New Tool on the Block: Segment Level Statistics

By Arup Nanda

Diagnosis of certain events like buffer busy waits require identifying the offending segments, a task easier said than done. Oracle 9iR2 provides a set of very useful views, called segment level statistics, which make this trivial. This article discusses how to set them up for collection of statistics and use the collections, illustrated with a practical case study.

A Database that is Slow

This article provides a way to solve a thorny problem DBAs face, a slow database, while tuning segment level performance statistics. A problem that up until now was virtually impossible to diagnose. With the advent of a new feature in Oracle 9*i* Release 2, diagnosing the problem has become a snap. Without the knowledge of the application, you are limited to diagnosing the problem at the database level only. Suppose that you our have pulled the latest STATSPACK report and here is an excerpt from the Top 5 Wait Events.

```
                                                    Avg
                                    Total Wait  wait    Waits
Event               Waits   Timeouts   Time (s)  (ms)     /txn
. . . .
buffer busy waits    1400          0         43   7.6     19.3. . . .
```

It shows that the *buffer busy waits* event occurred 1,400 times. Perhaps this contributed to the database sluggishness. A query from V$SYSTAT also reports the same information. It is necessary to resolve the problem of buffer busy waits.

Let's examine some of the reasons for buffer busy waits. When a session tries to update a row, the block containing the row is retrieved from the disk by the session's server process and placed in the db block buffer pool. What happens if another session, at the exact same time, tries to update a different row that happens to be in the same block? The server process of that session tries to get the block to the disk too, but since the block is being pulled by the first session, the second session has to wait, which creates a buffer busy wait.

There are numerous ways to reduce the buffer busy waits. These include increasing the freelist groups and freelists of the affected segment or rearranging the distribution of the rows in the table in such a way that the blocks are not repeatedly picked up at the same time from two different sessions. However, to do any of these you must know the exact segment that you are attempting to tune. The STATSPACK report does not tell us which objects contributed to the buffer busy waits event. This information is necessary to continue the tuning. If you query the v$session_wait for the sessions, you can learn that the offending segment is the columns P1 and P2 in that view, which correspond to the file_id and block_id of the segment. But problems are typically reported after the fact and the evidence in v$session_wait disappears after a session exits.

The traditional approach is to place event 10046 for each of the sessions and see all the wait events in the generated trace files, which tend to be extremely large and certainly not in a user-friendly format. In a typical system, which may contain several hundred applications, this approach may not even be feasible. Additionally if the applications connect through Multi Threaded Server, it becomes difficult to isolate a single segment for problems even if trace analysis is possible.

In Oracle 9*i*R2, this information is now obtained from the new performance diagnosis views V$SEGSTAT and V$SEGMENT_STATISTICS after setting up the collection properly. This article will explore such instrumentation and will present to the reader, specifically the DBA faced with troubleshooting the performance problem, the means necessary to find out the wait events on the specific segments.

Setting the Statistics Levels

In order for Oracle to collect those statistics, you must have proper initialization parameters set for the instance. The parameter in init.ora is called STATISTICS_LEVEL. The good news is that it is modifiable via ALTER SYSTEM command and some underlying parameters are even modifiable via ALTER SESSION. Let's explore this parameter further; it can take three values.

BASIC: At this setting Oracle does not collect any stats. Although this is not recommended, you may decide to set this in a fine-tuned production system to save some overhead.

TYPICAL: This is the default value. In this setting, Oracle collects the following statistics.

Buffer Cache: This statistics are utilized by the DBA to assess how best to tune the multiple buffer pools. These statistics can also be collected by setting another parameter DB_CACHE_ADVICE independently using initialization file, stored parameter file, ALTER SYSTEM or ALTER SESSION. If it's independently set then that setting takes preference over the statistics level setting.

Mean Time to Recover:These statistics help the DBA set an acceptable Mean Time to Recover (MTTR) setting, sometimes due to the requirements from Service Level Agreements with the users.

Shared Pool Sizing: Oracle can provide valuable clues to size the shared pool effectively based on usage and these statistics provide that information.

Segment Level Statistics: These statistics are collected at the segment level to help determine the wait events occurring at each segment. The statistics in this paper are of interest.

PGA Target: These statistics help tune the Program Global Area effectively based on the usage.

Timed Statistics: This is an old concept. The timed statistics were enabled in earlier versions with the initialization parameter *timed_statistics*. However, the statistic was so useful that Oracle made it default with the setting of *statistic_level*. It can be set independently, too. If set, it overrides the *statistics_level* setting.

