

A Simple Way to Speed-up Access to your iSeries Data

How to speed up your queries and SQL statements by understanding the workings of the iSeries Query Optimizer

To complement the reports and enquiries provided by their standard ERP system, most iSeries shops use some kind of reporting tool to provide extra information to their users.

Whether we use Query to drive an iSeries report, SQL or Open Query File to provide green screen or web enquiries, or run a PC data analysis tool via ODBC, the system always uses the same API set to retrieve and return the data to the calling application. Probably the most important component, and frequently the least understood element of this API set, is the Query Optimizer. Although this routine has only one purpose, namely to decide the quickest method of accessing the data requested, it is one of the most sophisticated and intelligent routines that runs on the iSeries.

The likelihood is that the Query optimiser is sitting in the background all the time on your system, being called many times every hour and making thousands of decisions every day about how to access your data in the most efficient method.

By default this logic is hidden from view – the aim of this article is to show you the easiest way to be able to see the decisions the optimizer makes and show you how to improve its performance by the judicious creation of logical files or indexes. If you follow it to the end you will see how this works using a real example. Firstly lets look at how we can access the decisions the Query Optimizer has made.

How to Gather Information about Query Optimizer Decisions

As is always the case with the iSeries there is more than one method available of achieving this. If you turn debug on for a job (STRDBG UPDPROD(*YES)) and then run a query, the decisions the optimizer makes will be placed into the joblog. Alternatively you can modify the query options file QAQQINI to log the information. You can do it in code using APIs, but by far the easiest method I have found is to use the STRDBMON command:

```
STRDBMON OUTFILE(DBQRYLOG/DBMON) JOB(*ALL) TYPE(*DETAIL)
```

Just running this command once will cause OS400 to log details of all query optimization decisions made across your whole system to a file called DBMON in library DBQRYLOG. The library has to already exist, but the file will be created if necessary. This command will stay in force until you stop it, even after an IPL of your system.

The file created will give you all the information you could ever require about the use of queries on your system. Please bear in mind that whenever I refer to Query, this is a generic term meaning Query400, SQL or Open Query File. When you wish to end the monitoring, just enter the command:

```
ENDDDBMON JOB(*ALL)
```

The file is structured so that there are different record formats, providing varied information about actual query usage. The first field on the file (QQRID) is the record identifier. An ID of 1000 is the “header” information about the query and can be used to give us the source of the Query (Query, SQL or OPNQRYF), the user and job details of the query and the date and time the query started. Similarly if you look at records with an ID of 3007, field QQ1000 contains details of all the access paths that were considered by the optimiser. Details here are the library name and file name followed by a 2 digit code. A code of zero means that this access path was used by the query. A non zero value means that this access path was not selected, for example a code of 5 means that the keys of the access path do not match while a code of 19 means the access path cannot be used for a join as it contains select/omits.

The records that we are going to concentrate on are those where Query Optimizer recommends that an access path be created. These are any that have a Y in field QQIDXA.

How to Interpret the Query Optimizer Data

By far the biggest impact on your data retrieval performance will be the creation of the indexes recommended by Query Optimizer. In order for us to work on the most meaningful information, I recommend that you let the monitor process take place for at least two weeks. This is because we will be looking to create the minimum number of indexes to achieve the most impact. For example, if on day one the Query Optimizer recommended we create an index over our outstanding orders file by sales area, we could go ahead and create an appropriate index and some queries would run faster immediately. It may well be though that overnight some jobs are run which recommend building an index on sales rep within sales area. Similarly when we get to the end of week routines it may recommend an index keyed on customer within sales rep within sales area. If we created an index at each stage the system would now be maintaining three indexes, rather than just one if we waited until the end of the week.

