Part 1: Tuning Informix SQL

Examples and analysis

Jianing Fan

SQL queries provide the backbone for Informix® database applications. This article discusses guidelines for SQL query tuning for Informix, looks at factors you need to consider when tuning SQL queries, and examines some real life tuning examples gathered from the author's experience.

Introduction

Tuning SQL query is often thought of as a major responsibility of programmers and developers, but database administrators should also be actively involved in this process. One of the major advantages of having database administrators involved in SQL query tuning is that they can offer a different perspective. Whereas the programmer approaches the problem from the point of view of the application performance, a DBA can approach the problem with an understanding of the database itself, and can offer advice and suggestions for the database layout, arrangement of tables and indexes, and effective use of Informix and system resources, including data fragmentation, PDQ priority, CPU time, memory utilization and data storage. Sometimes, programmers and developers just want to get a different view of their queries in terms of performance so that they can modify the query for higher efficiency.

This article will cover a number of approaches to tuning. Part 1 covers the criteria for tuning, tools, and a general methodology. In addition, it discusses the role of the optimizer, including the OPTCOMPIND parameter, query directives, and internal statistics.

Stay tuned for Part 2, when I'll cover performance considerations for the queries themselves, including access methods, join methods, extents for tables and indexes, index levels, fragmentation and PDQ priority. Part 2 will also include examples and analysis.

Criteria for tuning

The criteria for tuning depend largely on individual business requirements. In general, the performance of your system and database is the major concern. How do you then profile your system and database performance? There are many considerations, but the most critical are...
response time and resource utilization. Response time refers to the time that users wait for their requests, that is, for their SQL queries to complete. Resource utilization refers to the use of system resources such as CPU, memory and disks in performing SQL queries.

In general, if, after you've gone through the tuning process, the response time of the SQL queries after tuning is shorter and the utilization of system resources is less, then you can conclude with confidence that you have tuned the SQL query for higher efficiency. How do you measure response time and resource utilization of a SQL query? Unix and Informix have provided us some tools and utilities to help us making scientific and quantitative measurements of response time and resource utilization.

Tools for tuning

Unix tools

Time and timex

You can use the time utility to report the response time of system operations such as file transfer, database query execution, and other activities. Here is an example showing how to use the time utility to measure the response time of a simple database query:

```
sys3523:omcadmin > time dbaccess airgen_cm_db sel.sql
Database selected.
   (count(*)
      5958
1 row(s) retrieved.
Database closed.
real    0m0.09s
user    0m0.01s
sys     0m0.06s
```

The last part of the output above gives you the detailed time statistics of the operation, which in this case is the database query:

- The "real" field tells you the time elapsed between the beginning of the query and the end of query.
- The "user" field tells you the sum of CPU time used by the user processor for the operation.
- The "sys" field tells you the sum of CPU time used by the system as a whole.

The field we are most interested in is the "real" field; that is the field indicating the response time of the operation. Let's have a close look at the above example. From the output, we can determine the response time for our query is 0.09 seconds.

Timex is just another variation of the time utility that will present time in more readable format. Here is the timex output of the same query:
The `vmstat` utility delves into the system and reports system resource usage statistics about running processors, virtual memory, disk I/O, trap and CPU. It also displays a one line summary of virtual memory activities since the system was rebooted or booted. Here is an example of the output:

```
sys3523:omcadmin > vmstat 1 10
procs memory page disk faults cpu
 r b w swap free re mf pi po fr de sr m0 m1 m4 m5 in sy cs us sy id
 0 0 0 1950208 1294824 141 824 0 0 0 0 0 0 0 0 0 906 946 700 2 3 95
 0 0 0 1955568 1260288 0 46 0 0 0 0 0 0 0 0 0 834 386 213 0 0 100
 0 0 0 1955568 1260288 0 40 0 0 0 0 0 0 0 0 0 834 325 186 0 0 100
 0 0 0 1955568 1260288 0 40 0 0 0 0 0 0 0 0 0 834 325 186 0 0 100
 0 0 0 1955568 1260288 0 40 0 0 0 0 0 0 0 0 0 834 325 186 0 0 100
 0 0 0 1955568 1260288 0 40 0 0 0 0 0 0 0 0 0 834 325 186 0 0 100
 0 0 0 1955568 1260288 0 40 0 0 0 0 0 0 0 0 0 834 325 186 0 0 100
```

