Database Dashboard Project Report

Dias Lonappan - A0092741E, Nikil Vasudevan - A0092756U,
Shriram Devanathan - A0092736X
Group 3

April 9, 2012

1 Overview

1.1 DbDoctor – A DBA's delight

'They're beeping and they're flashing. They're flashing and they're beeping! I cant stand it anymore, they're blinking and they're flashing.'— Buck Murdock, in the 1982 movie 'Airplane II, The Sequel'.

A database administrator's job is comparable to that of an aircraft pilot. The system works on auto pilot most of the time, and like a pilot, the dba spends most of his time looking at dials and gauges. When disaster does strike, for example a sudden heavy workload or bottleneck appears in the system, the dba has to quickly analyze and provide solutions before the issue escalates. Our tool helps in identifying the problem area quickly and it also suggests necessary steps to improve the performance under one umbrella. An experienced DBA will prefer to have a customized tool that is configurable on-the-fly, and will give him meaningful information about the current performance statistics of the database. Configuration of certain parameters to further fine tune the database would be an added advantage.

DbDoctor intends to do just that, through an intuitive front-end. The DBA is given full freedom to plug and play and get dynamic performance statistics and recommendations without having to analyze humongous reports. He will instantly know when a database is going haywire, or even is on the verge of going bad. Our approach also allows the DBA to set threshold values and configure weights for each of them, to decide which parameter is more significant with regards to the behavior of a specific monitor. The DBA can chose to just monitor the overall performance of a monitor, like Shared Pool or Redo Log Buffer, or dig deeper into further breakdowns by monitoring and extrapolating the dynamic charts that will be generated based on current values. If he wishes to have more monitors, our tool is extensible to 'n' number of monitors without any hassles. In short, the tools is highly configurable, hence DbDoctor will be a DBA's delight. In the subsequent sections, we will talk about how we went about designing this tool, how we tested each and every monitor and constructed queries based on that, how we set baseline values, decided on recommendations and finally, how we concluded on the weights to be given.

Practical aspects of our tool -

1. Create Reports for higher management and for later verification;
2. Fully configurable;
3. ADDM analysis available;
4. Web application can be used by a team of database administrators.

2 Project Administration

Item	Design	Frontend	Backend (PLSQL)	Parameter Baselining and Estimation	Testing	Documentation	Overall
Total Effort Required(Person Days)	3	5	5	5	4	5	27
Dias Lonappan	1	0	4	1	1	2	9
Nikil Vasudevan	1	5	0	1	1	1	9
Shriram Devanthan	1	0	1	3	2	2	9

Table 1: Project administration

3 Overall Design and Architecture

3.1 System Specification

3.1.1 Database Server

Oracle 10g 10.2.0.3.0 running on a Dell Inspiron 1545 Pentium(R) Dual-Core CPU T4200@2.00GHZ System with 4GB RAM, 64 bit Operating System. Further to this the System Global Area details are tabulated in Table 1.

Memory Type	Size in bytes
Total System Global Area	293601280
Fixed Size	1290208
Variable Size	218103840
Database Buffers	67108864
Redo Buffers	7098368

Table 2: SGA Memory details

3.1.2 Application Server

Apache 2.2.21 VC9, PHP 5.3.9, JQuery 1.7.1 running on a i7 @ 2.20 GHZ System with 4GB RAM, 64 bit Operating System

3.1.3 Requirements

1. The connections should be in dedicated server mode.

2. The database should be Oracle 10g.
3. The client machines should have at least 2gigabytes of RAM.
4. The Database user is created with DBA role and table grants are given as required by the administrator.

3.1.4 Design Assumptions

1. The statistics are computed at sampling rate of Z which is configurable.
2. Z should be less than Y, the second level breakdown time interval. This in turn is less than X, the first level breakdown interval.
3. The ADDM advisor gives optimal recommendations using the AWR views. Therefore we should not build our queries on top of this because then our tool becomes redundant. That is we should not use any "dba_hist" views except when we need to validate our output.

3.1.5 Architecture Description

The High Level Architecture diagram Fig1 shows the multiple tiers of our web application, it is developed using a combination of PHP, Jquery and AJAX running on top of an Oracle Database. The design is such that the application is entirely configurable. On top of this, our usage of php ensures that the entire code is open source and can be modified by the open community.

