Article
Allyson Gerace · Feb 6, 2019 8m read

Index Handling

See Part 1 here.

Part 2: Index Handling

Now you have a good idea of what kind of indices you need for your class and how to define them. Next, how do you handle them?

Query Plan

(**REMEMBER**: Like any modifications to a class, adding indices in a live system has its risks – if users are accessing or updating data while an index is populated, they may encounter empty or incorrect query results, or even corrupt the indices that are being built. Please note that there are additional steps to defining and using indices on a live system – these steps will be touched on in this section and are detailed in our documentation.)

Once you have a new index in place, we can check to see if the SQL optimizer will decide it's the most efficient one to read through for executing the query. There's no need to actually run the query to check the plan. Given a query, you can check the plan programmatically:

```
      Set query = 1

     Set query(1) = "SELECT SSN,Name FROM Sample.Person WHERE Office_State =
'MA'"

      D $system.SQL.ShowPlan(.query)
```

Or by following the interface in the System Management Portal via System Explorer -> SQL.

From here, you can see which indices, if any, are being used before we load the table data (or "master map"). Consider: Is your new query in the plan as expected? Does the plan's logic make sense?

Once you know the SQL optimizer is using your indices, you can verify that these indices are functioning properly.

Building Indices

(If you are just in the planning stages and have no data at the moment, the steps detailed here won't be necessary right now.)

Defining an index will not automatically populate, or 'build', it with your table's data. If a query plan uses a new index that has not yet been built, you run the risk of incorrect or empty query results. You can 'deactivate' an index before it is ready to be used by setting its map selectability to 0 – essentially telling the SQL optimizer that it cannot use this index to run queries.

```
write $SYSTEM.SQL.SetMapSelectability("Sample.Person","QuickSearchIDX",0) ;
Set selectability of index QuickSearchIDX false
```

Note that you can use the above call even before you add a new index. The SQL optimizer will recognize this new index's name, know it is inactive, and not use it for any queries.

You can populate an index using the %BuildIndices method (`##class(<class>).%BuildIndices($lb("MyIDX"))`) or in the System Management Portal's SQL page (under the Actions drop-down).

The time taken to build indices depends on the amount of rows in the table and the types of indices you are building, with bitslice indices generally taking longer to build.

Once this process is complete, you can use the SetMapSelectivity method once more (this time to 1) to reactivate your newly populated index.

Note that building indices essentially means issuing KILL and SET commands to populate them. You may consider disabling journaling during this process to avoid filling up disk space.

You can find more detailed instructions on building new and existing indices, particularly on live systems, in our documentation linked below:

https://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=GSQLOPT_indices#GSQLOPT_indices_build_readwrite

Maintaining Indices

Now we can get to actually using our indices! Some points to consider are how efficiently a query will run, how often any given index is used, and what steps to take if your indices enter an inconsistent state.

Performance

First, we can consider how this index has affected query performance. If we know the SQL optimizer is using a new index for a query, we can run the query to gather its performance stats: number of global references, number of lines read, time to prepare and to execute the query, and time spent in disk.

Let's go back to a previous example:

“SELECT SSN,Name,DOB FROM Sample.Person WHERE Name %STARTSWITH 'Smith,J'”

We have the following index to help this query’s performance:

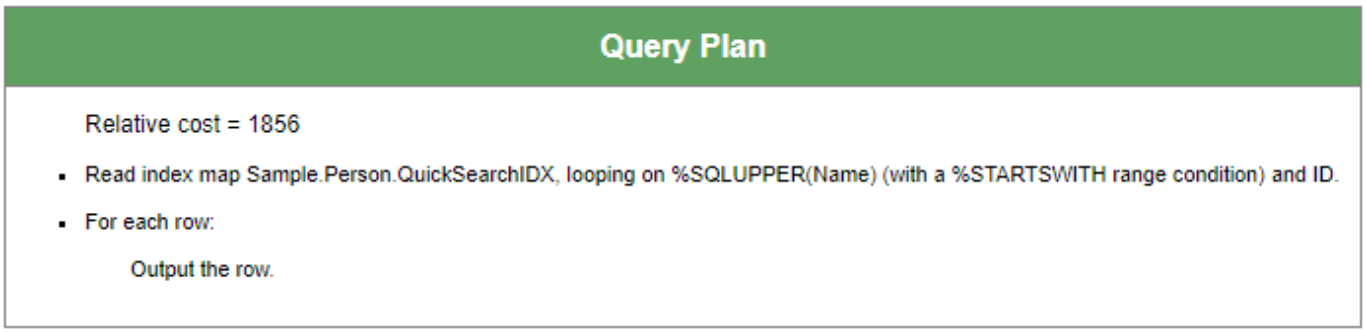
```
Index QuickSearchIDX On Name [ Data = (SSN, DOB, Name) ];
```

We already have an index NameIDX on the Name property.

