Surviving Client/Server: SQL Performance Tuning

by Steve Troxell

SQL was intended to make data more accessible to less sophisticated end users, thereby relieving some of the burden on the programming staff. It accomplishes this by providing a simple, concise syntax that allows the SQL programmer to focus on defining the data to retrieve without being encumbered by the technique used to access the data in the most efficient manner. This is not to say that performance is thrown by the wayside in favor of simplicity. Rather, most client/server SQL systems are smart enough to decide for themselves how to access the data most efficiently.

Query Optimizers

Many client/server SQL systems rely on an internal query optimizer that decides the best approach to take in processing any given query. The optimizer usually reorganizes the query and reduces it to a common form, so that syntactically different but functionally identical queries are handled the same way. It will make decisions such as whether an index can be used to sort the data or if it must use temporary storage to perform the sort, or whether a joined lookup table is small enough to load into memory or left on the disk.

Generally, however, the most crucial task the optimizer performs is deciding which, if any, index it will use in locating the rows you are looking for. It does this by examining the `WHERE` clause to determine which columns are being used to filter the result set and then checking the available indexes to see which one provides the best coverage for these columns. Some crude optimizers simply examine the fields in the `WHERE` clause in order and choose the first indexed column they find. Smarter optimizers make use of detailed statistics about the distribution of values across the table's indexes and will preferentially choose one index over another based on the perceived "cost" of performing the query.

For example, take the query shown in Figure 1 and assume an index on both the `Status` and `ZipCode` fields. Realistically, `Status` would contain very few unique values (`Active`, `Inactive`, `Hold`) and is said to have *low selectivity*. On the other hand, `ZipCode` would probably have a large number of unique values if the customer base were spread out over a large enough geographical area. `ZipCode` would then be said to have *high selectivity*. For this example, let's assume there are 7,500 customers in total, of which 5,000 are active. 100 customers are in the 80920 zip code, of which 87 are active.

Some optimizers will choose the `Status` index to process this query simply because it is the first field in the `WHERE` clause for which an index match was found. In this case, the query would not be very efficient, because there will be very many more active customers than there are customers in the 80920 zip code. The server will examine all 5,000 active customers and return the 87 that fall into the 80920 zip code.

A smart optimizer (one that is "cost-based") will use distribution statistics and recognize the selectivity and distribution of values in each of the indexes. It will judge whether it has to read more rows by following the `Status` column or by following the `ZipCode` column, and in this case it will correctly decide that it is cheaper in terms of I/O to follow the `ZipCode` column because there should be fewer rows to read to satisfy the query.

```
SELECT * FROM Customers
  WHERE Status = 'A' AND
        ZipCode = '80920'
```

➤ *Figure 1*

The server will examine all 100 of the 80920 zip code customers and return the 87 that are active.

To further illustrate how the optimizer works, let's assume we were doing the same query as shown in Figure 1, but we're looking for `hold` customers in the same zip code. Let's further assume that out of the total 7,500 customers, 25 are on hold and of those 5 are in the 80920 zip code. Given these values, the optimizer recognizes that it will have to examine fewer rows by following the `Status` index (25 `hold` records versus 100 `80920` records). The same query may utilize different indexes depending on the values used to filter the records.

You may be thinking that the overhead involved in all this decision-making about which indexes to use degrades the overall performance of the system and it would be faster to just tell SQL how to access the data instead of letting it decide for itself. In practice, most query optimizers can analyze a great number of possible access methods very quickly and generally do as well as or better than a human programmer at deciding the best access method (because they have more information to work with). The overhead of the query optimizer should not be a concern to client/server developers.

What should be of concern, however, is that no two SQL server products use exactly the same implementation of query optimizer. Some of the more sophisticated ones are difficult to defeat since

they will generally find the optimal method of access for even the most poorly conceived query. With some of the less sophisticated ones, you may have to "trick" the optimizer by carefully arranging your queries. A technique that may improve performance on one system won't necessarily have the same impact on another, simply because the optimizer is playing with a different rulebook.

Showing Execution Plans

Since it is important for us to know how our query optimizer behaves, we're going to look at a few techniques we can use to see what decisions the optimizer is making. Using Local Interbase Server, run WISQL and connect to the EMPLOYEE.GDB database (in file \IBLOCAL\EXAMPLES\EMPLOYEE.GDB).

We're going to run some queries against the `Employee` table, so we'll want to see what indexes are available. From the menu, choose `View | Metadata Information` and select `Index` for the object `Employee`. The output is shown in Figure 2.

