Tech Support Tid Bytes: Indexes - The Ultimate SQL Shortcut

By Jennifer Mills

Mastering the fundamentals of SQL can be a heady experience. All of a sudden, you can create a sophisticated query with a few lines of code, send it out into the database, and get back the data you need the way you need it. But the initial delight of using the simple and English-like SQL language can quickly fade into surprise and disappointment when queries take hours to execute.

After receiving several calls from BASIS ODBC Driver® users struggling with poor query performance times, we decided to do an in-house SQL query test. We selected all the records in our call log database--a medium-sized database of 14.6 megabytes with 48,145 records--and created a query that would return all the records in random order. The query was started at 5:00 p.m. and ran for about sixteen hours, finally finishing up at 1:30 p.m. the following afternoon.

What could have caused such poor query performance? The ODBC Driver itself contributed slightly to the slow response time. By incorporating a BASIS ODBC Driver we had added a middle layer into the system, but the time it added to the overall process could be measured in fractions of a second, not hours. The real culprit was a lack of optimization in our initial query. We had unleashed a simple query onto our call log files without any optimization, and the result was a very inefficient search.

SQL users can avoid the kind of poor performance we experienced by carefully optimizing their queries. Admittedly, optimizing SQL queries does require some up-front work in your database and requires a good understanding of tables, the BASIS Data Dictionary, and index theory, but the payoff is impressive. Our own in-house optimization tests have shown that optimized queries can run up to 6000% faster than their unoptimized counterparts. In this article, we will focus on the most effective and popular optimization method--indexing--and how you can incorporate this method into your own database.

Indexes--SQL Engine Shortcuts

A file index, also known as a key, is a special kind of shortcut, made up of a field or a combination of fields that lets an SQL engine quickly identify a record without having to read the record itself. By

eliminating the need for the engine to laboriously read and then compare each record to the search criteria, indexes dramatically improve query performance.

In any SQL-accessible database or file system, you will find two kinds of indexes--a primary index and a variety of secondary indexes. The primary index is a record field, like a customer number field, that uniquely identifies that record. A secondary index is more general and not necessarily unique to each record and a record can have several of them. A zip code is a good example of a secondary index.

Before you begin indexing your tables, make sure you carefully think through exactly which field or fields you need to index. Whenever a record is added or modified, the SQL engine has to modify the index and this can slow down performance. For that reason, you want to keep the number of indexed fields to a reasonable number. After deciding exactly what information you need to index, you should then consider how many fields you will include in each index. This is best determined by the types of queries--SELECTs, ORDER BYs, and JOINs--that you plan to use regularly.

Taking the Shortcuts

Once you have defined your indexes, you can begin taking advantage of these indexes in your SQL statement's WHERE and ORDER BY clauses. To show exactly how effective an optimized query can be, we return to our infamous sixteen-hour call log experience and compare an optimized and non-optimized query against our call log database.

Indexed Query

```
SELECT * from CALL_LOG where ID>'0000022000'
```

This query will select all the records in the call log table that have an ID number greater than 000022000. Because the primary index for each call log record is its ID number, the ODBC Driver can quickly retrieve the records. After executing the query, we can take a look at an excerpt from the log file to see exactly what happened in the query:

```
(file 1)=f:/odbc/Local_Call_Hist/data/call_log
order_knum=-1
[Selected] Predicate: 1 constraints
*!: (file1) (knum=0, kseg=1) ID (bracket head)
```

The `[Selected]` and the `!` in front of the ID line indicate that the query was successfully able to use the ID index to optimize its search.

Nonindexed Query

```
SELECT * from CALL_LOG where ID>'0000022000'
```

In this example, we again ask for all the calls that have an ID number greater than 0000022000. The Data Dictionary was modified so that the table's index referenced a different field. Additionally, the physical data file was rewritten so that the ID field was not defined as a keyed field. The following is an excerpt from the log file:

```
(file 1)=f:/odbc/Local_Call_Hist/data/call_log
order_knum=-1
Predicate: 1 constraints
*: (file1) (knum=0, kseg=1) ID (no bracketing)
```

The performance difference between the two queries is amazing. Because the indexed query was able to specify that only records higher than 000022000 be searched, the SQL engine only had to look at 239 records and whizzed through the query in only .25 second. On the other hand, the non-indexed query ended up having to read through all 48,145 records to see if they met the criteria and took 14.71 seconds to complete. In other words, the indexed query was 5884% faster than the nonindexed query!

At this point, we do want to mention that we specifically set up this query to show the maximum amount of time difference between the two queries. You will not always get a near- 6000% performance improvement, but this example does clearly show the power of indexes and query optimization. In our next Tid Bytes article, we will delve further into indexes by examining multiple-field indexes and how to use them in optimized queries. We will also offer some advice on how to further test and optimize your queries with query testing.