TEN SURPRISING PERFORMANCE IDEAS

"We All Knew About That!"

Have you ever been surprised by the simplicity of a new performance technique? I have. It causes me to wonder what *other* techniques I have missed. I think a key trait that makes a good Oracle performance analyst is *humility*. We need to be open to ideas that we may have missed. In my experience, good ideas often originate in the heads of pretty junior DBAs.

This year I decided to spend some time exploring new tuning ideas, and try to fill some gaps in my understanding. It's fun exploring techniques that others have successfully used. Some of the ideas discussed here will likely be old-hat to many readers. That's okay. I think, however, most readers will find at least a few innovative ideas.

In this paper I discuss ten performance tuning discoveries/tactics that I recently began using. Some techniques are very simple to use, whereas others require multiple steps. I hope that everyone finds at least a few "nuggets" that prove useful.

Tip 1: Optimizer: Same Plan, Different Plan?

This one really surprised me, and upset some assumptions I had made. See if you are as surprised as I was.

Here's a simple sql I was working on:[1]

```
Select 'X' From Tab1 T1 Where Code = 'xyz'
--
And Exists (Select 'X' From Tab2 T2          --CLAUSE 1
  Where T2.Op_Id = T1.Op_Id And T2.Flg= 'a'
--
And Exists (Select 'X' From Tab2 T2          --CLAUSE 2
  Where T2.Op_Id = T1.Op_Id And T2.Flg= 'b'
--
And Exists (Select 'X' From Tab2 T2          --CLAUSE 3
  Where T2.Op_Id = T1.Op_Id And T2.Flg= 'b';
```

In the sql above, Oracle will first consider the main body, and then the three *Exists* clauses. Let's assume that the optimizer considers Clause 1, then Clause 2, then Clause 3. For the main body, the index used is *Tab1_Index*. For the subqueries, the index is called *Tab2_Index*.

The execution plan is simple—*Tab1* first, followed by the subqueries:

[1] All code listings in this paper have been simplified, with different (and simpler) names used.

```
SELECT STATEMENT
 NESTED LOOPS
  NESTED LOOPS
   NESTED LOOPS
    NESTED LOOPS
     NESTED LOOPS
      TABLE ACCESS BY INDEX ROWID       TAB1
       INDEX RANGE SCAN                 TAB1_INDEX
      TABLE ACCESS  BY INDEX ROWID      TAB2
     INDEX RANGE SCAN                   TAB2_INDEX
    TABLE ACCESS  BY INDEX ROWID        TAB2
   INDEX RANGE SCAN                     TAB2_INDEX
  TABLE ACCESS  BY INDEX ROWID          TAB2
 INDEX RANGE SCAN                       TAB2_INDEX
```

Note that each *Exists* clause queries the same table—*Tab2*. In my actual case, It turns out that the order in which Oracle applies each Exists clause is critical to performance, because one of the clauses drastically reduces the result set. Yet you cannot tell by looking at the execution plan which clause is considered next. No matter which order Oracle applies the subqueries, *the plan will look exactly the same.*

In my particular case, it was critical for Clause 2 be considered before the other *Exists* clauses. That information cannot be gleaned from the execution plan.

Even Weirder

I thought at first that I was making some mistake, and that if I just looked at other columns in Plan_Table, I could detect a difference in the execution plans. Not so--the Plan Hash Value (PHV) is identical in each execution plan, even though the optimizer is doing something different! I suppose this is consistent in a fashion, since the plan details displayed are indeed identical in each case.

Admittedly, having the same PHV for different actions is a rarity. Other DBAs have done considerable research on the subject of plan hash values, and have uncovered many cases where different optimizer action yields the same PHV. Some examples: Sql using different degrees of parallelism, or different filtering criteria.

Tip 2: "A Little Too Active?"

For busy OLTP systems, it's sometimes nice to quickly spot occasional queries that are running more than a few seconds. (For example, when a user omits search criteria.) Note that AWR reports may not be too useful in spotting the occasional OLTP outlier. These queries are typically not listed in AWR, as they consume such a small consumer of database resources. Thus, they often fly "under the radar."

Beginning in Oracle 11g, there is a new field in V$Session, *Sql_Exec_Start*, which we can use to easily spot such problem sessions[2]. Here's a simple script that shows the Sql_Id, SID, and Sql_Text for all sessions running the same sql for over 5 seconds:

```
Select Sid, S.Sql_Id, (Sysdate-Sql_Exec_Start)*24*60*60 SECS,
Sql_Text
From V$Session S, V$Sqltext T
Where S.Sql_Id = T.Sql_Id
And Sql_Exec_Start Is Not Null And Piece = 0
And (Sysdate-Sql_Exec_Start)*24*60*60 > 5 ;
```

```
   SID SQL_ID          SECS SQL_TEXT
------ ------------- ------ --------------------------------------------------
   873 0mhm08114j5dy     15 SELECT /*+RULE*/ DISTINCT TAB1.FIELDX, TAB2. ...
  2937 d8st16qay2g7a    128 SELECT ID FROM PXP WHERE ACCT_ID = :1 AND (
```

I have found it useful to run this "active_time" script occasionally to get a feel for problem queries. Although generally very accurate, there is one caveat to remember when using this script. I have discovered that occasionally, Oracle will be fooled, and think that a session is active when it is not. So far, this quirk seems to only happen when the user is running a tool such as Sql Developer.

TIP 3: Dba_Hist Tables: All Are Equal, But Some Are More Equal than Others

I use the *Dba_Hist* tables very often. Queries on these tables are among the most powerful tools I have for getting to the root of performance issues. I count a total of 126 tables in Oracle 11g, so there's something for everybody. For example, I consider these two particularly valuable:

Dba_Hist_SqlStat: Historical runtime statistics for a particular sql_id.

Dba_Hist_Active_Sess_History: (ASH) History of active sessions.

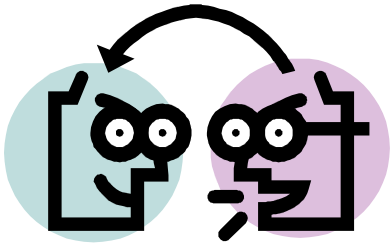
I have included in the appendix a few sample scripts on how to use these tables.
Note that the *Dba_Hist* tables generally do not contain *all* the sql that ran in the past. For some of the tables, only the biggest resource consumers are included. The criteria is controlled by the parameter *Topnsql,* via the package *Dbms_Workload_Repository*. You can see the current settings by querying *Dba_Hist_Wr_Control*.