The `vmstat` utility takes two parameters, time interval and count, both integers. Time interval specifies the interval `vmstat` refreshes, and count specifies the maximum times it will refresh. If no parameters are specified, `vmstat` only lists previous statistics about the system and won't refresh. In this case, the statistics it provides are not accurate and up to date. If interval is specified, `vmstat` will summarize system activities over the last interval in seconds repeatedly. If a count is given, `vmstat` will pull system statistics repeatedly until it reaches the number of count times specified.

What most interests us in the output are "r", "po" and "id" fields.

- The "r" field tells us how many jobs are in the system ready queue, waiting for resources to be executed.
- The "po" field tells us how many pages in the memory are currently paged out. If this number is extremely large and continues to increase, it is usually an indication of insufficient physical memory or RAM, and you may need to install more memory.
- The "id" field tells us how much system CPU resources are being used.

These fields together will give you a pretty good idea of how the system resources are currently used.
Informix tools

The most comprehensive tool that Informix provides for collecting detailed SQL query plans and execution statistics is the SET EXPLAIN utility. This utility will generate a file called sqexplain.out, and records in detail every step of the query execution. In addition it provides the estimated costs of the query and estimates the query results. By examining the SET EXPLAIN output file, you can determine if steps can be taken to improve the performance of the query. The following example shows the set explain output for a pretty complex query:

QUERY:
------
SELECT --+AVOID_FULL(omchn)+AVOID_FULL(daphn)

    omchn.omc_hn_uanc,
    nvl(daphn.gtt_version,"0000000000000000000"),
    nvl(idachn.egt4_version,"0000000000000000000"),
    nvl(ihlrhn.hlr_timestamp,"00000000000000"),
    vsgw_hn.hn_igw_uanc,
    nvl(vsgw_hn.hn_igw_version, "00000000000000")
FROM omchn, daphn, idachn, ihlrhn, vsgw_hn
WHERE daphn.dap_hn_inst  = omchn.omc_hn_inst
AND idachn.idac_hn_inst = omchn.omc_hn_inst
AND ihlrhn.hlr_hn_inst = omchn.omc_hn_inst
AND vsgw_hn.vsgw_hn_inst = omchn.omc_hn_inst

DIRECTIVES FOLLOWED:
AVOID_FULL ( omchn )
AVOID_FULL ( daphn )

DIRECTIVES NOT FOLLOWED:

Estimated Cost: 8
Estimated # of Rows Returned: 1

1) root.idachn: SEQUENTIAL SCAN

2) root.daphn: INDEX PATH
   (1) Index Keys: dap_hn_inst   (Serial, fragments: ALL)
       Lower Index Filter: root.daphn.dap_hn_inst = root.idachn.idac_hn_inst
   NESTED LOOP JOIN

3) root.vsgw_hn: SEQUENTIAL SCAN
   NESTED LOOP JOIN

4) root.omchn: INDEX PATH
   Filters: root.vsgw_hn.vsgw_hn_inst = root.omchn.omc_hn_inst
   (1) Index Keys: omc_hn_inst   (Serial, fragments: ALL)
       Lower Index Filter: root.idachn.idac_hn_inst = root.omchn.omc_hn_inst
   NESTED LOOP JOIN

5) root.ihlrhn: INDEX PATH
   (1) Index Keys: hlr_hn_inst   (Serial, fragments: ALL)
       Lower Index Filter: root.ihlrhn.hlr_hn_inst = root.omchn.omc_hn_inst
   NESTED LOOP JOIN

The above output may be divided into three parts:

- The first part shows the query syntax.
- The second part shows the estimated costs of the query.
- The third part is the detailed explanation of every step the query executed.
What we are most interested in are the second and third parts. The estimated costs are cost units that the optimizer uses to compare query plans. These units do not translate directly to time; they represent relative time for a typical disk access.

