solarwinds

SQL QUERY TUNING FOR ORACLE

Getting It Right the First Time

SOLARWINDS
WHITEPAPER

INTRODUCTION

As part of my job as a Senior DBA, I get to review Oracle database performance data with hundreds of customers a year. During the review process I provide performance improvement recommendations based on the response time data from ***SolarWinds Database Performance Analyzer (DPA).*** *However, I also try to go above and beyond the raw data to provide valuable performance tuning tips for our customers. Over the years we have developed a process that works time and time again. This process is the focus of this white paper and follows four fundamental steps:*

1. *Focus on the correct SQL statements*
2. *Utilize response time analysis*
3. *Gather accurate execution plans*
4. *Use SQL diagramming*

WHY FOCUS ON SQL STATEMENTS?

When I think about performance tuning for a database environment, the following three types of tuning approaches come to mind:

- Application Tuning—tune the application code to process data more efficiently.
- Instance Tuning—tune Oracle itself via modification of database parameters or altering the environment in which the database executes.
- SQL Statement Tuning—tune the SQL statements used to retrieve data.

The third approach, SQL tuning, seems to be a point of contention with many of our customers because it is often unclear which group (database administration or development) is responsible. This is also the area where I tend to focus my efforts for reasons discussed throughout this paper.

I am often asked why I focus on SQL statement tuning rather than instance or application tuning. Instance and application tuning are definitely beneficial in the right circumstances; however, I typically find that SQL tuning provides the most "bang for the buck" because it is often the underlying performance issue. My experience is that approximately 75-85% of the performance problems were solved using SQL tuning techniques.

Most applications accessing databases on the backend require simple manipulation of data. There are typically no complex formulas or algorithms that require significant application time and thus tuning. These applications also deal with smaller amounts of data so even if the processing of that data is inefficient, it does not become a significant portion of the total waiting time for the end user. For example, a web application that displays the status of an order may only manipulate a few rows of data.

On the flip side, the database is responsible for examining large amounts of data to retrieve the status of that order. There may be several tables involved containing millions of rows of data each and inefficiencies can quickly become huge bottlenecks. Tuning the underlying query in this case typically provides the most performance benefit rather than focusing on the application code.

WHICH SQL STATEMENT SHOULD BE TUNED?

If SQL statement tuning can provide the most benefit, the next question is "Which SQL statement should I focus on"? Often the DBA or developer did a great job of tuning a SQL statement, but later discovered it was not the root cause of the performance problem the end users were complaining about. Tuning the wrong SQL statement is clearly a waste of time, so what is the best way to know which SQL to tune? Choosing the correct SQL statement can involve looking for SQL statements that:

- Perform the most logical I/O
- Consume the most CPU
- Perform costly full table or index range scans
- Use an execution plan with a cost over X
- And many more…

With some of the strategies from above, what if the underlying problem was a blocking issue? The problematic queries may not appear in any of these lists and you would miss them. How do you know which of these are the queries causing your performance issues? The answer lies in measuring total elapsed times rather than using the above measurements, i.e. which SQL statements spend the most time executing in the database. A query similar to the following will retrieve a list from Oracle with the queries taking the longest to execute:

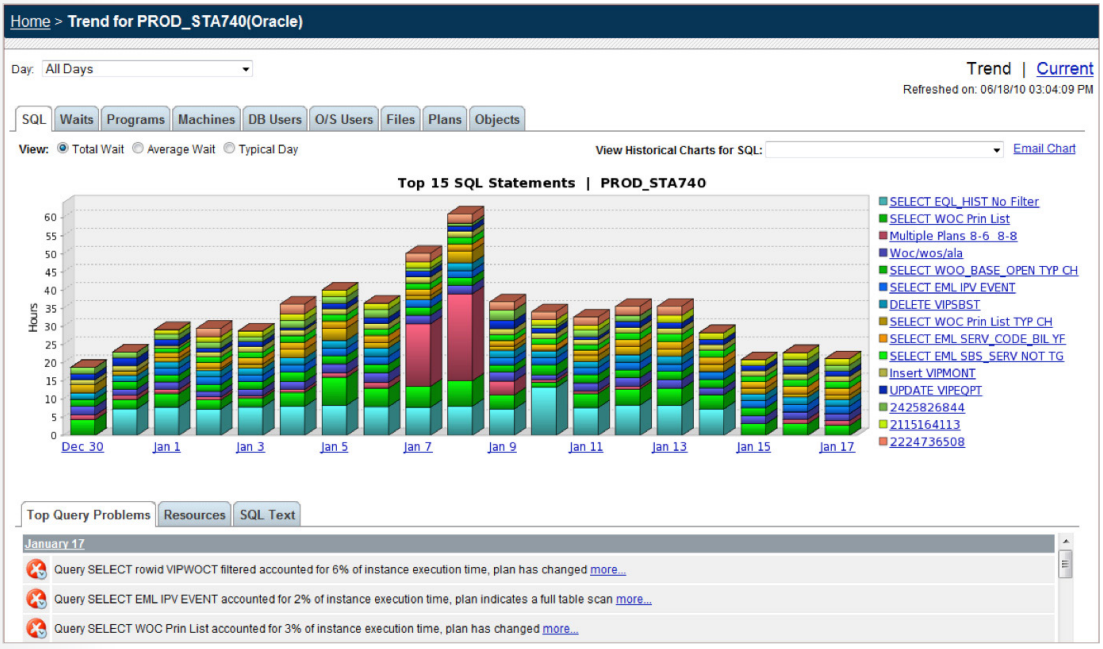
```
select sql_text, elapsed_time, sql_id, hash_value,
plan_hash_value, program_id, module, action, …
from v$sql
where module = <module having problems>
and username = <username>
and…
order by elapsed_time
```

