Craig S. Mullins

Database Performance Management

Return to Home Page

DB2 update

February 2007

Efficient SQL Coding Basics
By Craig S. Mullins

When it comes to assuring optimal performance of DB2 applications, coding properly formulated SQL is an imperative. Most relational experts agree that poorly coded SQL and application code is the cause of most performance problems – perhaps as high as 75% of poor relational performance is caused by "bad" SQL and application code.

But writing efficient SQL statements can be a tricky proposition. This is especially so for programmers and developers new to a relational database environment. So, before we delve into the specifics of coding SQL for performance, it is best that we take a few moments to review SQL basics.

The Basics

SQL, an acronym for Structured Query Language, is a powerful tool for manipulating data. It is the de facto standard query language for relational database management systems and is used not just by DB2, but also by the other leading RDBMS products such as Oracle, Sybase, and Microsoft SQL Server.

SQL is a high-level language that provides a greater degree of abstraction than do procedural languages. Most programming languages require that the programmer navigate data structures. This means that program logic needs to be coded to proceed record-by-record through data elements in an order determined by the application programmer or systems analyst. This information is encoded in program and is difficult to change after it has been programmed.

SQL, on the other hand, is fashioned so that the programmer can specify ***what*** data is needed, and not ***how*** to retrieve it. SQL is coded without embedded data-navigational instructions. DB2 analyzes the SQL and formulates data-navigational instructions "behind the scenes." These data-navigational instructions are called access paths. By having the DBMS determine the optimal access path to the data, a heavy burden is removed from the programmer. In addition, the database can have a better understanding of the state of the data it stores, and thereby can produce a more efficient and dynamic access path to the data. The result is that SQL, used properly, can provide for quicker application development.

Another feature of SQL is that it is not merely a query language. The same language used to query data is used also to define data structures, control access to the data, and insert, modify, and delete occurrences of the data. This consolidation of functions into a single language eases communication between different types of users. DBAs, systems programmers, application programmers, systems analysts, and end users all speak a common language: SQL. When all the participants in a project are speaking the same language, a synergy is created that can reduce overall system-development time.

Arguably, though, the single most important feature of SQL that has solidified its success is its capability to retrieve data easily using English-like syntax. It is much easier to understand the following than it is to understand pages and pages of program source code.

```
SELECT   LASTNAME
FROM     EMP
WHERE    EMPNO = '000010';
```

Think about it; when accessing data from a file the programmer would have to code instructions to open the file, start a loop, read a record,

check to see if the EMPNO field equals the proper value, check for end of file, go back to the beginning of the loop, and so on.

SQL is, by nature, quite flexible. It uses a free-form structure that gives the user the ability to develop SQL statements in a way best suited to the given user. Each SQL request is parsed by the DBMS before execution to check for proper syntax and to optimize the request. Therefore, SQL statements do not need to start in any given column and can be strung together on one line or broken apart on several lines. For example, the following SQL statement is equivalent to the previously listed SQL statement:

```
SELECT LASTNAME FROM EMP WHERE EMPNO = '000010';
```

Another flexible feature of SQL is that a single request can be formulated in a number of different and functionally equivalent ways. One example of this SQL capability is that it can join tables or nest queries. A nested query always can be converted to an equivalent join. Other examples of this flexibility can be seen in the vast array of functions and predicates. Examples of features with equivalent functionality are:

- BETWEEN versus <= / >=
- IN versus a series of predicates tied together with AND
- INNER JOIN versus tables strung together in the FROM clause separated by commas
- OUTER JOIN versus a simple SELECT, with a UNION, and a correlated subselect
- CASE expressions versus complex UNION ALL statements

This flexibility exhibited by SQL is not always desirable as different but equivalent SQL formulations can result in extremely differing performance. The ramifications of this flexibility are discussed later in this paper with guidelines for developing efficient SQL.

As mentioned, SQL specifies what data to retrieve or manipulate, but does not specify how you accomplish these tasks. This keeps SQL intrinsically simple. If you can remember the set-at-a-time orientation of a relational database, you will begin to grasp the essence and nature of SQL. A single SQL statement can act upon multiple rows. The capability to act on a set of data coupled with the lack of need for establishing how to retrieve and manipulate data defines SQL as a non-procedural language.

Because SQL is a non-procedural language a single statement can take the place of a series of procedures. Again, this is possible because SQL uses set-level processing and DB2 optimizes the query to determine the data-navigation logic. Sometimes one or two SQL statements can accomplish tasks that otherwise would require entire procedural programs to do.

The Optimizer

The optimizer is the heart and soul of DB2. It analyzes SQL statements and determines the most efficient access path available for satisfying each statement. It accomplishes this by parsing the SQL statement to determine which tables and columns must be accessed. It then queries system information and statistics stored in the DB2 system catalog to determine the best method of accomplishing the tasks necessary to satisfy the SQL request.

