Soaring Eagle Resource

DATABASE PERFORMANCE FOR EXECUTIVES

We received a call recently from a business partner who told us, in essence, "We have a customer, a large hospital, who complained of database performance issues. We spent $3M of their money upgrading their systems, and the performance is the same. The CIO is worried about his job, and we are worried that we are going to be sued for the $3M. Can you help us?"

This one incident (we hear of them on a monthly, if not weekly basis) is indicative of a host of problems. First & foremost is that in order to solve a problem, first you have to identify it. This premise sounds simple, but I assure you it's extraordinarily common to "throw hardware at a problem." The problem is that if hardware was not your problem hardware won't solve the problem. This is always intuitive after the fact, but you also have to understand the issue of making changes in database production environments.

I was on site at a large health insurance company a few years ago, who had a relatively small window (6 weeks) to attempt a database upgrade. At any organization of any size, a database upgrade is something you test thoroughly before you put it into production. At this customer site there was a performance issue in the new software they'd uncovered. A process that was taking about 2 minutes (which was barely acceptable) was suddenly taking 23 minutes (which was completely unacceptable). If they could solve it, they could roll out (internally certify and upgrade the software), if not, they'd have to wait almost a year for the next window. In comes the expert (us). We found that while they did indeed find a very specific software bug, there was another option. The code was written badly (in fact was one reason that the bug was uncovered). A small change in the code, and the 2 minute job could run in 2 ms. I was pleased with the finding, brought it to the customer expecting a pat on the head, only to have the customer tell me that solution was unacceptable. With a 6-week window, I had expected this would be an easy code change to fix and roll in, and was very surprised that my solution was rejected. After all this was about 60,000 times faster than the prior approach, and 600,000 times faster than their existing dilemma. The customer said, "It takes us 8 weeks to review and roll out any changes. Try again."

The message here is that for IT Executives, it is easier – politically and technologically – to make a hardware change than a software change. CIOs often "Throw money at a problem" in order to make it go

away because it is more cost effective to guarantee an immediate resolution to a problem than it is to identify the root cause and fix it.

The catch is, if the problem wasn't hardware, it may be that faster CPUs, more memory, or faster storage access times are not what was needed, as that hospital found out the hard way. Sometimes you have no alternative to fixing the code, or tuning the database.

This is one place where the is no substitution for experience and expertise – note that those are two separate, mandatory requirements.

ROOT CAUSE ANALYSIS

In order to solve a problem, you first need to identify the problem. Buying faster hardware may solve the problem, if that is in fact the problem, but in real life this approach simply masks the problem for a while, the problem continues to snowball, then to continue the metaphor the snowball freezes and hits you with a hard ice ball instead of a soft, poofy snowball. (I don't think it feels like a soft poofy snowball when it first hits, but compared to the ice ball… read the above issue with the hospital).

So, how do we get there from here?

Tools

First you need tools that enable you to measure what's happening on your system. There are dozens of tools out there, and most of them are JUNK. I've lost count of the number of times I've been brought in to solve a performance problem, I've been proudly shown the tools the shop has purchased, which are useless, which was the first reason they couldn't figure out the problem without me.

We will not be listing any tools here. I don't want to catch flak from vendors I list as bad, and while we are reselling the very few tools we think are good (i.e. make us successful), we do change those over time, and would much rather you call us & ask for a current recommendation.

That said, your tool needs to do many things (in no particular order):

1) Measure the utilization of resources over time… CPU, memory (yes, very different from CPU), storage bandwidth, network bandwidth. That list of 4 is a simplification, as there are multiple metrics that are tracked for each, but that's the starting list.

2) Measure the impact of database activity… this includes the ability to identify which queries are running, and what they are doing. "What they are doing" in this context does not mean what report they are running, but what resources they are using and how they are using them. For example it's important to know if a query is CPU-bound (limited by CPU), memory-bound, blocked by another process with locks on it… etc.
3) Measure the impact of other activities as they interact… "Is my report slow because lots of folks are running it, or because it's being blocked, or because I simply can't push that much data over the internet?"
4) Graphically and easily identify which queries are taking up the most resources
5) Graphically and easily identify which queries are taking up the most elapsed time
6) Graphically and easily identify which queries are running most frequently
    a. There's a difference between one query running for an hour and one query running once per second for an hour and one query running 1000 times/second for an hour
7) Graphically and easily identify which logins are involved in the above (critical in tracking down who did what & when)

There's more but this is a start… if your tool can't do these things, and simply enough that a production support manager (i.e. manager and not data expert) can look, identify a problem, and ask for resolution, your tool is likely shelf-ware (i.e. software the lives on a shelf and doesn't get used)

FIXING THE UNDERLYING PROBLEMS

So once you've identified the underlying problem, that's where you target your resources, whether that be buying hardware (as of this writing, the most common hardware purchases we are recommending are faster disk – for when we have positively quantified an IO bandwidth issue – and memory, also when we've quantified an issue – most folks overbuy CPU as it's very commodity today, even with DBMS vendors pricing their wares by CPU).

