SQL Query Optimization Part 1
Determining the Workload of a SQL Query on Actian Zen

A White Paper From

GOLDSTAR SOFTWARE

www.GoldstarSoftware.com

For more information, see our web site at
http://www.goldstarsoftware.com

SQL Query Optimization Part 1:

Determining the Workload of a SQL Query on Actian Zen

Last Updated: 02/04/2022

Note #1: *This paper is part 1 of a multi-part series on SQL query optimization. You may wish to start with Part 1 before looking at the other parts, as each takes a specific area of query optimization and breaks it down into manageable steps.*

Note #2: *This paper includes screenshots from the Actian Zen v15 product specifically, but the information presented herein relates to all versions of the Actian Zen, Actian PSQL, and Pervasive PSQL – all the way back to the Pervasive.SQL 2000i, in fact!*

Introduction

One common complaint we hear about the Actian PSQL/Zen database environment is that SQL queries are running "slowly" for some reason. Now, it is known that the SQL Relational Database Engine (SRDE) is internally single-threaded, so working on large data sets won't see parallelization that other SQL engines may offer, but "slow" queries are usually the result of a SQL query that is either improperly designed, overly complicated, or simply a query that is requiring a lot of effort to complete.

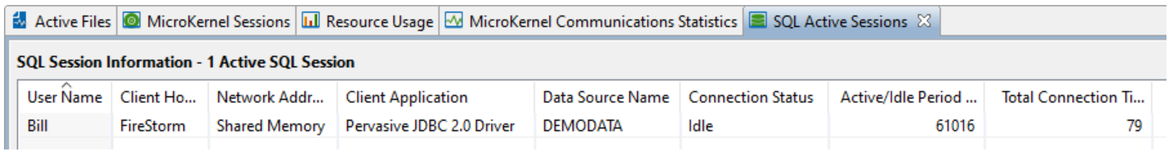
Before a query can be optimized, we need to first understand the workload of our query so that we can understand exactly what the database engine is attempting to do when it runs the query. Once we know how much work was required to run the query in its current state, we can then analyze the query in detail to see exactly what decisions the database engine made about the data and how the data was eventually accessed at the Microkernel level. Finally, we can make small changes to the query in an attempt to optimize the query. As we make the changes, we can then observe the difference in the workload and decide if the change we made is good or bad.

This paper covers the first part of the process, determining the workload of the SQL query itself.

Selecting the Right Metric

Most people use the clock as their optimization metric. (Admittedly, some queries may require a calendar.) This is easy to understand, because the query run time is easy enough to measure, and when you are waiting 5 minutes for your report to appear on the screen, it becomes obvious that there is a lot of work going on behind the scenes.

It is possible to monitor incoming SQL queries with the **Zen Monitor** utility in the *SQL Active Sessions* tab, which looks like this:



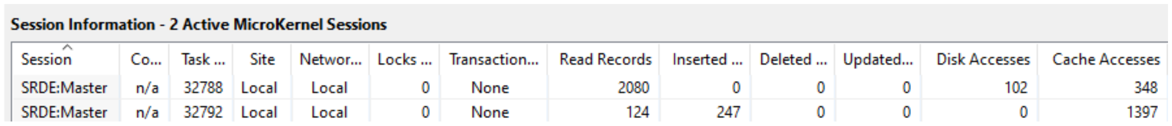
SQL Session Information - 1 Active SQL Session

User Name	Client Ho...	Network Addr...	Client Application	Data Source Name	Connection Status	Active/Idle Period ...	Total Connection Ti...
Bill	FireStorm	Shared Memory	Pervasive JDBC 2.0 Driver	DEMODATA	Idle	61016	79

If you look for connections with a *Connection Status* of "Active", the time immediately to the right (the *Active/Idle Period*) will tell you for how long that query has been running. Queries that have been running for more than 30 seconds are indeed suspect queries and worth a second look. Remember, though, that this screen refreshes only every 5 seconds by default, and once a query goes away, it is hard to tell exactly how long it took.

Although this sounds quite easy, the "clock time" required to run a query is not really a good metric. In order to complete a given query, a set amount of data must be examined. If this data is not already in the database engine cache memory, then each page that is accessed will see a delay waiting for the disk drive to provide that page, and the query will seem to run quite slowly. If you run the same query a second time, then the pages are already in the cache, eliminating the disk latency, and the query runs quite rapidly. However, don't be fooled – you didn't just optimize the query – you just traded slowness due to disk for high CPU utilization.