The optimizer is equivalent in function to an expert system. An expert system is a set of standard rules that when combined with situational data can return an expert opinion. For example, a medical expert system takes the set of rules determining which medication is useful for which illness, combines it with data describing the symptoms of ailments, and applies that knowledge base to a list of input symptoms. The DB2 optimizer renders expert opinions on data retrieval methods based on the situational data housed in DB2's system catalog and a query input in SQL format.

The notion of optimizing data access in the DBMS is one of the most powerful capabilities of DB2. Remember, access to DB2 data is achieved by telling DB2 what to retrieve, not how to retrieve it. Regardless of how the data is physically stored and manipulated, DB2 and SQL can still access that data. This separation of access criteria from physical storage

characteristics is called physical data independence. DB2's optimizer is the component that accomplishes this physical data independence.

If indexes are removed, DB2 can still access the data (albeit less efficiently). If a column is added to the table being accessed, the data can still be manipulated by DB2 without changing the program code. This is all possible because the physical access paths to DB2 data are not coded by programmers in application programs, but are generated by DB2.

Compare this with non-DBMS systems in which the programmer must know the physical structure of the data. If there is an index, the programmer must write appropriate code so that the index is used. If the index is removed, the program will not work unless changes are made. Not so with DB2 and SQL. All this flexibility is attributable to DB2's capability to optimize data manipulation requests automatically.

The optimizer performs complex calculations based on a host of information. To simplify the functionality of the optimizer, you can picture it as performing a four-step process:

1. Receive and verify the syntax of the SQL statement.

2. Analyze the environment and optimize the method of satisfying the SQL statement.

3. Create machine-readable instructions to execute the optimized SQL.

4. Execute the instructions or store them for future execution.

The second step of this process is the most intriguing. How does the optimizer decide how to execute the vast array of SQL statements that can be sent its way?

The optimizer has many types of strategies for optimizing SQL. How does it choose which of these strategies to use in the optimized access paths? IBM does not publish the actual, in-depth details of how the optimizer determines the best access path, but the optimizer is a ***cost-based*** optimizer. This means that the optimizer will always attempt to formulate an access path for each query that reduces overall cost. To accomplish this, the DB2 optimizer applies query cost formulas that evaluate and weigh four factors for each potential access path: the CPU cost, the I/O cost, statistical information in the DB2 system catalog, and the actual SQL statement.

Guidelines for Performance

So, keeping the information about the DB2 optimizer in mind, the following guidelines can be implemented to facilitate better SQL performance:

1) Keep DB2 statistics up-to-date: Without the statistics stored in DB2's system catalog, the optimizer will have a difficult time optimizing anything. These statistics provide the optimizer with information pertinent to the state of the tables that will be accessed by the SQL statement that is being optimized. The type of statistical information stored in the system catalog include:

- Information about **tables** including the total number of rows, information about compression, and total number of pages;
- Information about **columns** including number of discrete values for the column and the distribution range of values stored in the column;

- Information about **table spaces** including the number of active pages;

- Current status of the **index** including whether an index exists or not, the organization of the index (number of leaf pages and number of levels), the number of discrete values for the index key, and whether the index is clustered;

- Information about the table space and index nodegroups or partitions.

Statistics are gathered and stored in DB2's system catalog when the RUNSTATS or RUN STATISTICS utility is executed. This utility can be invoked from the Control Center, in batch jobs, or using the command line processor. Be sure to work with your DBA to ensure that statistics are accumulated at the appropriate time, especially in a production environment.

2) Build appropriate indexes: Perhaps the single most important thing that can be done to assure optimal DB2 application performance is creating correct indexes for your tables based on the queries used by your applications. Of course, this is easier said than done. But we can start with some basics. For example, consider the following SQL statement:

```
SELECT   LASTNAME, SALARY
FROM     EMP
WHERE    EMPNO = '000010'
AND      DEPTNO =  'D01';
```

What index or indexes would make sense for this simple query? The short answer is "it depends." Let's discuss what it depends upon! First, think about all of the possible indexes that could be created. Your first short list probably looks something like this:

- Index1 on EMPNO

- Index2 on DEPTNO

- Index3 on EMPNO and DEPTNO

This is a good start and Index3 is probably the best of the lot. It allows DB2 to use the index to immediately lookup the row or rows that satisfy the two simple predicates in the WHERE clause. Of course, if you already have a lot of indexes on the EMP table you might want to examine the impact of creating yet another index on the table. Factors to consider include:

- ***Modification impact***: DB2 will automatically maintain every index that you create. This means that every INSERT and every DELETE to this table will cause data to be inserted and deleted not just from the table, but also from its indexes. And if you UPDATE the value of a column that is in an index, the index will also be updated. So, indexes speed the process of retrieval but slow down modification.
- ***Columns in the existing indexes***: If an index already exists on EMPNO or DEPTNO it might not be wise to create another index on the combination. However, it might make sense to change the other index to add the missing column. But not always because the order of the columns in the index can make a big difference depending on the query. For example, consider the following query:

```
SELECT   LASTNAME, SALARY
FROM     EMP
WHERE    EMPNO = '000010'
AND      DEPTNO >  'D01';
```

  In this case, EMPNO should be listed first in the index. And DEPTNO should be listed second allowing DB2 to do a direct index lookup on the first column (EMPNO) and then a scan on the second (DEPTNO) for the greater-than.

