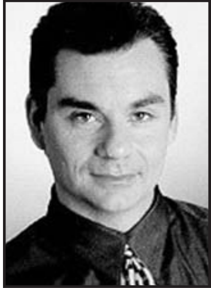
Optimizing Performance by Identifying Problem SQL



By Robin Schumacher

As a DBA, you need to have the right game plan in place for finding and fixing problem SQL code in your database. Fortunately, Oracle is better than most DBMSs at providing information in the data dictionary to help a DBA locate and analyze potentially bad SQL, so by using the roadmap and scripts provided in this article, you should be able to pinpoint any bad SQL that's run through your system.

What is *Bad SQL?*

Before you can identify problem SQL in your database, you have to ask the question what is *bad SQL?* What criteria do you use when you begin the hunt for problem SQL in your critical systems? Understand that even the seasoned experts disagree on what constitutes efficient and inefficient SQL; so there's no way to sufficiently answer this question to every Oracle professional's satisfaction. What follows are some general criteria you can use when evaluating the output from various database monitors or personal diagnostic scripts:

- **Overall Response (Elapsed) Time –** how much time the query took to parse, execute, and fetch the data needed to satisfy the query. It should not include the network time needed to make the round trip from the requesting client workstation to the database server.
- **CPU Time –** how much CPU time the query took to parse, execute, and fetch the data needed to satisfy the query.
- **Physical I/O –** often used as the major statistic in terms of identifying good vs. bad SQL. It is a measure of how many disk reads the query caused to satisfy the user's request. While you certainly want to control disk I/O where possible, it's important that you not focus solely on physical I/O as the single benchmark of inefficient SQL. Make no mistake, disk access is slower than memory access and also consumes processing time making the physical to logical transition. However, you need to look at the entire I/O picture of a SQL statement, which includes looking at a statements' logical I/O as well.
- **Logical I/O –** a measure of how many memory reads the query took to satisfy the user's request. The goal of tuning I/O for a query should be to examine both logical and physical I/O, and use appropriate mechanisms to keep both to a minimum.
- **Repetition –** a measure of how often the query has been executed. A problem in this area isn't as easy to spot as the others unless you know your application well. A query that takes a fraction of a second to execute may still be causing a headache on your system if it's executed erroneously over and over again. An example would be for example, a query that executes in a runaway PL/SQL loop.

There are other criteria that you can examine like sort activity or access plan statistics (that show items like Cartesian joins and the like), but more often than not, these measures are reflected in the criteria listed above.

Fortunately, Oracle records all the above measures (at least 9*i* does), which makes tracking the SQL that's been submitted against an Oracle database a lot easier.

Pinpointing Bad SQL

When you begin to look for inefficient SQL in a database, there are two primary questions you want answered:

- What has been the worst SQL that's historically been run in my database?
- What's the worst SQL that's running right now in my database?

Historical SQL Analysis

The easiest way to perform historical SQL analysis is to use either a third-party software vendor tool, or a homegrown solution to periodically collect SQL execution statistics into a database repository, and then analyze the results. While Oracle does record SQL execution information, such data can be lost if a DBA flushes the shared pool or shuts down the database, so ad-hoc historical analysis using straight SQL scripts might produce misleading results. However, if your database is predictable in terms of SQL code being kept in the shared pool, you can certainly obtain good metrics to determine if inefficient SQL is being executed in your system.

A good *Top SQL* script to use for Oracle 9*i* is the top9isql.sql query. It will pull the top 20 SQL statements as determined (initially) by disk reads per execution, but you can change the sort order to sort on logical I/O, elapsed time, etc.:

continued on page 32

Optimizing Performance by Identifying Problem SQL continued from page 31

top9isql.sql

```
select
        sql_text,
        username,
        disk_reads_per_exec,
        buffer_gets_per_exec,
        buffer_gets,
        disk_reads,
        parse_calls,
        sorts,
        executions,
        loads,
        rows_processed,
        hit_ratio,
        first_load_time,
        sharable_mem,
        persistent_mem,
        runtime_mem,
        cpu_time_secs,
        cpu_time_secs_per_execute,
        elapsed_time_secs,
        elapsed_time_secs_per_execute,
        address,
        hash_value
from
(select
        sql_text,
        b.username,
        round((a.disk_reads/
        decode(a.executions,0,1,a.executions)),2)
        disk_reads_per_exec,
        a.disk_reads,
        a.buffer_gets,
        round((a.buffer_gets/
        decode(a.executions,0,1,a.executions)),2)
        buffer_gets_per_exec,
        a.parse_calls,
        a.sorts,
        a.executions,
        a.loads,
        a.rows_processed,
        100 - round(100 *
        a.disk_reads/
        greatest(a.buffer_gets,1),2) hit_ratio,
        a.first_load_time,
        sharable_mem,
        persistent_mem,
        runtime_mem,
        round(cpu_time / 1000000,3) cpu_time_secs,
        round((cpu_time / 1000000)/
        decode(a.executions,0,1,a.executions),3)
        cpu_time_secs_per_execute,
        round(elapsed_time / 1000000,3) elapsed_time_secs,
        round((elapsed_time /
        1000000)/decode(a.executions,0,1,a.executions),3)
        elapsed_time_secs_per_execute,
        address,
        hash_value
from
        sys.v_$sqlarea a,
        sys.all_users b
where
        a.parsing_user_id=b.user_id and
        b.username not in ('SYS','SYSTEM')
        order by 3 desc)
where
      rownum < 21;
```

Output from this query might resemble the following:

	SQL_TEXT	USERNAME	DISK_READS_PER_EXEC	BUFFER_GETS_PER_EXEC	BUFFER_GETS	DISK_READS	PARSE_CALLS	SORTS	EX
1	begin PERFCNTR_24x7_QUERIES.fetchcursor20_2(VAR1_	USR1	122.5	78886.5	157773	245	2	0	
2	SELECT 956, (INVALID_OBJECTS + UNUSABLE_INDEXES) AS TOTAL FROM (SELECT	USR1	94	45723.5	91447	188	1	0	
3	SELECT 'PERFCNTR_24x7_QUERIES', PERFCNTR_24x7_QUERIES.GetVersion from DUAL UNION	USR1	67	975	975	67	1	1	
4	SELECT 900, (total_space - total_free_space)/1048576 FROM (SELECT SUM(bytes) AS total_space FROM	USR1	18.5	238	476	37	1	0	
5	select 99 / 100, 1 - to_number(to_char(to_date('1997-11-0	USR1	11	86	86	11	1	0	
6	SELECT 955, COUNT(*) FROM SYS.DBA_TABLES WHER	USR1	5.5	30784	61568	11	1	0	
7	SELECT 998, COUNT(*) FROM (SELECT USERNAME FROM SYS.DBA_USERS WHERE	USR1	3.5	239.5	479	7	1	18	
8	select a.machine, b.count from (SELECT DISTINCT MACHINE FROM V$SESSION WHERE TYPE =	USR1	3	63	63	3	1	1	
9	SELECT 977, a.active active_jobs,b.due-a.active jobs_waiting,c.snp_processes - a.active idle_jobs,c.snp_processes total_jobs FROM (SELECT	USR1	2.5	70.5	141	5	1	2	
10	begin PERFCNTR_24x7_QUERIES.fetchcursor22_5(VAR1_	USR1	2.5	70.5	141	5	2	0	
11	begin PERFCNTR_24x7_QUERIES.fetchcursor16_3(VAR1_	USR1	1.67	114.5	687	10	6	0	
12	SELECT 948, ACTIVE_COUNT,ROUND(100 * (ACTIVE_COUNT / TOTAL_COUNT),2) AS ACTIVE_PCT	USR1	1.67	76.17	457	10	1	0	
13	begin PERFCNTR_24x7_QUERIES.fetchcursor1_2(VAR1_	USR1	1.33	156.17	937	8	6	0	
14	begin PERFCNTR_24x7_QUERIES.fetchcursor3_3(VAR1_	USR1	.83	107.83	647	5	6	0	

Figure 1. Output From the Top 20 SQL Query

If you are using a version of Oracle less than 9*i*, you will have to modify the query above and remove the references to ELAPSED_TIME and CPU_TIME, which were new columns added to the v$sqlarea view in 9*i*.

It's important to examine the output of this query and see how it uses the criteria set forth at the beginning of this article to pinpoint problematic SQL.

First, start by looking at Figure 2 and focus on the circled columns.

