

Real-Time SQL Monitoring

December 2009

INTRODUCTION

Real-time SQL monitoring, introduced in Oracle Database 11g, provides a very effective way to identify run-time performance problems with resource intensive long-running and parallel SQL statements. Interactive Enterprise Manager screens display details of SQL execution using new, fine-grained SQL statistics that are tracked out-of-the-box with no performance penalty to production systems. Statistics at each step of the execution plan are tracked by key performance metrics, including elapsed time, CPU time, number of reads and writes, I/O wait time and various other wait times. This allows DBAs to analyze SQL execution with greater detail than it was previously possible and decide whether to terminate the long-running SQL, let the SQL complete, or send the SQL for tuning.

IMPLEMENTATION DETAILS

The Real-Time SQL Monitoring feature of Oracle Database enables monitoring the performance of SQL statements while they are executing. By default, SQL monitoring is automatically started when a SQL statement either runs in parallel or has consumed at least 5 seconds of combined CPU and I/O time in a single execution.

V\$ views

There are 2 new views introduced in Oracle 11 g to support this: V\$SQL_MONITOR and V\$SQL_PLAN_MONITOR.¹ These new views can be used in conjunction with the following views to get additional information about the execution being monitored:

- V\$ACTIVE_SESSION_HISTORY
- V\$SESSION_LONGOPS

¹ Although, it is by far the least friendly way to use SQL Monitoring, make sure to see sections on Active Reports and SQL Monitoring in the Enterprise Manager for the graphical representation of the same data.

- V\$SQL
- V\$SQL_PLAN

The V\$SQL_MONITOR view contains a subset of the statistics available in V\$SQL. However, unlike V\$SQL, monitoring statistics are not cumulative over several executions. Instead, one entry in V\$SQL_MONITOR is dedicated to a single execution of a SQL statement. If two executions of the same SQL statement are being monitored, each of these executions will have a separate entry in V\$SQL_MONITOR.

To uniquely identify two executions of the same SQL statement, a composite key called an execution key is generated. This execution key is composed of three attributes, each corresponding to a column in V\$SQL_MONITOR:

- SQL identifier to identify the SQL statement (SQL_ID)
- Start execution timestamp (SQL_EXEC_START)
- An internally generated identifier to ensure that this primary key is truly unique (SQL_EXEC_ID)

SQL Monitoring Data Retention

Once monitoring is initiated, an entry is added to the dynamic performance view V\$SQL_MONITOR. This entry tracks key performance metrics collected for the execution, including the elapsed time, CPU time, number of reads and writes, I/O wait time and various other wait times. These statistics are refreshed in near real-time, once every second, as the statement executes. Once the execution ends, monitoring information is retained in the V\$SQL_MONITOR view for at least a minute. The SQL Monitoring data in the SGA is not vulnerable to cursor age-outs from V\$SQL; however it exists in a size-constrained in-memory buffer and will eventually be overwritten by statistics coming from new statements that are monitored. So, if the system has a lot of long-running SQL statements and Parallel Queries, not all SQL statements are guaranteed to be monitored because of two factors: space limitation of the in-memory buffer and a 5-minute retention guarantee for the SQL statements that made the monitored list. However, practically this should never happen.

SQL Plan Monitoring

Real-time SQL monitoring also includes monitoring statistics for each operation in the execution plan of the SQL statement being monitored. This data is visible in the V\$SQL_PLAN_MONITOR view. Similar to the V\$SQL_MONITOR view, statistics in V\$SQL_PLAN_MONITOR are updated every second as the SQL statement is being executed. These statistics also persist for at least 5 minutes after the execution ends. There will be multiple entries in V\$SQL_PLAN_MONITOR for every SQL statement being monitored; each entry will correspond to an operation in the execution plan of the statement.

Parallel Execution Monitoring

Parallel queries, DML and DDL statements are automatically monitored as soon as execution begins. Monitoring information for each process participating in the parallel execution is recorded as separate entries in the V\$SQL_MONITOR and V\$SQL_PLAN_MONITOR views.