[2] Thanks to Ken Jordan of PG&E for showing me this.

For example, this command sets the criteria to retain data for 43200 minutes, take a snapshot every 30 minutes, and record the top-100 sql:

```
Exec DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS
(Retention => 43200, Interval => 30, Topnsql => 100;
```

Here's the point I find interesting: The method of sql inclusion is *not the same* across all the *Dba_Hist* tables. At first, this sounds inconsistent, but upon reflection, you will see that it must be so. Here's why: Some *Dba_Hist* tables store metrics for Sql that *in total,* for the snapshot period, exceed certain thresholds. However, the ASH tables don't work that way. They include sql that happen to be caught at a certain time, whilst active. Thus, you might have a very fast sql show up a few times in an Ash table, but not be included in the *Dba_Hist_SqlStat* table. This means that you can sometimes use the Dba_Hist ASH tables to find fast-running sql (if it's been run many times)—even if the cumulative runtime is quite small.



Tip 4: Watch Adaptive Cursor Sharing

This tricky feature is new for Oracle 11g. The purpose of adaptive cursor sharing is to consider multiple execution plans for a given sql, and use past runtime statistics to choose the best plan. The optimizer takes into account current and past bind variables.

So, rather than using “one size fits all” cursor sharing, the optimizer is given the freedom to switch execution plans, given solid empirical evidence to do so. The important point is that Oracle's optimizer may switch plans--even though you have not generated new table statistics, or made any other database change.

With this feature, each cursor is evaluated, and may then be flagged as follows:

BIND-SENSITIVE: This is the first step. The optimizer peeks the bind variable(s), stores the selectivity of the predicate, and also stores the runtime metrics for that predicate. This allows the optimizer to build-up data to be used in future evaluations. Based on these runs, the optimizer decides if an execution plan change should be considered. If so, the cursor is marked *Bind-aware.*

BIND-AWARE: For cursors in the category, the optimizer will compare the selectivity of a new bind value with the stored bind values. If they differ greatly, the optimizer will generate a new execution plan, for potential use in the future. Over time, the optimizer will thus build up a set of selectivity/execution plans from which to choose.

Adaptive Cursor Sharing Scripts

It's a good idea to occasionally monitor adaptive cursor sharing, and check to see if the optimizer is actively switching execution plans. (So far, on our most critical databases, I don't see the optimizer marking any sql as *bind aware*.) Here are some scripts that I find useful in monitoring this new functionality. Most of the objects queried have the letters "CS" in the title.

Adapt.sql: This is probably the best place to start. This script shows you adaptive status of the sql in the shared pool. In particular, shows what sql have been marked as bind sensitive or advanced to the next stage, "aware."

```
Col SENS Format A5
Col AWARE Format A5
Select Sql_Id, Executions, Is_Bind_Sensitive SENS, Is_Bind_Aware AWARE
From V$Sql
Where Is_Bind_Sensitive = 'Y';
```

SQL_ID	EXECUTIONS	SENS	AWARE
g2mc5zm4ph0a0	2940	Y	N
4ydyk19ca00zm	1	Y	N
cfk4qsrg681sz	155	Y	N
dfqx4wrp8h31s	17	Y	N

The following scripts provide more details to show you exactly how the new feature is operating. In some of the scripts, I assume a RAC application, and thus change the view queried from "V" to "GV."

Peeked.sql: This shows you what sql the optimizer is peeking.

```
Select Sql_Id,  Executions, Rows_Processed, Hash_Value
From V$Sql_Cs_Statistics
Where Peeked = 'Y';
```

SQL_ID	EXECUTIONS	ROWS_PROCESSED	HASH_VALUE
14wnf35dahb7v	3	1	1520971003
42cfrr6x5t75c	3	3	3126631596
a2k4qkh681fzx	4	2	209763325
3hzs1jx4uu5fy	3	2	1236080094

Buckets.sql: This shows the distribution of the execution count into the histogram buckets. Here I pick an arbitrary sql_id:

```
Select Sql_Id, Hash_Value, Bucket_Id, Count
From Gv$Sql_Cs_Histogram
Where Sql_Id = '57xxjzx5214nz';
```

SQL_ID	HASH_VALUE	BUCKET_ID	COUNT

```
57xxjzx5214nz 1243648671          0          0
57xxjzx5214nz 1243648671          1         80
57xxjzx5214nz 1243648671          2          0
57xxjzx5214nz 1243648671          0          0
57xxjzx5214nz 1243648671          1         41
57xxjzx5214nz 1243648671          2          4
```

Selectivity.sql: For a given sql_id, this scripts shows the selectivity groupings. This is the information that the optimizer will reference to determine future execution plans.

```
Col Pred Format A8 truncate
Col CHLD format 99999
Col INST format 99
Select Inst_Id INST, Child_Number CHLD, substr(Predicate,1,10) PRED,
Range_Id, Low, High From Gv$Sql_Cs_Selectivity
Where Inst_Id = 2
And Sql_Id = '75y46brtwzyn7';
```

INST	CHLD	PRED	RANGE_ID	LOW	HIGH
2	2	>1	0	0.900000	1.100000
2	2	>1	0	0.900000	1.100000
2	2	>1	0	0.900000	1.100000
2	2	=3	0	0.900000	1.100000
2	2	=2	0	0.029032	0.035484
2	1	>1	0	0.133633	0.163329
2	1	>1	0	0.133633	0.163329
2	1	>1	0	0.133633	0.163329
2	1	=3	0	0.900000	1.100000
2	1	=2	0	0.029032	0.035484

With the above scripts, a DBA can determine whether execution plan changes are being controlled by this new feature. Despite the similarity in names, note that this feature is *not* controlled by the parameter, *Cursor_Sharing*. Adaptive cursor sharing is active no matter what the setting of that parameter.

Tip 5: Examine *Changes* to the Sequential Read Rate

I have long monitored the single-block (in Oracle-ease, "Sequential") read rate. I have found this to be an excellent metric—perhaps my #1 metric.