When end-users complain about performance they will say something similar to: "When I click the submit button on this web page, it takes 30-40 seconds." They use elapsed time as a way to describe the problem, so why not use elapsed time when finding the queries to tune.

The query above will provide a definitive list of SQL statements with the highest elapsed time since instance startup. Often this is based on a timeframe of several months or even longer, so collecting this data periodically with deltas will provide the best information. For example, running a query similar to the above (along with other pertinent data based on your application) every

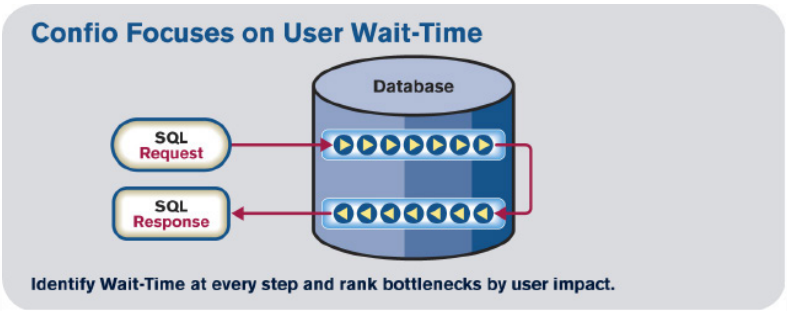
minute and saving the data will allow you to go back in time to find a problem. If an end-user complains about performance from yesterday at 3:00 p.m., reviewing the archived data from a timeframe around 3:00 yesterday will help you understand what was performing poorly during the problematic time.

A much easier method for collecting this type of data and finding the problematic SQL statement is to use a response time based performance analysis tool like DPA. DPA collects response time data once per second about all SQL statements and can quickly rank them according to the ones causing the most response time. The screenshot below shows the Top 15 SQL statements in an Oracle database from a total elapsed time perspective.



DATABASE RESPONSE TIME VIEW OF PERFORMANCE

Once the most problematic SQL is known, understanding why the SQL is slow is paramount. Is the problem an unnecessary full table scan, a locking issue, disk I/O related, etc? Oracle wait events very clearly show the problem.



The picture above depicts a SQL statement entering the Oracle database and going through a series of steps. These steps may include reading data from disk, waiting on a locking issue, waiting for Oracle to commit data, and many more. Measuring the total execution time for a SQL statement reveals which SQL to tune and measuring how much time that SQL waits on specific events provides the best clues for fixing the issue. For example, if a SQL statement executes for 40 seconds, what is the major cause of that delay? A SQL that waits on an event named "db file scattered read" means it is most likely performing a full table scan, while waits on "enq: TX – row lock contention" indicate the SQL is experiencing a locking issue. The solutions for these problems are much different and are where wait events help the most.

The following table shows the most common wait events (based on DPA data summarized from hundreds of customers) along with ideas for tuning SQL statements waiting on these events.

WAIT EVENT / SUGGESTIONS	WAIT CLASS	PCT
db file sequential read • tune indexes • tune disks • increase buffer cache	User I/O	28%
db file scattered read • add indexes • tune disks • refresh statistics • create materialized view	User I/O	27%

direct path read / direct path read temp
• review P1 (file id), P2 (starting dba), P3 (blocks)
• if reading temporary data
  • increase pga_aggregate_target (workarea_size_policy=AUTO)
  • increase sort_area_size (workarea_size_policy<>AUTO)
  • cache temporary datafiles at O/S level
• if reading application data
  • could be related to parallel query
	User I/O	10%

global cache cr request • RAC event similar to buffer busy waits • tune SQL to request less data • tune network latency between RAC nodes • localize data access	Cluster	5%
buffer busy waits / read by other session • tune SQL (often see this event with full table scans) • review P1 (file id), P2 (block id) for hot blocks • if the SQL is inserting data, consider increasing FREELISTS and/or INITRANS • if the waits are on segment header blocks, consider increasing extent sizes	Concurrency	5%

