High-Level Tips & Techniques for DB2 SQL Performance
By Craig S. Mullins

Keeping the basics of DB2 SQL and information about the DB2 optimizer in mind (see last month's blog entry titled **Coding DB2 SQL for Performance)**, 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 do with the indexes that already exist. Of course, the decision depends 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 be-all, end-all of SQL performance tuning – not by a long shot. Additional, in-depth tuning may be required. But following the above rules will ensure that you are not making "rookie" mistakes that can kill application performance.

Summary

Even after working our way through these high-level guidelines we have still only scraped the surface when it comes to DB2 SQL performance. 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. Keep tuning in to my blog entries on the IBM DB2 Community at Dell's Toad World and we'll explore these topics and more…