You can easily summarize this metric by querying gv$System_Event, and looking for "db file sequential read." For example, here we look for the average rate since instance startup for nodes 3-7 in our 8-node RAC cluster:

```
Select Inst_Id, EVENT, TOTAL_WAITS,  TIME_WAITED ,
Round(100*Total_Waits/Time_Waited) Rate,
Round(10* Time_Waited/Total_Waits,1) Latency
From Gv$System_Event
Where Event Like '%db file sequential read%'
And Inst_Id In (3,4,5,6,7)
Order By 1;
```

```
INST_ID EVENT                      TOTAL_WAITS TIME_WAITED       RATE    LATENCY
------- ------------------------ ----------- ----------- ---------- ----------
      3 db file sequential read   1951632651  1051443605        186        5.4
      4 db file sequential read    439614733   226170078        194        5.1
      5 db file sequential read    316738153   160298101        198        5.1
      6 db file sequential read    112183814    68933688        163        6.1
      7 db file sequential read    435084224   225583674        193        5.2
```

In the above output, we see a latency of about 5 ms. In my experience, a typical latency is about 5-10 ms. Note that these are summaries for the time the instance(s) have been running.

But Wait—There's More!

I had occasion recently to investigate inconsistencies in disk read rate. It appeared that we were getting wildly different disk performance at different days/times. Using the above script, however, I can't see how the latency has changed. One option would be to check the AWR report over a period—but that would be very time consuming.

To get more valuable information, we're going to use *Dba_Hist_Filestatxs*. This contains disk i/o information, sorted by filename and Snap_id. For RAC systems, it also includes the *Instance_Number*. Querying this table is a little tricky, because we need to get the "delta" information for a snapshot period—not the cumulative information.

Getting Precise Disk I/O

Using query subfactoring—the "with" syntax, we first find the total disk reads and time for a certain snapshot period and certain RAC node. This will make subsequent steps much simpler. We'll call these metrics ***Megreads*** and ***Ms*** (milliseconds.) Of course, it's not really mandatory to use query subfactoring; I just like breaking things up into bite-sized chunks for easier debugging, and to make the query easier to understand:

```
With S1 as (Select Snap_Id,
Round(Sum(Singleblkrds)/1000000,1) Megreads,
Round(Sum(Singleblkrdtim)*10) Ms
From  Dba_Hist_Filestatxs
Where Snap_Id > TBD
And Instance_Number = TBD
Group By Snap_Id)
```

Now let's change from cumulative to delta using the analytical function "lag" to go back just 1 row, which for us really means go back 1 snapshot.. We use lag twice to get the delta values, ***Totrds*** and ***Tot_Ms.***

```
S2 as ( Select
```

```
Snap_Id, Megreads - Lag(Megreads,1) Over(Order By Snap_Id) Totrds,
Ms- Lag(Ms,1) Over (Order By Snap_Id) Tot_Ms
From S1 )
```

Finally, let's just do a simple calculation to get the metrics we really want. We really only want to see busy periods, so let's filter out periods with less than 1 million reads:

```
Select Snap_Id,  Totrds Megards,
Round(Tot_Ms/Totrds/1000000,1) "Latency(Ms)"
From S2
Where Totrds > 1
```

Our Final Query

Here's what the final query looks like.

```
With S1 As (
Select /*+Parallel(X 10) */  Snap_Id, Round(Sum(Singleblkrds)/1000000,1)
Megreads, Round(Sum(Singleblkrdtim)*10) Ms
From  Dba_Hist_Filestatxs X
Where Snap_Id > 35400  And Instance_Number In (3)
And Upper(Filename) Like '%DB0%'  --optional filter
Group By Snap_Id),
--
S2 As
(Select Snap_Id, Megreads - Lag(Megreads,1)
Over(Order By Snap_Id) Totrds,
Ms- Lag(Ms,1) Over (Order By Snap_Id) Tot_Ms
From S1 )
--
Select Snap_Id, Totrds Megards, Round(Tot_Ms/Totrds/1000000,1)
"Latency(Ms)" From S2
Where Totrds > 1
```

Sample Output

SNAP_ID	MEGARDS	Latency(Ms)
1502	12.9	2.4
1503	74.5	1.1
1504	77.9	1
1505	16.1	1.1
1525	12.8	4.3
1526	28.3	3.7
1527	15.2	3.9
1528	7.6	4.9
1529	17	2.6
1530	12.3	2.3
1531	54.3	3
1532	46.6	3.4
1533	34.3	5.4
1534	73.1	3.3
1535	102	1.5
1536	10	3.1

