

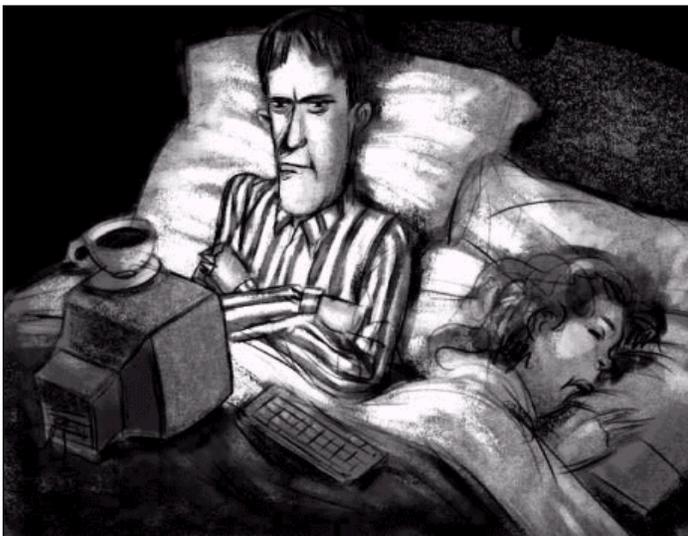
## Oracle Instance Tuning Techniques

Oracle professionals know that you must optimizer your database by tuning global parameters before detailed application tuning can proceed. This excerpt reviews proven techniques for tuning any Oracle instance and has scripts to ensure that your database is optimized for its application load.

This is an excerpt from the bestselling book “Oracle Tuning: The Definitive Reference” ([http://www.rampant-books.com/book\\_2005\\_1\\_awr\\_proactive\\_tuning.htm](http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm)) by Alexey Danchenkov and Donald Burleson, technical editor Mladen Gogala. To supplement the script

### Viewing table and index access with AWR

One of the problems in Oracle9i was the single bit-flag that was used to monitor index usage. The flag can be set with the *alter index xxx monitoring usage* command, and see if the index was accessed by querying the *v\$object\_usage* view.



The goal of any index access is to use the most selective index for a query. This would be the one that produces the smallest number of rows. The Oracle data dictionary is usually quite good at this, but it is up to the DBA to define the index. Missing function-based indexes are a common source of suboptimal SQL execution because Oracle will not use an indexed column unless the WHERE clause matches the index column exactly.

The [WISE tool](http://www.wise-oracle.com/product_wise_enterprise.htm) ([http://www.wise-oracle.com/product\\_wise\\_enterprise.htm](http://www.wise-oracle.com/product_wise_enterprise.htm)) is a great way to quickly plot Oracle time series data and gather signatures for Oracle metrics. The figure below shows how the WISE tool displays this data. WISE is also able to plot performance data on daily or monthly average basis. See <http://www.wise-oracle.com> for details.

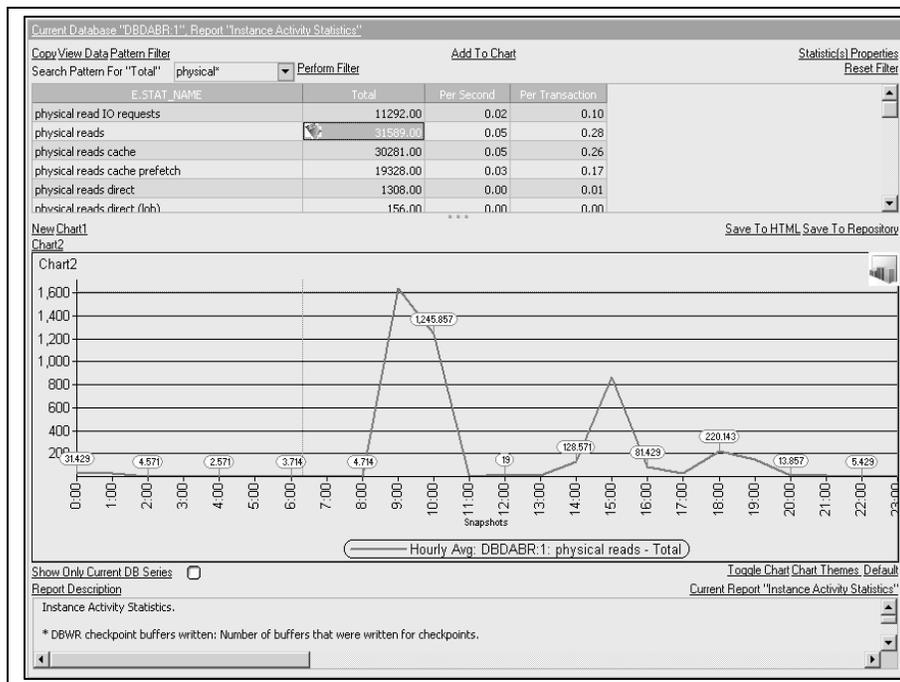


Figure Output from the [WISE viewer](http://www.wise-oracle.com/product wise enterprise.htm) (<http://www.wise-oracle.com/product wise enterprise.htm>)

## Tracking SQL nested loop joins

As a review, nested loop joins are the most common method for Oracle to match rows in multiple tables. Nested loop joins always invoke an index and they are never parallelized. The following *awr\_nested\_join\_alert.sql* script to count nested loop joins per hour:

### **awr\_nested\_join\_alert.sql**

```
col c1 heading 'Date'                format a20
col c2 heading 'Nested|Loops|Count'  format 99,999,999
col c3 heading 'Rows|Processed'      format 99,999,999
col c4 heading 'Disk|Reads'          format 99,999,999
col c5 heading 'CPU|Time'             format 99,999,999

accept nested_thr char prompt 'Enter Nested Join Threshold: '

title 'Nested Join Threshold|&nested_thr'

select
  to_char(sn.begin_interval_time, 'yy-mm-dd hh24') c1,
  count(*) c2,
  sum(st.rows_processed_delta) c3,
  sum(st.disk_reads_delta) c4,
  sum(st.cpu_time_delta) c5
from
  dba_hist_snapshot sn,
```

```

    dba_hist_sql_plan p,
    dba_hist_sqlstat st
where
    st.sql_id = p.sql_id
and
    sn.snap_id = st.snap_id
and
    p.operation = 'NESTED LOOPS'
having
    count(*) > &hash_thr
group by
    begin_interval_time;
SEE CODE DEPOT FOR MORE SCRIPTS
http://www.rampant-books.com/book\_2005\_1\_awr\_proactive\_tuning.htm

```

The output below shows the number of total nested loop joins during the snapshot period along with a count of the rows processed and the associated disk I/O. This report is useful where the DBA wants to know if increasing *pga\_aggregate\_target* will improve performance.