Luckily, the **Zen Monitor** utility provides additional data in the *MicroKernel Sessions* tab instead.



Session Information - 2 Active MicroKernel Sessions

Session	Co...	Task ...	Site	Networ...	Locks ...	Transaction...	Read Records	Inserted ...	Deleted ...	Updated...	Disk Accesses	Cache Accesses
SRDE:Master	n/a	32788	Local	Local	0	None	2080	0	0	0	102	348
SRDE:Master	n/a	32792	Local	Local	0	None	124	247	0	0	0	1397

Note that there are TWO separate MKDE sessions for this Zen Control Center SQL session. You will have to click on each MKE session to see the data files (in the lower section of the screen) that are currently opened in order to tell them apart, but one will usually have your data file(s) open, while the other will have some temporary files opened as support for the SRDE and the Control Center itself. In this example, the first session is the actual one that the SRDE is using to retrieve data from the files.

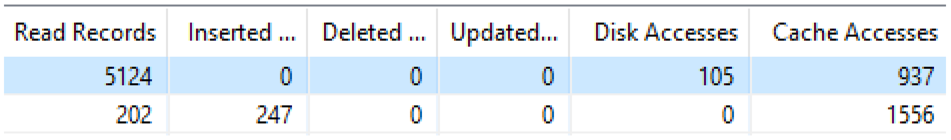
At first glance, you might think that the *Read Records* reported by the **Zen Monitor** is a good metric. However, this is not always the case. This counter does not always count each record accessed, but only those calls actually reading records, so GetNextExtended calls can reduce the apparent number of records reported here.

So, what should we use instead? As mentioned above, the SRDE needs to access a certain number of database file pages in order to examine all of the data to satisfy a query. This is reported in the last two columns, *Disk Accesses* and *Cache Accesses*. If you add these two values together, you will get a total for the number of pages actually accessed by the query, regardless of whether they were satisfied by the database engine cache or if they required a read from the disk.

Determining the Workload

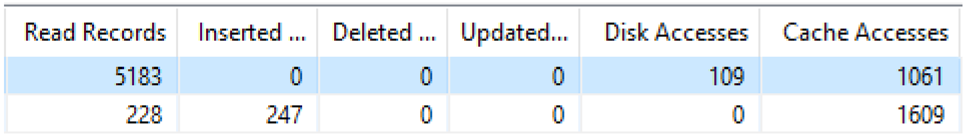
Now that we have a usable metric, we can use the number of pages accessed as a way to determine the workload required to run a specific query. If we use the **Zen Monitor** to look at these values both before and after the query, then some simple math tells us the page access count.

Let's look at a specific example. We first start by examining the counters prior to starting the test query:



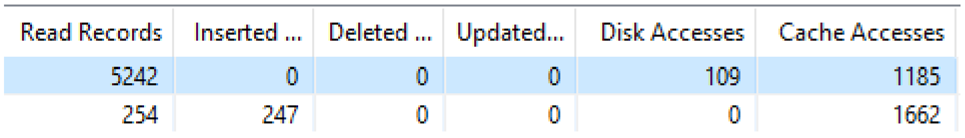
Read Records	Inserted ...	Deleted ...	Updated...	Disk Accesses	Cache Accesses
5124	0	0	0	105	937
202	247	0	0	0	1556

We can then run our query in the same Control Center window and observe the changes:



Read Records	Inserted ...	Deleted ...	Updated...	Disk Accesses	Cache Accesses
5183	0	0	0	109	1061
228	247	0	0	0	1609

Based on this result, we can see that the query itself generated 109 – 105 = 4 page accesses from the disk and 1061 – 937 = 124 page accesses from the cache. Of course, if your engine cache is large enough to hold the entire data set, the math gets quite a bit easier once the data is completely in cache. (You can force the data into the cache by running the query once, record the current values, then re-run the query a second time and record the new numbers:



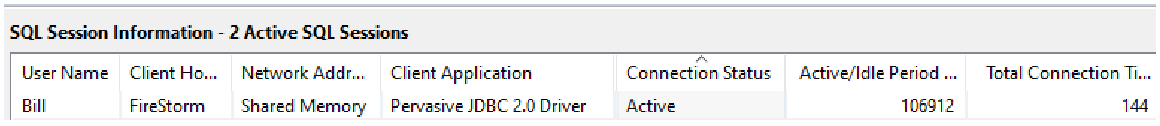
Read Records	Inserted ...	Deleted ...	Updated...	Disk Accesses	Cache Accesses
5242	0	0	0	109	1185
254	247	0	0	0	1662

This time, we see that the *Disk Accesses* values do not change, as all of the data was already cached. We now need to only keep track of a single number changing and calculate the difference between the two runs, which gives us 1185 – 1061 = 124 page accesses from cache for this sample query.