The optimizer chose this query plan because the estimated cost for its execution was the lowest among all the evaluated plans. A query with higher estimated costs generally takes longer to run than one with lower estimated costs. The third part is of vital importance in query tuning, because it provides us with a great deal of useful information, such as the data access method and the join method that the query used. The above example shows that sequential scan and indexes are used to do the data retrieval, and the nested-loop join method is used to join all tables. I will discuss this in detail later in this article.

This utility is easy to use. If you want to know the detailed query execution plan for a certain SQL query, just add SET EXPLAIN ON statement before your original query as follows:

```
set explain on;
select count(*) from acg;
```

Informix server will then generate a file named sqexplain.out in your user home directory which will record the detailed query execution plan and its costs as we talked about above. This file is cumulative, in other words, if you have multiple SQL queries after SET EXPLAIN ON statement, each query execution plan and its costs would be appended to this file until it is removed. As for the stored procedures, you need to execute UPDATE STATISTICS for the original stored procedure to get the detailed execution plan, since stored procedures can only update their query execution plan when update statistics. For example, if you want to see the detailed execution plan for a stored procedure dap_int, you need to do the following:

```
set explain on;
update statistics for stored procedure dap_int();
```

From version 9.3 on, there is a nice enhancement of this utility; you can get the detailed query execution plan without ever executing the query. This makes it possible to get query execution plan in the real production environment. To use this new feature, you need to use the key word AVOID_EXECUTE in SET EXPLAIN ON statement as follows:

```
set explain on avoid_execute;
select count(*) from act;
```

For detailed information how to use this utility, refer to IBM Informix Dynamic Server product documentation.

**General methodology**

How do we then apply those tools in tuning SQL queries? Different people may have different approaches, but in general, you should follow the methodology and steps outlined below:
1. Collect statistics on your original SQL query. At this step, you need to use the tools discussed above to get statistics on the query: its response time, its detailed execution plan, and costs for later in-depth analysis.

2. Analyze statistics. At this step, you need to dig into the statistics you collected above and take a good look at the query execution plan. As stated above, performance is the major concern in query tuning. You need to consider all factors that would impact performance when examining the query plan: the access method, the join method, subqueries, table and index extents, table and index fragmentation and so on. I will discuss each of these factors in detail later in the part two of this article.

3. Set up the test environment. This is a very important step. The test environment should be set up exactly the same or very similar to the production environment where the query runs in terms of hardware and software configurations. For example, if the production machine has 6 400HM CPUs, the test machine should also have 6 400HM CPUs; otherwise your subsequent tests will be invalid and not reliable. Bear in mind, all those queries will finally be run in production.

4. Make changes and test your new query. This is a major step and also can be the most tedious step in tuning. Make changes to your original query one at a time, and test to see if the performance has improved (decreased response time). Record details of your test, such as changes you made, response time, and execution plan. If after making the change, the performance of your query is not better than your original query, reverse the change. Your tests must be valid and reliable; in other words, your tests must be repeatable. For instance, if you did two identical tests on the same query, and the first time, it yields a very good response time (say 10 seconds), but the second time, the response time increased to 30 seconds, than your tests are not repeatable since the difference of the response time is too great. You need to re-examine your test procedure and identify the differences between those two tests. If your tests are repeatable, the differences between test results should be minimal.

5. Analyze test results. When analyzing test results, we need to examine the validity and reliability of the test results. We need to examine the hardware, software, the load and all other factors to ensure our test results are valid and reliable.

6. Implement your improvements in the production system. Before you implement, you need to perform the last detailed review and make sure the new queries won't cause any problems in production.