Nested Loop Join Thresholds

Date	Nested Loops Count	Rows Processed	Disk Reads	CPU Time
04-10-10 16	22	750	796	4,017,301
04-10-10 17	25	846	6	3,903,560
04-10-10 19	26	751	1,430	4,165,270
04-10-10 20	24	920	3	3,940,002
04-10-10 21	25	782	5	3,816,152
04-10-11 02	26	905	0	3,935,547
04-10-11 03	22	1,001	0	3,918,891
04-10-11 04	29	757	8	3,939,071

In the report above, nested loops are favored by SQL that returns a small number of *rows\_processed* than hash joins, which tend to return largest result sets.

The following *awr\_sql\_index.sql* script exposes the cumulative usage of database indexes:

### **awr\_sql\_index.sql**

```

col c0 heading 'Begin|Interval|time' format a8
col c1 heading 'Index|Name' format a20
col c2 heading 'Disk|Reads' format 99,999,999
col c3 heading 'Rows|Processed' format 99,999,999
select
    to_char(s.begin_interval_time,'mm-dd hh24') c0,
    p.object_name c1,
    sum(t.disk_reads_total) c2,
    sum(t.rows_processed_total) c3
from
    dba_hist_sql_plan p,
    dba_hist_sqlstat t,
    dba_hist_snapshot s
where
    p.sql_id = t.sql_id
    and
    t.snap_id = s.snap_id
    and
    p.object_type like '%INDEX%'
group by
    to_char(s.begin_interval_time,'mm-dd hh24'),

```

```

    p.object_name
order by
    c0,c1,c2 desc
;
SEE CODE DEPOT FOR MORE SCRIPTS
http://www.rampant-books.com/book\_2005\_1\_awr\_proactive\_tuning.htm

```

The following is a sample of the output where the stress on every important index is shown over time. This information is important for placing index blocks into the KEEP pool to reduce disk reads and for determining the optimal setting for the important *optimizer\_index\_caching* parameter.

```

Begin
Interval Index          Disk      Rows
time      Name            Reads     Processed
-----
10-14 12 I_CACHE_STATS_1          114
10-14 12 I_COL_USAGE$           201      8,984
10-14 12 I_FILE1                 2         0
10-14 12 I_IND1                93        604
10-14 12 I_JOB_NEXT            1     247,816
10-14 11 I_KOPM1                4      2,935
10-14 11 I_MON_MODS$_OBJ       12     28,498
10-14 11 I_OBJ1              72,852      604
10-14 11 I_PARTOBJ$           93        604
10-14 11 I_SCHEDULER_JOB2      4         0
10-14 11 SYS_C002433          302     4,629
10-14 11 SYS_IOT_TOP_8540       0     75,544
10-14 11 SYS_IOT_TOP_8542       1     4,629
10-14 11 WRH$_DATAFILE_PK        2         0
10-14 10 WRH$_SEG_STAT_OBJ_PK    93        604
10-14 10 WRH$_TEMPFILE_PK        0         0
10-14 10 WRI$_ADV_ACTIONS_PK     38     1,760

```

The above report shows the highest impact tables.

The following *awr\_sql\_index\_access.sql* script will summarize index access by snapshot period.

#### **awr\_sql\_index\_access.sql**

```

col c1 heading 'Begin|Interval|Time'   format a20
col c2 heading 'Index|Range|Scans'     format 999,999
col c3 heading 'Index|Unique|Scans'    format 999,999
col c4 heading 'Index|Full|Scans'      format 999,999

select
  r.c1  c1,
  r.c2  c2,
  u.c2  c3,
  f.c2  c4
from
  (
select
  to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
  count(1) c2
from
  dba_hist_sql_plan p,
  dba_hist_sqlstat s,
  dba_hist_snapshot sn
where

```

```

    p.object_owner <> 'SYS'
and
    p.operation like '%INDEX%'
and
    p.options like '%RANGE%'
and
    p.sql_id = s.sql_id
and
    s.snap_id = sn.snap_id
group by
    to_char(sn.begin_interval_time,'yy-mm-dd hh24')
order by
1 ) r,
(
select
    to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
    count(1) c2
from
    dba_hist_sql_plan p,
    dba_hist_sqlstat s,
    dba_hist_snapshot sn
where
    p.object_owner <> 'SYS'
and
    p.operation like '%INDEX%'
and
    p.options like '%UNIQUE%'
and
    p.sql_id = s.sql_id
and
    s.snap_id = sn.snap_id
group by
    to_char(sn.begin_interval_time,'yy-mm-dd hh24')
order by
1 ) u,
(
select
    to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
    count(1) c2
from
    dba_hist_sql_plan p,
    dba_hist_sqlstat s,
    dba_hist_snapshot sn
where
    p.object_owner <> 'SYS'
and
    p.operation like '%INDEX%'
and
    p.options like '%FULL%'
and
    p.sql_id = s.sql_id
and
    s.snap_id = sn.snap_id
group by
    to_char(sn.begin_interval_time,'yy-mm-dd hh24')
order by
1 ) f
where
    r.c1 = u.c1
and
    r.c1 = f.c1
;

```

[SEE CODE DEPOT FOR MORE SCRIPTS](http://www.rampant-books.com/book_2005_1_aws_proactive_tuning.htm)  
[http://www.rampant-books.com/book\\_2005\\_1\\_aws\\_proactive\\_tuning.htm](http://www.rampant-books.com/book_2005_1_aws_proactive_tuning.htm)