	PERSISTENT_MEM	RUNTIME_MEM	CPU_TIME_SECS	CPU_TIME_SECS_PER_EXECUTE	ELAPSED_TIME_SECS	ELAPSED_TIME_SECS_PER_EXECUTE	ADDRESS	HASH_VALUE
1	540	276	1.642	.821	4.814	2.407	672621C0	3077230681
2	696	30300	[illegible]	.26	1.204	.602	6746A26C	2965734772
3	1400	3684	.11	.11	.399	.399	674D57CC	2741343822
4	2388	22848	.06	.03	1.368	.684	6746A848	3829441909
5	700	876	.02	.02	.079	.079	674FA26C	1276527007
6	660	12748	.26	.13	.293	.147	6746A460	2911359602
7	692	100596	.05	.025	.07	.035	67469D84	3547812303
8	664	4036	.02	.02	.034	.034	674F4504	2966647892
9	852	4388	.02	.01	.054	.027	674688A0	482951707

Figure 2. Output From the Top 20 SQL Query That Shows Timing Statistics

The output displays both CPU and elapsed times for each query. The times are shown both cumulatively (in seconds) and per execution, indicating, for example, that the first query in the result set has accumulated almost five seconds of total execution time and runs for about two and half seconds each time it's executed. You can change the query to sort by any of these timed statistics, depending on the criteria you need to use to bubble the worst-running SQL to the top of the result set. Again, sadly, you lose these metrics when you use any database version under Oracle9*i*.

If you look at Figure 1, you can see the columns that will help you examine the I/O characteristics of each SQL statement. You can see the number of disk reads (physical I/O) and buffer gets (logical I/O), along with numbers that display the average I/O consumption of each SQL statement. Note that queries that have been executed once may have misleading statistics with respect to disk reads, as the data needed for the first run of the query was likely read in from disk to memory. Therefore, the number of disk reads per execution should drop for subsequent executions and the hit ratio for the query should rise.

The executions column of the top SQL's result set will provide clues to the repetition metric for the query. When troubleshooting a slow system, you should be on the lookout for any query that shows an execution count that's significantly larger than any other query on the system. It may be that the

query is in an inefficient PL/SQL loop, or other problematic programming construct. Only by bringing the query to the attention of the application developers will you know if the query is being mishandled from a programming standpoint.

Once you find the SQL statements through Oracle's diagnostic views, you will want to get the entire SQL text for the statements that appear inefficient. You should note the HASH_VALUE values for each SQL statement, and then issue the fullsql.sql script to obtain the full SQL statement:

fullsql.sql

```
select
     sql_text
from
      sys.v_$sqltext
where
      hash_value = <enter hash value for sql statement>
order by piece;
```

Current SQL Analysis

If your phone begins to ring with complaints of a slow database, you can quickly check to see what SQL is currently executing to understand if any resource intensive SQL is dragging down your database's overall performance levels.

This is very easy to do and only involves making one change to the already discussed top9isql.sql query. You should add the following filter to the main query's where clause:

```
where
        a.parsing_user_id=b.user_id and
        b.username not in ('SYS','SYSTEM') and
        a.users_executing > 0
        order by 3 desc;
```

This query change will display the worst SQL that is currently running in the database, so you can quickly tell if any queries are to blame for a dip in database performance.

New Techniques for Analyzing SQL Execution

The techniques and queries showcased above are the more traditional means of pinpointing problem SQL in a database. But if you are using Oracle 9*i*, there are some new methods you can use to get a handle on how well the SQL in your database is executing.

For example, an Oracle 9*i* DBA may want to know how many total SQL statements are causing Cartesian joins on the system. The following 9icartcount.sql query can answer that:

9icartcount.sql

```
select
       count(distinct hash_value) cartesian_statements,
       count(*) total_cartesian_joins
from
       sys.v_$sql_plan
where
       options = 'CARTESIAN' and
       (operation like '%JOIN%' or
        operation = 'NESTED LOOPS');
```

Output from this query might resemble the following (note that it is possible for a single SQL statement to contain more than one Cartesian join):

```
CARTESIAN_STATEMENTS     TOTAL_CARTESIAN_JOINS
--------------------     ---------------------
                   4                         6
```

A DBA can then view the actual SQL statements containing the Cartesian joins, along with their performance metrics by using the 9icartsql.sql query:

9icartsql.sql

```
select
      *
from
      sys.v_$sql
where
      hash_value in
(select
      hash_value
from
      sys.v_$sql_plan
where
      options = 'CARTESIAN'
AND  (operation LIKE '%JOIN%' or
      operation = 'NESTED LOOPS') )
order by hash_value;
```

Another big area of interest for DBAs concerned with tuning SQL is table scan activity. Most DBAs don't worry about small table scans, as Oracle can oftentimes access small tables more efficiently through a full scan than through index access (the small table is just cached and accessed). Large table scans, however, are another matter. Most DBAs prefer to avoid those, where possible, through smart index placement or intelligent partitioning.