### Optimizer and its role

Like other relational database management systems such as Oracle and SQL Server, Informix has its internal optimizer, which is responsible for selecting the best query execution plan. After SQL queries are parsed, the optimizer will consider all possible ways to execute the query by analyzing factors such as disk I/O and CPU costs. It will then construct all feasible plans simultaneously using a bottom up, breath-first search strategy.

In other words, the optimizer will construct all possible join plans first, and then eliminate the more expensive of any redundant pairs, which are join pairs that contain the same tables and produce the same set of rows as another join pair. If a query uses additional tables, the optimizer joins each remaining pair to a new table to form all possible join triplets, eliminating the more expensive of
redundant triplets, and so on for each additional table to be joined. When a non-redundant set of possible join combinations has been generated, the optimizer selects the plan that appears to have the lowest execution costs. For example, the optimizer must determine whether indexes should be used. If the query includes a join, the optimizer must determine the join plan (hash, sort merge, or nested loop), and the order in which tables are evaluated or joined.

The optimizer bases query cost estimates on the number of rows to be retrieved from each table. In turn, the estimated number of rows is based on the selectivity of each conditional expression that is used within the WHERE clause. The optimizer uses data distribution information generated by UPDATE STATISTICS to calculate selectivity for the filters in the query. However, in the absence of data distribution information, the optimizer will calculate selectivity for filters of different types based on table indexes. For example, if the indexed column has literal values and NULL values, then the selectivity is equal to the number of distinct keys in the index. Refer to Chapter 10 of the Performance Guide for the detailed table that the optimizer uses to calculate selectivity in the absence of data distribution. But the selectivity calculated in this way is not as accurate as selectivity calculated using data distribution.

So it is obvious that the accuracy of selectivity estimation depends upon how often you run UPDATE STATISTICS. If you run UPDATE STATISTICS frequently, the optimizer will calculate selectivity more accurately, since the data distribution will be updated every time you run UPDATE STATISTICS, except when you run UPDATE STATISTIC with low option.

The optimizer uses the following system catalog information as it creates a query plan:

- The number of rows in a table, as of the most recent UPDATE STATISTICS statement
- Whether a column is constrained to be unique
- The distribution of column values, when requested with the MEDIUM or HIGH keyword in the UPDATE STATISTICS statement
- The number of disk pages that contain row data
- The indexes that exist on a table, including the columns that they index, whether they are ascending or descending, and whether they are clustered
- The depth of the index structure (a measure of the amount of work that is needed to perform an index lookup)
- The number of disk pages that index entries occupy
- The number of unique entries in an index, which can be used to estimate the number of rows that an equality filter returns
- Second-largest and second-smallest key values in an indexed column.

The behavior of the optimizer is influenced by three key factors: the value of OPTCOMPIND parameter in the Informix configuration file, query directives, and the accuracy of internal statistics.

**OPTCOMPIND parameter**

OPTCOMPIND is an environment variable or a parameter in the Informix configuration file. The optimizer uses its value to determine its choice of the data access method. It has one of three values (0, 1 and 2), which indicate the following:
• If the value is set to 0, then the optimizer chooses index scans over table scans if appropriate indexes exist, without even considering the estimated costs.
• If the value is set to 1, then the optimizer behaves as it does for value 0 if the transaction isolation mode is not Repeatable Read. If the transaction isolation mode is Repeatable Read, then the optimizer would base its choice purely on the estimated costs.
• If value is set to 2, the optimizer would uses estimated costs to determine an execution plan regardless of the transaction isolation mode.

You may either set OPTCOMPIND either as an environment variable or as a parameter in the configuration file, but setting it as a parameter will take precedence on execution.

Query directives

Another way to influence the optimizer is to use query directives. Query directives are hints in SQL queries that instruct the optimizer how to execute the query. There are four types of query directives as follows:

• Access plan directives that force the optimizer to use the designated access method for data retrieval, either sequential scan or index scan.
• Join order directives that force optimizer to join tables in the designated order.
• Join plan directives that force optimizer to use the designated join method to join tables in the query, either nested loop join, sort merge join, or dynamic hash join.
• Goal directives that force the optimizer to use the designated rules to return query results.