Although the names are a bit cryptic, we can see that there are four indexes on this table: `NAMEX`, `RDB$FOREIGN8`, `RDB$FOREIGN9` and `RDB$PRIMARY7` each covering the fields indicated. The last three indexes are automatically generated by Interbase as a result of defining primary and foreign keys, which accounts for their unusual and less than helpful names.

Next, since Interbase uses a cost-based optimizer, it would be nice to get a feel for the distribution of values across a few of these indexes. The query shown in Figure 3 tells us that `Emp_No` is the most selective (understandable since it is a unique index) and that `Dept_No` and `Job_Code` are somewhat less selective. Note that this is a brute force indication of the distribution statistics for these indexes, the actual statistics used by the optimizer are more sophisticated.

Finally, we want to activate the WISQL option that shows us what the Interbase query optimizer is up to. From the menu, choose `Session | Basic Settings` and check `Display Query Plan`. Now, whenever we run a query, the output window shows us what indexes the optimizer has chosen. Look at the query shown in Figure 4. Note the `PLAN...` line in between the query and the results. This refers to the execution plan for the query and shows us that Interbase decided to use index `RDB$FOREIGN8` (the `Dept_No` field) to process this query.

Now look at the query shown in Figure 5. In this case the term `NATURAL` means that Interbase could not make use of any indexes and decided to scan the entire table to process this query. Some SQL server products will elect to do a table scan rather than use an index even if a useful index is available. In these cases, the optimizer has determined (from the selectivity statistics for the indexes) that enough rows would be processed by the query to make reading the index pages and the data pages more costly than if it just ignored the index and read all the data pages alone.

Figure 6 shows three related queries and their execution plans (without the results). As would be expected of the first query, the index for `Emp_No` is used. However, the second query ignores the index

```
SHOW INDEX employee
NAMEX INDEX ON EMPLOYEE(LAST_NAME, FIRST_NAME)
RDB$FOREIGN8 INDEX ON EMPLOYEE(DEPT_NO)
RDB$FOREIGN9 INDEX ON EMPLOYEE(JOB_CODE, JOB_GRADE, JOB_COUNTRY)
RDB$PRIMARY7 UNIQUE INDEX ON EMPLOYEE(EMP_NO)
```

➤ *Figure 2*

```
SELECT COUNT(*) AS Total,
       COUNT(DISTINCT Emp_No) AS Emp_No,
       COUNT(DISTINCT Dept_No) AS Dept_No,
       COUNT(DISTINCT Job_Code) AS Job_Code
  FROM Employee

      TOTAL      EMP_NO     DEPT_NO    JOB_CODE
========== ========== ========== ==========
         42         42         19         13
```

➤ *Figure 3*

```
SELECT Emp_No, First_Name, Last_Name, Phone_Ext
  FROM Employee WHERE Dept_No = '623'

PLAN (EMPLOYEE INDEX (RDB$FOREIGN8))

EMP_NO FIRST_NAME      LAST_NAME            PHONE_EXT
====== =============== ==================== =========
    15 Katherine       Young                231
    29 Roger           De Souza             288
    44 Leslie          Phong                216
   114 Bill            Parker               247
   136 Scott           Johnson              265
```

➤ *Figure 4*

```
SELECT Emp_No, First_Name, Last_Name, Salary
  FROM Employee WHERE Salary > 500000

PLAN (EMPLOYEE NATURAL)

EMP_NO FIRST_NAME      LAST_NAME                      SALARY
====== =============== ==================== ================
   110 Yuki            Ichida                 6000000.00
   118 Takashi         Yamamoto               7480000.00
   121 Roberto         Ferrari               99000000.00
```

➤ *Figure 5*

```
SELECT Emp_No, First_Name, Last_Name, Salary
  FROM Employee WHERE Emp_No > 50

PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))

SELECT Emp_No, First_Name, Last_Name, Salary
  FROM Employee WHERE Emp_No + 10 > 50

PLAN (EMPLOYEE NATURAL)
```

```
SELECT Emp_No, First_Name, Last_Name, Salary
  FROM Employee WHERE Emp_No > 50 - 10

PLAN (EMPLOYEE INDEX (RDB$PRIMARY7))
```

➤ *Figure 6*

```
SELECT * FROM Employee
  WHERE Salary > 50000
  ORDER BY Dept_No

PLAN (EMPLOYEE ORDER RDB$FOREIGN8)
```

➤ *Figure 7*

```
SELECT Emp_No, First_Name, Last_Name, Salary
  FROM Employee
  WHERE Dept_No = '623' and
        Emp_No > 10

PLAN (EMPLOYEE INDEX (RDB$PRIMARY7,RDB$FOREIGN8))
```

➤ *Figure 8*

and elects to do a table scan. Why? Because a calculation is being performed on the field side of the expression and this automatically negates the use of any index that might be present on that field. To get around this, arrange to have all calculations done on the value side of the expression as shown in the last query in Figure 6.