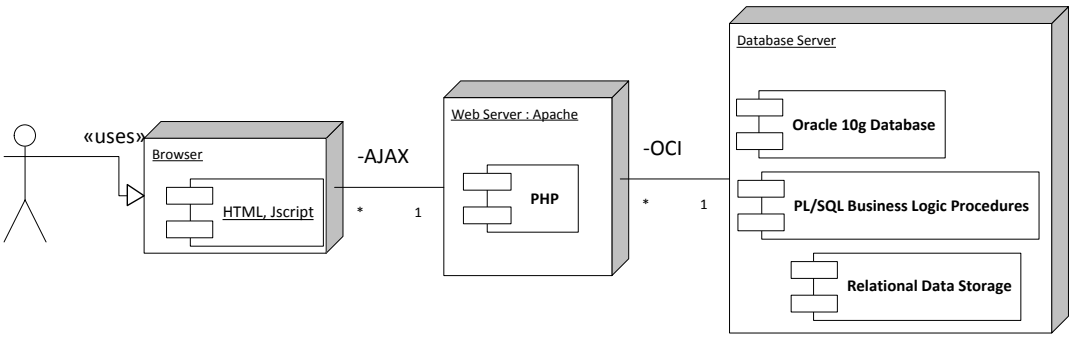


Figure 1: Architecture Diagram

4 Database design

4.1 Entity Relationship Diagram

Our view was that as a DBA, the user of the system would be well averse with the system and its functioning. We however have tested and base lined a few parameters to help the user monitor the

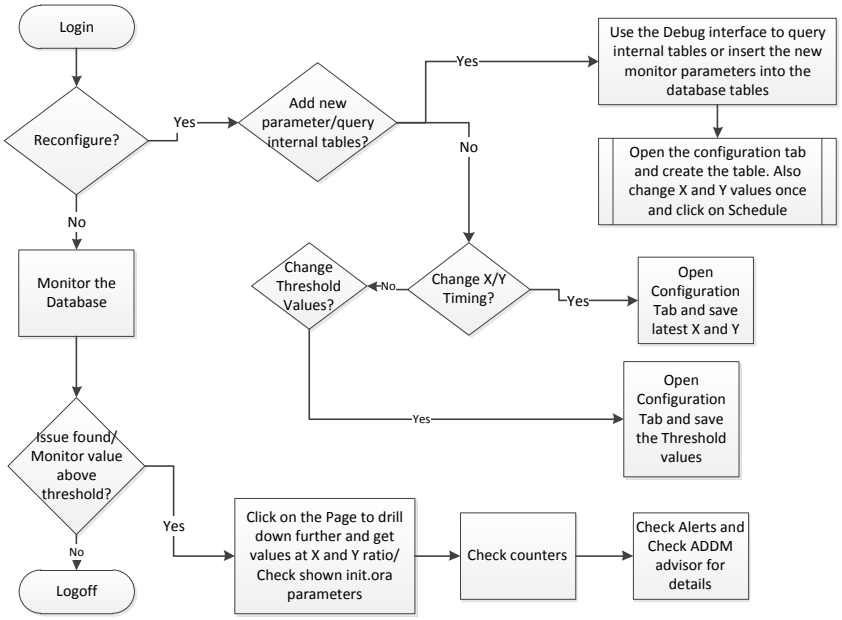


Figure 2: Flowchart of Processes

database initially. The intricate details of the back-end are as follows –

The PROC_WR_SCHED_INITIALIZE is called to initialize the X, Y and Z values. This procedure in turn inactivates the previous values of X,Y and Z. The current value of X , Y and Z is stored as the active values in the database. This procedure also activates the scheduler for the given parameters. The scheduler is an internal Oracle function used to maintain schedules and we are using this to call our sampling procedures at regular intervals.

When the time event for calling the procedure of that particular parameter occurs, the scheduler calls the particular PL/SQL program, in our case this is a procedure in a package. The called procedure in turn executes the Stored Sqls using dynamic SQL execution. The results of these SQL executions are stored in snapshot tables named according to the parameter of the results.The DBA accesses the front-end and when he requests for a particular graph or monitor, the front-end application in turn requests the database for the data by calling a procedure within the package. There are four different types of graphs and we have created different procedures for each of these tasks. The procedures return the data for the graphs to the end user by performing the following calculations

First execute the dynamically constructed query to get the value from the snapshot tables, it is important to note here that each Monitor can have multiple **child queries(Child Paramters)**. While inserting the data into the snapshot table, each of these queries insert their respective values into different columns in the snapshot table. DB_MON_QUERY_COL_SNAP gives us the relation between column names and attributes in the queries. Thus, we have multiple queries inserting a tuple of data points at a given time interval Z. Now we reconstruct the queries and the functions used to calculate the final data points using a procedure, also the way we store the ratio of the baseline value as 1, and then update it to the current value whenever we baseline . Before sending it to the front-end , we go through a very crucial and novel method developed by our team of using weighted means. That is the result of each Child Parameter is converted to a percentage. Subsequently, the sum of all Child Parameters is weighted to 1 before sending it to the graph loader. This allows us to have multiple Child Parameters under a Monitor.