Intuitively, we can tell that the query will be run using QuickSearchIDX; NameIDX would likely be a second choice because it is based on property Name but does not contain any of the data values for SSN or DOB.

Checking this query’s performance with QuickSearchIDX is as simple as running it.

(Aside: I have purged cached queries before executing these queries to better show performance differences. When a SQL query is run, we store the plan used to execute it, thus improving performance on its next execution – the finer details of that is outside the scope of this article , but I will include resources for other SQL performance consideration at the end of this article.)



Row count: **31** Performance: **0.003** seconds **154** global references **3264** lines executed **1** disk read latency (ms)

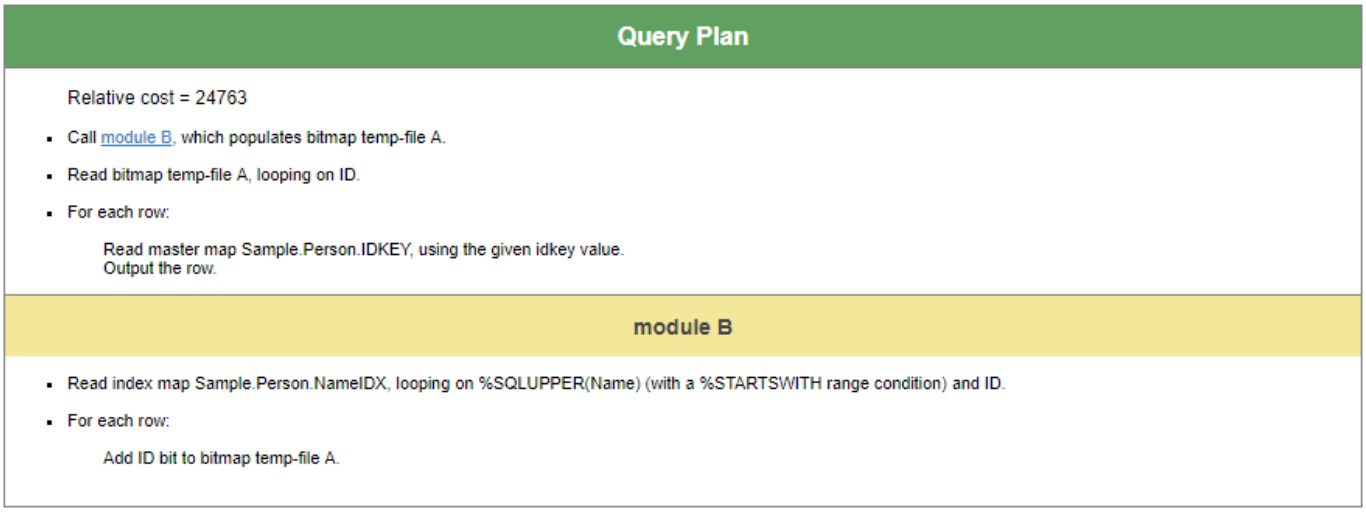
Straightforward: we only need QuickSearchIDX to execute the query.

Let’s compare the performance on using NameIDX rather than QuickSearchIDX – we can do this by adding a query keyword %IGNOREINDEX, which will prevent the SQL optimizer from choosing certain indices.

We rewrite the query as follows:

“SELECT SSN,Name,DOB FROM %IGNOREINDEX QuickSearchIDX Sample.Person WHERE Name %STARTSWITH 'Smith,J'”

The query plan now uses NameIDX, and we see that we must read from the data global (or “master map”) of Sample.Person using the relevant row IDs found via the index.



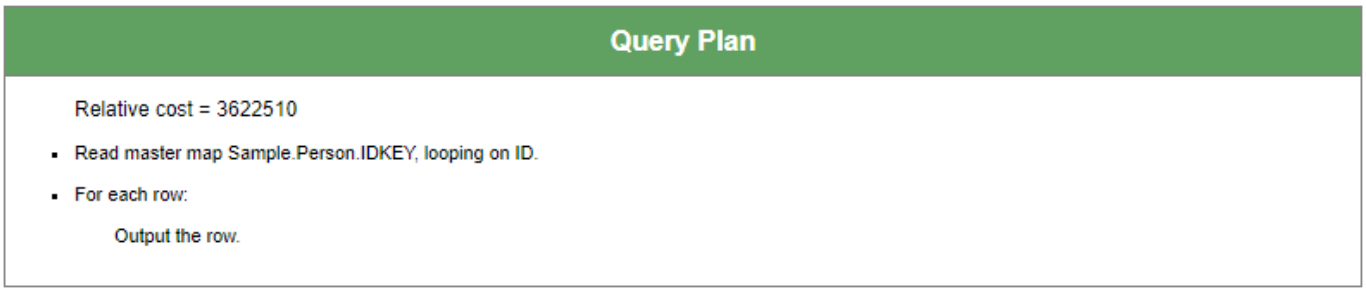
Row count: **31** Performance: **0.020** seconds **137** global references **3792** lines executed **17** disk read latency (ms)

We see greater amount of time needed to execute, more lines of code executed, and higher disk latency.