The sample output below shows those specific times when the database performs unique scans, index range scans and index fast full scans:

Begin Interval Time	Index Range Scans	Index Unique Scans	Index Full Scans
04-10-21 15	36	35	2
04-10-21 19	10	8	2
04-10-21 20		8	2
04-10-21 21		8	2
04-10-21 22	11	8	3
04-10-21 23	16	11	3
04-10-22 00	10	9	1
04-10-22 01	11	8	3
04-10-22 02	12	8	1
04-10-22 03	10	8	3
04-10-22 04	11	8	2
04-10-22 05		8	3
04-10-22 06		8	2
04-10-22 07	10	8	3
04-10-22 08		8	2
04-10-22 09		8	2

SQL object usage can also be summarized by day-of-the-week:

### **awr\_sql\_object\_avg\_dy.sql**

```
col c1 heading 'Object|Name'          format a30
col c2 heading 'Week Day'            format a15
col c3 heading 'Invocation|Count'    format 99,999,999

break on c1 skip 2
break on c2 skip 2

select

decode(c2,1,'Monday',2,'Tuesday',3,'Wednesday',4,'Thursday',5,'Friday',6,'Saturday',7,
'Sunday') c2,
c1,
c3
from
(
select
p.object_name          c1,
to_char(sn.end_interval_time,'d') c2,
count(1)              c3
from
dba_hist_sql_plan  p,
dba_hist_sqlstat   s,
dba_hist_snapshot  sn
where
p.object_owner <> 'SYS'
and
p.sql_id = s.sql_id
and
s.snap_id = sn.snap_id
group by
p.object_name,
to_char(sn.end_interval_time,'d')
order by
c2,c1
)
;
SEE CODE DEPOT FOR MORE SCRIPTS
http://www.ramport-books.com/book\_2005\_1\_awr\_proactive\_tuning.htm
```

The output below shows the top objects within the database during each snapshot period.

Week Day	Object Name	Invocation Count
Monday	CUSTOMER	44
	CUSTOMER_ORDERS	44
	CUSTOMER_ORDERS_PRIMARY	44
	MGMT_CURRENT_METRICS_PK	43
	MGMT_FAILOVER_TABLE	47
	MGMT_JOB	235
	MGMT_JOB_EMD_STATUS_QUEUE	91
	MGMT_JOB_EXECUTION	235
	MGMT_JOB_EXEC_IDX01	235
	MGMT_JOB_EXEC_SUMMARY	94
	MGMT_JOB_EXEC_SUMM_IDX04	94
	MGMT_JOB_PK	235
	MGMT_METRICS	65
	MGMT_METRICS_1HOUR_PK	43
Tuesday	CUSTOMER	40
	CUSTOMER_CHECK	2
	CUSTOMER_PRIMARY	1
	CUSTOMER_ORDERS	46
	CUSTOMER_ORDERS_PRIMARY	46
	LOGMNR_LOG\$	3
	LOGMNR_LOG\$_PK	3
	LOGSTDBY\$PARAMETERS	2
	MGMT_CURRENT_METRICS_PK	31
	MGMT_FAILOVER_TABLE	42
	MGMT_JOB	200
	MGMT_JOB_EMD_STATUS_QUEUE	78
	MGMT_JOB_EXECUTION	200
	MGMT_JOB_EXEC_IDX01	200
	MGMT_JOB_EXEC_SUMMARY	80
	MGMT_JOB_EXEC_SUMM_IDX04	80
MGMT_JOB_PK	200	
MGMT_METRICS	48	
Wednesday	CURRENT_SEVERITY_PRIMARY_KEY	1
	MGMT_CURRENT_METRICS_PK	17
	MGMT_CURRENT_SEVERITY	1
	MGMT_FAILOVER_TABLE	24
	MGMT_JOB	120
	MGMT_JOB_EMD_STATUS_QUEUE	46
	MGMT_JOB_EXECUTION	120
	MGMT_JOB_EXEC_IDX01	120
	MGMT_JOB_EXEC_SUMMARY	48
	MGMT_JOB_EXEC_SUMM_IDX04	48
	MGMT_JOB_PK	120
	MGMT_METRICS	36
	MGMT_METRICS_1HOUR_PK	14
	MGMT_METRICS_IDX_01	24
MGMT_METRICS_IDX_03	1	
MGMT_METRICS_PK	11	

When these results are posted, the result is a well-defined signature that emerges for particular tables, access plans and SQL statements. Most Oracle databases are remarkably predictable, with the exception of DSS and ad-hoc query systems, and the DBA can quickly track the usage of all SQL components.

Understanding the SQL signature can be extremely useful for determining what objects to place in the KEEP pool, and to determining the most active tables and indexes in the database.

Once a particular SQL statement for which details are desired has been identified, it is possible to view its execution plan used by optimizer to actually execute the statement. The query below retrieves an execution plan for a particular SQL statement of interest:

### awr\_sql\_details.sql

```

accept sqlid prompt 'Please enter SQL ID: '

col c1 heading 'Operation'          format a20
col c2 heading 'Options'            format a20
col c3 heading 'Object|Name'        format a25
col c4 heading 'Search Columns'     format 999,999
col c5 heading 'Cardinality'        format 999,999

select
  operation          c1,
  options            c2,
  object_name        c3,
  search_columns     c4,
  cardinality        c5
from
  dba_hist_sql_plan p
where
  p.sql_id = '&sqlid'
order by
  p.id;
SEE CODE DEPOT FOR MORE SCRIPTS
http://www.rampant-books.com/book\_2005\_1\_awr\_proactive\_tuning.htm

```

This is one of the most important of all of the SQL tuning tools. Here is a sample of the output from this script:

Operation	Options	Name	Search Cols	Cardinality
SELECT STATEMENT			0	
VIEW			3	4
SORT	ORDER BY		4	4
VIEW			2	4
UNION-ALL			0	
FILTER			6	
NESTED LOOPS	OUTER		0	3
NESTED LOOPS	ANTI		0	3
INDEX	UNIQUE SCAN	STATS\$IDLE_EVENT_PK	1	46
TABLE ACCESS	BY INDEX ROWID	STATS\$SYSTEM_EVENT	0	1
INDEX	UNIQUE SCAN	STATS\$SYSTEM_EVENT_PK	4	1
FILTER			0	
FAST DUAL			1	1

The following section will show how one can count the frequency that indexes are used within Oracle.