V\$SQL_MONITOR has one entry for the parallel execution coordinator process, and one entry for each parallel execution server process. Each entry has corresponding entries in V\$SQL_PLAN_MONITOR. Because the processes allocated for the parallel execution of a SQL statement are cooperating for the same execution, these entries share the same execution key (the composite SQL_ID, SQL_EXEC_START and SQL_EXEC_ID). It is therefore possible to aggregate over the execution key to determine the overall statistics for a parallel execution.

On RAC systems, for parallel execution going cross-instance, the V\$ view will show only those processes that ran on the current instance. In order to see the cross-instance statistics GV\$ views must be used.

Active Reports

Starting with Oracle Database version 11g Release 2, reporting for SQL Monitoring has been enhanced with the Active Reports functionality.

An Active Report is an interactive report that can be used for off-line analysis. It offers the same level of interactivity as the live Enterprise Manager screens, with drill-downs to various levels of detail. For example, the SQL Monitoring Active Report displays the expected and actual number of rows returned for each step of the execution plan. This information can be critical in understanding why the optimizer chose a specific execution plan.

The active report can be saved as a single HTML file and archived for later reference. Alternatively, the active report could be forwarded to appropriate personnel for further analysis (e.g., emailed to an in house performance expert, or to Oracle support).



Figure 1: SQL Monitoring Active Report.

The recipient does not need an Enterprise Manager or a Database installation to view an Active Report. To generate an active report the user can either use new button controls on top of the SQL Monitoring Execution Details page or

alternatively generate it with a REPORT_SQL_MONITOR() function call (part of the DBMS_SQLTUNE package).

SQL Monitoring Workflows in Oracle Enterprise Manager

SQL Monitoring was instrumented and available for command line use in the first release of the Oracle Database 11g. The graphical user interface to monitor active running SQL made its debut in Grid Control 10.2.0.5 and Database Control 11.1.0.7. Basically, the data from the GV\$ views, described in the previous subsections, is aggregated and presented in a user-friendly interactive fashion in EM.

There are three different views at the SQL Monitoring data in EM. The first and the most comprehensive (in a sense that it provides a system-wide view) is through a link at the bottom of the Performance Page:



Figure 2: SQL Monitoring link.

That link leads to the Monitored SQL Executions page that contains a list of all SQL statements that are and have been monitored on the system:

Status	Duration	SQL ID	Session	Parallel	Database Time	IO	Start	Ended	SQL Text
21.0s	ff3yqnaeh5yr	133	10.7s		06:31:44 PM		select * use_n3() use_n3() full (a) full (b) */P SQL		
6.4m	BuwnyC9u1m6s6	128	7.8m	2	06:25:42 PM		select * parallel(a, 2) use_n3() full (a) full (b) */P		
6.0m	l8dvgt1gn6zk	133	6.0m		06:25:42 PM	131	select * use_merge(b) use_merge(c) full (a) full (b)		
24.7m	BuwnyC9u1m6s6	131	29.5m	2	04:02:23 PM		select * parallel(a, 2) use_n3() full (a) full (b) */P		
2.7m	ff3yqnaeh5yr	149	2.7m		04:10:37 PM		select * use_n3() use_n3() full (a) full (b) */P SQL		
8.2m	l8dvgt1gn6zk	148	8.2m		04:02:23 PM	259	select * use_merge(b) use_merge(c) full (a) full (b)		
20.5m	BuwnyC9u1m6s6	140	36.2m	2	Thu Apr 23 2009 0:	1	select * parallel(a, 2) use_n3() full (a) full (b) */P		
9.3m	ff3yqnaeh5yr	142	9.3m		Thu Apr 23 2009 0:		select * use_n3() use_n3() full (a) full (b) */P SQL		
10.5m	l8dvgt1gn6zk	142	10.4m		Thu Apr 23 2009 0:	250	select * use_merge(b) use_merge(c) full (a) full (b)		
20.3m	BuwnyC9u1m6s6	140	39.1m	2	Thu Apr 23 2009 0:		select * parallel(a, 2) use_n3() full (a) full (b) */P		
9.3m	ff3yqnaeh5yr	137	9.3m		Thu Apr 23 2009 0:		select * use_n3() use_n3() full (a) full (b) */P SQL		
10.6m	l8dvgt1gn6zk	137	10.6m		Thu Apr 23 2009 0:		select * use_merge(b) use_merge(c) full (a) full (b)		
20.7m	BuwnyC9u1m6s6	140	39.5m	2	Thu Apr 23 2009 0:	1	select * parallel(a, 2) use_n3() full (a) full (b) */P		
2.4m	ff3yqnaeh5yr	142	9.4m		Thu Apr 23 2009 0:		select * use_n3() use_n3() full (a) full (b) */P SQL		
10.2m	l8dvgt1gn6zk	142	10.2m		Thu Apr 23 2009 0:	250	select * use_merge(b) use_merge(c) full (a) full (b)		
19.3m	BuwnyC9u1m6s6	132	37.2m	2	Thu Apr 23 2009 0:		select * parallel(a, 2) use_n3() full (a) full (b) */P		
9.3m	ff3yqnaeh5yr	134	9.3m		Thu Apr 23 2009 0:		select * use_n3() use_n3() full (a) full (b) */P SQL		
10.2m	l8dvgt1gn6zk	134	10.2m		Thu Apr 23 2009 0:	259	select * use_merge(b) use_merge(c) full (a) full (b)		
4.0s	d15odt0t3vtp	134	4.1s		Thu Apr 23 2009 0:	4491	SELECT TO_CHAR(current_timestamp AT TIME ZON		

Figure 3: Monitored SQL Executions page.

Each row on this page shows an instance of a SQL execution that has or is being monitored. By default, rows are ordered in the table such that most recent executions are shown first with the first few rows always showing SQL executions that are in progress. Information on this page presents only key data about each execution: global information like SQL id and text, parallel DOP and statistics on

the execution like status, start and end times, duration, database time breakdown, IO statistics. This page is refreshed on a periodic basis to update the statistics of execution(s) that are in progress or to add additional entries for newly monitored executions.

Clicking on an entry drills-down to the Monitored SQL Executions Detail page that contains additional information about a specific SQL execution. If the execution of interest is still in progress, it is possible to get real-time updates on the execution statistics as well as to see what operation of the plan is being executed at that time. Additionally, it is possible to see a lot of useful information about the SQL execution, including: SQL-level statistics for the execution (database time breakdown, CPU, I/O read/write breakdown), degree of parallelism for the query, start and finish time of the execution, etc.

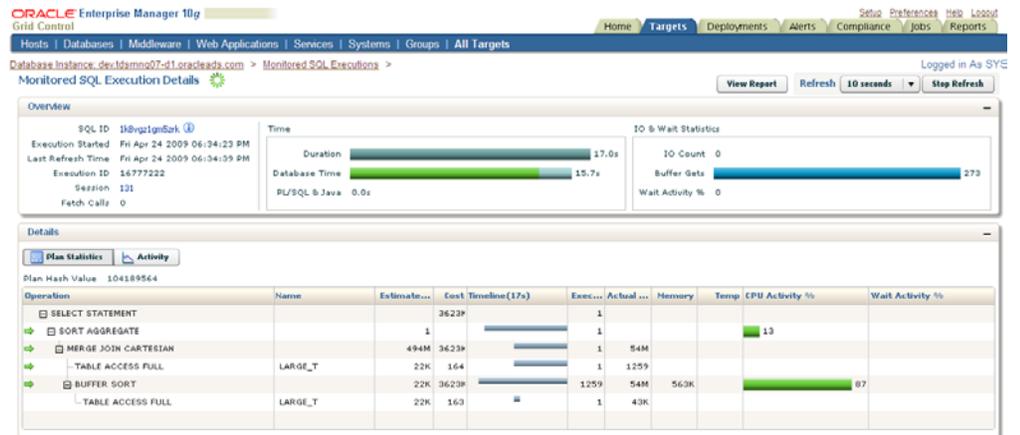


Figure 4: Monitored SQL Execution Details page.