ALL: In this setting all the above statistics are collected as well as an additional two.

Row Source Execution Statistics: These statistics help tune the sql statements by storing the execution statistics with the parser. This can provide an extremely useful tool in the development stages.

Timed OS Statistics: Along with the timed statistics, if the operating system permits it, Oracle can also collect timed stats from the host. Certain operating systems like UNIX allow it. It too can be set independently. If set, it overrides the *statistics_level* setting.

If you set these via any of the three methods, Initialization File, ALTER SYSTEM or ALTER SESSION, you can find the current setting by querying the view V$STATISTICS_LEVEL as follows:

```
SELECT ACTIVATION_LEVEL,
   STATISTICS_NAME, SYSTEM_STATUS,
   SESSION_STATUS
FROM V$STATISTICS_LEVEL
ORDER BY ACTIVATION_LEVEL, STATISTICS_NAME;
```

The output is as follows.

```
ACTIVAT STATISTICS_NAME          SYSTEM_S SESSION_
------- ------------------------ -------- --------
ALL     Plan Execution Statistics DISABLED DISABLED
ALL     Timed OS Statistics       DISABLED DISABLED
TYPICAL Buffer Cache Advice       ENABLED  ENABLED
TYPICAL MTTR Advice               ENABLED  ENABLED
TYPICAL PGA Advice                ENABLED  ENABLED
TYPICAL Segment Level Statistics  ENABLED  ENABLED
TYPICAL Shared Pool Advice        ENABLED  ENABLED
TYPICAL Timed Statistics          ENABLED  ENABLED
```

When starting the tuning process, set the STATISTICS_LEVEL to TYPICAL either by ALTER SYSTEM or by an initialization parameter file. If it's not set explicitly, the setting is TYPICAL by default.

Segment Level Statistics Collection

Now that you have set up the collection, let's examine what you can get from it. The main dynamic performance view that is populated is called V$SEGSTAT. Here is a description of the view.

Column	Explanation
TS#	Tablespace Number, corresponds to the TS# column in SYS.TS$
OBJ#	The Object ID, corresponds to the OBJECT_ID in SYS.DBA_OBJECTS.
DATAOBJ#	It corresponds to the DATA_OBJECT_ID in SYS.DBA_OBJECTS
STATISTIC_ NAME	The most important one, the name of the statistics of interest
STATISTIC#	A unique number to denote each statistics above. This is NOT the same as the V$SYSSTAT statistics number.
VALUE	The current value of that statistic. Please note: the value is cumulative, just like the statistic values in V$SYSSTAT. If you drop the segment and recreate it, the value is reset.

Table 1

As you can see, the columns are somewhat cryptic. Oracle provides another view called V$SEGMENT_STATISTICS, which is based on the above view. This view has a lot more columns and is more descriptive with respect to object identification. In addition to columns like the main view, it also references the names of the tablespace, the object, the owner etc. so that the user can quickly join the view with actual names. However, this view is a little slow because of a lot of joins. It's a better idea to get the object_id from the dba_objects and search v$segstat based on obj# column. Here is the description of the columns of the V$SEGMENT_STATISTICS view that are not present in the V$SEGSTAT view. The other columns are the same as in V$SEGSTAT.

OWNER	The owner of the segment
OBJECT_ NAME	The name of the segment
SUBOBJECT_ NAME	If the above is a partitioned table, each partition has separate statistics. The partition is referred to as sub-object.
TABLESPACE_ NAME	Tablespace where the segment resides
OBJECT_TYPE	Type of the segment, TABLE, INDEX, MATERIALIZED VIEW, etc.

Table 2

If you want to find out what statistics are being collected, you can check the view V$SEGSTAT_NAME, which describes the statistic name and the number.

continued on page 22

New Tool on the Block: Segment Level Statistics continued from page 21

Examining Detailed Statistics

To examine stats for a specific object, you can query as follows:

```
SELECT STATISTIC_NAME, VALUE
FROM V$SEGMENT_STATISTICS
WHERE OWNER = 'SCOTT'
And OBJECT_NAME = 'SALES';
```

This provides output similar to the next:

```
STATISTIC_NAME                    VALUE
------------------------------ ----------
logical reads                    1363168
buffer busy waits                   1649
db block changes                 1430448
physical reads                    238620
physical writes                    15572
physical reads direct                300
physical writes direct                 0
global cache cr blocks served          0
global cache current blocks served     0
ITL waits                              5
row lock waits                        30
```

Most of these events are self-descriptive. The above example shows that since the instance startup the table SALES has been subjected to buffer busy waits 1,649 times, 238,620 blocks have been read for this table, 5 times some sessions have waited due to lack of Interested Transaction Lists entries in this table, and 30 times some sessions have waited because some rows have been locked by other sessions. These are cumulative so the numbers go up as more operations continue on that segment. Like any system level statistics, these are reset to zero when the database is shutdown.