For detailed information how to use those directives, refer to Chapter 11 of IBM Informix Dynamic Server product documentation.

Internal statistics

By internal statistics, I mean the statistics in the system catalogs that the optimizer uses to determine the lowest-cost query execution plan. To ensure that the optimizer selects the best query plan, it is important to keep internal statistics current and accurate. The database server initializes a statistical profile of the database objects such as tables, indexes, stored procedures and triggers, and puts data distribution into system catalogs when the database tables are created, but does not update those statistics automatically.

To keep this statistic profile up to date, you need to run UPDATE STATISTICS regularly; otherwise, your system’s statistical profile may not reflect the current status of your system, and the optimizer may not be able to make right choice among numerous query execution plans. There are three modes for running UPDATE STATISTICS, and in general you need to run UPDATE STATISTICS after each time you perform big batch jobs that change large amounts of table data and each time after you add an index to a table. For detailed information how to perform UPDATE STATISTICS, refer to IBM Informix Dynamic Server documentation. The rule of thumb is that more often you run UPDATE STATISTICS, the more up to date and accurate your system’s statistical profile will be, and the greater the chance that the optimizer will make the best choice for a query execution plan.
Although the behavior of the optimizer is influenced by OPTCOMPIND and directives as we talked above, the optimizer generally chooses a query plan based on the following guidelines:

- The optimizer won't use an index if the query retrieves a large amount of data from a table. For example, if customers in your company are pretty much evenly spread among all 50 States, and you want to retrieve customer information for every state except for New York, and you execute the following query:

  ```sql
  SELECT * FROM customer WHERE STATE <> "NEW YORK";
  ```

  The optimizer will soon detect that you may retrieve 98 percent of the data in the table, and it thinks it is more efficient to read or scan the table sequentially rather than to traverse an index (and subsequently the data pages), and then to retrieve the related data.

- If multiple indexes are defined on the table, the optimizer uses the index that can rule out the most data in the table. For example, if your company has 200,000 customers in New York and only about 1000 customers place orders on the same day, say January 20, 1997, and you issue the following query to get their names and addresses:

  ```sql
  SELECT name, address FROM customer
  WHERE state = "NEW YORK" AND order_date = "01/20/97"
  ```

  The optimizer will most likely choose to use the index on order_date rather than the index on state.

- If there are no directives in the query, the optimizer will always retrieve data from tables with the most restrictive filter first. Let's look at the following query:

  ```sql
  SELECT * FROM customer, orders
  WHERE customer.customer_num = orders.customer_num
  AND customer.state = "NEVADA"
  ```

  In this example, the first thing the optimizer does is to evaluate the condition that state equals NEVADA, because that will rule out a large number of data rows in the table. It will then join two tables. The idea is to reduce as much as possible the load for the database server. If the optimizer joins two tables first, the join results could be huge and may use a large amount of system resources such as CPU and memory. If you have 1,000,000 active customers and on average, every one of them makes an order every month, then the join results will return at least one million records, and this will certainly hurt your system performance.

- The optimizer will choose dynamic hash join if none of the joined columns has index. In the previous example, if customer.customer_num and orders.customer_num are not indexed, dynamic hash join would be chosen by the optimizer as the best execution plan.

- The optimizer will choose nested-loop joins if:
  - The number of rows retrieved from the outer table after the database server applies any table filters is small, and the inner table has an index that can be used to perform the join.
  - The index on the outermost table can be used to return rows in the order of the ORDER BY clause, eliminating the need for a sort.
Conclusion

In part 1 of this article I've discussed the criteria for tuning, talked about the tools you'll need, and introduced a basic methodology for the tuning process. Part 2 will take you deeper into the process. We'll look at the queries themselves, and examine some examples.
Related topics

- IBM Informix Dynamic Server general product documentation.
- IBM Informix product family library.

© Copyright IBM Corporation 2004

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)