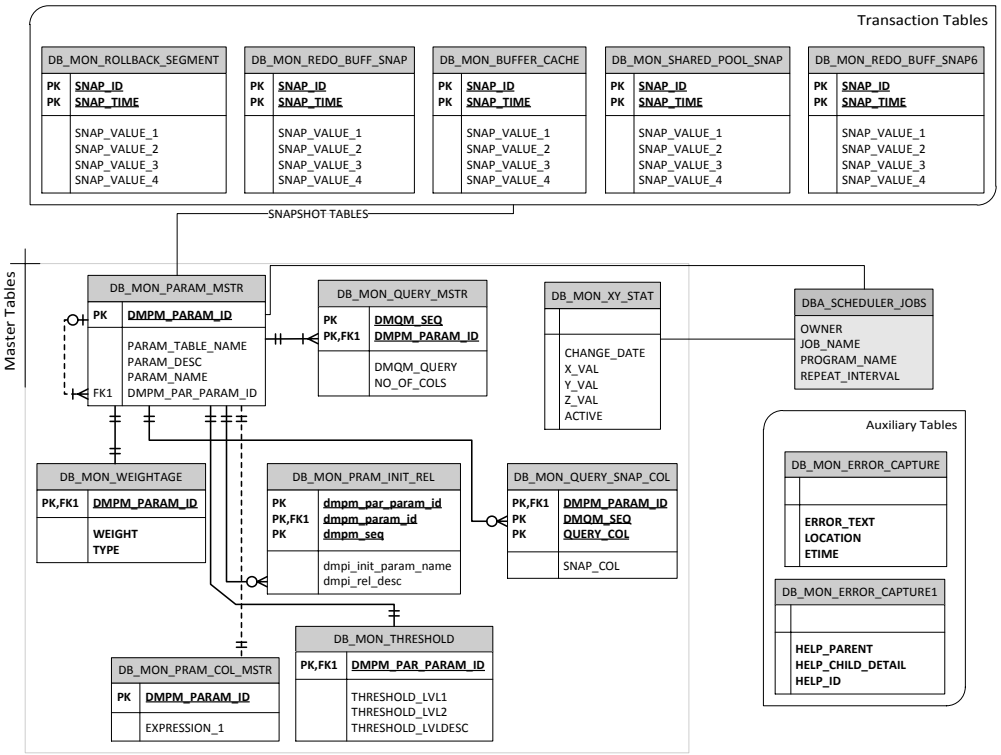


Figure 3: ER Diagram

4.2 Important Tables in the Schema - A brief look

DB_MON_PARAM_MSTR contains the list of the monitors, the child parameters of each and the tables that will be created dynamically by the procedure PROC_CREATE_TABLES_TO_BE. This is again dynamic, so the DBA can insert any number of monitors.

DB_MON_QUERY_MSTR consists of the queries for each monitor. It maintains the queries for each monitor and the number of columns that is being projected. This will be used to populate the DB_MON_QUERY_COL_SNAP table dynamically.

DB_MON_QUERY_COL_SNAP comprises of the projections (column names) from the given queries for the corresponding child parameter.

DB_MON_SHARED_POOL_SNAP is one of the tables corresponding to the monitor Shared pool that will be generated based on the name that is keyed in the DB_MON_PARAM_MSTR. Similarly, other monitor tables will be generated dynamically. These tables will consist of the column values of the corresponding queries and will follow the naming format 'SNAP_ID_[internal sequence number]'. This lays the ground for the next table where the actual computation expression needed will happen.

DB_MON_PRAM_COL_MSTR consists of the actual computation expression of the child parameter (expressed in terms of SNAP_ID_[internal_sequence_number]) needed to track the corresponding monitor. For instance, pin hit ratio could be calculated as SNAP_ID_1/SNAP_ID_2 and so on and so forth. This table also stores the baselined values.

DB_MON_WEIGHTAGE is one of the USP's of the DbDoctor tool. This is where the individual parameters, corresponding to a particular monitor, will be given weights before it is sent across to the front end for various operations. The total weight is 1, and it will be sliced to different parameters in accordance to its importance.

DB_MON_THRESHOLD defines the threshold values for the odometer.

5 Computation of Statistics

5.1 Testing and Baselining

A test case template was created, to record the idle state values before simulation began. Then, we ran the tests to capture the delta values. We utilized the Automatic Database Diagnostic Monitor (ADDM) advisor to get a baseline of the threshold value and validate our graphs and recommendations. An examples of this template is shown in Figure 4. The graph shows Shared Pool test cases. In the graph there is no data for replication 1, the reason for this is explained in Section 7. So when we ran the simulation tests, we realized that we have reached an optimal value at which we should baseline. Towards this, we created a function call which will update the current value of the parameter as the maximum value.

#Run	Time	Event	Addm Response	Interval	Shared Pool	Buffer Cache	Memory sort	Rollback Segment
1	11:00	DB normal load	Not enough Database Time		**54.6**	3.3	25	60
1	11:01	Shared Pool Baseliner run						
2	1:38	DB normal load	Not enough Database Time		**11.7**	11	30	5.53
2	1:43	Shared Pool Baseliner run			**45**	11	44	2.8
2	1:50	Ran ADDM Report	Snapshot created and results displayed.	1:35 - 1:45				

Figure 4: Test Case Template Shared Pool Performance Simulation

5.1.1 Shared pool

The Oracle shared pool contains the library cache, which is responsible for collecting, parsing, interpreting, and executing all of the SQL statements that query DB. It is shared by all oracle processes. Shared pool is used to store SQL statements to avoid repeated parse. Also, the plan that has already been created can be reused. Performance issues with Shared pool could be long running hard parses that could affect the library cache and the shared pool in turn. Therefore it is important to monitor the hard parse count.

