUNT ( T1 . Price ) , SUM ( T1 . Price ) FROM %NOPARALLEL FirstLook . StockTableDemoTwo T1
WHERE ( ( %SQLUPPER ( T1 . TransactionType ) = %SQLUPPER ( ? ) ) ) %PARTITION BY T1 . ID > ? AND T1 .
ID <= ?
```

Query Plan

Relative Cost = 1827000

- Call module B.
- Output the row.

Module: B

- Read master map FirstLook.StockTableDemoTwo.IDKEY, looping on ID (with a range condition).
- For each row:
  - Accumulate the count(Price).
  - Accumulate the sum(Price).

Interpreting these results is the subject of the next section.

2.5 Spotting Potential Performance Issues in Query Plan Results

Looking at the query plan results, you can see that there are some serious potential performance issues with this query. If you look at the plan for the subquery, which is where the actually work is done, you can see that the first task is "read master map." What this means is that the InterSystems SQL query optimizer will not use any indices to run the query;

instead, the query will loop over all of the IDs in the table. Especially in the case of a large table, this indicates a query that will not perform well.

As you optimize the query, you'll see its execution time decrease, and the query plan will change significantly as well.

Note: Relative cost can be a good predictor of performance, but relative only to a particular query. If you add an index to a table and see that the relative cost goes down, it's likely that query will now run much faster. However, relative cost is not intended to compare the performance of two different queries.

2.6 Testing Query Execution

To get some actual data as to how the unoptimized query will perform, run it in the SQL Shell:

```
SELECT AVG(Price) As AveragePrice FROM FirstLook.StockTableDemoTwo
   WHERE TransactionType = 'SELL'
   GO
```

The output will look something like this:

```
AveragePrice
266.1595139195757844

1 Rows(s) Affected
statement prepare time(s)/globals/cmds/disk: 0.0009s/6/1246/0ms
         execute time(s)/globals/cmds/disk: 0.2599s/1000075/8502571/0ms
                           cached query class: %sqlcq.USER.cls5
```

Statement preparation and execution metrics are listed separately. Take special notice of two items:

- Execution time was 0.2599 seconds. While this does not seem like a very long time, it can be vastly improved with the use of indices.

- The number of globals read in the execution step was 1,000,075. (Globals are multidimensional sparse arrays used by InterSystems IRIS to store data; for more information, see the "Introduction to Globals" chapter of *Introduction to InterSystems IRIS Programming*.) To improve query performance, this number should be decreased. You'll see that happen in the next section.

Important: Preparation is done only once: the first time a query is planned anew. Queries are automatically replanned if a relevant table is modified or if an index is added or removed. Most applications will prepare a query only once, but will execute it many times. So our focus in this demo will be on *tuning execution performance*.

3 Demo: Testing Query Optimizations

3.1 Adding a Bitslice Index to the Price Field

If your query will include aggregate functions on one or more fields, adding a bitslice index to one or more of those fields may improve performance.

A bitslice index represents each numeric data value in a field as a binary bit string, with a bitmap for each digit in the binary value to record which rows have a 1 for that binary digit.

Since we want to get the average price for all "SELL" transactions, it makes sense to add a bitslice index to the Price field. To create the bitslice index PriceIdx on the Price field, execute the following statement in the SQL Shell:

```
CREATE BITSLICE INDEX PriceIdx ON TABLE FirstLook.StockTableDemoTwo (Price)

9.       CREATE BITSLICE INDEX PriceIdx ON TABLE FirstLook.StockTableDemoTwo (Price)

0 Rows Affected
statement prepare time(s)/globals/cmds/disk: 0.0091s/2000/13151/0ms
         execute time(s)/globals/cmds/disk: 1.4268s/2087789/55765062/1ms
                        cached query class: %sqlcq.USER.cls7
```

Just because you've created the index does not necessarily mean that the InterSystems SQL query optimizer will use it, however, as you'll see below.

3.2 Testing the Effects of the Bitslice Index

To see if the new bitslice index makes any difference in how the query will be executed, or how fast it runs, use either method described above (the SQL Shell or the Management Portal) to show the query plan.

As you'll see, the query plan remains the same as before. The InterSystems SQL query optimizer will not use the new index.

Statement Text