Using the new 9*i v$sql_plan* view, a DBA can quickly identify any SQL statement that contains one or more large table scans, and even define *large* in their own terms. The following 9itabscan.sql query shows any SQL statement that contains a large table scan (defined in this query as a table over 1MB), along with a count of how many large scans it causes for each execution, the total number of times the statement has been executed, and then the sum total of all scans it has caused on the system:

9itabscan.sql

```
select
       sql_text,
       total_large_scans,
       executions,
       executions * total_large_scans sum_large_scans
from
(select
       sql_text,
       count(*) total_large_scans,
       executions
from
     sys.v_$sql_plan a,
     sys.dba_segments b,
     sys.v_$sql c
where
      a.object_owner (+) = b.owner
and   a.object_name (+) = b.segment_name
and   b.segment_type in ('TABLE', 'TABLE PARTITION')
and   a.operation like '%TABLE%'
and   a.options in ('FULL', 'ALL')
```

continued on page 34

Optimizing Performance by Identifying Problem SQL continued from page 33

```
and   c.hash_value = a.hash_value
and   b.bytes / 1024 > 1024
group by
      sql_text, executions)
order by
       4 desc;
```

Query | Results

	SQL_TEXT	TOTAL_LARGE_SCANS	EXECUTIONS	SUM_LARGE_SCANS
1	select o.owner#,o.obj#,decode(o.linkname,null, decode(u.name,null,'SYS',u.name),o.remoteowner),	1	19	19
2	SELECT 1 FROM SYS.DBA_OBJECTS WHERE ROWNUM = 1 MINUS SELECT 1 FROM SYS.DBA_EXTENTS WHERE ROWNUM = 1 MINUS	2	2	4
3	SELECT 1 FROM SYS.DBA_OBJECTS WHERE ROWNUM = 1 MINUS SELECT 1 FROM SYS.DBA_EXTENTS WHERE ROWNUM = 1 MINUS	2	1	2
4	SELECT OWNER,TABLE_NAME,NUM_ROWS,PCT_FREE,PCT_USED,TA	1	2	2
5	EXPLAIN PLAN SET STATEMENT_ID='10118429' INTO EMBARCADERO	1	1	1
6	SELECT OWNER,TABLE_NAME,NUM_ROWS,PCT_FREE,PCT_USED,TABLESPA	1	1	1
7	select count(*) from eradmin.emp	1	1	1
8	select distinct i.obj# from sys.idl_ub1$ i where i.obj#>=1 and i.obj# not	1	1	1

Figure 3. Output Showing Large Table Scan Activity From an Oracle 9*i* Database

This query provides important output and poses a number of interesting questions. As a DBA, should you worry more about a SQL statement that causes only one large table scan, but has been executed 1,000 times, or a SQL statement that has 10 large scans in it, but has only been executed a handful of times? Each DBA will likely have an opinion on this, but regardless, you can see how such a query can assist with identifying SQL statements that have the potential to cause system slowdowns.

Oracle 9.2 has introduced another new performance view – *v$sql_plan_statistics* – that can be used to get even more statistical data regarding the execution of inefficient SQL statements. This view can tell you how many buffer gets, disk reads, etc., that each step in a SQL execution plan caused, and even goes so far as to list the cumulative and last executed counts of all held metrics. DBAs can reference this view to get a great perspective of which step in a SQL execution plan is really responsible for most of the resource consumption. Note that to enable the collection of data for this view, you must set the Oracle configuration parameter *statistics_level* to ALL.

An example that utilizes this new 9*i* view is the following 9iplanstats.sql script that shows the statistics for one problem SQL statement:

9iplanstats.sql

```
select
       operation,
       options,
       object_owner,
       object_name,
       executions,
       last_output_rows,
       last_cr_buffer_gets,
       last_cu_buffer_gets,
       last_disk_reads,
       last_disk_writes,
       last_elapsed_time
from
       sys.v_$sql_plan a,
       sys.v_$sql_plan_statistics b
where
       a.hash_value = b.hash_value and
       a.id = b.operation_id and
       a.hash_value = <enter hash value>
order by a.id;
```

Query | Results

	OPERATION	OPTIONS	OBJECT_OWNER	OBJECT_NAME	EXECUTIONS	LAST_OUTPUT_ROWS	LAST_CR_BUFFER_GETS	LAST_CU_BUFFER_GETS	LAST_DISK_READS	LAST
1	MERGE JOIN	CARTESIAN	[NULL]	[NULL]	1	31849	46	0	18	
2	TABLE ACCESS	FULL	ERADMIN	PATIENT	1	22	24	0	5	
3	BUFFER	SORT	[NULL]	[NULL]	1	31849	22	0	13	
4	PARTITION RANGE	ALL	[NULL]	[NULL]	1	1507	22	0	13	
5	TABLE ACCESS	FULL	ERADMIN	ADMISSION	1	1507	22	0	13	