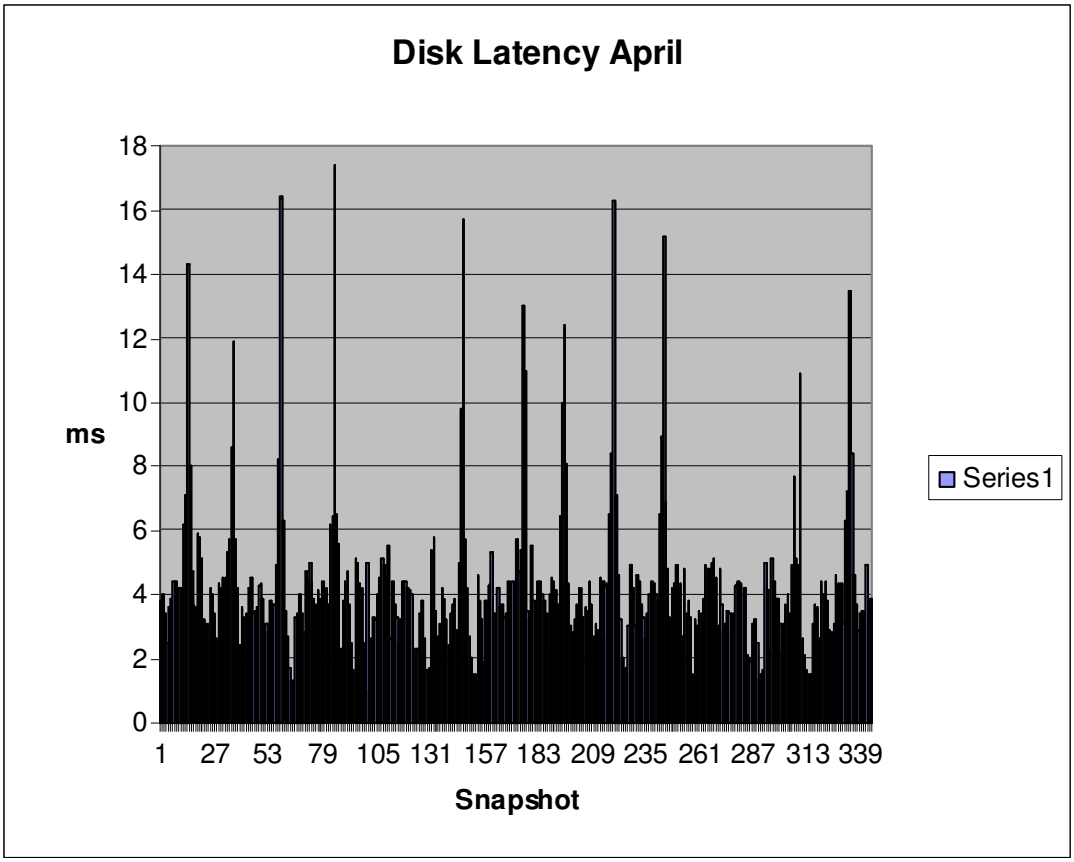
```
1537        12.3          1.4
```

A Few Notes on the Script

Sometimes, I include a filter on certain file names, in case I'm analyzing reads on particular file systems.
To speed things up, I generally use parallelism when querying the *Dba_Hist* tables, due to the large size of these tables on our system (especially monsters containing active session history.).

Charting the Results

To make it easier to spot patterns in the disk metrics, I frequently copy the values into Excel, and make a simple chart. When I used my new script to get historical values for sequential read rate, I obtained the following chart:



With the above chart, I was able to prove conclusively that the disk performance was inconsistent.

Tip 6: Examine Historical Database Load

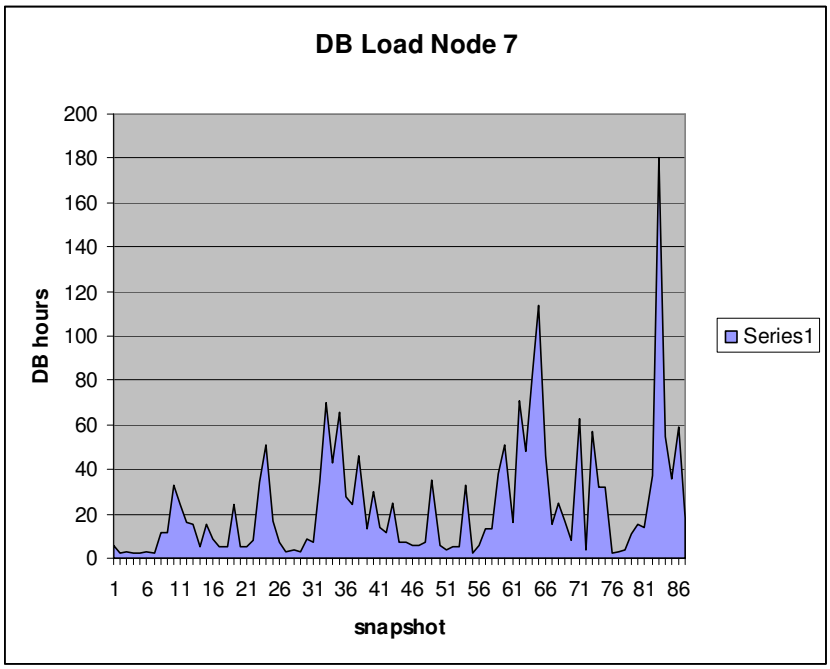
I find it helpful to occasionally check for the overall load on a particular node of our RAC cluster. In order to spot unwelcome patterns, it's important to look at the historical values. Here's a simple way to do that:

```
Col Stat_Name Format A10
Col Node format 999
With P1 As(
Select Snap_Id, Instance_Number NODE, Stat_Name,
Round((Value - Lag(Value,1) Over(Order By Snap_Id))/100/60/60) Dbhrs
From Dba_Hist_Sysstat
Where Snap_Id > 36500
And Instance_Number = 7  And Stat_Name In ('DB time'))
Select * From P1  Order By 1,2;
```

SNAP_ID	NODE	STAT_NAME	DBHRS
36502	7	DB time	6
36503	7	DB time	2
36504	7	DB time	3
36505	7	DB time	2
36506	7	DB time	2
36507	7	DB time	3
36508	7	DB time	2
36509	7	DB time	12
*	*	*	*

Be sure to customize this script for the particular node and snapshot. Note that we again use the "lag" function, since the underlying table holds cumulative statistics.

As with other scripts, I find it helpful to paste the result to into Excel to make a simple chart. In the example below, we have cause for concern about the activity on Node 7:



The next several ideas all originated with my investigation of sql profiles. That's how it is in Oracle—you learn one new approach, and it branches into multiple new ideas.

Tip 7: Consider Sql Profiles

I admit to being skeptical of Oracle's far reaching claims about "automatic" performance tuning features. I still remember the advice proffered by earlier tuning versions, such as, "Be sure you have gathered statistics," or "Try to avoid full table scans." It seems to me that the tools have one thing in common: learning the steps required by a new tool--rather than gaining understanding about the essence of performance tuning.

It's All Automatic!

Oracle's 10g Tuning Guide made especially grandiose claims about the latest performance tuning advisors. If one were to believe the manual, a DBA need only push a magic button:

> "Automatic SQL Tuning is a new capability of the query optimizer that automates the entire SQL tuning process. Using the newly enhanced query optimizer to tune SQL statements, the automatic process replaces manual SQL tuning, which is a complex, repetitive, and time-consuming function."

However, in the 11g Tuning Guide, the former hyperbole has been toned down. At least now, they don't claim that the "entire" process has been automated:

> "Automatic SQL tuning automates the manual process, which is complex, repetitive, and time-consuming."

Enter Sql Profiles

Sql profiles are based on Oracle's latest performance tuning heuristics. I was "forced" to investigate sql profiles because I encountered a query (originating in grid control) that would not work properly with a stored outline. (The difficulty was due to a comment at the very beginning of the query that kept interfering with the outline.) To my pleasant surprise, the sql profile worked the first time, and corrected the plan of the troublesome query.

Sql profiles use an interesting method to improve performance. Sql profiles work because they have the luxury of expending enormous amounts of cpu time investigating different plan options, and gleaning better statistics. They use the same optimizer that presumably produced the poor

execution plan in the first place—but they spend a lot more time to glean more information in order to optimizer the execution path.

The essence of a sql profile is not actually an execution plan per se. Instead, it is really a set of sql hints, stored in the data dictionary, ready to be applied the next time the sql is run. This is similar to stored outlines, but the sql hints are of a different kind.