The fact that some wait events against the table involve ITL waits indicates the low setting of INITRANS on that table. The logical and physical read statistics give important pointers to hot segments. These segment level statistics break down the mystery surrounding the statistics collected from V$SYSSTAT or from STATSPACK reports. When baffled with a number of wait events that have already happened the DBA can fall back on these statistics to dig deeper and identify the exact segments that experienced these waits which in turn contributed to the overall system wide wait for that event.

STATSPACK Connection

Statspack is an invaluable tool to collect statistics. The amount of information collected by statspack is controlled by the parameter "level". Level 5, the default, collects most of the useful information and level 10 collects more detailed statistics on latches. In Oracle 9*i* Release 2, STATSPACK has an undocumented level 7, which captures the above mentioned wait events. However, this needs the view V$SEGSTAT to be populated – meaning the STATTISTICS_LEVEL must be set as described above. To set this level in STATSPACK data collection, use the command:

```
execute statspack.modify_statspack_parameter (i_snap_level=>7)
```

This sets the collection to always report the waits for all sessions. If you need to collect stats for a specific period only, use the following while calling the snap package only.

```
execute statspack.snap (I_snap_level=>7)
```

This will select from v$segstat and store them. Use the next snap with the same snap_level to ensure that the v$segstat data is also collected and will be displayed in the final report.

Enhancements

Examining the view definition of V$SEGMENT_STATISTICS, you will notice that the view refers to an internal table called X$KSOLSFTS. This internal table contains a very useful column, namely a column denoting the time when the statistics were collected. This column, FTS_STMP, can be used to our advantage to provide further information on the wait events. You can create a new view called SEGSTAT_WITH_TIME. This view is built from the definition of the V$SEGMENT_STATISTICS after adding the column FTS_STMP. You can read this column to determine if the statistics are stale and help you assess whether to rely on them completely. The other important column this view adds is the INSTANCE_ID, which identifies the instance in a Real Application Cluster (RAC) environment. Adding these two columns and removing all but the most useful columns, the view definition is as shown below:

```
create or replace view segstat_with_time as
select s.inst_id    Instance_id,
   u.name         Owner,
   o.name         Object_name,
   o.subname      Sub_object_name,
   ts.name         Tablespace_name,
   decode(o.type#,
      0, 'NEXT OBJECT',
      1, 'INDEX',
      2, 'TABLE',
      3, 'CLUSTER',
      4, 'VIEW',
      5, 'SYNONYM',
      6, 'SEQUENCE',
      7, 'PROCEDURE',
      8, 'FUNCTION',
      9, 'PACKAGE',
      11, 'PACKAGE BODY',
      12, 'TRIGGER',
      13, 'TYPE',
      14, 'TYPE BODY',
      19, 'TABLE PARTITION',
      20, 'INDEX PARTITION',
      21, 'LOB',
      22, 'LIBRARY',
      23, 'DIRECTORY',
      24, 'QUEUE',
      28, 'JAVA SOURCE',
      29, 'JAVA CLASS',
      30, 'JAVA RESOURCE',
      32, 'INDEXTYPE',
      33, 'OPERATOR',
      34, 'TABLE SUBPARTITION',
      35, 'INDEX SUBPARTITION',
      40, 'LOB PARTITION',
      41, 'LOB SUBPARTITION',
      42, 'MATERIALIZED VIEW',
      43, 'DIMENSION',
      44, 'CONTEXT',
      47, 'RESOURCE PLAN',
      48, 'CONSUMER GROUP',
      51, 'SUBSCRIPTION',
      52, 'LOCATION',
      55, 'XML SCHEMA',
      56, 'JAVA DATA',
      57, 'SECURITY PROFILE',
      'UNDEFINED')    Object_type,
   s.fts_statnam      Statistic_name,
   s.fts_staval      Value,
   to_char(fts_stmp, 'mm/dd/yyyy hh24:mi:ss')
            time_stamp
from obj$ o, user$ u, x$ksolsfts s, ts$ ts
```

```
where o.owner# = u.user#
and s.fts_inte = 0
and s.fts_objn = o.obj#
and s.fts_tsn = ts.ts#
and s.fts_objd = o.dataobj#
and o.linkname is null
and
(o.type# not in
   (1 /* INDEX - handled below */,
   10 /* NON-EXISTENT */)
   or
   (o.type# = 1
   and 1 =
      (select 1
      from ind$ I
      where i.obj# = o.obj#
      and i.type# in (1, 2, 3, 4, 6, 7, 9)
      )
   )
)
and o.name != '_NEXT_OBJECT'
and o.name != '_default_auditing_options_'
```