SQL*Net more data from dblink • may not be a problem • reduce amount of data transferred across dblink • tune network between databases	Network	4%
log file sync • tune applications to commit less often • tune disks where redo logs exist • try using nologging / unrecoverable options • log buffer could be too large	Commit	3%
direct path write / direct path write temp • review P1 (file id), P2 (starting dba), P3 (blocks) • similar approaches as "direct path read" • could be related to direct path loads	User I/O	3%

LOOKING AT THE EXECUTION PLAN

Execution plans further the understanding of where a SQL statement can be made more efficient. If there are five tables involved and the query waits mostly on "db file scattered read" indicating a full table scan, the plan will help determine where that is occurring. Plans supply costing information, data access paths, join operations and many other things.

However, not all plans are useful. Do you know that EXPLAIN PLAN can be wrong and not match how Oracle is really executing the statement? It can be wrong because the EXPLAIN PLAN command is typically executed in a completely different environment like SQL*Plus and not from the application code and environment. The application may set session variables that are not set from SQL*Plus, bind variable data types may be defined differently, and many other things. If EXPLAIN PLAN can be wrong, where is a better place to retrieve the correct plan? The best places to get execution plan information are:

V$SQL_PLAN – contains raw data for execution plans of SQL statements. It provides the plan Oracle used so why not go straight to the source. Use the DBMS_XPLAN system package to get the execution plan in a readable format.

Tracing – provides great information as well as executions plans for an entire session or process.

Historical Data – if possible, collect and save execution plan information so you can go back to 3:00 pm yesterday to understand why the SQL statement performed poorly.

Once you have an accurate execution plan, determine how each of the SQL components is being executed. Based on response time data, you should already have a feel if full table scans are significant, whether more time is spent reading indexes or if you have other issues. Execution plans help determine more about those problems.

NOT ALL PLANS CREATED EQUAL

Here is an example where, based on response time data, we believed the query was doing a full table scan since the SQL was spending over 95% of its execution time waiting for "db file scattered read". However, when we reviewed the EXPLAIN PLAN output, the query looked very efficient and seemed to be using a unique index to retrieve one row of data. Here is the statement and the EXPLAIN PLAN information:

```
SELECT company, attribute FROM data_out WHERE segment = :B1;
Data from EXPLAIN PLAN Command

SELECT STATEMENT Optimizer=ALL ROWS (cost=1 card=1 bytes=117)
TABLE ACCESS (BY INDEX ROWID) OF 'DATA_OUT' (TABLE)
       (Cost=1 Card=1 Bytes=117)
INDEX (UNIQUE SCAN) OF 'IX1_DATA_OUT' (INDEX(UNIQUE))
       (Cost=1 Card=1)
```

How is this possible? It is a very simple query with one criterion in the WHERE clause, so if it is executing very efficiently, why are the end-users waiting 20 seconds for this query each time it executes? The answer is revealed when we review the execution plan from V$SQL_PLAN using DBMS_XPLAN:

```
select * from table(dbms_xplan.display_cursor('az7r9s3wpqg7n',0));

+-------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |       |       |   370 (100)|          |
|*  1 |  TABLE ACCESS FULL| DATA_OUT |     1 |   117 |   370   (4)| 00:00:05 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(TO_BINARY_DOUBLE("SEGMENT")=:B1)
```

In this example, we use the DISPLAY_CURSOR function from the DBMS_XPLAN package and pass it the SQL_ID and CHILD_NUMBER as parameters.

If we were only reviewing EXPLAIN PLAN data, we may have dismissed the issue and referred the problem back to the development team claiming the problem lies in the application tier. The execution plan from V$SQL_PLAN above quickly points to the full table scan on the DATA_OUT table and the predicate information reveals the extent of the problem. The :B1 bind variable is being passed as a BINARY DOUBLE number from Java code. The segment column is defined as a standard NUMBER data type so Oracle is forced to implicitly convert the data in the table to BINARY DOUBLE to apply the criteria. Anytime a function is used around a column, even if it is an implicit function, standard indexes on that column will not be used. You now have a plan of action:

1. **Change Application** – show the developers the problem and ask them to modify the Java code to pass in a standard number.

2. **Modify Table Definition** - redefine the column in the DATA_OUT column to be a BINARY DOUBLE if needed. Oracle will not be required to implicitly convert the data passed in from the application.

3. **Create Function-Based Index** – this will immediately help performance while the development team determines the most effective method for code modification.

Now that we've covered the theory behind this simple tuning process, I'd like to provide some practical examples to illustrate the value of response time analysis in your SQL tuning process.