There is a subtle difference between stored outlines and sql profiles. A stored outline uses a set of sql hints that tend to preserve certain steps in an execution plan (e.g., sql hints such as "full" or "index.") A sql profile, on the other hand uses hints that give the optimizer *extra information that it normally would not have*.

Sample Operation

Let's see how it works. In this example, I have identified a sql statement that runs poorly. It happens to be the sql originated from OEM discussed in an earlier section that checks for sessions blocked.

We must first create a "tuning task." I'll call mine "chris1." The bad performer is sql_id *5k2b3qsy3b30r*.

Step 1: Create the Tuning Task

```
Declare
  L_Sql_Tune_Task_Id  Varchar2(100);
Begin
  L_Sql_Tune_Task_Id := Dbms_Sqltune.Create_Tuning_Task (
  Sql_Id      => '5k2b3qsy3b30r',
  Scope       => Dbms_Sqltune.Scope_Comprehensive,
  Time_Limit  => 60,
  Task_Name   => 'chris1',
  Description => 'chris1 ');
  Dbms_Output.Put_Line('L_Sql_Tune_Task_Id: ' || L_Sql_Tune_Task_Id);
End;
/
```

Note that instead of supplying a sql_id, there's an alternate way to create the tuning task, by supplying the actual sql code, rather than the sql_id, but the above method is much easier.

Step 2: Run The Tuning Task

Let's now run the task I just defined:

```
Begin
  Dbms_Sqltune.Execute_Tuning_Task( Task_Name => 'chris1' );
End;
/
```

Step 3: Get Recommendations

Let's see what Oracle has found:

```
Set Long 9999
Set Longchunksize 1000
Set Linesize 100
Select Dbms_Sqltune.Report_Tuning_Task( 'chris1')
  From Dual;
```

This could be a long report—it will show the sql, and include a summary of recommendations. The part we're interested in looks like this:

```
                * * *
6- SQL Profile Finding (see explain plans section below)
--------------------------------------------------------
  A potentially better execution plan was found for this statement.

  Recommendation (estimated benefit: 67.35%)
  ------------------------------------------
  - Consider accepting the recommended SQL profile.
    execute dbms_sqltune.accept_sql_profile(task_name => 'chris1', replace =>
            TRUE);

                * * *
```

Step 4: Apply The Profile

Okay, let's activate the profile:

```
execute dbms_sqltune.accept_sql_profile(task_name => 'chris1', replace =>
TRUE);
```

Step 5: Confirm Profile is Enabled

Here's a simple way to check which profiles have been created:

```
Col Created Format A30
Select Name, Created, Type, Status
From Dba_Sql_Profiles
Where Last_Modified > Sysdate - 1
```

```
NAME                           CREATED                          TYPE    STATUS
------------------------------ -------------------------------- ------- --------
SYS_SQLPROF_01313de63cc50000   18-JUL-11 08.38.44.000000 AM     MANUAL  ENABLED
```

Behind The Scenes

What in the world is Oracle doing with a sql profile? What kinds of sql hints are being applied?

Here's a simple way[3] to see what types of hints Oracle 11g is using for a certain sql profile, and get a feel for what is happening behind the scenes. In this example, I display the hints that are used for the profile I created above, called *SYS_SQLPROF_01313de63cc50000*.

```
Select Extractvalue(Value(H),'.') As Hint
```

[3] Thanks to Christian Antognini for this useful script.

```
From Sys.Sqlobj$Data Od, Sys.Sqlobj$ So,
Table(xmlsequence(extract(xmltype(od.comp_data),'/outline_data/hint'))) h
WHERE so.name = 'SYS_SQLPROF_01313de63cc50000'
And So.Signature = Od.Signature
And So.Category = Od.Category
And So.Obj_Type = Od.Obj_Type
And So.Plan_Id = Od.Plan_Id

HINT
--------------------------------------------------------------------------------
OPT_ESTIMATE(@"SEL$AF73C875", TABLE, "S"@"SEL$4", SCALE_ROWS=3024)
OPT_ESTIMATE(@"SEL$26", TABLE, "X$KSQEQ"@"SEL$26", SCALE_ROWS=8205128.205)
OPT_ESTIMATE(@"SEL$34", TABLE, "X$KTCXB"@"SEL$34", SCALE_ROWS=164102.5641)
OPT_ESTIMATE(@"SEL$54967A98", TABLE, "S"@"SEL$21", SCALE_ROWS=3024)
OPT_ESTIMATE(@"SEL$54967A98", TABLE, "S"@"SEL$38", SCALE_ROWS=42800)
OPTIMIZER_FEATURES_ENABLE(default)
```

The key point in the above output is the hint *Opt_Estimate*. We can see in the above output that the profile is giving the optimizer statistical information. The purpose of the *Opt_Estimate* hint is to supply cardinality information. The *Scale_Rows* parameter means to scale up (or down) the estimate of the rows to be returned.

Note that Oracle's method of storing profile hints drastically changed from 10g to 11g. The method I illustrated here only works with Oracle 11g.

Sql Profiles Can Handle *Literals*

One interesting facet of sql profiles, in contrast to stored outlines, is that a profile can work even if different literals are used. Of course, this is where a stored outline falls down.

The argument, *Force_Match* should be used to enable this feature, as shown here:

```
execute dbms_sqltune.accept_sql_profile(task_name => 'chris1', -
replace => TRUE, force_match => TRUE);
```

A Disadvantage

One disadvantage of sql profiles—at least for the command line approach—is that the syntax is cumbersome if you need to supply the actual sql text (that is, not just give it the sql_id.) This would be necessary, for instance, if the database has recently started. In that case, you can't start a tuning task for a sql that has never been run. Getting the syntax correct is especially awkward if the sql itself has quotation marks. A stored outline, on the other hand, is trivial to create for a specific sql, regardless of the actual sql text.

Tip 8: Recognize Query Block Hints

A query block identifies a particular area of sql code, which can be distinguished from other parts—such as subqueries. This is useful to know for several reasons. Firstly, query blocks can be used in sql hints so that the hint just applies to certain parts of the sql (i.e., certain query blocks)

Secondly, query block syntax is used behind the scenes when Oracle builds stored outlines or sql profiles. When I first saw the code used in sql profiles, I was confused by the query block syntax. I didn't know what it was, but it's not that complicated. Once you understand the format of query blocks, the hints that you see in the stored outline or profile views will make sense.