The query shown in Figure 7 demonstrates that an index was used even though no index exists for the field we are filtering on (the `Salary` field). Here we have an `ORDER BY` clause to sort the result set. Because the field we want to sort on is indexed, the optimizer uses it to traverse the table rather than perform an independent sort operation.

Finally, Figure 8 illustrates a more complex query involving multiple indexes. You'll notice that two indexes were used to access the `Employee` table. Because more than one index is available for the filtering fields defined in the `WHERE` clause, the optimizer makes use of both indexes by finding the matching rows for each case and reducing to the intersection of the two sets. Generally speaking, a set of pointers to all rows matching on the first index (in this case, `RDB$PRIMARY5`, the `Emp_No` field) is created. Then a set of pointers to all rows matching on the second index (`RDB$FOREIGN8`, the `Dept_No` field) is created. Finally, the two sets are sorted and any duplicating pointers are used to construct the final result set. This can actually be very fast and allows for greater flexibility in defining indexes, because you can typically define more single-field indexes and worry less about projecting which fields would be needed in combination for multiple-field indexes.

Care And Feeding Of Cost-Based Optimizers

Cost-based optimizers such as this are the most commonly found type in modern SQL databases and are usually the most desirable. However, they do require a bit of attention to be effective.

The distribution statistics that provide the optimizer with the information it needs to make an intelligent choice of indexes generally are not maintained automatically. As data is added, deleted and modified in a table, obviously the distribution statistics become less and less accurate. The optimizer may be misled into following a particular index which may actually result in poorer performance than some other data access approach.

Why isn't the system smart enough to keep its own statistics updated? The added overhead of mandatory statistics compilation on every data modification would degrade overall performance. In reality, the distribution statistics do not need to be 100% accurate. As you can see from Figure 3, the selectivity of the indexes as a percentage of the total size of the table can be ballpark figures and still produce effective results.

The task of updating the distribution statistics is handled through the `SET STATISTICS` statement (for Interbase; for Microsoft SQL Server use `UPDATE STATISTICS`). This command is used on each index or table in the database needing to be updated. Usually this task is done by the database administrator with batch processes or, if the server supports it, by scheduling the tasks to be automatically executed at the appropriate intervals.

The frequency with which you must update the distribution statistics varies depending on the activity in the table. In a predominantly read-only system, the indexes need only be updated after the initial population of data, or after significant updates. A predominantly write-only data collection system with little or no selective filtering of records may not require any updating. A system combining heavy reads and writes requires more frequent updating depending on how frequently and significantly the distribution of values in the indexes changes.

Of course it doesn't hurt to update the statistics as often as you can. However, the update process usually locks the table while it's working and large tables could take quite some time to complete.

Another issue involves the use of stored procedures. Since stored procedures are compiled queries, the execution plan for the query is determined at the time the stored procedure is created. The optimizer will look at the distribution statistics for the indexes available at compile time in deciding what the best execution plan will be. The information available at this time may not be a very good representation of the production environment. Tables may be sparsely populated (if at all) and the distribution statistics may lead the optimizer to elect to perform table scans on everything. The stored procedure will not be aware of any new indexes added after it was created nor will it change its execution plan if the distribution statistics change significantly (despite you having performed `SET STATISTICS` religiously).

There are many steps you can take to address this problem. First, when deploying a system it is a good idea to run all stored procedure scripts again after the database has been initially populated (and `SET STATISTICS` has been performed) so that the optimizer has another chance to devise a more accurate execution plan. Also, stored procedures could be recreated on the same schedule as the `SET STATISTICS` regimen. Or, some servers allow the option of forcing stored procedures to be recompiled every time they are executed. This has the advantage of relieving you of tending to the problem but does have the disadvantage of negating some of the performance benefits of stored procedures.

