WHITEPAPER

INGRES

FINDING AND FIXING TROUBLESOME LONG-RUNNING INGRES QUERIES

BY CHIP NICKOLETT, INGRES CORPORATION

TABLE OF CONTENTS:

About the Author

Chip Nickolett is the Director of Consulting Services for the Americas at Ingres. He has over 23 years of business and IT experience, and has been using Ingres RDBMS technology for over 20 years. He is a recognized expert in performance tuning and management of mission-critical Ingres installations.

FINDING AND FIXING TROUBLESOME LONG-RUNNING INGRES QUERIES

PREFACE

Performance is a relative thing. Most Database Administrators (DBAs) know what is expected within their environment, but they may have problems with identification of slow and possibly problematic queries. Having a method to identify problems is half the battle when it comes to proactively managing that environment. This white paper provides a method (and free downloadable tool) to do just this within the Ingres environment. While the tool and examples are Ingres specific, a similar methodology could be used for Oracle or other RDBMS products.

OVERVIEW

Disaster "long-running query" is one that takes too long to run. So how long is "too long"? The amount of time that equates to "too long" is dependent on your specific environment. In our experience, long-runs tend to fall into one of three categories:

- Queries that take an excessive amount of time to complete. These queries are often related to reporting activity. A good example is a year-to-date production summary report. These are generally queries that are considered important.
- Queries that take a relatively short time to complete but are run all the time. These are usually OLTP related (for example, a database procedure that allocates a tracking number).
- Ad Hoc queries run by privileged users (including developers and help desk personnel).

This white paper describes the method for detecting, analyzing, and fixing long-running queries that we currently use. The advantage to this procedure is that with the proper scripting long-running query detection can be automated.

Additionally, problem queries are automatically e-mailed or written to a file to be addressed when the DBA has the time. While this is not a completely proactive approach it does provide enough information to identify and address problems before they get too bad. This article also includes information on what to look for once you have collected your long-running queries, how to go about fixing the problems, and what to test. Hopefully this method will be as effective for you as it has been for us.

THE SYSTEM

In order to improve query performance you must first identify what we call "the long-runs". This can be performed using the Ingres utilities "iinamu" and "iimonitor". For example:

- Use "iinamu" to find the connection addresses for all DBMS servers in the system. (In Unix, "echo 'show' | iinamu")
- Use "iimonitor" to list all of the sessions ("show sessions formatted") within the server to a file. (In Unix, "echo 'show sessions formatted' | iimonitor XXXX")
- Wait a period of time (typically 5 – 10 minutes).
- Use "iimonitor" to list all of the sessions to a file again.
- Compare the two sets of iimonitor output to determine which queries have been running over the wait interval. These are the "long-runs". Because of the manner in which they are collected, you can be sure they have been running at least as long as the wait time from above.
- Repeat

Ideally, this process can be automated through the use of a script that generates e-mail to the DBA group (and/or writes to a file) as soon as a query is seen to pass over a given threshold. The script should be coded in such a way that the threshold value can be easily adjusted. This way, as you fix problem queries, you can force the acceptable threshold down to detect the next set of problems.

QEP REVIEW

This document assumes that the reader has a basic understanding of Ingres query execution plans, or QEPs. This understanding should include knowing how to generate and interpret the QEP (see an example). For those of you requiring a "brush up" on the process, we recommend reviewing the Ingres "DBA's Guide" or the Ingres Tech Support document "US: 38697, EN".

Here are some of the basic tips for reading QEPs:

- QEPs are read in post-order form. (That is, you start from the bottom left and work your way over and up.)
- The most restrictive portion of the query should be "pushed down" to the bottom of the plan (by doing this, you carry the minimum amount of data through the rest of the query). This could be either a base table or an index.
- When generating a QEP, the Ingres optimizer facility (OPF) is going to rely on table statistics to help make accurate choices. If the statistics are incorrect (or nonexistent) the OPF can make bad choices. It should be noted that sometimes "perfect" statistics will generate worse performing queries than old or bad statistics. For that reason it is important to save your old statistics (using "statdump –o") prior to running the optimizedb utility.
- When processing a complex query, it is possible that the OPF will time out. This occurs when theoptimizer decides that it would take more time to generate a new plan then it would take to run the best plan it currently has.
- Ingres can only use one access method to a table at a time. For instance, if a query has restrictions on col1, and col2 and is running against a table that is btree on col1 and has a btree secondary on col2, Ingres is only able to use the btree structure of the table or the secondary – not both. This concept is very important! The OPF will decide which way is more efficient.
- When you generate a QEP, you get estimates for Disk I/O (D) and CPU statistics (C). The CPU statistics are basically useless – you are concerned with the number of Disk I/O's required. The smaller the number, the faster the query. It is important to compare the actual disk I/O required versus the estimated disk I/O.
- The cost estimates for CPU and Disk I/O is cumulative up the tree.