## Counting index usage inside SQL

Prior to Oracle9i, it was very difficult to see if an index was being used by the SQL in the database. It required explaining all of the SQL in the library cache into a holding area and then parsing through the execution plans for the index name. Things were simplified slightly in Oracle9i when the primitive ALTER INDEX XXX MONITORING command and the ability to see if the index was invoked were introduced.

One problem has always been that it is very difficult to know what indexes are the most popular. In Oracle10g, it is easy to see what indexes are used, when they are used and the context in which they are used. The following is a simple AWR query that can be used to plot index usage:

#### index\_usage\_hr.sql

---

```
col c1 heading 'Begin|Interval|time' format a20
col c2 heading 'Search Columns'          format 999
col c3 heading 'Invocation|Count'       format 99,999,999

break on c1 skip 2

accept idxname char prompt 'Enter Index Name: '

tttitle 'Invocation Counts for index|&idxname'

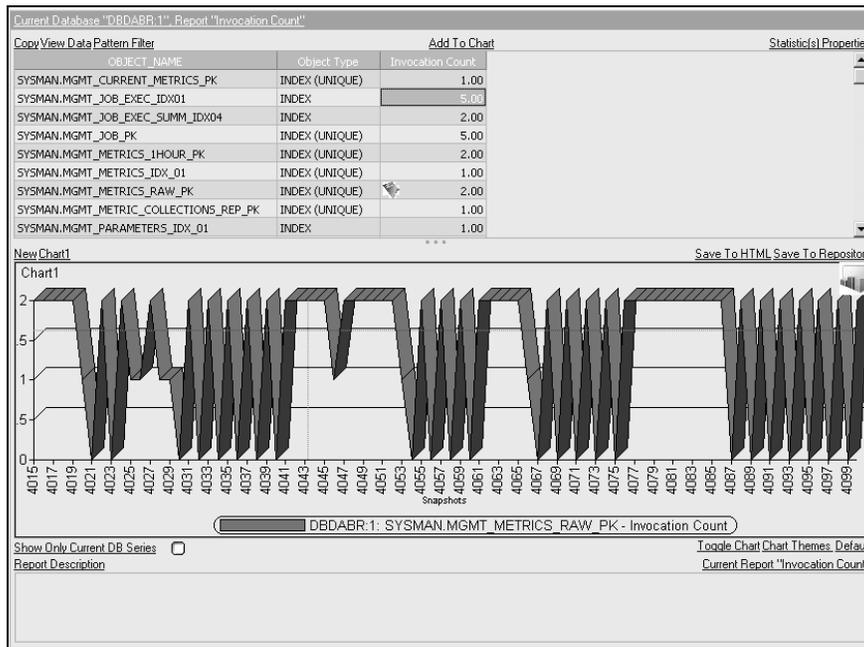
select
  to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
  p.search_columns                               c2,
  count(*)                                       c3
from
  dba_hist_snapshot  sn,
  dba_hist_sql_plan  p,
  dba_hist_sqlstat   st
where
  st.sql_id = p.sql_id
and
  sn.snap_id = st.snap_id
and
  p.object_name = '&idxname'
group by
  begin_interval_time,search_columns;
SEE CODE DEPOT FOR MORE SCRIPTS
http://www.rampant-books.com/book\_2005\_1\_awr\_proactive\_tuning.htm
```

The query will produce an output showing a summary count of the index specified during the snapshot interval. This can be compared to the number of times that a table was invoked from SQL. Here is a sample of the output from the script:

```
Invocation Counts for cust_index

Begin
Interval
time          Search Columns          Invocation
-----
04-10-21 15          1              3
04-10-10 16          0              1
04-10-10 19          1              1
04-10-11 02          0              2
04-10-11 04          2              1
04-10-11 06          3              1
04-10-11 11          0              1
04-10-11 12          0              2
04-10-11 13          2              1
04-10-11 15          0              3
04-10-11 17          0             14
04-10-11 18          4              1
04-10-11 19          0              1
04-10-11 20          3              7
04-10-11 21          0              1
```

Figure 15.24 shows a sample screenshot of a time-series plot produced by the [WISE tool](http://www.wise-oracle.com/product_wise_professional.htm) ([http://www.wise-oracle.com/product\\_wise\\_professional.htm](http://www.wise-oracle.com/product_wise_professional.htm)) for index access.



**Figure 15.24:** Index invocation count time-series plot in WISE tool ([http://www.wise-oracle.com/product\\_wise\\_professional.htm](http://www.wise-oracle.com/product_wise_professional.htm)).

The AWR SQL tuning tables offer a wealth of important time metrics. This data can also be summed up by snapshot period giving an overall view of how Oracle is accessing the table data.

#### **awr\_access\_counts.sql**

```

title 'Table Access|Operation Counts|Per Snapshot Period'

col c1 heading 'Begin|Interval|time' format a20
col c2 heading 'Operation' format a15
col c3 heading 'Option' format a15
col c4 heading 'Object|Count' format 999,999

break on c1 skip 2
break on c2 skip 2

select
  to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
  p.operation c2,
  p.options c3,
  count(1) c4
from
  dba_hist_sql_plan p,
  dba_hist_sqlstat s,
  dba_hist_snapshot sn
where
  p.object_owner <> 'SYS'
and
  p.sql_id = s.sql_id

```

```

and
  s.snap_id = sn.snap_id
group by
  to_char(sn.begin_interval_time, 'yy-mm-dd hh24') ,
  p.operation,
  p.options
order by
  1,2,3;
SEE CODE DEPOT FOR MORE SCRIPTS
http://www.rampant-books.com/book\_2005\_1\_awr\_proactive\_tuning.htm

```

The output of the query is shown below, and it includes overall total counts for each object and table access method.