An extra problem with stored procedures is that they are much more likely to use variables instead of constants in their `WHERE` clauses for filtering values. For example, the stored procedure shown in Figure 9 uses two variables to filter for rows based on the `Dept_No` field and the `Job_Code` field. Since the desired values for the two fields are not known when the stored procedure is created (when the execution plan is formed), the optimizer cannot make use of index statistics

```
CREATE PROCEDURE CountEmployees(iDept_No char(3), iJob_Code varchar(5))
  RETURNS (oCount smallint)
AS
BEGIN
  SELECT COUNT(*) FROM Employee
    WHERE Dept_No = :iDept_No AND Job_Code = :iJob_Code
    INTO :oCount;
END
```

➤ *Figure 9*

to decide if `Dept_No` or `Job_Code` would be the better index to follow. It will use one of the indexes, but the one it chooses may not be the optimal one for any particular combination of values.

This problem generally does not occur with parameterized queries passed through from a Delphi application (see *TTable vs TQuery*, in the January 1996 issue). The parameter values are bound to the query when it is sent to the server and the optimizer can make use of those values when deciding how to form the execution plan. However, if you are explicitly using the `Prepare/Unprepare` methods to improve the performance of a series of calls to the same query, the optimizer will formulate an execution plan based on the first set of values it receives. It uses the same plan for all future executions of the same query (until the `Unprepare` method is called).

Indexes

Useful indexes are crucial to good performance in any database system, but because the optimizer is normally selecting indexes outside of your direct control, you may need to be a bit more careful in deciding what indexes to provide. You should analyze the `WHERE` clauses in your system to determine what should be indexed. Obviously, fields most commonly used for filtering data are candidates for indexing. The more indexes the optimizer has to select from, the more possibilities there are for it to pick an optimal execution plan for a variety of queries.

Decision support systems where data is churned through many different ways will benefit from having several indexes. On the other hand, a data collection system with many writes will usually be slowed down by the additional overhead of maintaining several indexes when the key fields change. When updating records, you may have to compromise when deciding to add additional indexes. Since the records to update have to be located first, there may be improved performance with additional indexes that may balance out the added overhead of index maintenance.

In SQL databases, indexes can be added or dropped independently of the table they are attached to. So you can easily experiment with various indexing strategies to see what impact a particular approach has on the system.

Clustered Indexes

Many SQL databases offer clustered indexes in which the data is physically stored in the order indicated by the index. This can be very helpful in cases where data is generally accessed in groups, but doesn't provide much benefit if rows are retrieved singly. For example, in an order entry system there may be a "master" table containing one row per order and a "detail" table containing one row per item in the order (a one-to-many relationship). The detail rows are keyed by the order number and a line number (1 through n for each item in the order).

If the detail table used a nonclustered index, then the detail rows could potentially be spread across several data pages, requiring additional I/O to gather all the detail rows for any particular order. In any event, the server reads the index to find each detail row and then locates the row in the data pages. If a clustered index is used, then all the detail rows are guaranteed to be consecutive within the table. The server can use the index to locate the first detail row and a

sequential table scan to gather the remaining rows. In contrast, if the master table had a clustered index on the order number, there won't be much additional benefit in retrieving a single order.

Clustered indexes may hinder write performance somewhat, but generally not as much as you might expect. If a new record is inserted in the middle of the table, the affected data page will be split in two without impacting the physical placement of the rest of the table. If frequent, this may result in a high degree of table fragmentation.

When deciding whether or not to use a clustered index, be very careful in your analysis of what the heaviest access method may be. You may think that a table of one row per order may not benefit from a clustered index. But if there is a reporting requirement for rapid retrieval of all orders handled by a particular user, then a clustered index on user ID may be appropriate. Good project analysis takes into consideration all aspects of the system (or as many as can be defined) when determining the "best" technique.

Conclusion

Careful indexing is one of the most important aspects of performance tuning for SQL databases. Some of the most useful information I've found on this topic comes from Joe Celko's book *SQL for Smarties: Advanced SQL Programming* (Morgan Kaufmann Publishers, ISBN 1-55860-323-9), a must-have for serious SQL programmers. We'll return to performance enhancement, but proper indexing and understanding the access methods of your queries should give you the most dramatic results.

Steve Troxell is a Software Engineer with TurboPower Software where he is developing Delphi Client/Server applications using InterBase and Microsoft SQL Server for parent company Casino Data Systems. Steve can be reached on the internet at stevet@tpower.com and also on CompuServe at 74071,2207