TUNING EXERCISE #1

The first sample SQL statement answers the question: "Who registered for the SQL Tuning class within the last day?":

```
SELECT s.fname, s.lname, r.signup_date
FROM student s
INNER JOIN registration r ON s.student_id = r.student_id
INNER JOIN class c ON r.class_id = c.class_id
WHERE c.name = 'SQL TUNING'
AND r.signup_date BETWEEN TRUNC(SYSDATE-1) AND TRUNC(SYSDATE)
AND r.cancelled = 'N'
```

REVIEW EXECUTION PLAN AND QUERY STATISTICS

The execution plan was retrieved from V$SQL_PLAN for our sample SQL statement and is given below:

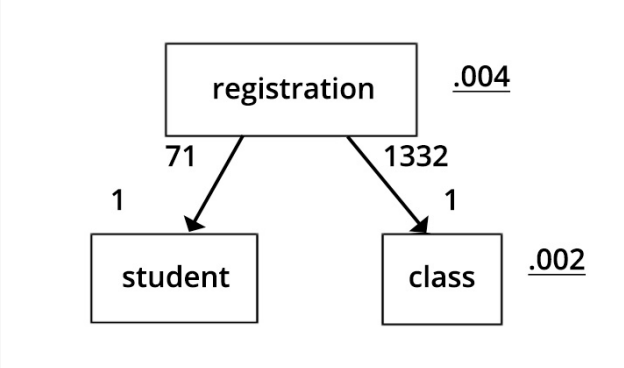
```
-------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |              |     7 |   819 |  3173   (5)| 00:00:39 |
|*  1 |  FILTER                      |              |       |       |            |          |
|   2 |   NESTED LOOPS               |              |       |       |            |          |
|   3 |    NESTED LOOPS              |              |     7 |   819 |  3173   (5)| 00:00:39 |
|*  4 |     HASH JOIN                |              |     7 |   448 |  3166   (5)| 00:00:38 |
|*  5 |      TABLE ACCESS FULL       | CLASS        |     1 |    45 |    22   (0)| 00:00:01 |
|*  6 |      TABLE ACCESS FULL       | REGISTRATION |  8334 |   154K|  3144   (5)| 00:00:38 |
|*  7 |     INDEX UNIQUE SCAN        | PK_STUDENT   |     1 |       |     0   (0)| 00:00:01 |
|   8 |    TABLE ACCESS BY INDEX ROWID| STUDENT      |     1 |    53 |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(TRUNC(SYSDATE@!-1)<=TRUNC(SYSDATE@!))
   4 - access("R"."CLASS_ID"="C"."CLASS_ID")
   5 - filter("C"."NAME"='SQL TUNING')
   6 - filter("R"."SIGNUP_DATE">=TRUNC(SYSDATE@!-1) AND "R"."CANCELLED"='N' AND
              "R"."SIGNUP_DATE"<=TRUNC(SYSDATE@!))
   7 - access("S"."STUDENT_ID"="R"."STUDENT_ID")
```

Learning to read the execution plan is outside the scope of this white paper, but in simple terms, start from the innermost lines to understand what Oracle does first. In this case, two full table scans are performance against the CLASS and REGISTRATION tables.

Another important piece of data about the query execution is the number of logical reads, also known as buffer gets. This query is performing 11,239 logical reads. The logical reads metric provides a very good understanding of the amount of work done by Oracle to execute the query. Why not use simple execution times to measure a SQL statement? The problem with using time is that it is greatly influenced by other processes running while you are testing the query. This can significantly impact the time a query takes to execute and thus give false impressions about the performance gains. Logical reads will not be affected in this way.

CREATE THE SQL DIAGRAM

I would like to introduce a technique that I use to tune SQL statements correctly without the trial and error that I was often faced with before. The concept of SQL diagramming was introduced to me through a book entitled SQL Diagramming by Dan Tow. For our purposes, I will explain the basics of the following SQL diagram:



The diagram looks similar to an ERD diagram. To build it:

- Start with any table in the FROM clause and put it on paper (the easiest way to draw these diagrams is by hand). In my case I started with the first table in the FROM clause named STUDENT.
- Take the next table, which in my case is REGISTRATION, and place it either above or below the existing tables based on the relationship. Since the REGISTRATION relates to the STUDENT table uniquely, i.e. one row in REGISTRATION points to one row in the STUDENT table, I put it above and draw an arrow downwards.
- Take the next table, CLASS - one row in REGISTRATION points to one row in CLASS so I put it below REGISTRATION with another downward pointing arrow.