The second view is at the SQL Monitoring statistics for different instances of a SQL statement. It is exposed through the SQL Details page. A new SQL Monitoring tab has been introduced on that page. It contains the list of all previously monitored executions for that specific SQL statement:

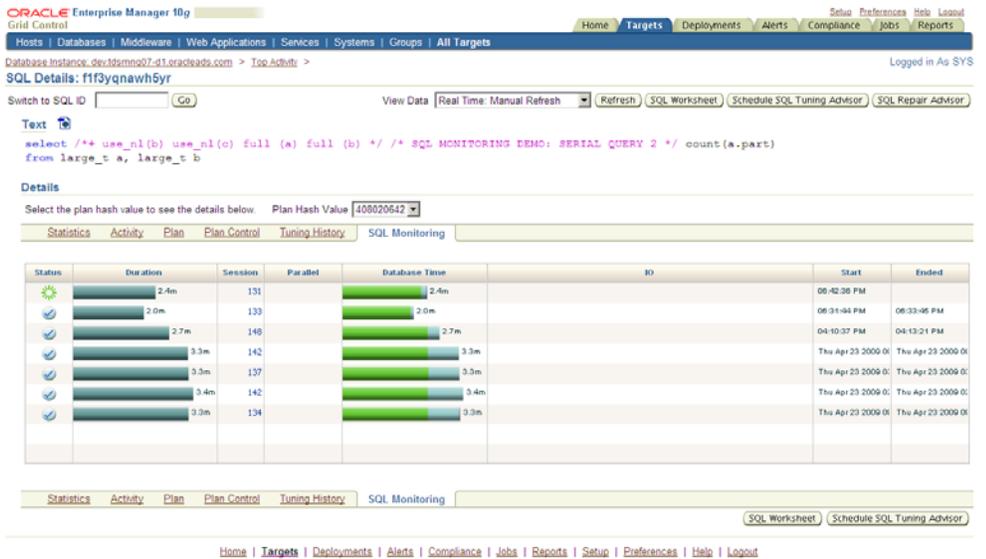


Figure 5: SQL Details page.

The third view is at the SQL Monitoring statistics for a specific session. It is exposed through the Session Details Page. Just like with the SQL Details page, a new SQL Monitoring tab has been added to the Session Details page. It contains all monitored SQL statements that were part of that session.

Also note that it is possible to terminate the whole session from the Session Details page (with the Kill Session button) as a prompt reactive measure of dealing with a runaway SQL statement that is monopolizing system resources to the detriment of all other users:



Figure 6: Session Details page.

It is also possible to get to the SQL and Session Detail pages from the main Monitored SQL Executions page, if more information about the SQL statement is desired. Thus all three views at the SQL Monitoring data are interlinked in EM.

Note that the SYSTEM→SQL→SESSION approach to viewing database activity for SQL monitoring is not new with this feature, it is the principal approach Oracle Database has exposed in the EM GUI since version 10gR1 and SQL Monitoring is simply fully integrated with that methodology.

MONITORED SQL EXECUTION DETAILS PAGE

This section is dedicated to describing different components of the new Real-Time SQL Monitoring user interface available in the Enterprise Manager with a detailed example of a complex parallel query. Its goal is to teach the DBA how to navigate through the abundant run-time data available for a monitored SQL statement. Monitored SQL Execution Details page is a drill down for an individual SQL statement from the Monitored SQL Executions page.

On the very top of the screen, next to the title of the page, there is a status icon for the SQL statement that this page is monitoring. Three types of icons to describe the state of the SQL statement are available. They are: Done, Running, and Failed. As the mouse goes over the icon we get the description of it:



Figure 7: status of a monitored SQL statement.