Begin Interval time	Operation	Option	Object Count
04-10-15 16	INDEX	UNIQUE SCAN	1
04-10-15 16	TABLE ACCESS	BY INDEX ROWID	1
04-10-15 16		FULL	2
04-10-15 17	INDEX	UNIQUE SCAN	1
04-10-15 17	TABLE ACCESS	BY INDEX ROWID	1
04-10-15 17		FULL	2
04-10-15 18	INDEX	UNIQUE SCAN	1
04-10-15 18	TABLE ACCESS	BY INDEX ROWID	1
04-10-15 18		FULL	2
04-10-15 19	INDEX	UNIQUE SCAN	1
04-10-15 19	TABLE ACCESS	BY INDEX ROWID	1
04-10-15 19		FULL	2
04-10-15 20	INDEX	UNIQUE SCAN	1
04-10-15 20	TABLE ACCESS	BY INDEX ROWID	1
04-10-15 20		FULL	2
04-10-15 21	INDEX	UNIQUE SCAN	1
04-10-15 21	TABLE ACCESS	BY INDEX ROWID	1
04-10-15 21		FULL	2

If the DBA has a non-OLTP database that regularly performs large full-table and full-index scans, it is helpful to know those times when the full scan activity is high. The following query will yield that information:

### **awr\_sql\_full\_scans.sql**

```

-- *****
-- Copyright © 2005 by Rampant TechPress
-- This script is free for non-commercial purposes
-- with no warranties. Use at your own risk.
--
-- To license this script for a commercial purpose,
-- contact info@rampant.cc
-- *****

```

```

col c1 heading 'Begin|Interval|Time' format a20
col c2 heading 'Index|Table|Scans' format 999,999
col c3 heading 'Full|Table|Scans' format 999,999

select
  i.c1 c1,
  i.c2 c2,
  f.c2 c3
from
  (
select
  to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
  count(1) c2
from
  dba_hist_sql_plan p,
  dba_hist_sqlstat s,
  dba_hist_snapshot sn
where
  p.object_owner <> 'SYS'
and
  p.operation like '%TABLE ACCESS%'
and
  p.options like '%INDEX%'
and
  p.sql_id = s.sql_id
and
  s.snap_id = sn.snap_id
group by
  to_char(sn.begin_interval_time,'yy-mm-dd hh24')
order by
  1 ) i,
  (
select
  to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
  count(1) c2
from
  dba_hist_sql_plan p,
  dba_hist_sqlstat s,
  dba_hist_snapshot sn
where
  p.object_owner <> 'SYS'
and
  p.operation like '%TABLE ACCESS%'
and
  p.options = 'FULL'
and
  p.sql_id = s.sql_id
and
  s.snap_id = sn.snap_id
group by
  to_char(sn.begin_interval_time,'yy-mm-dd hh24')
order by
  1 ) f
where
  i.c1 = f.c1
;

```

[SEE CODE DEPOT FOR MORE SCRIPTS](http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm)  
[http://www.rampant-books.com/book\\_2005\\_1\\_awr\\_proactive\\_tuning.htm](http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm)

The output below shows a comparison of index-full scans versus full-table scans.

Begin Interval Time	Index Table Scans	Full Table Scans
-----	-----	-----
04-10-21 15	53	18

04-10-21 17	3	3
04-10-21 18	1	2
04-10-21 19	15	6
04-10-21 20		6
04-10-21 21		6
04-10-21 22	16	6
04-10-21 23	21	9
04-10-22 00	16	6
04-10-22 01		6
04-10-22 02	17	6
04-10-22 03	15	6

Knowing the signature for large-table full-table scans can help in both SQL tuning and instance tuning. For SQL tuning, this report will tell when to drill down to verify that all of the large-table full-table scans are legitimate. Once verified, this same data can be used to dynamically reconfigure the Oracle instance to accommodate the large scans.

With that introduction to the indexing component, it will be useful to learn how to use the AWR data to track full-scan behavior over time.

## Tracking full scan access with AWR

All of the specific SQL access methods can be counted and their behavior tracked over time. This is especially important for large-table full-table scans (LTFTS) because they are a common symptom of suboptimal execution plans (i.e. missing indexes).

Once it has been determined that the large-table full-table scans are legitimate, the DBA must know those times when they are executed so that a selective parallel query can be implemented, depending on the existing CPU consumption on the server. OPQ drives up CPU consumption, and should be invoked when the server can handle the additional load.

### awr\_full\_table\_scans.sql

---

```

title 'Large Full-table scans|Per Snapshot Period'

col c1 heading 'Begin|Interval|time' format a20
col c4 heading 'FTS|Count' format 999,999

break on c1 skip 2
break on c2 skip 2

select
  to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
  count(1) c4
from
  dba_hist_sql_plan p,
  dba_hist_sqlstat s,
  dba_hist_snapshot sn,
  dba_segments o
where
  p.object_owner <> 'SYS'
and
  p.object_owner = o.owner
and
  p.object_name = o.segment_name
and

```

```

o.blocks > 1000
and
p.operation like '%TABLE ACCESS%'
and
p.options like '%FULL%'
and
p.sql_id = s.sql_id
and
s.snap_id = sn.snap_id
group by
to_char(sn.begin_interval_time, 'yy-mm-dd hh24')
order by
1;

```

[SEE CODE DEPOT FOR MORE SCRIPTS](http://www.rampant-books.com/book_2005_1_aws_proactive_tuning.htm)  
[http://www.rampant-books.com/book\\_2005\\_1\\_aws\\_proactive\\_tuning.htm](http://www.rampant-books.com/book_2005_1_aws_proactive_tuning.htm)

The output below shows the overall total counts for tables that experience large-table full-table scans because the scans may be due to a missing index.