The query to find out the free memory available in the shared pool is as follows:

```
select ((sum(s.value)/100000)/.7) "HARD PARSE COUNT" from v$sesstat s, v$sesstat t where s.sid=t.sid and s.statistic#=(select statistic# from v$statname where name='parse count (hard)') and t.statistic#=(select statistic# from v$statname where name='execute count') and s.value>0
```

In the above query, we utilize v$sesstat which displays the user sessions statistics. We have kept a hard parse count limit of 100,000. v$statname displays decoded statistic names for the statistics shown in the v$sesstat and v$sysstat tables. A hard parse count that is approaching 100,000 is bad according to

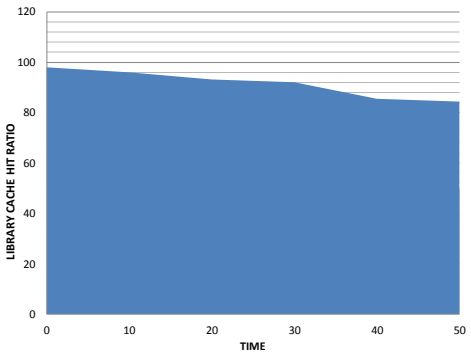


Figure 5: Library Cache Hit Monitor

the given system specification. This could be changed by the DBA according to the resource available. The next query to look out for is the library cache hit ratio that gives us an idea of the number of times all of the metadata pieces of the library object were found in memory to the number of times a PIN was requested for objects of this namespace. v$librarycache contains statistics about library cache performance and activity. Thus the percentage of number of times objects are found in the library cache is tracked and hence, the miss ratio.

```
select sum(pinhits)/sum(pins) from v$librarycache
```

This ratio, ideally, has to be anywhere between 95 to 100 %. A deviation from this behavior suggests that there are many hard parses going on and the library cache is facing a performance issue due to the number of misses while objects are being accessed.

Also, to check if the CPU is being used for parsing operations, it is important to know the ratio of the parse time taken by the CPU and the total CPU used by this session. This ratio must be close to 0, so that we can be sure that parsing is not what the CPU is being used for. Another way to check for hard parses is to have a ratio between hard parse executions and total executions. This was evaluated from the V$sesstat view. We converted them into counters and baselined it to 100000 hard parses based on our system specifications and a value of 0.7 to reflect a critical condition

```
select round((a.value/b.value),4) CPU_RATIO from v$sysstat a, v$sysstat b where a.name = 'parse time cpu' and b.name = 'CPU used by this session'
```

[http://avdeo.com/2007/06/10/tuning-shared-pool-oracle-database-10g/]

The query to find out the free memory available in the shared pool is as follows:

```
select bytes from v$sgastat where name='free memory' and pool='shared pool'
```

This view shows the statistics for sga [ID 62143.1] Result of the above query when the test simulation was run is shown in Figure 5.

Gap between the queries is roughly 2 seconds. As per the observation, we found that the free memory was going down drastically and the test worked. So this will be baselined against the total memory (current total), which will be gathered by:

```
select sum(bytes) from v$sgastat where pool='shared pool'
```

The ratio of the above two queries should be subtracted from 1, and the value will be weighted out of 100. In addition, we have identified a query which gives us the SQL area used currently and this is given by the following. These two queries shall serve as LED parameter which shows the usage

```
select sum(a.sharable_mem)/1024/1024 "total MB used" from v$sqlarea a where a.sharable_mem > 0
```

1. To test the shared pool, we simulated a hard-parse scenario using a java console application, wherein we executed a SELECT statement with an iteratively increasing predicate value.

2. This means that although the query is the same, it would still be hard parsed because the predicate value is being changed every single time. We ran the iteration 10,000 times just to make sure the erratic behavior is being reflected in our odometer. The table dummy consists of more than 20,000 records.

3. Now when we ran this code, we found that the hard parse count was consistently increasing and hence the library cache hit ratio went way below 95% as shown in the graph.

5.1.2 Buffer Cache

Buffer Cache is a Cache memory area wherein data blocks are stored so as to minimize physical I/O reads from the disk. It cannot reside permanently in buffer cache and has to leave the buffer cache whenever a physical read for data in the disk happens. It is ordered in an MRU(Most recently Used) to LRU(Least Recently Used) fashion so the LRU blocks have to leave first. The query that we used was as follows.

```
select round((1-(1-(phy.value / (cur.value + con.value))))*100,2) "Hit Ratio - 1" from v$sysstat cur, v$sysstat con, v$sysstat phy where cur.name = 'db block gets' and con.name = 'consistent gets' and phy.name = 'physical reads'
```

The above query will give us the "Cache Hit ratio" which should always be low.