```
SELECT AVG ( Price ) AS AveragePrice FROM FirstLook . StockTableDemoTwo WHERE TransactionType = ?
/*#OPTIONS {"DynamicSQLTypeList":"1"} */
```

Query Plan

Relative Cost = 1827000

- Call module B.
- Output the row.

Module: B

- Process query in parallel, partitioning master map FirstLook.StockTableDemoTwo.IDKEY into
   subranges of T1.ID values, piping results to temp-file A:
   - SELECT count(T1.Price),sum(T1.Price) FROM %NOPARALLEL FirstLook.StockTableDemoTwo T1 where
((%SQLUPPER(T1.TransactionType) = %SQLUPPER(?)))
- Read temp-file A, looping on a counter.
- For each row:
   - Accumulate the count([value]).
   - Accumulate the sum([value]).

Statement Text

```
SELECT COUNT ( T1 . Price ) , SUM ( T1 . Price ) FROM %NOPARALLEL FirstLook . StockTableDemoTwo T1
WHERE ( ( %SQLUPPER ( T1 . TransactionType ) = %SQLUPPER ( ? ) ) ) %PARTITION BY T1 . ID > ? AND T1 .
ID <= ?
```

Query Plan

Relative Cost = 1827000

- Call module B.
- Output the row.

Module: B

- Read master map FirstLook.StockTableDemoTwo.IDKEY, looping on ID (with a range condition).
- For each row:
   - Accumulate the count(Price).
   - Accumulate the sum(Price).

Running the query yields nearly the same performance statistics as it did before you created the bitslice index (0.2559 seconds of execution time compared with 0.2599). InterSystems IRIS intelligently caches query plans and data, so subsequent runs of the same query may result in improved performance, as may have been the case here given the slight difference in query performance times. Other applications running on the machine can affect performance as well.

```
        SELECT AVG(Price) As AveragePrice FROM FirstLook.StockTableDemoTwo
        WHERE TransactionType = 'SELL'
        GO

10.     SELECT AVG(Price) As AveragePrice FROM FirstLook.StockTableDemoTwo
        WHERE TransactionType = 'SELL'
```

```
AveragePrice
266.1595139195757844

1 Rows(s) Affected

statement prepare time(s)/globals/cmds/disk: 0.0569s/35431/227191/0ms
          execute time(s)/globals/cmds/disk: 0.2559s/1000075/8502571/0ms
                            cached query class: %sqlcq.USER.cls8
```

If you remove the WHERE clause from the query, you'll see quite a different result when you show the query plan:

Statement Text

SELECT AVG (Price) AS AveragePrice FROM FirstLook . StockTableDemoTwo

Query Plan

Relative Cost = 16362

- Call module D once.
- Call module G once.
- Output the row.

Module: D

- Read bitslice index FirstLook.StockTableDemoTwo.PriceIdx, looping on bitslice power.
- For each bitslice power:
  Call module E.
  - Accumulate the sum(Price).

Module: G

- Read extent bitmap FirstLook.StockTableDemoTwo.%%DDLBEIndex, looping on bitmap chunks.
- For each bitmap chunk:
  - Read a chunk from bitslice index FirstLook.StockTableDemoTwo.PriceIdx and perform a bitwise AND NOT operation.
  - Accumulate the count(bits).

Module: E

- Read bitslice index FirstLook.StockTableDemoTwo.PriceIdx, using the given bitslice power, and looping on bitslice chunks.
- For each bitslice chunk:
  - Accumulate the aggregate for the power.

As you can see, the bitslice index is read as the first step of the query plan. The "master map" is not read in this plan.

The SQL query optimizer also uses a second index, FirstLook.StockTableDemoTwo.$StockTableDemoTwo. This is a *bitmap extent index*, which is automatically created whenever the CREATE TABLE SQL statement is executed. It is a bitmap index of all the rows in the table, not just one field, and the value of each bit reflects whether or not the row actually exists.

However, the query that we truly want to run contains a WHERE clause. So we'll have to find a way to get the SQL query optimizer to use the index when the WHERE clause is present.

3.3 Adding a Bitmap Index To the TransactionType Field

If you read the InterSystems SQL Optimization Guide, you'll find that the InterSystems SQL query optimizer will often use a bitslice index when it is combined with a bitmap index on the field in a WHERE clause.

This is because aggregate queries without the WHERE clause can simply aggregate all the data in the index. However, to aggregate only the rows that satisfy a WHERE condition, a query must *mask* those bits out of the bitslice index for rows that do not satisfy the condition. A bitmap index on the field in the WHERE clause allows this mask to be constructed efficiently.

Fortunately, the other field in the query, TransactionType, is a good candidate for a bitmap index because its count of possible values is two ("SELL" and "BUY").

To add a bitmap index to the TransactionType field, execute the following statement in the SQL Shell:

```
CREATE BITMAP INDEX TransactionTypeIdx ON TABLE FirstLook.StockTableDemoTwo
        (TransactionType)

11.     CREATE BITMAP INDEX TransactionTypeIdx ON TABLE FirstLook.StockTableDemoTwo
        (TransactionType)

0 Rows Affected
statement prepare time(s)/globals/cmds/disk: 0.0069s/2001/13291/0ms
        execute time(s)/globals/cmds/disk: 1.1046s/2088960/19771584/0ms
                        cached query class: %sqlcq.USER.cls7
```

3.4 Retesting Query Performance

Now that you have added bitslice and bitmap indices: if you show the query plan for

```
SELECT AVG(Price) as AveragePrice FROM FirstLook.StockTableDemoTwo
    WHERE TransactionType = 'SELL'
```

in SQL Shell or in the Management Portal, you'll see that the query optimizer uses the two indices you created to obtain the best performance.

Note as well that the relative cost of 18742 is a small fraction of the unoptimized query, whose cost was 1827000.

Statement Text

```
SELECT AVG ( Price ) AS AveragePrice FROM FirstLook . StockTableDemoTwo WHERE TransactionType = ? /*#OPTIONS
{"DynamicSQLTypeList":"1"} */
```

Query Plan

Relative Cost = 18742

- Call module C once, which populates bitmap temp-file A.
- Call module E once.
- Call module H once.
- Output the row.

Module: C

- Generate a stream of bitmap chunks using the multi-index combination:
  ((bitmap index FirstLook.StockTableDemoTwo.TransactionTypeIdx) INTERSECT (extent bitmap FirstLook.StockTableDemoTwo.%%
- For each bitmap chunk:
  - OR the bitmap chunk into bitmap temp-file A.

Module: E

- Read bitslice index FirstLook.StockTableDemoTwo.PriceIdx, looping on bitslice power.
- For each bitslice power:
  Call module F.
  - Accumulate the sum(Price).

Module: H

- Read bitmap temp-file A, looping on bitmap chunks.
- For each bitmap chunk:
  - Read a chunk from bitslice index FirstLook.StockTableDemoTwo.PriceIdx and perform a bitwise AND NOT operation.
  - Accumulate the count(bits).

Module: F

- Read bitslice index FirstLook.StockTableDemoTwo.PriceIdx, using the given bitslice power, and looping on bitslice chunks.
- For each bitslice chunk:
  - Read a chunk from bitmap temp-file A and perform a bitwise AND operation.
  - Accumulate the aggregate for the power.

Finally, if you run the query in SQL Shell, you'll see a much more efficient use of globals (594 as opposed to 1000075).

Most critically, the indexed query ran nearly 85 times faster than the unindexed query (0.0031 seconds of execution time as opposed to 0.2599).

```
        SELECT AVG(Price) As AveragePrice FROM FirstLook.StockTableDemoTwo
        WHERE TransactionType = 'SELL'
        GO
12.    SELECT AVG(Price) As AveragePrice FROM FirstLook.StockTableDemoTwo WHERE TransactionType =
'SELL'