Figure 4. Example Output Showing Statistical Metrics for Each Step in a 9*i* Query Execution Plan

SQL Tuning Roadmap

There are large volumes of SQL tuning books on the market that provide minute detail on how to build and tune SQL code. As a DBA, you can stay immersed in such manuals (and many of them are very good) for a long time, but chances are, you don't have the time. If that's the case for you, then walk through the quick generic roadmap below. You can use them to remedy some of the problem SQL statements that you've identified, using the techniques and scripts already outlined in this article.

Once you have one or more problem queries in hand, you can then start the tuning process. It basically consists of these three broad steps:

- Understand the query and dependent objects
- Look for SQL rewrite possibilities
- Look for object-based solutions

Understand the Query and Dependent Objects

The first thing to do is get a handle on what the query is trying to do, how the Oracle optimizer is satisfying the query's request, and what kind of objects the query is referencing.

In terms of understanding what the query is trying to accomplish and how Oracle will handle the query, the EXPLAIN plan is your first step. However, while the EXPLAIN plan has been around for as long as SQL itself, you might be surprised at how many seasoned database professionals aren't very good at reading an EXPLAIN plan output. Now, many can do the basics like recognize table scans, spot Cartesian joins, and zero in on unnecessary sort operations, but when the EXPLAIN output *wave* starts rolling back and forth in large SQL EXPLAINs, some tend to get a little lost. Some of the better SQL analysis tools are now sporting a new EXPLAIN format (better graphics and English-based explanations) that makes it a lot easier to follow the access path trail. For DBAs who aren't good at reading traditional EXPLAIN output, it makes getting to the root of a bad SQL statement much simpler. Additionally, it can save you or one of your developers from submitting the query from you-know-where.

If you aren't using a third-party SQL analysis product with built-in EXPLAIN functionality, then you will have to use the standard EXPLAIN plan table and methods for performing a SQL statement EXPLAIN. Users of Oracle 9*i*, however, can get an EXPLAIN of any SQL statement that's already been executed in the database. A script like the 9iexpl.sql can be used:

9iexpl.sql

```
select
        lpad(' ',level-1)||operation||' '||options||' '||
        object_name "Plan",
        cost,
        cardinality,
        bytes,
        io_cost,
        cpu_cost
   from
        sys.v_$sql_plan
   connect by
        prior id = parent_id
        and prior hash_value = hash_value
  start with
        id = 0 and hash_value = <enter hash value>
  order by id;
```

	Plan	COST	CARDINALITY	BYTES	IO_COST	CPU_COST
1	SELECT STATEMENT	1502	[NULL]	[NULL]	[NULL]	[NULL]
2	MERGE JOIN CARTESIAN	1502	750000	52500000	1502	[NULL]
3	TABLE ACCESS FULL PATIENT	2	500	23500	2	[NULL]
4	BUFFER SORT	1500	1500	34500	1500	[NULL]
5	PARTITION RANGE ALL	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
6	TABLE ACCESS FULL ADMISSION	3	1500	34500	3	[NULL]

Figure 5. Example Output for an EXPLAIN Plan Generated for a SQL Statement Already in Oracle 9*i*'s Shared Pool

Once you gain understanding through an EXPLAIN, you should then begin to look into the objects the EXPLAIN plan is referencing. When writing efficient SQL, it's imperative to know the demographics and shape of the objects your code will bump against. For most databases, all the information you will ever need can be found in the data dictionary. But when querying the dictionary for object statistics, you need to make sure you're looking at accurate information.

While some databases like SQL Server 2000 have automatic updating of object statistics into the dictionary, other RDBMS engines like Oracle require you to manually (and periodically) refresh the database catalog with up-to-date object data. Fortunately, this is pretty easy to accomplish. Oracle now offers special packages to assist with the updating of objects, in addition to the standard *analyze* command. The *dbms_utility* package contains several procedures to help database professionals update their schema objects. To update the dictionary for a single schema, you can use the *dbms_utility.analyze_schema* procedure. The *dbms_utility.analyze_database*. procedure has been recently introduced for larger updates. Just be careful when executing such a procedure against a monolithic database like Oracle's applications.

