

A Whole New Level of DB2 for i5/OS Query Optimization

These techniques give you hands-on query optimization for DB2 for i5/OS

by Mike Cain

WANT TO ILLUMINATE AND EXPLORE the concept and benefit of *query rewrite*. Given that virtually all users of the System i are using DB2 for i5/OS to store, access, and process data, and much of this work is driven by SQL queries, it might be advantageous to have a basic understanding of the art and science of query optimizations.

SQE

Beginning with V5R2, IBM has been delivering and enhancing a state-of-the-art query optimizer and database engine referred to as SQL Query Engine (SQE). What makes it “state of the art”?

To answer this question, let’s start with the premise that SQL is the standard language used to interact with the database management system. Let’s also remember that with SQL requests, we tell the DBMS what to do, not how to do it. For example, to join rows from tables Orders and Customers based on CustomerNo and calculate the total sales for the customers in Minnesota, we can issue the SQL request in Figure 1.

This SQL request says nothing about how to access the respective tables, identify rows containing “Minnesota,” perform the join, or group and sum the data by customer. Determining how to do this work is the domain of the query optimizer. To follow this example further, it is the database engine’s job to execute the plan provided by the query optimizer. So, what does this have to do with query rewrite? Everything!

Let’s assume that DB2 for i5/OS provides different methods to handle the various parts of a query. To identify and access the appropriate rows in a table, the optimizer can consider a table scan, an index scan with table probe, an index probe with table probe, a dynamic bitmap from an index and table probe from the bitmap, or other approach.

Each of these methods has a “cost” represented in terms of time. This time might be processing time

FIGURE 1

Sample SQL request

```
SELECT C.CustomerNo, SUM(O.Sales)
FROM Orders O, Customers C
WHERE O.CustomerNo = C.CustomerNo
AND C.Location = 'Minnesota'
GROUP BY C.CustomerNo;
```

by the CPU or I/O time handled by the memory and storage subsystems. The query optimizer compares each of the available methods to determine what it believes is the fastest method.

If a table in the join request has no local selection predicate, then every row in the table must be accessed, and the sequential table scan is the most efficient and fastest technique. If the SQL specifies a local selection predicate for the table, and we have an appropriate index available, the optimizer can rewrite your request to query the *index* instead querying the table. Yes, believe it or not, this is a form of query rewrite.

As a matter of fact, with SQL, you cannot tell DB2 to query an index directly. It is up to the query optimizer to figure this out. What if your join query request specifies local selection for only one of the two tables specified? Are we limited to a full table scan for the table with no local selection? Aha! The DB2 query optimizer can rewrite the query so that this table has local selection. Additional access techniques such as using an index can now be considered, allowing even faster processing.

This month's Networking &
Systems Management sponsor



Vision Solutions. Inc.
www.visionsolutions.com
(800) 957-4511

■ QUERY OPTIMIZATION

Recalling our example, the local selection predicate WHERE C.Location = 'Minnesota' applies only to the Customers table. Using an IBM patented technique known as look-ahead predicate generation (LPG), SQE can rewrite the query so there is also local selection on the Orders table.

Another benefit of LPG is that the order in which the tables are joined is no longer a major concern. SQE can perform its join optimization to find the best order: Orders|Customers, or Customers|Orders. Each table has local selection, so the number of rows accessed and joined can be reduced to only those required for the final query processing.

More importantly, if the optimizer can do all of this without hints or specific SQL coding, you have the basis for a truly state-of-the-art database management system. You have the advantages of DB2 for i5/OS.

Query Rewrite Tricks

One major goal of i5/OS is to provide sophisticated and powerful data-centric capabilities for business. Crafting DB2 for i5/OS to deliver these capabilities with little or no DBA intervention has been a key IBM design point for years. When it comes to applying SQL query optimization techniques, the same philosophy applies. All database management systems provide a set of “tricks” from a query optimization perspective. With the exception of DB2 for i5/OS, virtually all other systems rely on DBAs to help do the “costing” — in effect, overriding the particular optimizer’s own rules and/or costing process.

In other words, DBAs help pick the “trick” or rewrite the SQL request to fit the tricks available. Search the Internet for “query rewrite,” and you’ll find page after page identifying these tricks and describing how to apply them effectively. The new and improved SQE in DB2 for i5/OS offers an extraordinary set of query rewrite techniques and strategies without the need for the same level of administration and direction.

To fully understand the breadth and depth of DB2’s optimization capabilities, we need to look at the life cycle of an SQL query from the query rewrite perspective. Query rewrite techniques can be applied before optimization, during optimization, during execution, or during subsequent executions.

When you submit an SQL request for processing, the SQL statement itself can be rewritten to make it more efficient or to simplify it before the optimization phase. For example, you can automatically replace any literals with parameter markers (variables) so that the statement text appears more generic, enabling reuse.