A Simple Example

Let's take a look at a simple sql statement to understand the concept. Consider the following query:

```
Select Ename From Scott.Emp Where Empno = 123;
```

There's really only one part to this query—so we would expect just one main query block. Let's do a simple *explain plan* query, but include this new field. The query block is available in the *Plan_Table*, column *Qblock_Name*.

```
Col Qblock_Name Format A15
Col Operation Format A20
Select Qblock_Name,     Id, Parent_Id,
Lpad (' ', Level - 1) || Operation || ' ' ||
           Options Operation, Object_Name
From       Plan_Table
Where      Statement_Id = '&Stmt_Id'
Start With Id = 0
And        Statement_Id = '&Stmt_Id'
Connect By Prior
           Id = Parent_Id
And        Statement_Id = '&Stmt_Id';
```

```
QBLOCK_NAME       ID PARENT OPERATION            OBJECT_NAME
--------------- ---- ------ -------------------- ---------------
                  0        SELECT STATEMENT
SEL$1             1      0  TABLE ACCESS BY IND EMP
SEL$1             2      1   INDEX UNIQUE SCAN  PK_EMP
```

In the above results, note that a query block name, when automatically given by the optimizer, is of the form "SEL $n." So in our example, Oracle has called it, *SEL$1*. Note that the query block number assigned by Oracle will not always be so simple as 1,2,3, etc.

Query Blocks Everywhere

Here's an example that shows a lot of query blocks.

```
Select Ename From Scott.Emp Where Empno = 123
and Empno in (Select Object_id from Dba_Objects);
```

Using our explain plan script above, we see what is actually happening behind the scenes:

```
QBLOCK_NAME       ID PARENT OPERATION                    OBJECT_NAME
--------------- ---- ------ ------------------------ --------------
                  0        SELECT STATEMENT
SEL$5DA710D3      1      0  NESTED LOOPS SEMI
SEL$5DA710D3      2      1   TABLE ACCESS BY INDEX R EMP
SEL$5DA710D3      3      2    INDEX UNIQUE SCAN      PK_EMP
SEL$683B0107      4      1   VIEW                    VW_NSO_1
SET$1             5      4    VIEW                   DBA_OBJECTS
SET$1             6      5     UNION-ALL
SEL$24D5D062      7      6      FILTER
                  8      7       NESTED LOOPS
                  9      8        NESTED LOOPS
SEL$24D5D062     10      9         TABLE ACCESS BY I OBJ$
SEL$24D5D062     11     10          INDEX RANGE SCAN I_OBJ1
SEL$24D5D062     12      9         INDEX RANGE SCAN  I_USER2
SEL$24D5D062     13      8        INDEX RANGE SCAN   I_USER2
SEL$8            14      7       TABLE ACCESS BY IND IND$
SEL$8            15     14        INDEX UNIQUE SCAN  I_IND1
SEL$7            16      7       NESTED LOOPS
SEL$7            17     16        INDEX RANGE SCAN   I_OBJ4
SEL$7            18     16        INDEX RANGE SCAN   I_USER2
SEL$9            19      6      FILTER
                 20     19       HASH JOIN
SEL$9            21     20        INDEX FULL SCAN    I_LINK1
SEL$9            22     20        INDEX FAST FULL SC I_USER2
```

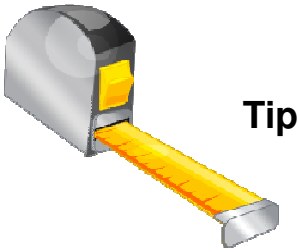
Although the sql is short, my reference to *Dba_Objects* means a lot of different query blocks:

Query Blocks in Sql Profiles

Here's an example of how query blocks are used in sql profiles. The code below (actually a sql hint) is from a sql profile:

```
OPT_ESTIMATE(@"SEL$AF73C875", TABLE, "S"@"SEL$4", SCALE_ROWS=3024)
```

We can now understand what this hint means. We see that the hint references two different query blocks— *SEL$AF73C875* and *SEL$4*. (The *Scale_Rows* parameter means to scale up (or down) the estimate of the rows to be returned.)



Tip 9: Get Familiar with *Extended Hints*

Extended hints resemble normal sql hints—but they are more cryptic-looking. Here's are some examples of extended sql hints. These hints were used in some stored outlines I had created[4]:

```
USE_NL(@"SEL$1" "ABNQ"@"SEL$1")
LEADING(@"SEL$1" "RQ4"@"SEL$1" "ABO"@"SEL$1" "V1D"@"SEL$1" "XBL"@"SEL$1")
```

At first, the syntax looks bewildering, but based on what we learned about query blocks, we can get the gist of what the hints are doing, Most importantly, we know that the "@SEL$1" syntax is referring to a query block (explained in the prior section.) So most of the hints are being applied to query block *SEL$1*. So the optimizer is being instructed to join the tables (identified by their respective aliases) in the order shown.

If we look at the actual sql for the outline, we do indeed see those table aliases contained:

```
Select [List Of Fields]
From Vi_Data_Rgaf     V1d,
Xvi_Md_Tbl            Xbl,
Abo_Aoot_Obj          Abo,
Vmti_Md_Cco_Tbl       Rq4
Where
* * *
```

Clearly, the hint 's purpose is to ensure a certain join order. So, we can see that the extended hint makes sense.

Hints Similar, But a Little Different

In the prior section I illustrated some hints that were used in a sql profile:

```
OPT_ESTIMATE(@"SEL$AF73C875", TABLE, "S"@"SEL$4", SCALE_ROWS=3024)
```

In this example, the optimizer is being instructed via the *Scale_Rows* keyword to change its cardinality estimate for a table by 3024. That is, the *Scale_Rows* hint works similar to the *cardinality* hint—but uses a scaling factor instead.

[4] I have changed and simplified the table names and aliases

Finding Extended Hints

Here's a way to see extended hints for particular sql of interest. In the *V$Sql* view, the field *Other_Xml* contains this hint information.

```
select extractvalue(value(d), '/hint') as Ext_Hint
from xmltable('/*/outline_data/hint'
  passing (
    select xmltype(other_xml) as xmlval
    from v$sql_plan
    where sql_id='3rwnb6n8mj3tp'   -- for example
    and child_number = 0           -- for example
      and other_xml IS NOT NULL )) d
/

EXT_HINT
--------------------------------------------
IGNORE_OPTIM_EMBEDDED_HINTS
OPTIMIZER_FEATURES_ENABLE('11.2.0.2')
DB_VERSION('11.2.0.2')
OPT_PARAM('optimizer_index_cost_adj' 1)
OPT_PARAM('optimizer_index_caching' 100)
```

Tip 10: Try Pushing Subqueries

For this last tip, I have saved a commentary on a unusual sql hint—one I seldom see used. This hint (and its counterpart) has always confused me, and I bet others have been confused as well. I hope this section clears things up.