Next, consider running this query with no indices at all. We adjust our query as follows:

“`SELECT SSN,Name,DOB FROM %IGNOREINDEX * Sample.Person WHERE Name %STARTSWITH 'Smith,J'`”

Without the use of indices, we must to check rows of data for our condition.



Row count: **31** Performance: **0.765** seconds **149999** global references **1202681** lines executed **517** disk read latency (ms)

It shows that a specialized index, QuickSearchIDX, allows our query to run over 100x faster than with no index and nearly 10x faster than with the more general NameIDX – and using NameIDX runs at least 30x faster than using no index at all.

For this particular example, the performance difference between using QuickSearchIDX and NameIDX may be negligible, but for queries that are run hundreds of times in a day with millions of rows to query on, we’d see valuable time saved day after day.

Analyzing Existing Indices Using SQLUtilites

%SYS.PTools.SQLUtilities contains procedures, such as IndexUsage, JoinIndices, TablesScans, and TempIndices. These analyze existing queries in any given namespace and reports information on how often any given index is used, which queries are electing to iterate over each row of a table, and which queries are generating temporary files to simulate indices.

You can use these procedures to determine gaps that an index may be able to address and any indices that you may want to consider deleting due to lack of usage or inefficiency.

Details on these procedures and examples of their usage can be found in our class documentation below:

https://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=GSQLOPT_optquery_indexanalysis

Index problems?

Validating indices confirms whether each index exists and is defined correctly for each row in a class. No class should get into a state where indices are corrupted, but if you find that queries are returning empty or incorrect result sets, you may consider checking whether the classes' existing indices are currently valid.

You can validate indices programmatically as follows:

```
                Set status = ##class(<class>).%ValidateIndices(indices,autoCo
rrect,lockOption,multiProcess)
```

Here, the indices parameter is an empty string by default, meaning we validate all indices, or a $listbuild object containing the names of indices.

Note that autoCorrect defaults to 0. If 1, any errors encountered during the validation process will be corrected. While this is functionally the same as rebuilding indices, the performance of ValidateIndices is slower in comparison.

Please refer to the %Library.Storage class documentation for more information:

https://docs.intersystems.com/latest/csp/documatic/%25CSP.Documatic.cls?PAGE=CLASS&LIBRARY=%25SYS&CLASSNAME=%25Library.Storage#%ValidateIndices

Removing Indices

If you no longer have need for an index – or if you're planning a large amount of modifications to a table and want to save the performance impact of building the relevant indices for later – you can simply remove the index definition from the class in Studio and delete the appropriate index global node. Alternatively, you can run a DROP INDEX command via DDL, which will also clear the index's definition and data. From there, purge cached queries to ensure that no existing plans will be using the now-removed index.

Now what?

Indices are just one part of SQL performance. Along this vein, there are other options for monitoring the performance and usage of your indices. To understand your SQL performance, you can also consider learning about:

Tune Tables – A utility we run once a table is populated with representative data or if the distribution of data takes a massive shift. This fills your class definition with metadata, e.g. how long do we expect a field to be or how many unique values are in one field, which helps the SQL optimizer choose a query plan for efficient execution.

Kyle Baxter has an article on this here: https://community.intersystems.com/post/one-query-performance-trick-you-need-know-tune-table

Query Plans – The logical representation of how our underlying code executes SQL queries. If you have a slow query, we can consider what query plan is being generated, whether it makes sense for your query, and what can be done to further optimize this plan.

Cached queries – Prepared dynamic SQL statements – cached queries are essentially the code underneath query plans.

Further Reading

Documentation on defining and building indices. Includes additional steps for consideration on live read-write
systems. https://docs.intersystems.com/irislatest/csp/docbook/DocBook.UI.Page.cls?KEY=GSQLOPT_indices

ISC SQL commands – see CREATE INDEX and DROP INDEX for syntactical references of handling indices via DDL. Includes proper user permissions for running these
commands.
https://docs.intersystems.com/irislatest/csp/docbook/DocBook.UI.Page.cls?KEY=RSQL_COMMANDS

Details on SQL Collation in ISC classes. By default, string values are stored in index globals as SQLUPPER (“ STRING”):

https://docs.intersystems.com/irislatest/csp/docbook/DocBook.UI.Page.cls?KEY=GSQL_collation

[EDIT 05/06/2020: Corrections on index build performance and the DROP INDEX command.]

#Best Practices #Indexing #Performance #SQL #Caché #InterSystems IRIS

Source URL: https://community.intersystems.com/post/index-handling