Also, it is worth noting that if the SQL has failed, the Oracle error number and message will be displayed via the icon tooltip. This enhancement was introduced into SQL monitoring starting with Oracle database 11gR2 release.

Next thing is the SQL ID when we hover the mouse pointer over it we get the SQL text corresponding to the running query:

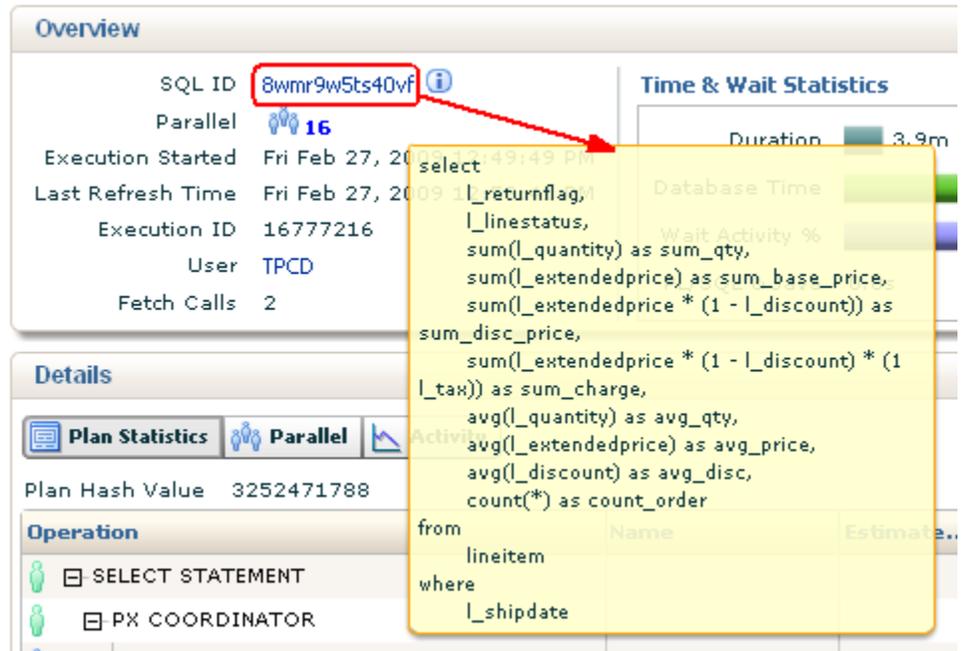


Figure 8: SQL text of a monitored SQL statement.

On the live system when a user clicks on the SQL ID link they navigate to the SQL Details page that shows real-time as well as historical information about the SQL. For long SQL statements and for purposes of copying SQL text from the SQL Monitoring UI into the clipboard, the user can click on the information icon next to the SQL ID:

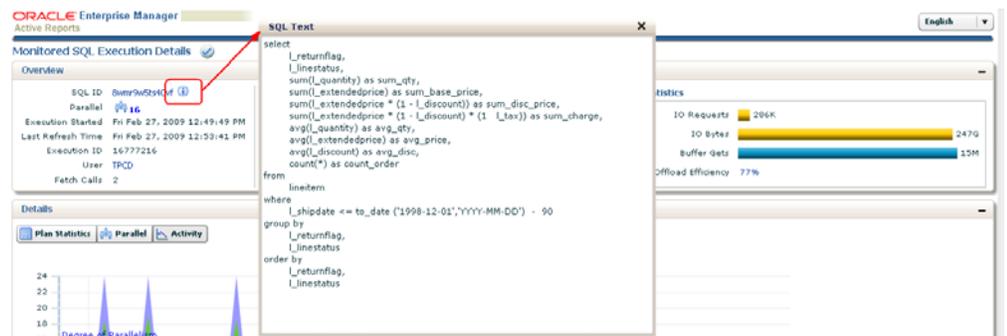


Figure 9: SQL text of a monitored SQL statement.

