dbe SOFTWARE

DBE Software, Inc.
6842 Elm Street
McLean, VA 22101
USA

E LCS@dbesoftware.com
P +1-703-847-9500
F +1-703-991-2500

It's the Code, Stupid!

The main performance problem in the huge majority of database applications is bad SQL code. It is not lack of hardware. It is not network traffic. It is not slow front ends. I would say that most bad code is the result of a bad schema design.

The next obvious question is: Why do we have bad SQL code? A big reason is that it is very easy to write bad SQL and difficult to write good SQL. If you can get a query to run at all, it will return something. If you get back the something you wanted to get, then life is good. But very often, you cannot tell if a result is right just by looking at the result set.

Testing is important, of course, but you need to cover all the possibilities. In a procedural language, you can filter the data one step at a time and inspect it at each step. If I have new data that is not expected, there is a good chance that it will cause a program fault or error message in a well-designed program. In effect, the program says "I don't know what to do, so I am passing it to the user," and then it is up to the user to think of something. In SQL, the weird data will not be selected and I keep getting the previous results.

We have centuries of data processing principles to assure correct procedural code that began with paper systems. We also have a lot of software tools and templates with which to build the front-end side of the application.

That culture does not really exist in SQL yet. Yes, there were declarative and functional languages before SQL, but they were not popular enough to get the support culture that procedural languages have. Other than formal logic, there are only heuristics to help the SQL programmer. To make this clearer, when I want to do a sort in a procedural language, I have algorithms, which are so well known that they have names such as QuickSort, heap sort, Bubble sort and so forth. Can you recall an SQL programming method that has a name?

In SQL, you are actually writing a program specification. What do we, as programmers, know about writing good specifications? This must be really difficult because we seldom saw a good, clear one when we were coding in procedural languages! In the 1970s, Zohar Manna demonstrated this fact at conferences by taking the problem of sorting an array of integers. He would come up with several "solutions" that met the spec, but destroyed the data. By the way, his classic Mathematical Theory of Computation is back in print (Dover Publications, ISBN 0-486-43238-6).

Traditionally, analysis and coding were separate skills. Did anyone give the SQL programmer training for both skills? I would bet not. The usual way that someone becomes an SQL programmer is that their shop gets an RDBMS product, the programmers get a minimal introduction to the product and then they start writing embedded SQL code. It takes six years to become a Union Journeyman Carpenter. I would be willing to bet that most production SQL databases are built by programmers with far less experience.

What do the newbie SQL programmers bring to the job? For the most part, the "wrong stuff" instead of the "right stuff." A skill set developed for procedural languages is actually dangerous in the relational world. A procedural programmer sees the world as sequences of discrete records; an SQL programmer thinks in sets. The newbie SQL programmer will often use cursors to sequentially process a table like a file rather than use a simple single set-oriented SQL statement. Cursors are orders of magnitude slower and mess up concurrency, but a cursor FETCH looks and acts like a READ statement.

How bad can this be? From my personal experience at an educational testing company, I was able to make one procedure 2,300 times faster and 300 lines of code shorter. This was extreme, but improvements of two to three orders of magnitude are common. The bottleneck was one programmer. All of his code was so bad that it did not matter that his routines were not called that often - a bomb does not have to go off very often to do damage.

In a procedural language, such as COBOL, FORTRAN, Java or C++, the same program will compile to the same executable code on the same platform for the same compiler every time. SQL is a declarative language that must decide how to implement the statements it is given. The optimizer looks for access methods (usually indexes) and statistics about the table sizes, data distributions in the columns and a dozen other factors to create the execution plan for the query.

The SQL programmer might write code that is good for the current contents of the database. Later, the statistics change and the code performs poorly. Most of the time, recompiling the code with the new statistics will solve a short-term problem. Exactly how and when you can recompile your code will vary with each product.