Whatever method you choose to update your objects, you ought to make a practice of keeping data in the dictionary current, especially for databases that are very dynamic in nature. Scheduling the object updates in a nightly maintenance plan would probably be a good thing to do for such a database, especially if you use the cost-based optimizer in Oracle. Fresh statistics help the optimizer make more informed choices of what path to follow when routing your queries to the requested data. Obviously, if your database thinks you only have 100 rows in a table that actually contains a million, the map used by the optimizer might not be the right one and your response times will show it.

When tuning SQL, what types of metrics should you look for in your objects to help you make intelligent coding choices? Although this list is certainly not exhaustive, for tables you can start by eyeballing these items:

- **Row Counts –** No heavy explanation is needed for why you should be looking at this statistic. You will want to avoid full scans on beefy tables with large row counts. Proper index placement becomes quite important on such tables. Other reasons for reviewing row counts include physical redesign decisions. Perhaps a table has grown larger than anticipated and is now eligible for partitioning? Scanning a single partition in a table is a lot less work than running through the entire table.

- **Chained Row Counts –** Row chaining and migration can be a thorn in the side of an otherwise well-written SQL statement. Chained rows are usually the result of invalid page or block size choices (rows for a wide table just plain won't fit on a single page or block). Migration is caused when a row expands beyond the remaining size boundary of the original block it was placed into. The database is forced to move the row to another block and leaves a pointer behind to indicate its new location.

  While chaining and migration are different, they have one thing in common: extra I/O is needed to retrieve the row that is either chained or migrated. Knowing how many of these your table has can help you determine if an object reorganization is needed. Extreme cases may require a full database rebuild with a larger blocksize. Oracle 9*i* users, however, can create new tablespaces with larger block sizes and move/reorganize their objects into them.

- **Space Extents –** For some databases, objects that have moved into multiple extents can be slower to access than same-size objects that are contained within a single contiguous extent of space. Oracle, however, doesn't suffer from this multi-extent problem.

- **High Water Marks –** Tables that experience much insert and delete activity can be special problem children. Oracle will always scan up to a table's *high water mark,* which is the last block of space it *thinks* contains data. For example, a table that use to contain a million rows, but now only has a hundred may be scanned like it still has a million! You can determine if you need to reload a table (usually done by a reorg or truncate and load) by checking the high water marks of tables to see if they still are set to abnormally high values.

- **Miscellaneous Properties –** There are several other performance boosting properties that you may want to set for tables. For instance, large tables that are being scanned may benefit from having parallelism enabled so the table can be scanned (hopefully) much quicker. Small lookup tables may benefit from being permanently cached in memory to speed access times. In Oracle, this may be done by placing them into the KEEP buffer pool. The CACHE parameter may also be used, although it is not as permanent a fix as the KEEP buffer pool option.

Indexes have their own unique set of items that need occasional review. Some of these include:

- **Selectivity/Unique Keys –** Indexes by their very nature normally work best when selectivity is high – in other words, the numbers of unique values are many. The exception to this rule is the bitmap index, which is designed to work on columns with very low cardinality (like a Yes/No column). The selectivity of indexes should be periodically examined to see if an index that used to contain many unique values is now one that is losing its uniqueness rank.

- **Depth –** The tree depth of an index will tell you if the index has undergone a lot of splits and other destructive activity. Typically, indexes with tree depths greater than three or four are good candidates for rebuilds, an activity that hopefully will improve access speed.

- **Deleted Row Counts –** Indexes that suffer from high parent table maintenance may contain a lot of *dead air* in the form of deleted rows in the leaf pages. Again, a rebuild may be in order for indexes with high counts of deleted leaf rows.

continued on page 36

Optimizing Performance by Identifying Problem SQL continued from page 35

There are of course other items you can review on the table and index statistical front, as well as at the individual column level.

Understanding the current state and shape of the objects being used in the queries you are trying to tune can unlock clues about how you may want to restructure your SQL code. For example, you may realize you haven't indexed critical foreign keys that are used over and over again in various sets of join operations. Or you might find that your million-row table is a perfect candidate for a bitmap index given the current where predicate.

Look for SQL Rewrite Possibilities

For complex and problematic systems, analyzing and attempting the rewrite of many SQL statements can consume a lot of a database professional's time. A book of this nature cannot possibly go into this vast subject, as there are a plethora of techniques and SQL hints that can be used to turn a query that initially runs like molasses into one that runs as fast as greased lightning.

To save time, you might want to make use of one of the third-party SQL tuning tools that can help with rewriting SQL statements. Some of these tools will even generate automatic rewrites that you can trial and review. Some have found that using such tools can indeed cut down on the SQL tuning process if used properly, since they offer easy generation of hints and (normally) a good benchmarking facility that allows easy execution and review of performance statistics.