AveragePrice
266.1595139195757844

1 Rows(s) Affected
statement prepare time(s)/globals/cmds/disk: 0.0554s/34877/186130/0ms
        execute time(s)/globals/cmds/disk: 0.0031s/594/2878/0ms
                        cached query class: %sqlcq.USER.cls8
```

To track the performance of the query over time, InterSystems IRIS provides query statistics, which you'll learn how to view in the next section.

4 Viewing Query Performance Over Time

To track down slow-running queries or see how a new query is doing in production, you can use the **SQL Statements** view in the Management Portal. To navigate to this view, open the SQL query interface in the Management Portal and click **SQL Statements**.

If, for example, the query you tuned above ran nine times under its original (unoptimized) plan, you might see something like:

Catalog Details | Execute Query | Browse | **SQL Statements**

All SQL Statements in this namespace

Filter: FirstLook.StockTableDemoTwo   Page size: 100   Max rows: 1000   Results: 4   Page: |< « 1 » >| of 1   **Refresh**

#	Table/View/Procedure Name(s)	Plan State	New plan	Natural Query	Count	Average Count	Total time	Average time	Std Dev	Location(s)	SQL Statement Text
1	FirstLook.StockTableDemoTwo	Unfrozen		0						FirstLook.StockTableDemoTwo.1	DECLARE QEXTENT CUR
2	FirstLook.StockTableDemoTwo	Unfrozen/Parallel		0	36	36	8.2143	0.22818	0.028498	%sqlcq.USER.cls5.1	DECLARE P0AMBM2L2T (
3	FirstLook.StockTableDemoTwo	Unfrozen		0						FirstLook.Loader.1	INSERT %NOLOCK INTO
» 4	FirstLook.StockTableDemoTwo	Unfrozen/Parallel		0	9	9	2.3103	0.25670	0.013373	%sqlcq.USER.cls5.1	DECLARE QRS CURSOR

Clicking on the statement's link in the **SQL Statement Text column** allows you to view the query in SQL form:

Statement Text and Query Plan

Statement Text

SELECT AVG (PRICE) AS AVERAGEPRICE FROM FIRSTLOOK . STOCKTABLEDEMOTWO WHERE TRANSACTIONTYPE = ? /*#OPTIONS {"DynamicSQLTypeList":1} */

You can also tie SQL statement execution to the **SQL Statements** view using the name of the cached query class, which is the last line of output in the SQL Shell and is listed in the **Location(s)** column of **SQL Statements**.

After you optimize the query and run it a few times, you can expect to see improvements in the **Total time** and **Average time** columns.

Catalog Details | Execute Query | Browse | **SQL Statements**

All SQL Statements in this namespace

Filter: FirstLook.StockTableDemoTwo   Page size: 100   Max rows: 1000   Results: 4   Page: |< « 1 » >| of 1   **Refresh**

#	Table/View/Procedure Name(s)	Plan State	New plan	Natural Query	Count	Average Count	Total time	Average time	Std Dev	Location(s)	SQL Statement Text
1	FirstLook.StockTableDemoTwo	Unfrozen		0						FirstLook.StockTableDemoTwo.1	DECLARE QEXTENT CUI
2	FirstLook.StockTableDemoTwo	Unfrozen		0						FirstLook.StockTableDemoTwo.1	SELECT %ID INTO :id FR
3	FirstLook.StockTableDemoTwo	Unfrozen		0						FirstLook.Loader.1	INSERT %NOLOCK INTO
» 4	FirstLook.StockTableDemoTwo	Unfrozen		0	8	4	0.028930	0.0036163	0.0007167	%sqlcq.USER.cls8.1	DECLARE QRS CURSOR

Note that the value of **Count** has dropped. This is because the addition of the bitmap and bitslice indices caused the query plan to change, which in turn triggered a removal of cached queries for the associated class. The query has run under the new query plan a total of eight times, four times on average per day.

5 Learn More About InterSystems SQL

To learn more about SQL and InterSystems IRIS, see:

5.1 Introductory Material

- First Look: InterSystems SQL
- Using InterSystems SQL
- InterSystems SQL Reference
- InterSystems SQL Overview

5.2 SQL Development

- SQL – Things You Should Know
- Learn InterSystems SQL: Design and Execution
- Developing with InterSystems Objects and SQL

5.3 Query Optimization

- InterSystems SQL Optimization Guide
- Academy – Optimizing SQL Performance
- Optimizing SQL Queries
- Learn InterSystems SQL: Performance

5.4 Sharding and Scalability

- First Look: Scaling for Data Volume with Sharding
- Scalability Guide

5.5 SQL Search

- First Look: SQL Search with InterSystems IRIS
- Using InterSystems SQL Search
- Creating iFind Indices for Searching Text Fields

5.6 JDBC

- First Look: JDBC and InterSystems IRIS
- Using Java JDBC with InterSystems IRIS (documentation)
- Java Overview
- Using JDBC with InterSystems IRIS (online learning)