WHAT TO LOOK FOR

Once you have built your "hit list" of queries, you need to determine which ones you are going to spend your time on. The process of generating and analyzing a query can take from 15 minutes, to several hours, to several days depending on the complexity of the query, the amount of time it takes to run, and your familiarity with the data and the database. A common misconception is that you need to be intimately familiar with the environment prior to recommending changes. Not so! Knowing how to interpret a QEP and how to retrieve table and index information from Ingres are enough to solve most problems.

In order to start prioritizing the queries, our first step is to look at the frequency that each query appears. Queries that are run more frequently will generally be addressed first. Next, place all of the queries into a script along with the commands "set qep; set optmizeonly;" at the top and let it run against the production database. Since you are just generating QEPs and not running the queries, there should be no ill effects on the production machine. Remember that the QEP estimates are just that, estimates. If a query is in your longrun list but generates a QEP that looks fine you will need to do further analysis. That analysis includes the use of the Ingres trace point "QE90" (more information on this is provided below). We also recommend assigning each of the queries a number at this point as it makes tracking these queries easier.

One thing that you want to look for at this point is patterns in the queries. For example, is there one table (or group of tables) that is involved in several problem queries? If so, there is a possibility that a change you make to one of these tables could fix the other queries (of course, it could also make them much worse). Once you start capturing queries from the servers on a daily basis, it is possible to determine which queries are occurring with the most frequency. These make it to the top of our list, as does anything where we feel that we can make a "quick fix" (problems solved here are usually the result of developer error – disjoint queries, use of functions on key columns in the where clause, etc.).

Now that you have identified which queries you want to focus on, you should get table information "help table xxxx" for each table involved in the query. If your site uses indexes that carry data in the indexes as non-key data, you will need to do a "help index xxxx" to see what data columns are in that index. Once you have that information handy, you can now start looking for the following:

- Ad Hoc queries. These are some of the easiest problems to spot. More likely than not they are queries of the "select *…." variety. In general, ad hoc queries have no place in a production system. At the very least, users should be required to submit queries to the DBA to review prior to being allowed to run them. If you are seeing the same query frequently, perhaps it is something that development should code an application for. If it is impossible to control ad hoc queries (for whatever reason) the resource control within Ingres should be investigated.
- Cartesian products (almost always bad). This usually traces back to an unrestricted join in a program, and is a relatively easy fix for a developer. We have seen more and more of these as people beginusing the newer outer join syntax.

- Any type of detail table as the bottom node (usually it's best to start with a header table – remember, the most restrictive portion of the query should be pushed down the tree!).
- Any node showing as NU (Not Used) is suspicious, especially if you think the index or structure should be used. If the optimizer is failing to use an index, you can force the query to explicitly join against the index to see how the query runs. Note: This is not recommended for production systems!
- Bad estimates (QEP says 30 DIO, you know the query runs for 2 hours). This could be an indication that the statistics are off. You should try regenerating statistics for the tables involved in the query (generally, statistics should be created for all key columns, columns involved in joins, and columns present in a where clause). In the best case scenario, the optimizer picks a better plan and your query runs faster. In the worst case scenario, the QEP stays the same except the DIO statistics now read 50,000 rather than 30. The best way to identify bad statistics is to look at the tuple estimates at each node. Are they fairly accurate? If not there may be a statistics problem. Important note: Before generating new statistics it is highly recommended that you save the old statistics using the "statdump -o" Ingres command. New statistics do not always perform better and having the old statistics around will facilitate a quick recovery (using the"optimizedb -i" Ingres command).
- Table scans (look at the number of DIO for the node and compare it with the number of pages in the ORIG node for the table in the QEP). If your DIO count is greater than 25% of the number of pages then a table scan is most likely occurring.
- Functions such as "upper()" and "lower()" being used within a where clause. When this occurs on an index column, the index will not be used as an Index (although it may be scanned as data) and a table scan will be performed.
- Use of conventional temporary tables. With OpenIngres 1.2 and above, lightweight session tables are available for use. These tables are not logged, do not require updates to the system catalogs, and do not persist beyond the session. They are ideal for cases where a true temporary table is needed.

HOW TO FIX IT

There are many things you can do to improve a problem query. Unfortunately, there are tradeoffs involved in virtually all of them. What you need to do is to balance the improvements to the query in question versus the degradation that the change may cause in other areas. (This is usually referred to as an "impact analysis" of the change.)