To find the buffer busy waits we use the query

```
select AVERAGE_WAIT/100 from v$system_event where event ='buffer busy waits'
```

So this will be baselined against the total memory (current total), which will be gathered by[change this to a baseline value which we found out]:

Our testing mechanism here was to flood the buffer cache by performing numerous write operations. So we went about it by creating tables with data types like varchar2(4000), issuing insert commands first and then updating it again with a larger value.

5.1.3 Memory Area for Sort

When a process is started, it is given a separate private RAM. This is called the Program Global Area(PGA). This area is exclusively used for operations such as sorting SQL retrievals, and for managing joins. The PGA could suffer a severe performance hit if the workload of queries running in the database performs non optimal sorts and joins. SQL used :

```
select (1-(VALUE/100)) "VALUE",VALUE from V$PGASTAT where name = 'cache hit percentage'
```

The table v$pgastat gives us information about the usage and performance of pga. cache hit percentage is cumulative from instance startup. A 100% means that all the work areas executed by the system since instance start-up have used an optimal amount of PGA memory. Further to this we also display the usage of PGA

```
select sum(value) "curbytes (total)",trunc(avg(value))"curbytes (average)",count(*) "sessions"
from (select s.sid, s2.serial#, n.name, s.value, s2.logon_time
      from v$statname n, v$sesstat s, v$session s2
      where n.statistic# = s.statistic# and (s.sid = s2.sid) and name like 'session%memory'
      and name not like 'session%memory max%' )
where name like 'session pga memory%'
```

This query is rearranged from the queries given in [MOSC Note id: 835254.1], v$sesstat contains information regarding user session statistics, v$statname contains the name and description of the statistic available in v$sesstat and v$session contains information regarding the sessions connected to the oracle instance . Here the inner query gives the user and the logon time along with the serial#, it also provides the statistic value for each of the individual sessions. The predicates in the inner query determine the memory used by each session and not the max memory that has been used by the session. We compare this with total PGA set currently to show the current percentage PGA used.

```
select value-(value*.2) "(total)" from v$parameter where name = 'pga_aggregate_target'
```

The ratio of the above two queries is calculated as a percentage. We assign this a weight of 1.

To test the above queries we utilized a key value pair table of size 100mb and selected values from it in the reverse order of the key. We ran this query five times and at the fifth execution we baselined it. This caused the cache hit percentage to decrease and this was reflected in our monitor. We also display the number of non optimal executions using v$sql_workarea_histogram which displays t:

```
select sum(onepass_executions)+sum(multipasses_executions) "NONOPTIMAL_EXECUTIONS"
from v$sql_workarea_histogram where onepass_executions > 0
```

The DBA can then make use of the usage and non optimal usage and check the performance.

5.1.4 Redo Log Buffer

Redo log buffer is an area in the Shared Global Area(SGA) which holds information about changes made to the database. This information is stored in redo entries. Redo entries contain the information necessary to reconstruct, or redo changes made to the database . Redo entries are ultimately required to recover databases in the event of failure.[ID 147471.1]. Oracle will eventually flush the redo log buffer to disk. It is important to note that this is guaranteed to happen when a commit operation occurs, a checkpoint occurs, or when the log buffer is one-third full.

The init.ora parameter log_buffer is the key to determine the size of the redo log buffer. Contention in the redo log buffer will have an adverse effect on the performance of the database since all DML and DDL must record an entry in the redo log buffer. Contention could be a latch contention or something like an excessive request for free space in the log buffer. We made use of the following query to determine the immediate miss ratio.

```
select 100* ((immediate_misses+misses)/(immediate_gets+gets)) from v$latch l1, v$latchname l2 where
l2.name = 'redo allocation' and l1.latch# = l2.latch#
```

It gives us an idea about the ratio of the number of times a no-wait latch request did not succeed to that of the total number of latches requested in wait-mode. In case there are many transactions that did not succeed, it could result in data loss, apart from the fact that it is a performance issue.

```
select 100* ((immediate_misses+misses)/(immediate_gets+gets)) from v$latch l1, v$latchname l2 where
l2.name = 'redo copy' and l1.latch# = l2.latch#
```

If the above ratios exceed 1%, then we could conclude that there is latch contention. However, there is another important contention aspect that is to be considered with respect to Redo Log files and that is "redo log space requests". This statistic reflects the number of times a user process waits for space in the redo log file. The following query gives was used by us to get the value.

```
select value from v$sysstat where name = 'redo log space requests'
```

The above value was close to 0 in the idle state of the database. We tried to simulate redo log latch contention by running huge update statements with "commits", using threads in Java. But when we ran our load, we observed that the above value was consistently going up, signifying that many processes

have had to wait for space in the buffer. This was however, overcome by changing the init.ora parameter LOG_BUFFER to a higher value, depending on the load. This is the DBA's prerogative.