  Furthermore, if indexes already exist for both columns (one for EMPNO and one for DEPTNO) DB2 potentially can use them both to satisfy this query so creating another index may not be necessary.

- ***Importance of this particular query***: The more important the query the more you may want to tune by index creation. For example, if you are coding a query that will be run every day by the CIO, you will want to make sure that it performs optimally. Who wants to risk a call from the CIO complaining about performance? So building indexes for that particular query is very important. On the other hand, a query for a low-level clerk may not necessarily be weighted as high, so that query may have to make due with the indexes that already exist. Of course, the decision depend on the importance of the application to the business – not just on the importance of the user of the application.

There is much more to index design than we have covered so far. For example, you might consider index overloading to achieve index only access. If all of the data that a SQL query asks for is contained in the index, DB2 may be able to satisfy the request using only the index. Consider our previous SQL statement. We asked for LASTNAME and SALARY given information about EMPNO and DEPTNO. And we also started by creating an index on the EMPNO and DEPTNO columns. If we include LASTNAME and SALARY in the index as well then we never need to access the EMP table because all of the data we need exists in the index. This technique can significantly improve performance because it cuts down on the number of I/O requests.

Keep in mind, though, that it is not prudent (or even possible) to make every query an index only access. This technique should be saved for particularly troublesome or important SQL statements.

SQL Coding Guidelines

When you are writing your SQL statements to access DB2 data be sure to follow the subsequent guidelines for coding SQL for performance. These are certain very simple, yet important rules to follow when writing your SQL statements. Of course, SQL performance is a complex topic and to understand every nuance of how SQL performs can take a lifetime. That said, adhering to the following simple rules puts you on the right track to achieving high-performing DB2 applications.

1) The first rule is to always provide ***only the exact columns*** that you need to retrieve in the SELECT-list of each SQL SELECT statement. Another way of stating this is "do not use SELECT *". The shorthand SELECT * means retrieve all columns from the table(s) being accessed. This is fine for quick and dirty queries but is bad practice for inclusion in application programs because:

   - DB2 tables may need to be changed in the future to include additional columns. SELECT * will retrieve those new columns, too, and your program may not be capable of handling the additional data without requiring time-consuming changes.

   - DB2 will consume additional resources for every column that requested to be returned. If the program does not need the data, it should not ask for it. Even if the program needs every column, it is better to explicitly ask for each column by name in the SQL statement for clarity and to avoid the previous pitfall.

2) ***Do not ask for what you already know***. This may sound simplistic, but most programmers violate this rule at one time or another. For a typical example, consider what is wrong with the following SQL statement:

```
SELECT   EMPNO, LASTNAME, SALARY
FROM     EMP
WHERE    EMPNO = '000010';
```

   Give up? The problem is that EMPNO is included in the SELECT-list. You already know that EMPNO will be equal to the value '000010' because that is what the WHERE clause tells DB2 to do. But with EMPNO listed in the WHERE clause DB2 will dutifully retrieve that column too. This causes additional overhead to be incurred thereby degrading performance.

3) ***Use the WHERE clause to filter data*** in the SQL instead of bringing it all into your program to filter. This too is a common rookie mistake. It is much better for DB2 to filter the data before returning it to your program. This is so because DB2 uses additional I/O and CPU resources to obtain each row of data. The fewer rows passed to your program, the more efficient your SQL will be. So, the following SQL

```
SELECT   EMPNO, LASTNAME, SALARY
FROM     EMP
WHERE    SALARY > 50000.00;
```

Is better than simply reading all of the data without the WHERE clause and then checking each row to see if the SALARY is greater than 50000.00 in your program.

These rules, though, are not the final word in SQL performance tuning – not by a long shot. Additional, in-depth tuning will likely be required. But following the above rules will ensure that you are not making "rookie" mistakes that can kill application performance.

Summary

In this article we have learned the basics of SQL and coding SQL for performance. But we have only scraped the tip of the iceberg. You will need to learn about increasingly complex types of SQL including joins, subselects, unions, and more; and you will also need to learn how best to write these SQL statements. You will also need to learn how to discover the access paths DB2 chose to satisfy your SQL requests. Indeed, there is much more to learn. But content yourself with the knowledge that you have embarked on the path of understanding efficient DB2 SQL.

From DB2 Update, February 2007.

Home.