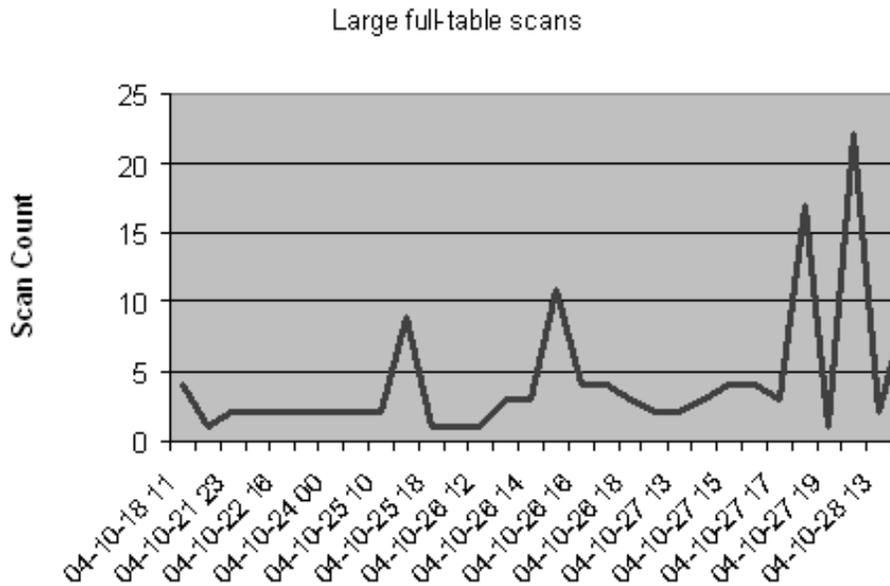
```

Large Full-table scans
Per Snapshot Period

Begin
Interval          FTS
time              Count
-----
04-10-18 11      4
04-10-21 17      1
04-10-21 23      2
04-10-22 15      2
04-10-22 16      2
04-10-22 23      2
04-10-24 00      2
04-10-25 00      2
04-10-25 10      2
04-10-25 17      9
04-10-25 18      1
04-10-25 21      1
04-10-26 12      1
04-10-26 13      3
04-10-26 14      3
04-10-26 15     11
04-10-26 16      4
04-10-26 17      4
04-10-26 18      3
04-10-26 23      2
04-10-27 13      2
04-10-27 14      3
04-10-27 15      4
04-10-27 16      4
04-10-27 17      3
04-10-27 18     17
04-10-27 19      1
04-10-28 12     22
04-10-28 13      2
04-10-29 13      9

```

This data can be easily plotted to see the trend for a database as shown in Figure 15.25:



**Figure 15.25: – Trends of large-table full-table scans**



**Search for Symptoms!** One of the most common manifestations of suboptimal SQL execution is a large-table full-table scan. Whenever an index is missing, Oracle may be forced to read every row in the table when an index might be faster.

If the large-table full-table scans are legitimate, the DBA will want to know the periods that they are invoked, so Oracle Parallel Query (OPQ) can be invoked to speed up the scans as shown in the *awr\_sql\_access\_br.sql* script that follows:

**awr\_sql\_access\_hr.sql**

```

title 'Large Table Full-table scans|Averages per Hour'

col c1 heading 'Day|Hour'          format a20
col c2 heading 'FTS|Count'        format 999,999

break on c1 skip 2
break on c2 skip 2

select
  to_char(sn.begin_interval_time,'hh24') c1,
  count(1)                               c2
from
  dba_hist_sql_plan p,
  dba_hist_sqlstat s,
  dba_hist_snapshot sn,

```

```

    dba_segments      o
where
  p.object_owner <> 'SYS'
and
  p.object_owner = o.owner
and
  p.object_name = o.segment_name
and
  o.blocks > 1000
and
  p.operation like '%TABLE ACCESS%'
and
  p.options like '%FULL%'
and
  p.sql_id = s.sql_id
and
  s.snap_id = sn.snap_id
group by
  to_char(sn.begin_interval_time,'hh24')
order by
  1;

```

[SEE CODE DEPOT FOR MORE SCRIPTS](http://www.rampant-books.com/book_2005_1_aws_proactive_tuning.htm)  
[http://www.rampant-books.com/book\\_2005\\_1\\_aws\\_proactive\\_tuning.htm](http://www.rampant-books.com/book_2005_1_aws_proactive_tuning.htm)

The following output shows the average number of large-table full-table scans per hour.

Large Table Full-table scans  
Averages per Hour

Day Hour	FTS Count
00	4
10	2
11	4
12	23
13	16
14	6
15	17
16	10
17	17
18	21
19	1
23	6

The script below shows the same data for day of the week:

 **aws\_sql\_access\_day.sql**

---

```

title 'Large Table Full-table scans|Averages per Week Day'

col c1 heading 'Week|Day'          format a20
col c2 heading 'FTS|Count'        format 999,999

break on c1 skip 2
break on c2 skip 2

select
  to_char(sn.begin_interval_time,'day') c1,
  count(1) c2
from
  dba_hist_sql_plan p,
  dba_hist_sqlstat s,
  dba_hist_snapshot sn,

```

```

    dba_segments      o
where
  p.object_owner <> 'SYS'
and
  p.object_owner = o.owner
and
  p.object_name = o.segment_name
and
  o.blocks > 1000
and
  p.operation like '%TABLE ACCESS%'
and
  p.options like '%FULL%'
and
  p.sql_id = s.sql_id
and
  s.snap_id = sn.snap_id
group by
  to_char(sn.begin_interval_time, 'day')
order by
1;
SEE CODE DEPOT FOR MORE SCRIPTS
http://www.rampant-books.com/book\_2005\_1\_awr\_proactive\_tuning.htm

```

The following sample query output shows specific times the database experienced large table scans.

Large Table Full-table scans  
Averages per Week Day

Week Day	FTS Count
-----	-----
sunday	2
monday	19
tuesday	31
wednesday	34
thursday	27
friday	15
Saturday	2

The *awr\_sql\_scan\_sums.sql* script will show the access patterns of usage over time. If a DBA is really driven to know their system, all they need to do is understand how SQL accesses the tables and indexes in the database to provide amazing insight. The optimal instance configuration for large-table full-table scans is quite different than the configuration for an OLTP databases, and the report generated by the *awr\_sql\_scan\_sums.sql* script will quickly identify changes in table access patterns.