5.1.5 Rollback Segment

Rollback segments, as the name suggests, is an area which contains previous values after some update has taken place so that, when a rollback statement is issued, they could be restored. They are flushed(or marked as invalidated, so-to-speak) as soon as a commit is issued. The following query gives us the total number of transactions waiting for a particular rollback segment. The V$WAITSTAT view gives us an idea of that. Ideally, it would be better to have only 30 transactions at a time requesting for the same rollback segment.

```
select sum(count) from v$waitstat where class in ('undo header','undo block')
```

We baseline this against the total number of data requests which can be given by

```
select sum(value) "Data Requests" from v$SYSSTAT where name in ('db block gets', 'consistent gets')
```

So if the number of transactions from query 1 exceeds even 1% of the total number of data requests that is retrieved from query 2, then we must consider adding more rollback segments in the database. The 1% is just an arbitrary value, but this could be tweaked according to the resources available for a DBA. The other thing that has to be monitored is the hit ratio which gives us an idea of percentage of requests that have had to wait for rollback segments. The V$rollstat view allows us to do that

```
select avg(round(((gets-waits)*100)/gets,2)) "hit_ratio" from v$rollstat a,v$rollname b where a.usn = b.usn
```

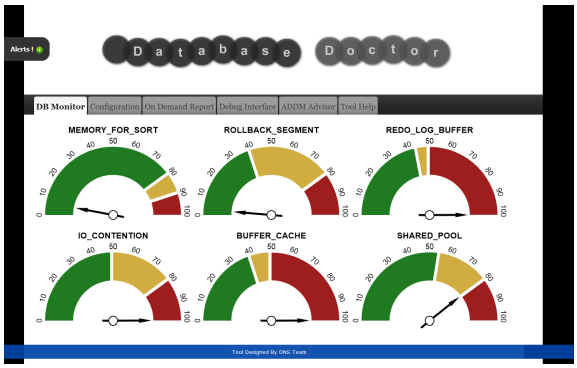
If this is anything below 99%, it is a performance issue and adding more rollback segments to the tablespace must be considered. It also makes sense to monitor the usage of the rollback segments. The following query gives us the ratio of the needed size to the ratio(based on the current load) of the actual size of the undo segments

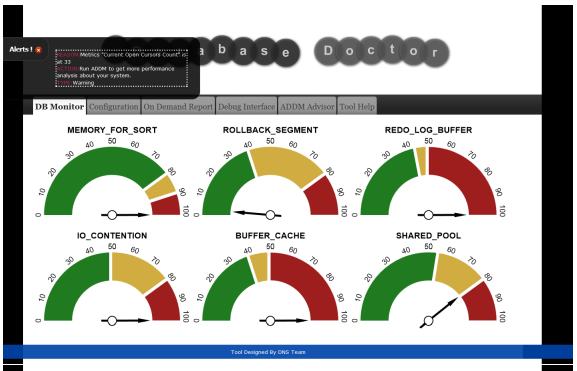
```
select ((to_number(e.value) * to_number(f.value) *g.undo_block_per_sec) / (1024*1024))/(d.undo_size/(1024*1024)) "ratio_needed_actual_undo_size"
from (
      select sum(a.bytes) undo_size from v$datafile a, v$tablespace b, dba_tablespaces c
      where c.contents = 'UNDO' and c.status = 'ONLINE'
      and b.name = c.tablespace_name and a.ts# = b.ts#) d,
v$parameter e, v$parameter f, (select max(undoblks/((end_time-begin_time)*3600*24))undo_block_per_sec from v$undostat) g
where e.name = 'undo_retention' and f.name = 'db_block_size'
```

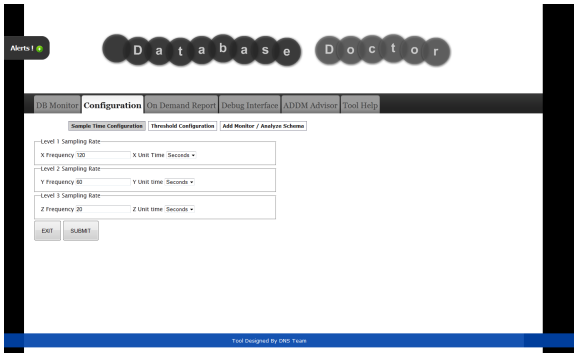
The above query has been worked out based out of the formula below undo size= undo_retention * db_block_size * undo_block_per_sec

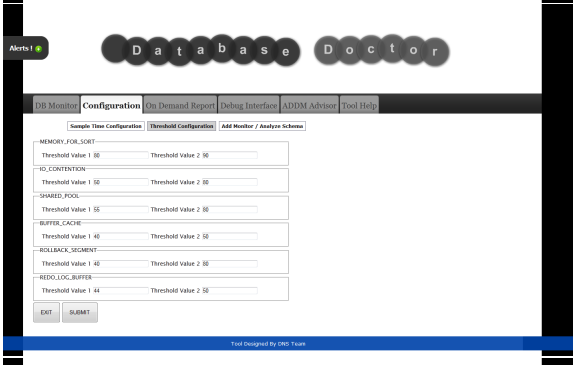
So depending on the load, we will have to make changes to the init.ora parameter UNDO_RETENTION or increase the rollback size. To test the performance issues with rollback segments, we again simulated a contention-simulation in Java by using multi-threading, each of which performed inserts and updates, using their own transactions every time. This resulted in a sharp increase in the hit ratio as each thread created a transaction each, trying to modify the same table. There were around 10 threads with 10 insert statements each, thus resulting in a contention. The load also increased the undo size that is needed(query 3), drastically, thus resulting in not just a performance issue, but a usage issue as well.