A More Complex Example

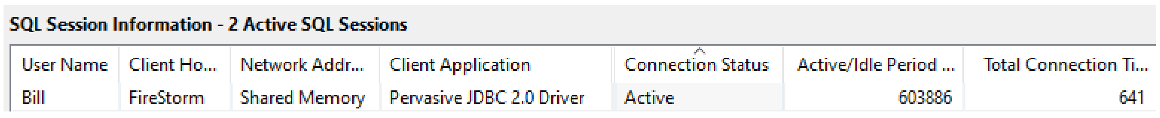
The above tests were done with small files, and relatively little workload. As such, any optimization we do is likely to not provide much in the way of gains. However, let's see what it looks like when a very nasty query is submitted to the engine.

While the query is running, we see the clock time increase in the *SQL Connections* tab:



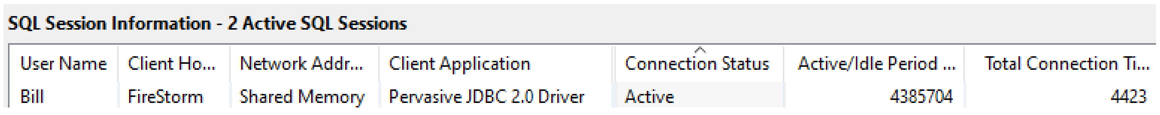
SQL Session Information - 2 Active SQL Sessions

User Name	Client Ho...	Network Addr...	Client Application	Connection Status	Active/Idle Period ...	Total Connection Ti...
Bill	FireStorm	Shared Memory	Pervasive JDBC 2.0 Driver	Active	106912	144



SQL Session Information - 2 Active SQL Sessions

User Name	Client Ho...	Network Addr...	Client Application	Connection Status	Active/Idle Period ...	Total Connection Ti...
Bill	FireStorm	Shared Memory	Pervasive JDBC 2.0 Driver	Active	603886	641

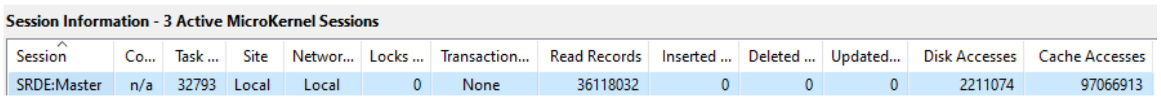


SQL Session Information - 2 Active SQL Sessions

User Name	Client Ho...	Network Addr...	Client Application	Connection Status	Active/Idle Period ...	Total Connection Ti...
Bill	FireStorm	Shared Memory	Pervasive JDBC 2.0 Driver	Active	4385704	4423

Obviously, this query is taking a REALLY long time to finish. Note that the *Active/Idle Period* is in milliseconds, but even the last snapshot above indicates that the query has been actively running for over 4385 seconds, or over 73 minutes. [In the end, I gave up waiting for the query and went to lunch, so I was unable to get a true runtime for this query. It is not surprising the end user was complaining about it being slow!]

When the query had finally finished, I switched over to the *MicroKernel Sessions* tab to get the page counts. There I saw the following:



Session Information - 3 Active MicroKernel Sessions

Session	Co...	Task ...	Site	Networ...	Locks ...	Transaction...	Read Records	Inserted ...	Deleted ...	Updated...	Disk Accesses	Cache Accesses
SRDE:Master	n/a	32793	Local	Local	0	None	36118032	0	0	0	2211074	97066913

Here you can see that the entire query required a LOT of work, generating over 36-million record reads, which in turn required 2.2-million page accesses from the disk and over 97-million page accesses from cache.

And, in a sad twist of fate, the date range we used did not hit ANY valid data within our database, so this effort returned 0 rows of data, and essentially generated a report with NO data on it. Yikes!

Conclusion

Understanding the SQL query workload is only the first step in the optimization process, of course. The next step is to capture the SQL query plan and examine it in more detail, and hopefully find out why this query takes so long to run. This is covered in the next white paper in our series, so look for Part 2!

If you still can't get it to work, contact Goldstar Software and let us help!