### **awr\_sql\_scan\_sums.sql**

```

col c1 heading 'Begin|Interval|Time'          format a20
col c2 heading 'Large|Table|Full Table|Scans' format 999,999
col c3 heading 'Small|Table|Full Table|Scans' format 999,999
col c4 heading 'Total|Index|Scans'            format 999,999

select
  f.c1 c1,
  f.c2 c2,
  s.c2 c3,

```

```

i.c2 c4
from
(
select
to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
count(1) c2
from
dba_hist_sql_plan p,
dba_hist_sqlstat s,
dba_hist_snapshot sn,
dba_segments o
where
p.object_owner <> 'SYS'
and
p.object_owner = o.owner
and
p.object_name = o.segment_name
and
o.blocks > 1000
and
p.operation like '%TABLE ACCESS%'
and
p.options like '%FULL%'
and
p.sql_id = s.sql_id
and
s.snap_id = sn.snap_id
group by
to_char(sn.begin_interval_time,'yy-mm-dd hh24')
order by
1 ) f,
(
select
to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
count(1) c2
from
dba_hist_sql_plan p,
dba_hist_sqlstat s,
dba_hist_snapshot sn,
dba_segments o
where
p.object_owner <> 'SYS'
and
p.object_owner = o.owner
and
p.object_name = o.segment_name
and
o.blocks < 1000
and
p.operation like '%INDEX%'
and
p.sql_id = s.sql_id
and
s.snap_id = sn.snap_id
group by
to_char(sn.begin_interval_time,'yy-mm-dd hh24')
order by
1 ) s,
(
select
to_char(sn.begin_interval_time,'yy-mm-dd hh24') c1,
count(1) c2
from
dba_hist_sql_plan p,
dba_hist_sqlstat s,
dba_hist_snapshot sn
where
p.object_owner <> 'SYS'
and
p.operation like '%INDEX%'
and

```

```

    p.sql_id = s.sql_id
and
    s.snap_id = sn.snap_id
group by
    to_char(sn.begin_interval_time,'yy-mm-dd hh24')
order by
1 ) i
where
    f.c1 = s.c1
and
    f.c1 = i.c1
;

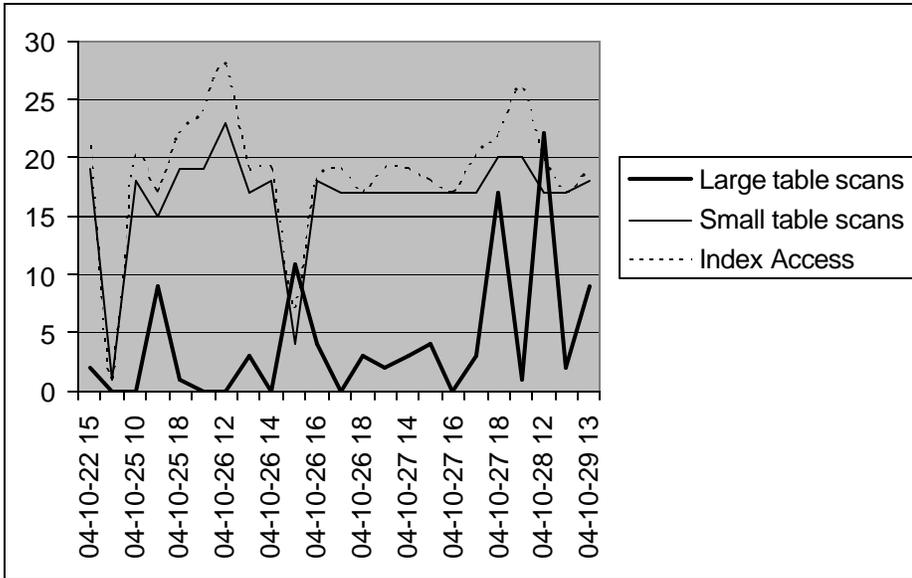
```

[SEE CODE DEPOT FOR MORE SCRIPTS](http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm)  
[http://www.rampant-books.com/book\\_2005\\_1\\_awr\\_proactive\\_tuning.htm](http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm)

The sample output looks like the following, where there is a comparison of index versus table scan access. This is a very important signature for any database because it shows, at a glance, the balance between index (OLTP) and data warehouse type access.

Begin Interval Time	Large Table		Small Table		Total Index Scans
	Full Scans	Table Full Scans	Full Scans	Table Scans	
04-10-22 15		2		19	21
04-10-22 16				1	1
04-10-25 10				18	20
04-10-25 17		9		15	17
04-10-25 18		1		19	22
04-10-25 21				19	24
04-10-26 12				23	28
04-10-26 13		3		17	19
04-10-26 14				18	19
04-10-26 15		11		4	7
04-10-26 16		4		18	18
04-10-26 17				17	19
04-10-26 18		3		17	17
04-10-27 13		2		17	19
04-10-27 14		3		17	19
04-10-27 15		4		17	18
04-10-27 16				17	17
04-10-27 17		3		17	20
04-10-27 18		17		20	22
04-10-27 19		1		20	26
04-10-28 12		22		17	20
04-10-28 13		2		17	17
04-10-29 13		9		18	19

This is a very important report because it shows the method with which Oracle is accessing data over time periods. This is especially important because it shows when the database processing modality shifts between OLTP (*first\_rows* index access) to a batch reporting mode (*all\_rows* full scans) as shown in Figure 15.26.



**Figure 15.26: Plot of full scans vs. index access**

The example in Figure 15.26 is typical of an OLTP database with the majority of access being via small-table full-table scans and index access. In this case, the large-table full-table scans must be carefully checked, their legitimacy verified for such things as missing indexes, and then they should be adjusted to maximize their throughput.

Of course, in a really busy database, there may be concurrent OLTP index access and full-table scans for reports and it is the DBA's job to know the specific times when the system shifts table access modes as well as the identity of those tables that experience the changes.

The following *awr\_sql\_full\_scans\_avg\_dy.sql* script can be used to roll-up average scans into daily averages.