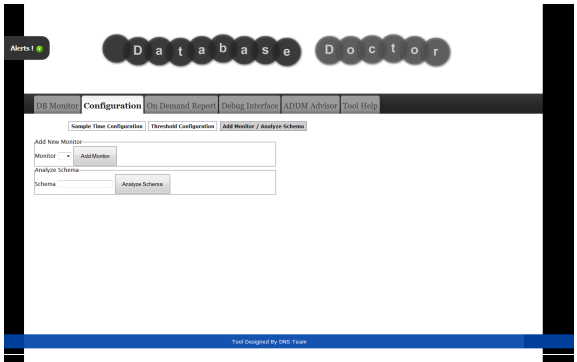
6 Screenshots of each feature

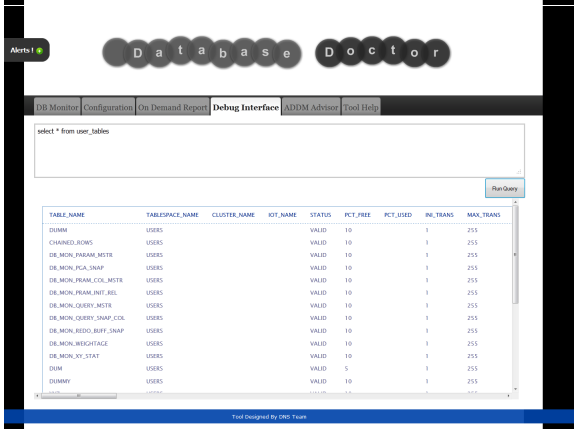


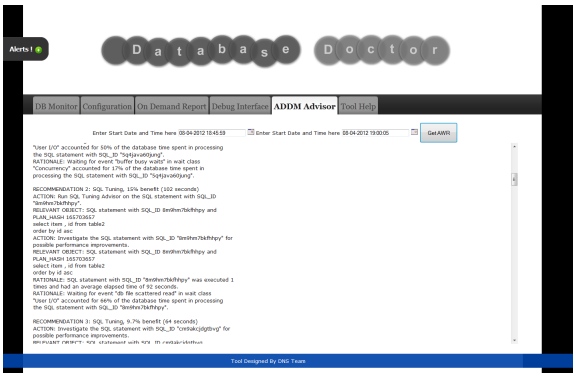


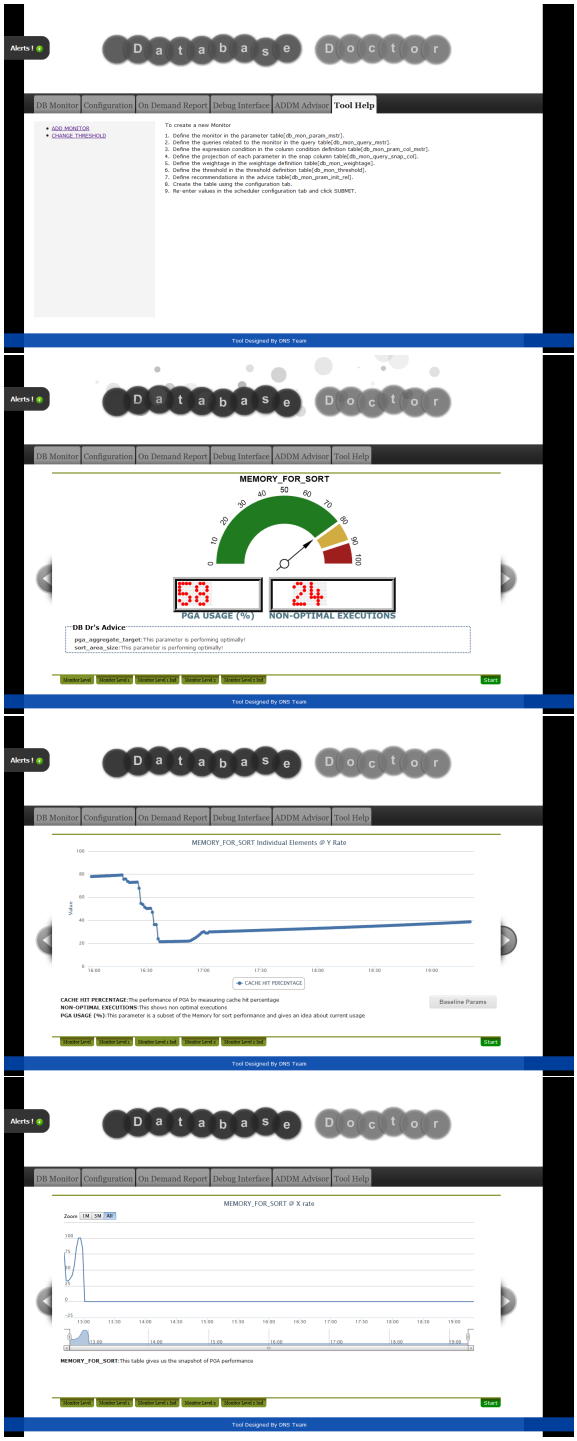


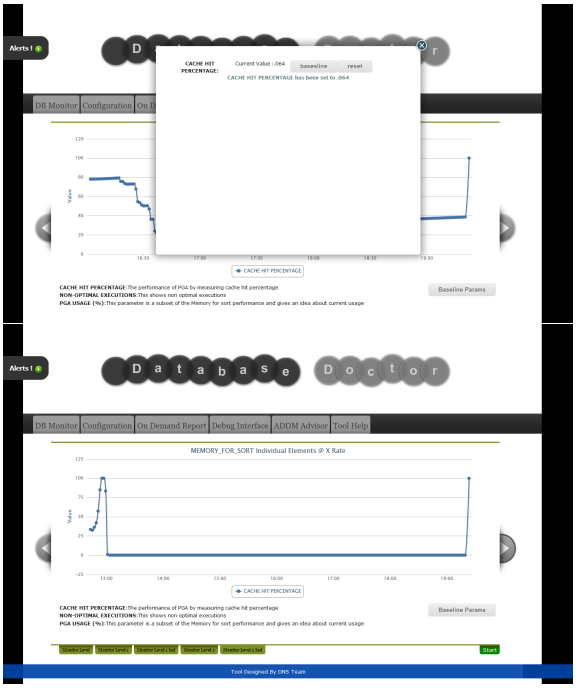












7 Novel Ideas

1. Multiple parameters add up to a parent monitor. That is the usage of weights.
2. Our own Advisor, this advisor is mapped to the threshold values and depending on current statistics will display correct advice.
3. Displaying counters to gain more information. Counters can be number of sessions, current cpu usage etc.

4. Usage of DBA_ALERT views to highlight alerts, additional alerts can be configured by the user, by setting the threshold of the alert.

5. Schema Analyzer

6. Onboard Help

7. ADDM Advisor

8 Issues Faced and Lessons Learned

1. The first issue we faced was when we started the simulation run for shared pool, we had created a table with two columns one an id and the other the text data for that particular id. When we ran this query we found that although the cost was low, the query took exceptionally long to execute. Roughly 20 seconds, this led us to investigate this particular query. We ran the ADDM advisor for this time period and found out that there was high I/O. That is although the query was indexed, which led to the low cost. A High I/O incurred because the data retrieved was quite large.

2. While testing shared pool we noticed that the shared pool size automatically resizes when other components in the SGA change. We realized that Oracle manages its internal structure quite well, we learned that performance tuning does not mean we need to continously tweak the init.ora paramters, it also includes thinking of redesigning the application that is causing the performance issue. That is when we ran this application at Z=5, ADDM gave us High Database Time Impact. So we decided to give a recommendation that querying the database for performance analysis should be done only every 20 seconds or more.

3. While trying to query the V$ views we faced some performance degradation, especially if we were to query our database every 5 seconds. A search for a better solution took us into some depth into the x$ fixed table architecture. We have tried to convert some of our v$ view queries into x$ table queries.