Before we do either of those, though, we look to tune the application. It's not unusual, with a few relatively minor and simple changes, to reduce CPU from 95% to 16% (a recent success), or memory requirements to a tiny percentage of what they were, or disk requests from 100 trillion/hour to 3000/hour (not making that one up either).

Note that many research organizations have quantified mean time to resolution (MTTR) for performance issues at 80% identification of problem, 20% resolution, another great reason to invest in a tool to track down issues.

How do we do this? There are 3 primary tasks, query tuning, architecture, and index selection.

Query tuning

What happens in real life (i.e. outside the classroom) is that software developers are very good at creating logical, sensible, procedural code that's relatively easy to maintain. This last part is very important. CIOs / CTOs will tell you that the average cost of maintenance of an application is seven times the cost of developing it. Restated, if you spend $1,000,000 building a software application, you'll spend $7,000,000 maintaining it before the application gets replaced. The easier the code is to maintain, the lower you can keep that number – business rules and needs ALWAYS change.

Here's your problem: The people who write your applications generally do NOT understand database performance, and as a result often create artistic examples of sophistry – creations which are eminently logical but flawed because they don't understand how the underlying technology works.

Solution: have your DBAs code review before any data manipulation language (DML) accesses anything on your production database applications.

Architecture

Architecture of your DBMS is very hardware and application dependent, but there are a variety of issues that can negatively drive performance. Here's a short-list.

High availability is a buzz-phrase that has made the circuit, and really means that if your primary DBMS fails, you have another one to back it up, often automatically. For example, if your primary server is in Miami, and Miami gets hit by a hurricane, are your users in Chicago going to be happy that the hurricane in Miami, which perhaps cut off power or an internet line, means that they are unable to do their jobs?

There are a variety of ways you can make this happen, many of them popular and successful today. Some of them are being done, though, with an eye towards perfection rather than performance.

For example, let's say that you have a primary data center in Miami, with a failover in Chicago. It's reasonable to assume that a hurricane hitting Miami won't take out the Chicago data center and that an ice storm taking out the Chicago data center is going to have no effect on Miami. Good decision so far.

Have you considered, though, the speed of light? How is that relevant?

Well, pushing information across a fast line to Chicago from Miami might take 30ms. The signal coming back that says "Got it!" would take the same 30ms. 60ms round-trip time doesn't seem like a lot. In fact, as an exercise for the reader, wait 60ms before you read this next sentence. Oh, wait, 60ms is a bit below our threshold of awareness. But multiply that out by thousands of processes per second. This might end up creating a backlog which affects performance on your primary server.

That example assumed "synchronous" data commitment – in other words, making a change at the one site could not complete before the change was made permanent at the remote site. What, though, if we chose

"asynchronous" data commitment? For asynchronous commitment, we commit (make permanent) the changes on the primary, without regard to whether the secondary is up, and trust the changes to make it to the secondary. It becomes possible that there is a lag between the data on the primary moving to the secondary, in case of a sudden unavailability of the primary and failover to the secondary… of about 30 ms of data.

Some places and times, this is no matter. Users will know there was a failover, and make sure whatever they were working on still made it. Others, that loss completely unacceptable (for example, you are transferring money from your checking account to your savings account, and an automatic withdrawal fails because of that, your payments are late and don't' make it… there will be lots of unhappy campers.

Indexing

Indexing is the life blood of the database performance expert.

We hope you're all old enough to remember phone books, as we haven't come up with a better metaphor for database b-tree indexes.

Consider the phone book. It is an index organized but last name, then first name, then middle name, which allows you to look up an address and telephone number.

This "phone book" structure is the key to getting information quickly from large databases.

Consider a table with 100,000,000 rows of data in it. This is not the least bit unusual anymore (100 million used to be a very large table, but we have customers with 20 employees with much larger tables… on the order of 10 to 100 times bigger). So, how do you want to find your address & phone number? By checking every row? Or by using the "phone book?" We will pick the phone book most of the time (not all of the time… this is where database performance experts earn their money), as we can then get to the information we want with just a few (8 or 10) page requests (a page is the basic unit of io), rather than millions.

That telephone book (OK, the indexes) are going to need to be constructed to meet the needs of each query that runs against the database. Stated more technically, and query that the DBMS needs to optimize and process will need a matching index in order to avoid the table scan – the systematic scanning of the entire table.

Matching the indexes to queries is a science and art, and can have a dramatic effect on query performance. A favorite example is reducing a 48-hour query to 23 seconds by modifying a query.

Another recent example had 3,000 users downloading data in 2 minutes rather than 30 – prompting 3,000 calls to the help desk, saying "Thank you! Thank you!"