The SQL Text window shows the full text of the statement as well as information about bind variables used in the execution, including their values. Bind information is shown in a separate tab of the SQL text window. Information on bind variables is important to know because their value can change selectivity of SQL predicates, which in turn impacts both the execution plan selected by the Oracle optimizer and execution statistics. Showing bind variables is an enhancement introduced into SQL monitoring starting with Oracle database 11g Release 2.

The next entry in the Overview box is called “Parallel” and indicates the degrees of parallelism (DOP) for the query. In our case the query had DOP of 16:

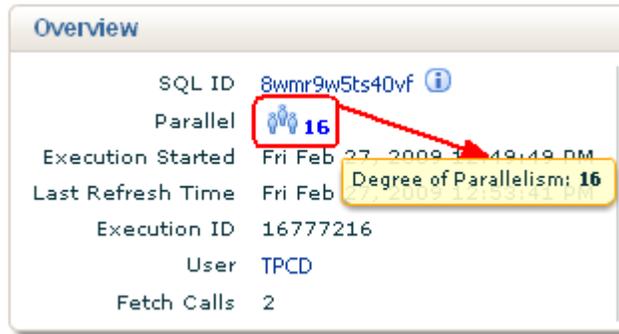


Figure 10: degree of parallelism for a monitored SQL statement.

This particular query was run on a single instance Oracle Database. Thus we only have one number in the Parallel row. In case of RAC databases that support multiple instances, the number of instances involved in processing this query will be indicated next to the DOP icon.

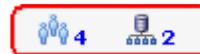


Figure 11: degree of parallelism for a monitored SQL statement.

Note that a Degree of Parallelism (DOP) of 16 does not necessarily mean that only 16 parallel servers were allocated to execute the query. Indeed, the DOP indicates how many parallel execution servers are allocated per set of parallel execution servers. Oracle allocates only one set to process simple queries while it will allocate two sets of DOP processes each for more complex queries. In the latter case, the reason we use the same DOP (i.e. 16 in our example and not 32) is because the speedup is going to be at most 16 and not 32, even though 32 parallel execution servers are allocated. That happens because of the normal dynamics of parallel execution: the two sets of parallel servers are exchanging data in pipeline where either producers or consumers active in turns.

An example of the first case could be a simple query, such as `select * from table_name`, where only one set of parallel servers is needed to scan the table. Usually parallelization occurs at the level of running one operation in parallel (in our example – a table scan). A more complex query would generally involve at least one pipeline. For example, assume that an aggregate is computed on the result of the above simple select query. In that case, rows produced by the full table scan operation will be pipelined to a second set of parallel servers assigned to a GROUP BY operation. Thus the incoming data from the first (producer) set is redistributed to the second (consumer) set. Please see the TABLE ACCESS and GROUP BY operations highlighted on the graphical SQL plan. The red arrows indicate the data communication between two sets of parallel servers:

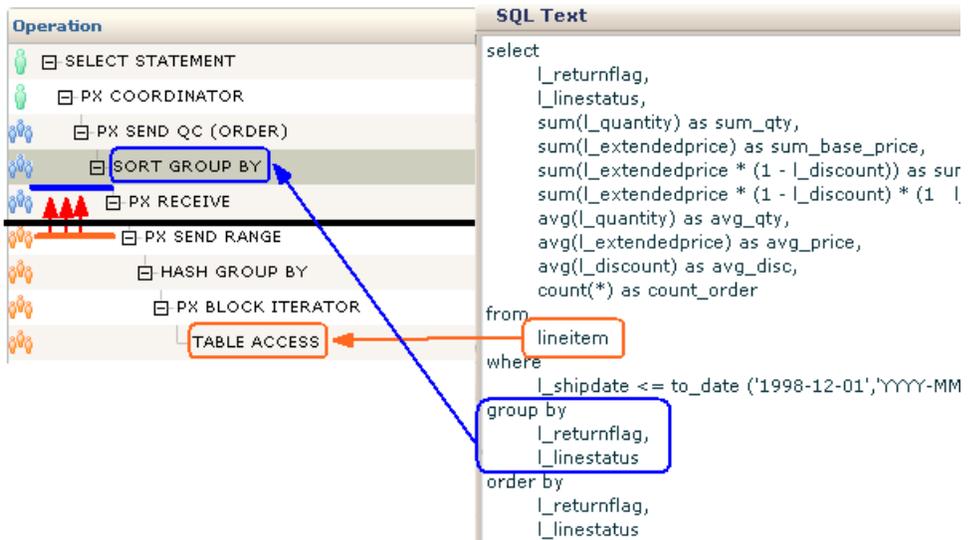


Figure 12: correlating SQL execution plan operations to SQL text.