I occasionally run into queries having multiple tables and clauses, in which I need the optimizer to evaluate certain clauses as early as possible—before joining to the other tables**.** Oracle provides a hint for doing this: *Push_Subq.* Naturally, there is also an inverse hint, *No_Push_Subq.* Accordingly to Oracle documentation, *No_Push_Subq* instructs the optimizer to evaluate the clause as late as feasible.[5]

In Oracle hints, the term "push" is an awkward choice, and has always confused me. Here's the key: All it really means is, evaluate a section of code as *early as possible*. In other words, *push* it to the *front of the line*. Note that in Oracle versions prior to Oracle 10, the *Push_Subq* hint was applied to the entire sql statement. Beginning in Oracle 10, you can selectively decide which parts to push or not push.

[5] The hint name, *No_Push_Subq*, is unfortunate, and is not consistent with its function. Actively moving something to the end of the line is not the same as *not pushing*.

Sample Query

Let's see how this works in practice I first create three simple tables. For purposes of our illustration, it is not important that they actually contain any data:

```
Create table parent  (id1 number, col2 number);
Create table child   (id1 number, col2 number);
Create table subtest (id1 number, col2 number, col3 number);
```

Now, let's create a query that has a main portion, and a subquery, that may be evaluated earlier, or later.

```
select par.col2,  chi.col2
from parent  par, child   chi
where par.id1 between 1001 and 2200
and chi.id1 = par.id1
and exists (
        select /*+ No_Unnest Qb_Name(Subq1) */
         'x' from    subtest sub1
        where  sub1.id1       = par.id1
        and sub1.col2 = par.col2
        and sub1.col3 >= '2')
```

In the query above, note that the optimizer has an option on when our subquery, *Subq1*, should be applied. It can join the parent and child tables, and then consider the clause afterwards; or it can evaluate the subquery earlier.

Also note that we included two hints, *No_Unnest*, and *Qb_Name*. The hint *No_Unnest* is used to prevent the optimizer from merging the subquery into the main body. This is a requirement for the *Push_Subq* hint to work.

Base Plan[6]

In the base plan, without any "pushing" hints, the optimizer considers the subquery last.

```
QBLOCK_NAME     OPERATION                 OBJECT_NAME
--------------- ------------------------- --------------
                SELECT STATEMENT
SEL$1            FILTER
                  HASH JOIN
SEL$1              TABLE ACCESS FULL      PARENT
SEL$1              TABLE ACCESS FULL      CHILD
SUBQ1              TABLE ACCESS FULL      SUBTEST
```

Notice that Oracle calls the subquery *SUBQ1*—exactly what we named it with our hint.

There is one subtle, but very critical line in this plan--the *Filter* operation. This indicates that the subquery remains unnested—it hasn't been integrated into the main body of the sql. Again, that is mandatory for out pushing hint to be accepted by the optimizer.

[6] The script I use to show the execution plan with query block name is shown in the appendix.

Try Pushing

Now, let's assume we want the subquery to be evaluated soon. Let's apply the *Push_Subq* hint and see what happens.

```
select /*+push_subq(@subq1) */
    par.col2,  chi.col2
from
    parent  par,
    child   chi
where par.id1 between 1001 and 2200
and chi.id1 = par.id1
and exists (
        select /*+ no_unnest qb_name(subq1) */   'x'
        from    subtest sub1
        where  sub1.id1       = par.id1
        and sub1.col2 = par.col2
        and sub1.col3 >= '2'
    )
```

As expected, we see that the subquery has indeed moved up in the order.

```
QBLOCK_NAME    OPERATION                      OBJECT_NAME
-------------- ------------------------------ ------------
               SELECT STATEMENT
SEL$1           HASH JOIN
SEL$1             TABLE ACCESS FULL            PARENT
SUBQ1              TABLE ACCESS FULL           SUBTEST
SEL$1             TABLE ACCESS FULL            CHILD
```

Try Pushing Back

Let's now switch focus; instead of pushing a subquery earlier, we are going to push it *later*. In practice, I have found it much easier to find a test case for using *No_Push_Subq*, than for *Push_Subq*—possibly due to the *Filter/unnesting* requirement discussed above.

This example uses the *Scott* schema and a very simple sql. Once again, we will use query block naming to put a label on each clause. In this example, I use the imaginative query block names, *Clause1* and *Clause2*.

```
Select E.Mgr, D.Loc from Emp E, Dept D
Where E.DeptNo = D.DeptNo
--
And E.Sal in (Select /*+QB_NAME(CLAUSE1) */  Max(Sal) from Bonus)
And E.Sal in (Select /*+QB_NAME(CLAUSE2) */ Max(HiSal) from Salgrade)
```

Base Plan

The base plan in this test case is interesting, because the subqueries are at the *front*. We can also see that our query block naming is successful, as shown in bold below. Oracle has generated the query block name, SEL$1 as the label for the main body of our query.

```
QBLOCK_NAME  OPERATION                          OBJECT_NAME
------------ --------------------------------- ------------
             SELECT STATEMENT
SEL$1         NESTED LOOPS
               NESTED LOOPS
SEL$1           TABLE ACCESS FULL              EMP
CLAUSE1          SORT AGGREGATE
CLAUSE1           TABLE ACCESS FULL             BONUS
CLAUSE2          SORT AGGREGATE
CLAUSE2           TABLE ACCESS FULL             SALGRADE
SEL$1           INDEX UNIQUE SCAN              PK_DEPT
SEL$1          TABLE ACCESS BY INDEX ROWID     DEPT
```

Now Do The Push

Now let's see if we can rearrange when the subqueries are considered. Let's try moving processing of *Clause1* to the end of the line, using *No_Push*. (Remember that in Oraclespeak, *No_Push* really means "Push it to the end.") Note the "@" symbol in referencing the subblock. With the hints, we now have this code:

```
Select /*+NO_PUSH_SUBQ(@CLAUSE1) */
E.Mgr, D.Loc from Emp E, Dept D
Where E.DeptNo = D.DeptNo
--
And E.Sal in (Select /*+QB_NAME(CLAUSE1) */  Max(Sal) from Bonus)
And E.Sal in (Select /*+QB_NAME(CLAUSE2) */ Max(HiSal) from Salgrade)
```