The next step is to further understand the criteria used in the query to limit the number of rows from each of the tables. The first criteria to explore is anything in the WHERE clause to limit the rows from REGISTRATION. This criteria is:

```
AND r.signup_date BETWEEN TRUNC(SYSDATE-1) AND TRUNC(SYSDATE)
AND r.cancelled = 'N'
```

To understand the selectivity, run a query similar to below against the table using the critiera:

```
select count(*) from registration r
WHERE r.signup_date BETWEEN TRUNC(SYSDATE-1) AND TRUNC(SYSDATE)
AND r.cancelled = 'N'
```

Results – 8,221 / 1,687,980 (total rows in REGISTRATION) = 0.004 = 0.4% selectivity

Use this number on the diagram next to the REGISTRATION table and underline it. This represents the selectivity of the criteria against that table.

The next table to review in the same manner is the CLASS table. The query I would run becomes:

```
SELECT count(1) FROM class
WHERE name = 'SQL TUNING'
```

Results – 2 / 1,267 (total rows in CLASS) = 0.001 = 0.1% selectivity

Add this to the diagram next to the CLASS table. The next table to explore is the STUDENT table, but there are no direct criteria against this table, so no underlined number next to that table.

The other numbers next to the arrows in the diagram are simple calculations based on average numbers of rows that one row in a table references in the others. In this case, the REGISTRATION table refers to exactly one row in the CLASS and STUDENT tables so the number 1 goes next to the bottom of the arrows. One row in the CLASS table refers to on average 1,332 rows in the REGISTRATION table (1,687,980 total rows REGISTRATION divided by 1,276 rows in CLASS) and one row in the STUDENT table refers to on average 71 rows in REGISTRATION (1,687,980 / 23,768).

The diagram has now been completed for our example, however, Dan Tow goes into more detail than I will include at this point.

ANALYZE THE SQL DIAGRAM

Analyzing the SQL diagram begins by looking for the smallest, underlined number. In this example it is 0.001 next to the CLASS table. To limit the number of rows the query needs to process, starting here will trim our result sets the soonest. If we can make Oracle start with this table the query can be executed optimally. When the query starts with the CLASS table for this query, it will read two rows and then visit the REGISTRATION table and read 2,664 rows (2 * 1,332). From

there it will read 2,664 more rows from the CLASS table for a total of 5,330 rows (2 + 2,664 + 2,664). If the query started on the REGISTRATION table first, it would read 8,221 rows and then 8,221 rows from the CLASS and STUDENT tables for a total of 24,663 rows (3 * 8,221). Based on this data we can understand why starting on the CLASS table is best for this query.

TUNING BASED ON SQL DIAGRAMS

We know we want Oracle to start with the CLASS table, but how do we do that? The easiest way is to ensure an index exists and fits our criteria of "name = 'SQL TUNING". In this case that means the NAME column of the CLASS table.

```
create index cl_name on class(name)
```

The new execution plan is:

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |              |     7 |   819 |  3153   (5)| 00:00:38 |
|*  1 |  FILTER                        |              |       |       |            |          |
|   2 |   NESTED LOOPS                 |              |       |       |            |          |
|   3 |    NESTED LOOPS                |              |     7 |   819 |  3153   (5)| 00:00:38 |
|*  4 |     HASH JOIN                  |              |     7 |   448 |  3146   (5)| 00:00:38 |
|   5 |      TABLE ACCESS BY INDEX ROWID| CLASS        |     1 |    45 |     2   (0)| 00:00:01 |
|*  6 |       INDEX RANGE SCAN         | CL_NAME      |     1 |       |     1   (0)| 00:00:01 |
|*  7 |      TABLE ACCESS FULL         | REGISTRATION |  8334 |  154K|  3144   (5)| 00:00:38 |
|*  8 |     INDEX UNIQUE SCAN          | PK_STUDENT   |     1 |       |     0   (0)| 00:00:01 |
|   9 |    TABLE ACCESS BY INDEX ROWID | STUDENT      |     1 |    53 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
```

The number of logical reads was reduced from 11,239 to 11,168 which represent a gain by using an index to retrieve data from the CLASS table versus a table scan. There is some improvement, but not as much as we had hoped. The CL_NAME index is now used to find the two rows in the CLASS table with a NAME='SQL TUNING'. We achieved partial results but a full table scan is still done against the REGISTRATION table. Why would that be?

When we review the indexes on REGISTRATION we find one index named REG_PK that contains the STUDENT_ID and CLASS_ID columns in that order. Since our query is hitting the CLASS table first it cannot use this index that starts with STUDENT_ID so it must perform a full table scan. This index could be modified to switch the columns around, but that could affect many other queries and may require significant testing. In this case, since I know I need an index with a leading edge of CLASS_ID, I created an index on just that column:

```
create index reg_alt on registration (class_id)
```

The new execution plan is:

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                      | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |              |     7 |   819 |  2415   (1)| 00:00:29 |
|*  1 |  FILTER                        |              |       |       |            |          |
|   2 |   NESTED LOOPS                 |              |       |       |            |          |
|   3 |    NESTED LOOPS                |              |     7 |   819 |  2415   (1)| 00:00:29 |
|   4 |     NESTED LOOPS               |              |     7 |   448 |  2408   (1)| 00:00:29 |
|   5 |      TABLE ACCESS BY INDEX ROWID| CLASS        |     1 |    45 |     2   (0)| 00:00:01 |
|*  6 |       INDEX RANGE SCAN         | CL_NAME      |     1 |       |     1   (0)| 00:00:01 |
|*  7 |      TABLE ACCESS BY INDEX ROWID| REGISTRATION |     7 |   133 |  2406   (1)| 00:00:29 |
|*  8 |       INDEX RANGE SCAN         | REG_ALT      |  2667 |       |     9   (0)| 00:00:01 |
|*  9 |     INDEX UNIQUE SCAN          | PK_STUDENT   |     1 |       |     0   (0)| 00:00:01 |
|  10 |    TABLE ACCESS BY INDEX ROWID | STUDENT      |     1 |    53 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
```

This query now performs 4,831 logical reads so we have made significant progress. Before utilizing the SQL diagramming technique I used to focus on the execution plan and attempt to reduce the cost of expensive steps in the plan. Based on the initial execution plan, I would focus on the REGISTRATION table. However, as we have seen, Oracle would have been required to read through more than four times the number of rows to get the same results.

TUNING EXERCISE #2

The second sample SQL statement we will tune provides a list of open orders for a customer.

```
SELECT o.OrderID, c.LastName, p.ProductID, p.Description,
      sd.ActualShipDate, sd.ShipStatus, sd.ExpectedShipDate
FROM Orders o
INNER JOIN Item i ON i.OrderID = o.OrderID
INNER JOIN Customer c ON c.CustomerID = o.CustomerID
INNER JOIN ShipmentDetails sd ON sd.ShipmentID = i.ShipmentID
INNER JOIN Product p ON p.ProductID = i.ProductID
INNER JOIN Address a ON a.AddressID = sd.AddressID
WHERE c.LastName LIKE NVL(:1,'') || '%'
AND c.FirstName LIKE NVL(:2,'') || '%'
AND o.OrderDate >= SYSDATE - 30
AND o.OrderStatus <> 'C'
```

The first question that needs to be asked is whether the bind variables :1 and :2 are always provided. In this case, both are optional but at least one of them is required by the application. Therefore, the first step is to rewrite the query. If only one criteria is provided, the query would perform a full table scan on CUSTOMER every time. For example, if the LASTNAME value was provided but FIRSTNAME was not, the criteria would turn into:

```
WHERE c.LastName LIKE 'SMI%'
AND c.FirstName LIKE '%'
```

The FIRSTNAME criteria would cause Oracle to perform a full table scan to retrieve the results. A possible solution is to rewrite the query to be constructed dynamically and include only the criteria in the query where the bind variable has a value. It's important to remember that tuning a SQL statement is not always about technical details.

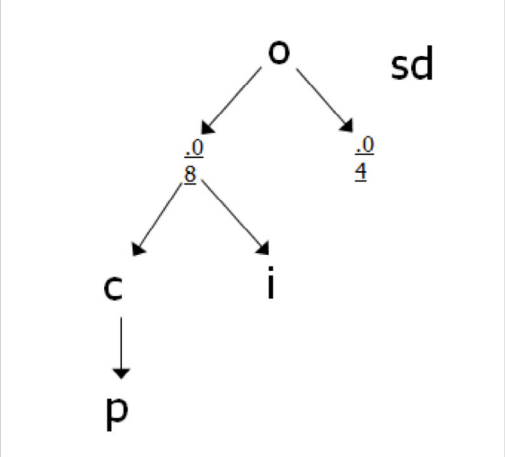
REVIEW EXECUTION PLAN AND QUERY STATISTICS

The execution plan was retrieved from V$SQL_PLAN for our sample SQL statement and is given below:

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                      | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                  |     6 |  1164 |  4892   (2)| 00:00:59 |
|   1 |  NESTED LOOPS                  |                  |       |       |            |          |
|   2 |   NESTED LOOPS                 |                  |     6 |  1164 |  4892   (2)| 00:00:59 |
|   3 |    NESTED LOOPS                |                  |     6 |   510 |  4886   (2)| 00:00:59 |
|*  4 |     HASH JOIN                  |                  |   105 |  5460 |  4782   (2)| 00:00:58 |
|*  5 |      HASH JOIN                 |                  |  1320 | 44880 |  4118   (2)| 00:00:50 |
|*  6 |       TABLE ACCESS FULL        | SHIPMENTDETAILS  |   304 |  6080 |  1070   (2)| 00:00:13 |
|   7 |       TABLE ACCESS FULL        | ITEM             |  3588K|    47M|  3036   (1)| 00:00:37 |
|*  8 |      TABLE ACCESS FULL         | ORDERS           | 54903 |   965K|   663   (3)| 00:00:08 |
|*  9 |     TABLE ACCESS BY INDEX ROWID| CUSTOMER         |     1 |    33 |     1   (0)| 00:00:01 |
|* 10 |      INDEX UNIQUE SCAN         | PK_CUSTOMER      |     1 |       |     0   (0)| 00:00:01 |
|* 11 |    INDEX UNIQUE SCAN           | PK_PRODUCT       |     1 |       |     0   (0)| 00:00:01 |
|  12 |   TABLE ACCESS BY INDEX ROWID  | PRODUCT          |     1 |   109 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("I"."ORDERID"="O"."ORDERID")
   5 - access("SD"."SHIPMENTID"="I"."SHIPMENTID")
   6 - filter("SD"."SHIPSTATUS"<>'C')
   8 - filter("O"."ORDERDATE">=SYSDATE@!-30)
   9 - filter("C"."LASTNAME" LIKE 'SMI%')
  10 - access("C"."CUSTOMERID"="O"."CUSTOMERID")
  11 - access("P"."PRODUCTID"="I"."PRODUCTID")
```

The expensive steps from this plan are several full table scans performed against fairly large tables named ORDERS, ITEM and SHIPMENTDETAILS. The number of logical reads is 73,600.

CREATE THE SQL DIAGRAM



Note: I typically draw SQL diagrams like this because they are much simpler and cleaner but still provides the important details of selectivity.

In this example, the end-users are trained to input only the first three characters of a name. For the selectivity calculations on the CUSTOMER table we should use "SMI%" as the criteria because it is the most occurring first three characters of the LASTNAME.

```
SELECT COUNT(1) FROM Customer
WHERE LastName LIKE 'SMI%'
Results – 1,917 / 52,189 = .04
```

The next table to review is the ORDERS table. Whenever I have two criteria against a table in a query, I run them together as well as individually so I understand selectivity of each. For this SQL statement I ran the following queries:

```
SELECT count(1) FROM orders o
WHERE o.OrderDate >= SYSDATE - 30
AND o.OrderStatus <> 'C'
```

Results – 0.005 = 0.5% selectivity

```
SELECT count(1) FROM orders o
WHERE o.OrderDate >= SYSDATE - 30
```

Results – 0.08 = 8% selectivity

```
SELECT count(1) FROM orders o
WHERE o.OrderStatus <> 'C'
```

Results – 0.005 = 0.5% selectivity

These results show the ORDERDATE column selectivity is not near as good as the selectivity of the ORDERSTATUS column individually. Also, when including both criteria, the selectivity matches the results from using only the ORDERSTATUS column. Because of this, we should focus on that criterion alone without ORDERDATE.

TUNING BASED ON SQL DIAGRAMS

Since this is a non-equality criterion, adding an index on the ORDERSTATUS column will not help because Oracle will not use it. I always review data skew when I notice a status or flag column being used in a query. For this column, here are a list of values and the number of rows that contain the value:

```
SELECT OrderStatus, COUNT(1)
FROM Orders
GROUP BY OrderStatus
I      3760
C      68911
```

Based on this data, the criteria could be rewritten as:

```
AND o.OrderStatus = 'I'
```

This is a little dangerous to do without discussions about the data with the applications group. A problem can arise if the ORDERSTATUS column can have other values which would break the new query's logic. In this case, 'I' and 'C' are the only two possible values so our new logic is sound. This also means that an index on ORDERSTATUS seems to make sense. Remember that when a column contains skewed data, you will also need to collect histograms on the new index as well (not shown in the statements below):

```
CREATE INDEX order_status ON Orders (OrderStatus);
```

The new execution plan is:

```
-------------------------------------------------------------------------------------------
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT | | 1581 | 301K| 3722 (1)| 00:00:45 |
|* 1 | HASH JOIN | | 1581 | 301K| 3722 (1)| 00:00:45 |
|* 2 | HASH JOIN | | 1581 | 270K| 2655 (1)| 00:00:32 |
| 3 | NESTED LOOPS | | | | |
| 4 | NESTED LOOPS | | 1581 | 101K| 2587 (1)| 00:00:32 |
|* 5 | HASH JOIN | | 303 | 15756 | 170 (1)| 00:00:03 |
|* 6 | TABLE ACCESS BY INDEX ROWID| ORDERS | 303 | 5757 | 33 (0)| 00:00:01 |
|* 7 | INDEX RANGE SCAN | ORDER_STATUS | 3811 | | 9 (0)| 00:00:01 |
|* 8 | VIEW | index$_join$_001 | 2055 | 67815 | 137 (1)| 00:00:02 |
|* 9 | HASH JOIN | | | | |
|* 10 | INDEX RANGE SCAN | CUSTOMER_LN | 2055 | 67815 | 14 (8)| 00:00:01 |
| 11 | INDEX FAST FULL SCAN | PK_CUSTOMER | 2055 | 67815 | 123 (0)| 00:00:02 |
|* 12 | INDEX RANGE SCAN | PK_ORDERINFO | 5 | | 2 (0)| 00:00:01 |
| 13 | TABLE ACCESS BY INDEX ROWID | ITEM | 5 | 70 | 8 (0)| 00:00:01 |
| 14 | TABLE ACCESS FULL | PRODUCT | 8914 | 948K| 68 (0)| 00:00:01 |
| 15 | TABLE ACCESS FULL | SHIPMENTDETAILS | 836K| 15M| 1064 (1)| 00:00:13 |
```

This query now performs 7,221 logical reads. This is a significant improvement. Again, SQL diagramming is instrumental in identifying how to tune the query correctly the first time. Reviewing the execution plan shows that there are still full table scans on the PRODUCT and SHIPMENTDETAILS table, so there probably is room for additional tuning.

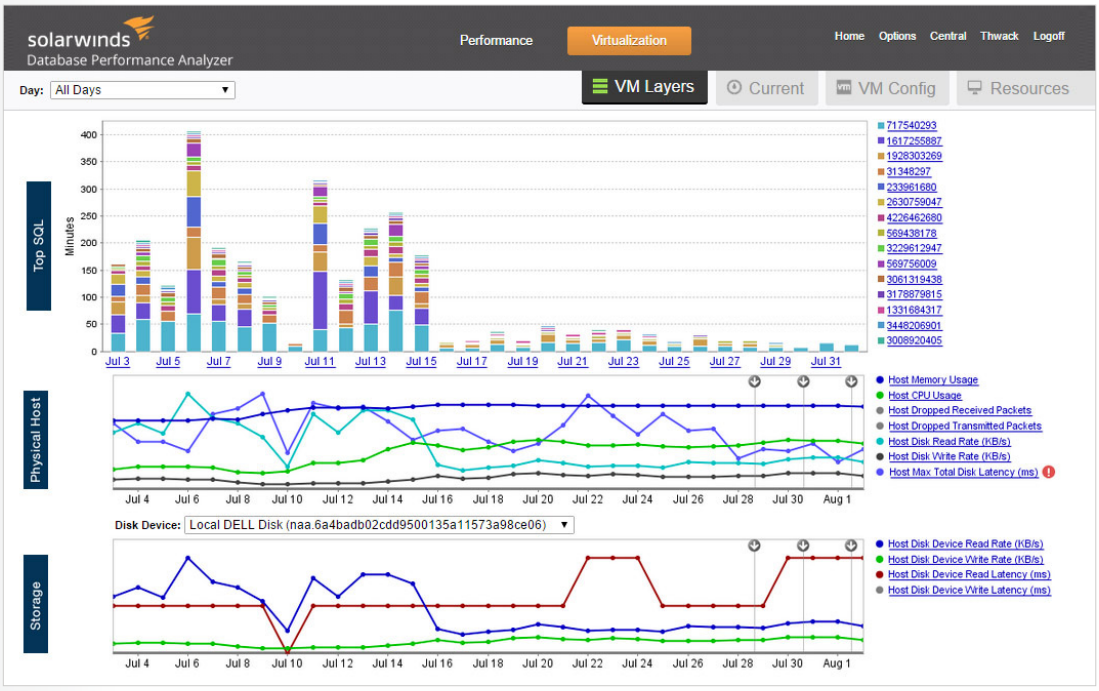
SUMMARY

As a DBA, I get to work through database performance issues with hundreds of DBAs and developers every year. There is one commonality amongst all these professionals: no one has time to spare. Everyone wants to streamline processes and solve problems as quickly as possible and move on to the next issue of the day. A simple, repeatable process for performance tuning is a valuable time saver because you'll get it right the first time.

1. Focus on the correct SQL statements
2. Utilize response time analysis
3. Gather accurate execution plans
4. Use SQL diagramming

HOW CAN DATABASE PERFORMANCE ANALYZER HELP?

Database Performance Analyzer (DPA) from SolarWinds (NYSE: SWI) provides the fastest way to identify and resolve database performance issues. DPA is part of the SolarWinds family of powerful and affordable IT solutions that eliminate the complexity in IT management software. DPA's unique Multi-dimensional Database Performance Analysis enables you to quickly get to the root of database problems that impact application performance with continuous monitoring of SQL Server, Oracle, SAP ASE and DB2 databases on physical, Cloud-based and VMware servers.



For additional information, please contact SolarWinds at 866.530.8100 or e-mail **sales@solarwinds.com.**