 **awr\_sql\_full\_scans\_avg\_dy.sql**

```
col c1 heading 'Begin|Interval|Time' format a20
col c2 heading 'Index|Table|Scans' format 999,999
col c3 heading 'Full|Table|Scans' format 999,999

select
  i.c1 c1,
  i.c2 c2,
  f.c2 c3
from
  (
  select
    to_char(sn.begin_interval_time,'day') c1,
    count(1) c2
  from
    dba_hist_sql_plan p,
    dba_hist_sqlstat s,
    dba_hist_snapshot sn
  where
```

```

    p.object_owner <> 'SYS'
and
    p.operation like '%TABLE ACCESS%'
and
    p.options like '%INDEX%'
and
    p.sql_id = s.sql_id
and
    s.snap_id = sn.snap_id
group by
    to_char(sn.begin_interval_time,'day')
order by
1 ) i,
(
select
    to_char(sn.begin_interval_time,'day') c1,
    count(1) c2
from
    dba_hist_sql_plan p,
    dba_hist_sqlstat s,
    dba_hist_snapshot sn
where
    p.object_owner <> 'SYS'
and
    p.operation like '%TABLE ACCESS%'
and
    p.options = 'FULL'
and
    p.sql_id = s.sql_id
and
    s.snap_id = sn.snap_id
group by
    to_char(sn.begin_interval_time,'day')
order by
1 ) f
where
    i.c1 = f.c1
;

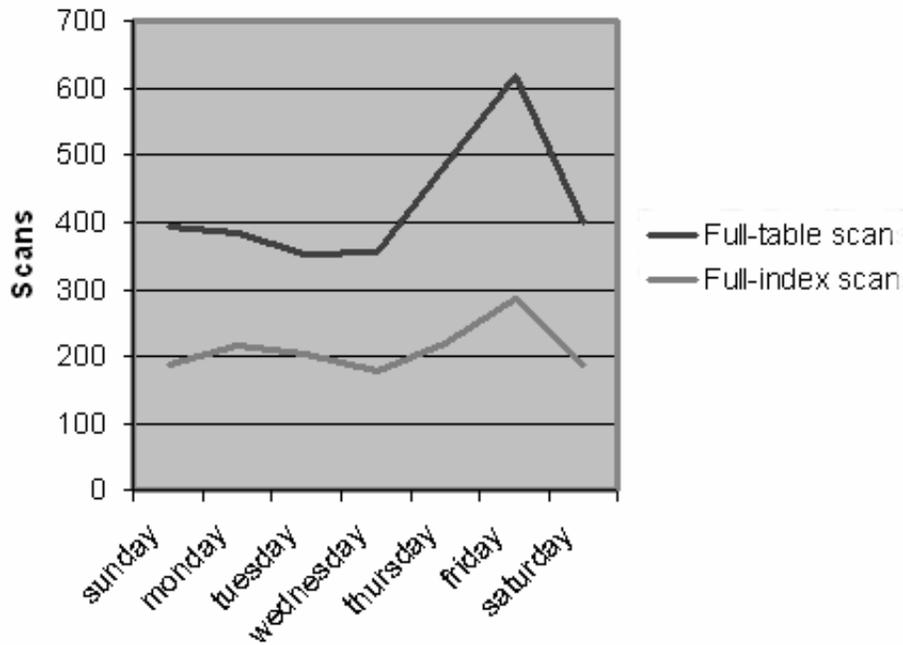
```

[SEE CODE DEPOT FOR MORE SCRIPTS](http://www.rampanant-books.com/book_2005_1_aws_proactive_tuning.htm)  
[http://www.rampanant-books.com/book\\_2005\\_1\\_aws\\_proactive\\_tuning.htm](http://www.rampanant-books.com/book_2005_1_aws_proactive_tuning.htm)

The sample output is shown below:

Begin Interval Time	Index Table Scans	Full Table Scans
sunday	393	189
monday	383	216
tuesday	353	206
wednesday	357	178
thursday	488	219
friday	618	285
saturday	400	189

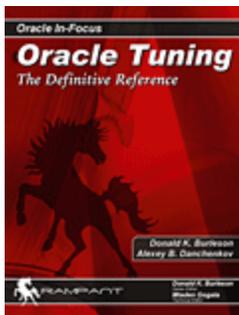
For example, the signature shown in Figure 15.27 below indicates that Fridays are very high in full-table scans, probably as the result of weekly reporting.



**Figure 15.27: Plot of full scans**

With this knowledge, the DBA can anticipate the changes in processing from index access to LTFIS access by adjusting instance configurations.

Whenever the database changes into a mode dominated by LTFIS, the data buffer sizes, such as *db\_cache\_size* and *db\_nk\_cache\_size*, can be decreased. Since parallel LTFIS bypass the data buffers, the intermediate rows are kept in the *pga\_aggregate\_target* region. Hence, it may be desirable to use *dbms\_scheduler* to anticipate this change and resize the SGA just in time to accommodate the regularly repeating change in access patterns.



This is an excerpt from the bestselling book “Oracle Tuning: The Definitive Reference” ([http://www.rampant-books.com/book\\_2005\\_1\\_awr\\_proactive\\_tuning.htm](http://www.rampant-books.com/book_2005_1_awr_proactive_tuning.htm)) by Alexey Danchenkov (<http://www.wise-oracle.com/>) and Donald Burleson (<http://www.dba-oracle.com/books.htm>), technical editor Mladen Gogala.

Incorporating the principles of artificial intelligence, Oracle10g has developed a sophisticated mechanism for capturing and tracking database performance over time periods. This new complexity has introduced dozens of new v\$ and DBA views, plus dozens of Automatic Workload Repository (AWR) tables.

The AWR and its interaction with the Automatic Database Diagnostic Monitor (ADDM) is a revolution in database tuning. By understanding the internal workings of the AWR tables, the senior DBA can develop time-series tuning models to predict upcoming outages and dynamically change the instance to accommodate the impending resource changes.

This is not a book for beginners. Targeted at the senior Oracle DBA, this book dives deep into the internals of the v\$ views, the AWR table structures and the new DBA history views. Packed with ready-to-run scripts, you can quickly monitor and identify the most challenging performance issues.