Once again querying *Plan Table,* we see that the *No_Push_Subq* hint caused Oracle to move *Clause1* to the end:

```
QBLOCK_NAME          OPERATION                     OBJECT_NAME
-------------------- ----------------------------- ---------------
                     SELECT STATEMENT
SEL$1                 FILTER
                       NESTED LOOPS
                        NESTED LOOPS
SEL$1                    TABLE ACCESS FULL        EMP
CLAUSE2                   SORT AGGREGATE
CLAUSE2                    TABLE ACCESS FULL       SALGRADE
SEL$1                     INDEX UNIQUE SCAN        PK_DEPT
SEL$1                    TABLE ACCESS BY INDEX RO  DEPT
CLAUSE1                  SORT AGGREGATE
CLAUSE1                   TABLE ACCESS FULL        BONUS
```

A Second Push

Similarly, can we move *Clause 2* to the end? We'll just change the hint to this:

```
Select /*+NO_PUSH_SUBQ(@CLAUSE2) */
```

Here is the new plan. We can see that Oracle has moved Clause2 to the end, as expected.

```
QBLOCK_NAME          OPERATION                          OBJECT_NAME
-------------------- ---------------------------------- --------------
                     SELECT STATEMENT
SEL$1                 FILTER
                       NESTED LOOPS
                        NESTED LOOPS
SEL$1                    TABLE ACCESS FULL              EMP
CLAUSE1                   SORT AGGREGATE
CLAUSE1                    TABLE ACCESS FULL            BONUS
SEL$1                     INDEX UNIQUE SCAN             PK_DEPT
SEL$1                     TABLE ACCESS BY INDEX ROWID   DEPT
CLAUSE2                   SORT AGGREGATE
CLAUSE2                    TABLE ACCESS FULL            SALGRADE
```

SUMMARY: DID YOU *REALLY* KNOW ALL THESE TIPS?

I hope you found this to be a fun excursion through these tips. I learned a lot while researching these ideas; I hope you have picked-up a few notions that will help you solve your performance puzzles.

Here are the ten tips again:

1. Oracle can claim *same* plan but actually be a *different* plan
2. Easily spot long-running sessions with a new column in 11g
3. *DBA_Hist* tables use different approach to store data
4. Recognize impact of new feature, *Adaptive Cursor Sharing*
5. Watch *changes* to sequential read rate
6. Measure and graph historical database load
7. Try *sql profiles* as alternative to stored outlines
8. Recognize format of *query block* hints
9. Be familiar with *extended sql* hints
10. See how to *push* subqueries back and forth

BE WILLING TO SHARE IDEAS

Sharing good tips is a hallmark of a top DBA, and we all should be open to sharing our best ideas, and be open to learning from others. I hope to hear from *you* soon. You may contact me at: Chris@OracleMagician.com.

REFERENCES:

I relied quite a bit on research from others. Thanks to the following authors:

On Execution plans:

- Eingestellt von Randolph *How Stable are your Executions Plans?*
- Riyaj Shamsudeen, *Is Plan Hash Value a Final Say?*

On 11g new performance features

- Karl Reitschuster, *Oracle 11g : A deeper granularity level to SQL performance metrics*

On using sql profiles:

- Christian Antognini, *SQL Profiles*
- Intermediatesql.com, *How to Add Hint to Query without Touching its Text*
- Eingestellt von Randolf, *Plan stability in 10g - using existing cursors to create Stored Outlines and SQL profiles*

On use of the *Opt_Estimate* hint:

- Christo Kutrovsky *Oracle's Opt_Estimate Hint: Usage guide*

On use of query blocks and the Push_Subq hint:

- Jonathan Lewis, *http://jonathanlewis.wordpress.com/2007/03/09/push_subq*
- Vazha Mantua, http://ocp.community.ge/post/no_unnest-hint-useful-hint-for-subqueries!.aspx

SCRIPTS

Here are some other scripts that I reference in the text.

Old_Sql_Stats.sql

```
Col Beg Format A20
Col Inst Format 999
Select S.Snap_Id,To_Char(Begin_Interval_Time, 'Dd-Mon-Yy-Hh24:Mi') Beg,
S.Instance_Number Inst, Executions_Delta EXECS,
Rows_Processed_Delta ROWSP,
Round(Elapsed_Time_Delta/1000000/60) Totalmins,
Round(Elapsed_Time_Delta/1000/(Executions_Delta+.01)) Pertimems
From Dba_Hist_Sqlstat S, Dba_Hist_Snapshot T
Where Sql_Id = [sql_id]
And S.Instance_Number = T.Instance_Number
And S.Snap_Id = T.Snap_Id
And Executions_Delta > 1
Order By 1;
```

Ash.sql

```
With P1 As (Select /*+Parallel(A 8) */    Distinct A.*
From Dba_Hist_Active_Sess_History A
Where Snap_Id Between 35006 And 35007
And Instance_Number In (3,4,5,6,7)
And Sql_Id = [sql_id]
) Select Sample_Time, Instance_Number, Sql_Exec_Id,Current_Obj#,
Current_Block#, Current_File#   Order By 1
```

Showqueryblock.sql

```
SET VERIFY OFF
SET PAGESIZE 100
ACCEPT stmt_id CHAR PROMPT "Enter statement_id: "
COL id          FORMAT 999
COL parent_id   FORMAT 999 HEADING "PARENT"
COL operation   FORMAT a45 TRUNCATE
COL object_name FORMAT a30

Select     qblock_name, id, parent_id, LPAD (' ', LEVEL - 1) || operation
|| ' ' ||
           options operation, object_name
From       plan_table
Where      statement_id = '&stmt_id'
START WITH id = 0
And        statement_id = '&stmt_id'
CONNECT BY PRIOR
           id = parent_id
AND        statement_id = '&stmt_id';
```

* * *

Chris Lawson is an Oracle *Ace* and performance specialist in the San Francisco Bay Area. He is the author of *The Art & Science of Oracle Performance Tuning*, as well as *Snappy Interviews: 100 Questions to Ask Oracle DBAs*. When he's not solving performance problems, Chris is an avid hiker and geocacher, where he is known as *Bassocantor*. Chris can be reached at, Chris@OracleMagician.com