To see which indexes have been recommended, run the following SQL statement (or the equivalent query).

```
SELECT QQPTLN,QQPTFN,QQIDX FROM DBQRYLOG/DBMON
WHERE QQIDXA='Y'
ORDER BY QQPTLN,QQPTFN,QQIDX
```

This statement pulls out records from our monitor file where the optimiser has recommended creation of an index to improve performance (QQIDXA = 'Y') and displays the library (QQPTLN) and name of the physical file used (QQPTFN) and the recommended keys (QQIDX). When you run this statement you will most likely see a status message saying "Building access path" or "Building hash table". This is because the optimizer has decided to build an index for us to present the data in the sequence we requested.

If you still have the DBMON logging active and you run our SQL statement again you should see entries for the first run of the SQL statement, recommending we build an access path keyed on QQPTLN, QQPTFN and QQIDX. (It may be that on your system the access path recommendation is different – this will be entirely dependent upon the makeup of the data in the log file. If yours is different please proceed but assume your recommendations are the same as my example). As a working example of what this article is all about, lets follow its recommendations and create an index to do this using SQL:

```
CREATE INDEX DBQRYLOG/DBMONL1 ON DBQRYLOG/DBMON (QQPTLN, QQPTFN,
QQIDX)
```

This creates an index called DBMONL1 in library DBQRYLOG, it is based on file DBMON in library DBQRYLOG and its key fields are QQPTLN, QQPTFN and QQIDX. A similar result could be achieved by keying in DDS and creating a logical file.

When you run the SQL SELECT statement again you should now notice that the query is satisfied much quicker, although there may still be a slight delay as we are doing a record selection on the file. To go that extra mile if we delete the index:

```
DROP INDEX DBQRYLOG/DBMONL1
```

And we then re-create it with the record selection field specified as an extra key, you should see our statement will run just that little bit faster again. This is because the database engine can now read just the records required rather than reading them all and performing dynamic record selection.

```
CREATE INDEX DBQRYLOG/DBMONL1 ON DBQRYLOG/DBMON (QQPTLN, QQPTFN,
QQIDX, QQIDXA).
```

How to Decide which Logical Files/Indexes to Create

It is not practical to create an index for every recommendation issued by the optimiser – if you did you would end up thousands of extra logicals on your system. What we need to decide is what will give us the most benefit for the least overhead. My favourite way of doing this is to summarize the log file, grouping by physical file library and name, and also by access path recommendation. If we also total the number of rows we can see which access paths the system is spending the most time building. The SQL statement to achieve this is:-

```
SELECT QQPTLN,QQPTFN,SUM(QQTOTR),QQIDXD FROM DBQRYLOG/DBMON
WHERE QQIDXA='Y' GROUP BY QQPTLN,QQPTFN,QQIDXD
ORDER BY QQPTLN,QQPTFN,QQIDXD.
```

Look for the entries with the highest number of rows, check to see if there is another physical file which may have the same recommended keys but with some extra trailing keys and then create the index using either SQL or DDS

Worried about Logical File Proliferation?

In all my time working on the iSeries and its predecessors, I have heard many conflicting opinions about how many logical files, or indexes, it is wise to have over any particular physical. I remember one respected figure saying that in no circumstances should you ever have more than 20. When considering this issue though, the main criterion is undoubtedly how volatile the physical file is. Once the logical has been created the only overhead associated with it is when a record is inserted, deleted, or whenever a key value in the logical changes. So, for example, if you had a very large table containing all previous years sales history, which you added to only once a year you could create as many logicals as you wished over this data as the only overhead would ever be when the records were being added. At the opposite end of the spectrum would be something like an inventory movements file, where every transaction is recorded and you may be adding thousands of records every hour. Having many hundreds of logicals over this file would probably result in some kind of performance degradation.

To view the IBM documentation for STRDBMON and the other methods visit

<http://publib.boulder.ibm.com/series/v5r2/ic2924/index.htm?info/rzajq/rzajqmst02.htm>

About the author.

Steve Close is technical director of Utilities 400 Inc of Millbrook, NY and Utilities 400 Limited, England. He has been involved in the development and installation of system management and optimization products for the iSeries, AS400 and System/38 since 1979.

Steve can be reached at sclose@uti400.com