Otherwise, the same SQL request with different

FIGURE 2

Transitive closure technique

```
SELECT *
FROM   TableA A,
       TableB B,
       TableC C
WHERE  A.JoinColumn = B.JoinColumn
AND    B.JoinColumn = C.JoinColumn
AND    A.JoinColumn = C.JoinColumn
```

FIGURE 3A

Sample query request

```
SELECT C.Location, C.Customers, SUM(O.Sales)
FROM Orders O, Customers C
WHERE O.CustomerNo = C.CustomerNo
GROUP BY C.Location, C.Customer
HAVING C.Location = 'Minnesota';
```

FIGURE 3B

Sample query request using query rewrite

```
SELECT C.Location, C.Customers, SUM(O.Sales)
FROM Orders O, Customers C
WHERE O.CustomerNo = C.CustomerNo
AND C.Location = 'Minnesota'
GROUP BY C.Location, C.Customer;
```

literal values will be treated as a different request. The more unique statements there are in the system, the more tracking and managing of SQL requests will occur. When the SQL request is optimized, you can apply query rewrite techniques to the various sections of the query or across the entire query.

In the FROM clause, you can replace tables with indexes or materialized query tables, expand views and common table expressions, and combine their definitions with the main query. You can even eliminate tables from the query if they are not required. With the WHERE clause, predicates can be added, removed, or moved to support access methods as well as match indexes.

Selected Tricks

Besides the LPG example I discussed earlier, another interesting and common rewrite technique employs “transitive closure.” For those of us who have long ago forgotten math class, here is a simple refresher: If $A = B$ and $B = C$, then $A = C$.

Making use of this principle allows DB2 to optimize query requests for faster, more efficient processing. For example, the three table join queries in Figure 2 define only the relationships of TableA–TableB and TableB–TableC. By applying transitive closure to the join condition, SQE can rewrite the query and add the relationship TableA–TableC. This gives us more latitude and options when optimizing the join order, as all combinations are now possible.

Another useful technique to minimize unnecessary I/O and data processing is “pulling up” a HAVING

FIGURE 4

Top customers view

```
CREATE VIEW The_2007_Top_Customers AS
  SELECT      C.Location,
             C.Customers,
             SUM(O.Sales) AS TotalSales
  FROM        Orders O,
             Customers C
  WHERE       O.CustomerNo = C.CustomerNo
  AND         O.Year = 2007
  GROUP BY   C.Location,
             C.Customer
  HAVING      SUM(O.Sales) > 1000;

SELECT T.Customers,
       T.TotalSales
FROM   The_2007_Top_Customers T
WHERE  T.Location = 'Minnesota'
ORDER BY T.TotalSales DESC;
```

FIGURE 5

Merged and rewritten query

```
SELECT      C.Location,
             C.Customers,
             SUM(O.Sales) AS TotalSales
FROM        Orders O,
             Customers C
WHERE       O.CustomerNo = C.CustomerNo
  AND       O.Year = 2007
GROUP BY   C.Location,
             C.Customer
HAVING      SUM(O.Sales) > 1000
  AND       C.Location = 'Minnesota'
ORDER BY   SUM(O.Sales) DESC;
```

FIGURE 6

Second merged and rewritten query

```
SELECT      C.Customers,
             SUM(O.Sales) AS TotalSales
FROM        Orders O,
             Customers C
WHERE       O.CustomerNo = C.CustomerNo
  AND       O.Year = 2007
  AND       C.Location = 'Minnesota'
GROUP BY   C.Customer
HAVING      SUM(O.Sales) > 1000
ORDER BY   SUM(O.Sales) DESC;
```

clause condition into the WHERE clause. This allows the database engine to perform local selection and row elimination before any join and grouping work. The query request in Figure 3A specifies grouping on Location and Customer and then dictates that only results for the “Minnesota” group be returned. It doesn’t make much sense to process all rows, only to throw most of them out afterward.

Using query rewrite, the optimized request is equivalent to the statement in Figure 3B. This in turn allows the database engine to access and aggregate only those rows whose location is Minnesota.

Another common example of clever optimization involves the merging of queries and pushing down predicates. This can occur when a VIEW or COMMON TABLE EXPRESSION (CTE) is specified in the query.

A VIEW contains no data. It is merely a logical rep-

resentation of a data set, as defined by the VIEW’s definition. In effect, the VIEW is only a query yet to be executed. When the VIEW or CTE is used in an SQL request, the query optimizer will rewrite the request to handle both queries as one, where possible.

This form of query rewrite allows the database engine to avoid unnecessary or duplicate processing. It can also minimize the use of temporary storage because the VIEW or CTE does not need to be materialized before running a user request. Let’s look at an example of how this might work.

Given the VIEW definition in Figure 4 to describe our top customers of 2007 based on their order history, we want to issue a query against this VIEW to return only customers in Minnesota. The query optimizer will merge our VIEW definition and our query into one request and “push down” or “pull up” the selection predicates to eliminate irrelevant rows as soon as possible. The merged and rewritten query might look something like Figure 5. And even more rewrites are possible with the example in Figure 6, so read on!