Even if you don't have access to third-party SQL tuning products, you can still use SQL*Plus to perform comparison benchmarks. By using the SET AUTOTRACE ON feature of SQL*Plus, you can get decent feedback from Oracle on how efficient a query is:

```
SQL>set autotrace on;
SQL>select count(*) from admission;

  COUNT(*)
-----
      1552

Execution Plan
----------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1)
   1    0   SORT (AGGREGATE)
   2    1     INDEX (FAST FULL SCAN) OF 'ADMISSION_PK' (UNIQUE) (Cost=
          1 Card=1542)


Statistics
----------------------------
          0  recursive calls
          4  db block gets
          5  consistent gets
          0  physical reads
          0  redo size
        368  bytes sent via SQL*Net to client
        425  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
                  1 rows processed
```

So what are some things you should look for in terms of rewriting SQL? While this is certainly a large topic, there are a few major items that stand out above the rest:

- **The Cartesian Product –** Although some optimizers will automatically try to rewrite SQL and will actually use a Cartesian product to accomplish the mission, seeing a Cartesian join in your EXPLAIN plan is usually not a good thing. If observed, check your WHERE predicate to ensure you are adhering to the N – 1 rule of thumb (for example, 10 tables in a FROM clause will require 9 proper join conditions).
- **The Table Scan –** So you thought you were using an index, did you? There are a lot of conditions that can negate the use of an index (use of NOT, failing to use the starting column of a concatenated index, use of expressions like WHERE TOTAL_SALARY = SALARY * 1.2). Or perhaps the SQL is fine, but an improper indexing scheme is being used and needs to be changed. Remember, however, that you are really looking for table scans on large tables. Small lookup tables are actually accessed many times faster when the database engine caches the whole table and scans it, rather than using an available index.
- **The Unnecessary Sort –** Can your query do without the DISTINCT clause you have in the code? Can the UNION be replaced with a UNION ALL? Knowing when and how to yank sort activity out of a SQL statement can go a long way in improving its response time.
- **The Nonselective Index Scan –** If you've followed your checklist and understand the demographics of the objects used in your query, then you should know what indexes are selective and which aren't. While cost based optimizers should ignore indexes with poor selectivity, rule based approaches may not. Be on the lookout for these types of scans because not every index scan is a good one.

When you've eliminated these noticeable coding flaws, it's now time to begin trying different code combinations in the hopes of improving your query's performance. This is where the use of hints can become a serious time saver. The SQL language now contains a plethora of hints that you can imbed in your query without really touching the actual code structure that exists. By using hints, you can in effect accomplish many iterations of rewrites in a short amount of time.

Of course, one of the dangers of using hints is not writing them accurately. Unlike typical SQL code that will cough up a syntax error if you make a mistake, an invalid hint won't afford you that luxury. *It just won't do anything*. Therefore, you need to make sure you code your hints accurately. Even the smallest of tuning techniques that use hints can produce dramatic results. For example, one large Oracle data dictionary query that is used in this book was tuned quite well by just introducing the RULE hint to the code. It reduced the number of physical reads from 1,400 to 51, and cut the overall response time of the query in half.

So what different coding approaches should you try using hints? While there is no way to give you a complete rundown of everything that is open to you, there are a few mainstays to try:

- **The Four Standby's –** These include RULE, FIRST ROWS, ALL ROWS, and COST. Believe it or not, many times a query has been dramatically improved just by going back to the rule base optimizer. Even with all the progress made by cost-based approaches, sometimes the old way is the best way.
- **Table Order Shuffle –** You may want to influence the order in which the optimizer joins the tables used in your query. The ORDERED hint can force the database to use the right tables to drive the join operation, and hopefully reduce inefficient I/O.
- **Divide and Conquer –** When databases introduced parallel operations, they opened up a whole new avenue in potential speed gains. The PARALLEL hint can be a powerful ally in splitting large table scans into chunks that may be worked on separately in parallel, and then merged back into a single result set. One thing to ensure is that your database is set up properly with respect to having enough parallel worker bees (or *slaves*) to handle the degree of parallelism you specify.

- **Index NOW –** From the EXPLAIN plan, you may discover that the optimizer is not using an available index. This may or may not be a good thing. The only way to really tell is to force an index access plan in place of a table scan with an index hint.

Look for Object-based Solutions

Object-based solutions are another option for SQL tuning analysts. This route involves things like intelligent index creation, partitioning, and more. But to do this, you have to first find the objects that will benefit from such modification, which in turn will enhance the overall runtime performance. For users of Oracle 9*i*, the new V$ views can help with this type of analysis.