However, recompiling does not always work; sometimes the entire query needs to be replaced. As an example of this problem, consider the nested EXISTS( ) predicates version of relational division made popular by Chris Date's textbooks and the COUNT(*) version of relational division, which I made popular. The winter 1996 edition of DB2 Online Magazine (http://www.db2mag.com/db_area/archives/1996/q4/9601lar.shtml) had an article entitled "Powerful SQL: Beyond the Basics" by Sheryl Larsen, which gave the results of testing both methods. Her conclusion for DB2 was that the nested EXISTS( ) version is better when the quotient has less than 25 percent of the dividend table's rows and the COUNT(*) version is better when the quotient is more than 25 percent of the dividend table.

In 1988, Fabian Pascal published an article on PC database systems at the time, "SQL Redundancy and DBMS Performance" in Database Programming & Design (vol. 1, #12; Dec 1988; pp. 22-28). Pascal constructs seven logical equivalent queries for a database. Both the database and the query set were very simple and were run on the same desktop hardware platform to get timings.

The Ingres optimizer was smart enough to find the equivalence, used the same execution plan and gave the best performance for all the queries. The other products at the time gave very uneven performances. The worst timing was an order of magnitude or more than the best. In the case of Oracle, the worst timing was more than 600 times the best. Products are better now, but this illustrates the point.

Bad Schema Design

Are inexperienced programmers the only source of bad SQL code? No, I would say that most bad SQL code is the result of bad schema design. It is fairly easy to write code against a normalized, well-designed database. Data integrity is enforced by the constraints, DRI actions and perhaps some triggers, but not by procedural code. The joins are obvious. The data uses industry standards. In short, it feels like a comfortable tool in your hand.

However, in a poorly designed schema, you need excessive joins to get to data. Temporary tables are used to hold intermediate results that should have been hidden in a derived table. Extra code is needed to remove redundancies that should not have existed in the first place. There are too many nulls and non-standard codes. Proprietary auto-numbering is used in place of actual relational keys, so there is no validation or verification of the data. The schema contains so many proprietary features that it cannot be ported or maintained by anyone who is not an expert in that dialect of SQL. In short, you cannot trust anything and must add code to protect yourself.

The best SQL programmer cannot overcome a bad schema. The most he can hope to achieve is a query that works right and runs "good enough" for now, even though we know it will not scale up. This is an important point with SQL versus procedural programming. Most procedural programs are linear - if you have twice as many records in a file, it takes approximately twice as long to process.

A good SQL query can have linear or less than linear growth as the table sizes increase. However, a bad SQL query can have exponential growth as the table size increases. This is usually a result of hidden cross joins, poor indexing and other factors that lead to multiple passes over the same data.

The first response to a performance problem is to buy monitoring and diagnostic software. These tools are useful and have their place, but they cannot solve the root problem. If you locate a procedure that is performing poorly, telling the programmer who wrote it to fix it is not going to make him smarter. He might be able to make some improvements, but unless he has made himself smarter, the new code will be written at the same competency level as the old code.

Another factor is that most organizations lack a feedback loop. If one front-end application programmer discovers that a business rule is not enforced on the database side of the house, he will put it into his code ("VIN numbers do not allow certain letters in certain positions"). That is mopping up the spill. The programmer should not be doing this; those rules belong on the database side or in a rules engine. That is fixing the leak.

The worst case scenario is that the database has no constraint at all. The database has a "Lamborghini Civic" manufactured in 1950, and that is just fine. Now several programmers have slightly different ideas about the rules for fixing this. In one program, the "Lamborghini Civic" is just fine; but in another, it is corrected to a "Honda Civic." In a third, it never appears, but shows up in the totals, etc.

Most organizations have no way for a programmer to send a memo to the DBA to tell him we need a CHECK( ) constraint on the table for this rule. Furthermore, if the DBA did add the constraint later, most organizations have no way to tell the programmers to drop the extra front-end code and catch the constraint exception in the usual way.

Solutions

First, spend the money to train programmers in SQL. They cannot train themselves. They cannot learn it from a few books in a weekend. This is a better use of money than software tools.

Second, set up a feedback mechanism between the application and the database sides of the house. This will let you know what the business rules actually are.

Article published in DM Review Magazine
Prepared by
Joe Celko (www.celko.com)
June 2005