Subqueries

It is common for SQL developers and data-centric applications to do as much as possible with one SQL statement. Specifying subqueries is a common way to accomplish this. Unfortunately subqueries can also be the bane of good query performance.

Once again, the approach DB2 for i5/OS takes is to break down the request to simplify it. This simplification into basic constructs allows for more straightforward application of core access and processing methods.

For requests involving subqueries, this often means that the request is rewritten into a “join composite.” In effect, the subquery is transformed and blended into the outer query. This new join composite query can be optimized using all the various and powerful techniques available to any join query.

This frees up DB2 from having a catalog of subquery-specific techniques and strategies and, more importantly, it frees you up from worrying too much about using this particular syntax. To illustrate a simple subquery optimization, I’ll use the SQL request in Figure 7. For this searched update operation, we must find all the customers who have an order in the second half of 2007 and update their master row to reflect a good standing.

The good news is, SQE can rewrite this update request as a join, optimize the join order, and optimize each table’s access. It is even possible to read the Orders table first, pulling out a distinct CustomerNo, and then join into the Customers table to identify the row to update.

Learn-and-Adapt Conduct

Previously, I stated that the optimizer determines and builds the query plan, and the data engine executes the plan. This would imply that the plan is fixed and that the database engine is stuck running with what it has been given. If the plan is a good one, there should be no problem, so carry on.

But what if the plan is not optimal? For example, what if a temporary data structure such as a hash table is used for grouping, and the number of groups expected is way off? As more groups are stored in the hash table, it eventually might not fit into your job's fair share of memory. As the temporary data structure is accessed, more and more faulting will occur, which in turn will increase the query response time.

One solution is to provide the optimizer with the proper indexes and statistics to give it a better understanding of the data. Another solution is for DB2 for i5/OS to recognize the problem and alter the query plan — while it's running! In V6R1, SQE now has the ability to enhance the data structure to minimize the impact of sizing inaccuracies. As you run the same query repeatedly, DB2 is watching and learning. At some point, your request might again be “enhanced” by the query optimizer. If the query and data are the same, why not remember and return the final result? There's no need to actually do all the processing over and over. This is exactly what SQE can do.

If your application or SQL interface specifies “optimize for 30 rows,” the optimizer will indeed build a plan based on this request. But what if your application behavior is to fetch all 1,000 rows of the result set before waiting? Normally, you would be stuck running a plan that is not necessarily the fastest for returning all the rows of the result set. With SQE, the actual runtime behavior is recognized and the query changed to “optimize for all rows.” This “learn and adapt” conduct is greatly enhanced in V6R1 and will help you get the most out of your environment.

So why should you care about all this query rewrite stuff? If your application is running well, and you are satisfied, there's nothing to care about. On the other hand, awareness of query optimization lets you understand what is going on with the SQL requests, and possibly gain even more benefits by providing the best indexes and/or using more complex statements. For example:

- If you know that LPG will be employed, you won't be surprised when DB2 recommends an index on a table that has no local selection predicates specified.
- If you know that transitive closure will take place, you won't be surprised that the join order is optimized,

FIGURE 7

Simple subquery optimization

```
UPDATE Customers C
SET C.Standing = 'Good'
WHERE C.CustomerNo IN
(SELECT O.CustomerNo
FROM Orders O
WHERE O.OrderDate BETWEEN '2007/07/01' and
'2007/12/31');
```

- and indexes on all join columns will be beneficial.
- If you know that subqueries can be turned into joins, you'll be more likely to embrace them and support them properly with a good indexing strategy.

Monitoring and Analysis Tools

To help you recognize some of the more common query rewrites, DB2 for i5/OS SQL monitoring and analysis tools provide insight and information. You can use the detailed SQL performance monitor, SQE plan cache, and Visual Explain to understand what occurred during query optimization and execution.

For example, if a table was replaced with a materialized query table, this information is reflected in the monitor data. If you're using Visual Explain to illustrate the query plan, you can have MQTs highlighted. If local predicate generation took place for a given table, again Visual Explain can highlight the table and/or indexes involved.

When it comes to observing the logic applied at any given part of a query plan, a more complex set of information is also available in Visual Explain. The attributes and information associated with each “node” of the query plan is displayed in a separate pane, on the right side. At the bottom, “pseudo SQL” can be listed to provide a clue as to what processing is going on in that node. You can use this information to see the result of query rewrite.

Grace and Sophistication

Query rewrite is query optimization, and query optimization is query rewrite. Essentially they are one in the same. When employed systemically in a cost-based environment, they allow DB2 for i5/OS to handle both simple and complex SQL requests with grace and sophistication.

For helping me glimpse the elegance of query optimization, my thanks goes to Rob Bestgen, architect and lead developer of the DB2 for i5/OS SQL Query Optimizer. ■

► **Mike Cain** is a senior technical staff member within the IBM Systems and Technology Group and team leader of the DB2 for i5/OS Center of Competency. Prior to his current position, Mike worked as an IBM AS/400 systems engineer and IT consultant. Mike is in Rochester, Minnesota, and can be reached at mcain@us.ibm.com.