Other possible solutions:

- Adding columns to an index. A significant cost is incurred when a query that has gone into a table through a secondary index needs to join back to the base table to get some piece of information. Sometimes it is possible to add additional data to an index to satisfy the query (which eliminates the join back to the index). Possible problems with this approach include disk space issues (you are adding additional data to the index) and concurrency issues (every time the base table gets an insert/update/delete transaction the index needs to be kept in sync).
- Changing the storage structure of a table or index.
- Generating statistics on a table. This allows the optimizer to make a more informed decision when choosing a QEP. However, there are instances where statistics can actually cause performance to degrade. For this reason, it is always advisable to copy current statistics out (via statdump) prior to regenerating statistics on a table.
- Adding a new index. This has the benefit of providing an alternate access method to the table. It carries with it all of the liabilities of adding columns to an index. As a rule of thumb, in an OLTP environment is ideal to keep the total number of indexes on a table below 6 (there is generally a significant degradation in performance around 8 or more secondary indexes).
- Adding a column to a table (denormalization) to eliminate a join. This has the obvious advantage of speeding up the query through the elimination of a join. Possible problems include data consistency issues (which can usually be managed by database rules and procedures) and disk space concerns.
- Using a session table to replace a conventional temporary table. This can speed up a query by eliminating some of the overhead associated with a conventional temporary table. It should be noted that these session tables will attempt to reside in the DMF cache and may displace good production data (lowering your cache hit ratio).

- Using a session table to hold intermediate data for a multi-query transaction (MQT). For example, if a MQT is summarizing inventory data for several different parts within a date range, it would make sense to build a session table with all the information for the date range and then run queries to retrieve data for each of the individual parts, rather than run multiple single-query transactions (SQT's) that restrict from the inventory table on date range and part.
- Rewrite the query. This is often the most difficult option, as it involves extensive testing to ensure the proper data is being returned.
- "Clustering" the data is sometimes useful. For example, assume that you have a table that is btree unique on emp_id, and contains a column that only contains a few values (such as a loc_cd) which are always included in the where clause of queries accessing the table. If the table is made btree on the loc_cd (with a unique secondary on emp_id), queries that access the table for only a specific loc_cd will be able to gather the data using fewer DIOs.

Clustering is a method of improving performance by minimizing I/O; it is generally not intended to provide direct access into or uniqueness on the structure. The goal is to physically locate data that would be frequently accessed together in close proximity to each other. This not only improves performance by minimizing I/O, it can improve concurrency by minimizing the number of data pages that need to be locked during execution of the query.

TESTING YOUR CHANGES

Once we begin testing changes intended to improve the performance of long-runs we print (or save to a file) a copy of the QEP from each change made. We annotate the QEP to indicate what the change was, why we made it, and whether this change was helpful or not. This way we are able to go back and recreate any test situation that we encountered. Additionally, it keeps us from duplicating the same test twice. When performing this type of testing, it is important only to change one thing at a time. Our personal preference is to make a change, test the change, and then determine if it worked or not. If it worked, we document it and move on to the next change. If the change does not result in the desired effect we document this fact (including the relevant metrics) and back the change out.

A helpful tracepoint to use when working with QEPs is "QE90". Setting this trace point causes Ingres to print actual disk and cpu statistics for the current query. This enables you to see how accurately the optimizer is estimating the disk requirement and row counts for a query. If these numbers are way off, it usually points to a problem with statistics. Usually, we will use "QE90" as part of our final testing, unless we suspect that there is a problem with the statistics during our initial QEP analysis.

An additional part of the final testing (prior to production implementation) should involve testing any other queries that use the table(s) being modified. You should test on a representative sample of data. If you have a 60 GB database and your test system is only 2 GB you are not going to be able to effectively test performance (unless it contains full copies of the tables being used). Performance problems usually do not follow a straight line progression. (e.g., if a query runs in 2 minutes on the 2 GB database, it does not follow that it will run in 60 minutes on the 60 GB database). Additionally, when you test you should vary the selection criteria in the where clause of the query you are testing. This will cause the optimizer to read different cells in the histogram for the table in question, and may help to point out problems with statistics.

Sometimes what looks to be a problem query may in actuality be a contention problem. If a query is waiting on a lock, it will appear in the server each time you look in iimonitor until the lock is released or the query times out. The easiest way to identify these problems is to run the query (and observe it through IPM) prior to making any changes. **Make sure that you are fixing the real problem.**

Don't be discouraged if your changes don't work! There are times when you will spend all day working on a query only to determine that it's inherently a bad query and there is nothing that you can do to fix it. Conversely, it is possible to look at a problem query and make a quick change that reduces the run time from 20 hours to 2 minutes. The more you practice this process the easier it becomes!

FREE DOWNLOAD

The script described in this white paper ("longrun") is GPL and can be found on the Ingres Community website at the following locations:

UNIX tar format: http://community.ingres.com/forums/home.php

Windows zip format: http://community.ingres.com/forums/home.php

About Ingres Corporation

Ingres Corporation is a leading provider of open source database management software. Built on over 25 years of technology investment, Ingres is a leader in software and service innovation, providing the enterprise with proven reliability combined with the value and flexibility of open source. The company's partnerships with leading open source providers further enhance the Ingres value proposition. Ingres has major development, sales and support centers throughout the world, supporting thousands of customers in the United States and internationally.

INGRES CORPORATION : 500 ARGUELLO STREET : SUITE 200 : REDWOOD CITY, CALIFORNIA 94063
PHONE 650.587.5500 : FAX 650.587.5550 : www.ingres.com : For more information, contact info@ingres.com

INGRES

WP-303A