4. While simulating the Memory Used for sorting we noticed that the Cache Hit Ratio decreased rapidly on running the simulation load just after the instance started. However after a little while when we retried the same load the rate of change was less dramatic. Hence we realized that initialization of the database does cause a load and for getting optimal baseline values we should execute the tests once we reach a steady state.

8.1 Additional Parameter

The additional parameter that we incorporated was to measure the I/O load. Based on Insight 1 discussed previously we decided to incorporate this as an additional monitor. I/O is a bottleneck in most systems today, new flash storage based devices do help in reducing I/O bottlenecks.-[Identifying Poorly Performing SQL - www.toadworld.com]

The following query tells us how to identify SQL statements with I/O hit ratio, we are doing a minus one to convert the data into a value from zero to hundred.

```
select (1-avg(round((buffer_gets - disk_reads) / buffer_gets, 2)))*100 hit_ratio from v$sqlarea where executions > 0 and buffer_gets > 0 and (buffer_gets - disk_reads) / buffer_gets < 0.80
```

The v$sqlarea view stores the sql's fired and details about it. buffer_gets is defined as "Sum of buffer gets over all child cursors" - [http://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_2129.htm], and disk_reads is the sum of the number of disk reads over all child cursors. Thus this ratio will tell us the

To test this we created a key value pair, inserted large values into the value attribute and then tried selecting this value by using a point query on the id attribute. This immediately reflected in the I/O rate going up.

9 Looking ahead

In conclusion, we learned quite a lot from this exercise, and in future we plan to make this tool compatible with all databases. In addition, archival and purging of old statistics. maintaining an audit of the change in weights, ASH and AWR snapshot report display are some of the features we are planning to incorporate.