Case Study

Now that you know how to setup and use the segment level stats, you should be able to diagnose some of your most difficult performance problems. To solidify your understanding, let's examine a case study you can experiment with on your own. A scenario with waits was created to enable us to study and then diagnose it with the segment level statistics. You'll note that although the case study simulated the problems as expected when tested by the author, it is not guaranteed to produce the same behavior elsewhere. It should, however, help the reader's understanding of the methodology.

Our example system is of OLTP nature. It shows consistent performance degradation. The objective of the exercise is to identify the problem and eliminate it. STATSPACK reports show high waits for *buffer busy waits* event. However, since the report that experience these waits does not provide information on specific tables or indexes, the process of segment tuning cannot begin.

For the sake of illustration, see the table called SALES. The table is created as per the example script.

```
create table sales
(
   sales_trans_id   number primary key,
   customer_id   number(2),
   product_id   char(10),
   price      number(10,2),
   quantity   number(5),
   comments   varchar2(20)
)
pctfree 10 pctused 40
storage (freelists 1 freelist groups 1 )
initrans 1 maxtrans 1
/
```

Initially, the table is populated using the following script:

```
declare
   i  number;
   j  number;
   k  number := 0;
begin
   for i in 1..60 loop
      for j in 1..10000 loop
         k := k + 1;
         insert into sales values
         (k, i, 'D'||mod(i,20),
            dbms_random.value
               (1000000,2000000),
         dbms_random.value(1,10),
         'INITIAL');
      end loop;
   end loop;
end;
/
```

Examining the script closely, you will notice that the customer_id column values are loaded one bunch at a time, making the records of a particular customer_id concentrated in a few blocks. Therefore, during an update where the records are picked up in the customer id sequence, they will be very likely to be picked from the same block by two different sessions. The test case transaction is described in the snippet of code following, named stress.sql.

```
declare
  v_cust_id   number(6) := 0;
begin
  for v_cust_id in 1..60 loop
     update sales
     set comments = 'CHANGED by &&1'
     where customer_id = v_cust_id
     and mod(sales_trans_id,2) = &&1;
     commit;
  end loop;
end;
/
exit
```

This program, a simple PL/SQL script updates records with either the odd or even numbered sales_trans_id depending upon the parameter passed to it, for each customer_id from 1 to 60. This script is run from two different sessions. The parameter passed is 0 from one session and 1 from the other, for example from the first session, issue the SQL command @stress 0. If the sessions are kicked off at the exact same time, both sessions will operate on the same customer_id but on different records due to the odd and even numbered sales_trans_id values, eliminating locking. However, both sessions will most likely try to update the records in the same block, because the records are arranged in the customer_id order and both the scripts access the records for the same customer_id. This creates a *buffer busy waits* scenario to identify and eliminate.

Once the table is loaded, execute a STATSPACK report collection. Typically in a production scenario, you would have enabled the jobs to run STATSPACK regularly. To collect the statistics, you would have to login as the STATSPACK user, usually PERFSTAT and issue a command EXECUTE STATSPACK.SNAP. This provides your baseline collection stats.

Now run the stress script from two different sessions, with parameter 0 in one session and 1 in other. To execute at the same time, start them using a

continued on page 24

scheduler like *cron* in UNIX or AT command in Windows. After they run, collect the STATSPACK statistics again by issuing EXECUTE STATSPACK.SNAP. To generate the report, run the script spreport.sql under $ORACLE_HOME/rdbms/admin directory, which will ask you the snap_id for the collections. Give the snap_ids just before and after the stress script. An excerpt from the generated report is shown in the next code box. Under the Section "Top 5 Timed Events", you will note that "buffer busy waits" is one. The system waited 3,378 times for 49 seconds, about 2.83% of all the waits times.

```
Top 5 Timed Events
~~~~~~~~~~~~~~~~~~                                     % Total
Event                    Waits    Time (s) Ela Time
------------------------ ----- ----------- --------
CPU time                                771    44.32
db file scattered read   5,845         428    24.60
db file sequential read  5,680         340    19.54
db file parallel write     894          66     3.80
buffer busy waits        3,378          49     2.83
```

Armed with this information, you can identify the segment that experienced this wait event. Before Oracle 9*i*R2, this was impossible. In 9*i*R2, if you have setup statistics collection by specifying the STATISTIC_LEVEL initialization parameter, you can run the following query:

```
SELECT OWNER, OBJECT_TYPE, OBJECT_NAME, VALUE
FROM V$SEGMENT_STATISTICS
WHERE OWNER = 'SYS'
AND STATISTIC_NAME = 'buffer busy waits';
```

The result is something like this. Of course, you may see a lot more in your environment.

```
OWNER  OBJECT_TYPE OBJECT_NAME    VALUE
-----  ----------- -----------   ------
SCOTT  TABLE       SALES           3302
```

This example shows that the buffer busy waits were experienced by the table SALES owned by user SCOTT. The figure 3302 also roughly corresponds to the figure obtained from the STATSPACK report. In addition to the SALES table, some other SYS owned tables also experienced the same type of wait. Hence, the number is larger in STATSPACK. You immediately know that the problem lies in the table SCOTT.SALES. In an actual production system, you would probably see a lot more tables with the buffer busy waits and the sum of all will roughly correspond to the figure obtained from STATSPACK report. This gives the DBA the capability to identify the segment that is either a victim or creator of a wait event and to take corrective action.

Solution

In the above example, because the offending segment was identified, you can take corrective steps to fix the problem. Notice that the buffer busy waits occurred because two sessions were trying to update the same block at the same time. More even distribution can resolve this. In addition, by making sure a block is less packed, the likelihood that a block will become hot is reduced. As a solution, recreate the table with larger PCTFREE and larger INITRANS and MAXTRANS parameters. This will make the table less dense. The table creation script is provided in the next code box:

```
create table sales
(
   .. .. ..
)
pctfree 40 pctused 60
storage (freelists 11 freelist groups 3 )
initrans 4 maxtrans 30
/
[END CODE BOX]
[TEXT]
Next, load the table in a different way as shown.
[BEGIN CODE BOX]

declare
   i     number;
begin
   for i in 1..600000 loop
      insert into sales values
      (i,mod(i,60)+1, 'D'||mod(i,20),
      dbms_random.value(1000000,2000000),
      dbms_random.value(1,10),'INITIAL');
    end loop;
end;
/
```

Examine the script closely. It loads the customer_id values one after the other until the maximum of 60 is reached and the cycle is repeated. This type of loading eliminates the likelihood that a particular block will be chosen at the same time by two sessions if the customer_id is the same.

After this change, note the value of VALUE in V$SEGMENT_STATISTICS for the table SALES. Since the value is cumulative, you will need a reference value to compare against. Now run the stress.sql script from two sessions the same way as outlined before, with parameter 0 and 1. Now examine the V$SEGMENT_STATISTICS view for the table SALES; the stats for buffer busy waits should be much less. The result of rearranging rows and reducing the packing factor was reduced probability that two sessions will contend for the same block thereby reducing the buffer busy waits.

Conclusion

Prior to Oracle 9*i*R2, this information on the segment level waits could not be obtained easily and in most cases was infeasible to collect using the 10046 event. With this new tool, it is easy to diagnose and resolve the problem of buffer busy waits.

Some common wait events like *free buffer waits* for example are not present in the V$SEGSTAT. Hopefully, Oracle will provide them in the future releases. This is no doubt an important step in the right direction. As a result, performance diagnosis at segment level becomes much easier for the DBA community.

For more information refer to: Oracle 9*i* Release 2 Manuals at http://otn.oracle.com/docs/products/oracle9i/doc_library/release2/index.htm. Search on v$segstat or v$segment_statistics for more information. The author maintains a support page for this article at www.proligence.com/pubsupp.html where the after print errata, user queries and answers are maintained.

About the Author

Arup Nanda is the founder of Proligence (www.proligence.com), a specialized Oracle database services provider in the New York metropolitan area, which provides tactical and strategic solutions in all phases of an Oracle project life cycle. An Oracle DBA for more than 10 years, he has touched almost all aspects of database issues. He specializes in Oracle performance diagnostics and high availability solutions planning. He has presented at several Oracle technology conferences including IOUG Live! and authored articles for several reputed journals, including *SELECT*. He would appreciate receiving feedback on this article at arup@proligence.com.