For example, to investigate better use of partitioning, you would first need to locate large tables that are the consistent targets of full table scans. The 9iltabscan.sql query below will identify the actual objects that are the target of such scans. It displays the table owner, table name, the table type (standard, partitioned), the table size in KB, the number of SQL statements that cause a scan to be performed, the number of total scans for the table each time the statement is executed, the number of SQL executions to date, and the total number of scans that the table has experienced (total single scans * executions):

9iltabscan.sql

```
select
    table_owner,
    table_name,
    table_type,
    size_kb,
    statement_count,
    reference_count,
    executions,
    executions * reference_count total_scans
from
   (select
    a.object_owner table_owner,
    a.object_name table_name,
    b.segment_type table_type,
    b.bytes / 1024 size_kb,
    sum(c.executions) executions,
    count(distinct a.hash_value) statement_count,
    count(*) reference_count
from
     sys.v_$sql_plan a,
     sys.dba_segments b,
     sys.v_$sql c
where
    a.object_owner (+) = b.owner
    and a.object_name (+) = b.segment_name
    and b.segment_type in ('TABLE', 'TABLE PARTITION')
    and a.operation like '%TABLE%'
    and a.options in ('FULL', 'ALL')
    and a.hash_value = c.hash_value
    and b.bytes / 1024 > 1024
group by
     a.object_owner, a.object_name, a.operation,
     b.bytes / 1024, b.segment_type
order by
     4 desc, 1, 2 );
```

	TABLE_OWNER	TABLE_NAME	TABLE_TYPE	SIZE_KB	STATEMENT_COUNT	REFERENCE_COUNT	EXECUTIONS	TOTAL_SCANS
1	ERADMIN	EMP	TABLE	19456	2	2	2	4
2	ERADMIN	PATIENT	TABLE	3496	1	1	1	1
3	ERADMIN	ADMISSION	TABLE	3136	4	7	31	217

Figure 6. Identifying Tables or Table Partitions That Have Been Scanned in Oracle 9*i*

The above query will help you determine what tables might benefit from better indexing or partitioning. When reviewing such output, you might begin to wonder if the tables being scanned have indexes, and if so, why don't the queries that are scanning the tables make use of them? While only examination of the actual SQL statements can answer the second part of that question, the first part can be answered through the following 9iunused_indx.sql query:

9iunused_indx.sql

```
select distinct
    a.object_owner table_owner,
    a.object_name table_name,
    b.segment_type table_type,
    b.bytes / 1024 size_kb,
    d.index_name
from
     sys.v_$sql_plan a,
     sys.dba_segments b,
     sys.dba_indexes d
where
    a.object_owner (+) = b.owner
    and a.object_name (+) = b.segment_name
    and b.segment_type in ('TABLE', 'TABLE PARTITION')
    and a.operation like '%TABLE%'
    and a.options in ('FULL', 'ALL')
    and b.bytes / 1024 > 1024
    and b.segment_name = d.table_name
    and b.owner = d.table_owner
order by
    1, 2;
```

	TABLE_OWNER	TABLE_NAME	TABLE_TYPE	SIZE_KB	INDEX_NAME
1	ERADMIN	ADMISSION	TABLE	2048	I_ADMISSION1
2	ERADMIN	ADMISSION	TABLE	2048	I_ADMISSION2
3	ERADMIN	PATIENT	TABLE	3072	I_PATIENT1
4	ERADMIN	PATIENT	TABLE	3072	I_PATIENT2
5	ERADMIN	PATIENT	TABLE	3072	I_PATIENT3

Figure 7. Output Showing Unused Indexes for Tables Being Scanned

Such a query can create a mini *unused indexes* report that you can use to ensure that any large tables being scanned on your system have the proper indexing scheme.

Conclusion

By using the techniques and scripts listed in this article to pinpoint and correct SQL submitted by novices, you can hopefully prevent problem SQL from wrecking your otherwise well-performing database. This article is an excerpt from my book *Oracle Performance Troubleshooting* published by Rampant TechPress.

About the Author

Robin Schumacher is vice president of Product Management for Embarcadero Technologies, Inc. Schumacher has over 14 years in the field as a database administrator and developer on the Oracle, Sybase, Microsoft SQL Server, DB2, and Teradata platforms and has authored numerous articles and software reviews for many different database-related publications. You can reach Schumacher at **Robin.Schumacher@Embarcardero.com.**