The lower part of the “Monitored SQL Execution Details” page has several tabs. The first one shown by default displays line by line statistics on the execution plan of the statement being monitored:

Operation	Name	Estimate...	Cost	Timeline(232s)	Exec...	Actaa...	Memo...	Temp ...	IO Requests	Cell O...	CPU Activity %	Wait Activity %
SELECT STATEMENT					33	4						
PX COORDINATOR					33	4					0.00	
PX SEND QC (ORDER)	:TQ10001	5	90K		16	4						
SORT GROUP BY		5	90K		16	4	0192					
PX RECEIVE		5	90K		16	54						
PX SEND RANGE	:TQ40000	5	90K		16	54	13M				95	
HASH GROUP BY		5	90K		16	1775M						
PX BLOCK ITERATOR		1737M	72K		16	1775M						
TABLE ACCESS	LINEITEM	1737M	72K		1296	1775M			206K	77	4.65	100

Figure 13: plan operations of a monitored SQL execution.

The first few columns (operation, name, estimated rows and cost) show optimizer information about the execution plan.

The column **Name** displays the name of the object associated with a plan operation. For example, the table name *LINEITEM* is associated with the *TABLE ACCESS* operation. Object names like *:TQ10000* represent the name of parallel table queues. Table queues are temporary in-memory objects created for the duration of the parallel query execution and form the bases of the mechanism to exchange rows in pipeline between two sets of parallel servers (see *:TQ10000* on our example) or between one set of parallel servers and the query coordinator (see *:TQ10001* on our example).

The **Estimated Rows** column displays the numbers of rows that are expected to be the input for the specific plan operation, as estimated by the Oracle optimizer when the statement was compiled.

Similarly, the **Cost** column displays the cost estimated by the optimizer for the entire sub-tree rooted by a specific operation. The cost is expressed in internal units.

The other columns (starting from the Timeline column) display run-time statistics collected by SQL monitor for each line of the plan. Additionally, when the query executes parallel, SQL monitor also indicates the type of parallel entities that executes each operation. This information is shown using a small icon placed on the left side of the operation name. There are three possibilities here, all illustrated on our example: the parallel query coordinator executes the operation (light green icon), the first set of parallel servers executes the operation (blue icons), and finally the second set of parallel servers executes the operation (orange icons). In our example, the second set of parallel servers (the orange set) scans the *LINEITEM* table in parallel. Each server in that set scans a subset of blocks in the table and then pre-aggregates the subset of rows it has selected. When the pre-aggregation is completed, the pre-aggregated rows are sent to the first set of parallel servers (the blue set) to perform in parallel the final aggregation. Rows are redistributed in a pipeline manner between the two sets of servers based on the value of the group by keys.

The **Timeline** column displays the relative start time, from when the query execution started, and the actual duration of each plan operation. Note that the total runtime of the query is displayed in the header of the column (e.g., in our case 232 seconds). Once the mouse pointer is over the timeline bar it is possible to see quantitative information about the duration of each operation, as well as its relative start time:

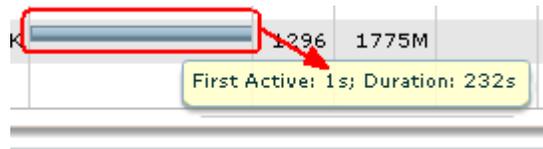


Figure 14: timeline of a SQL execution plan operation.

Note, that plan lines come in and out of being active at different stages of a query execution. For example, an operation on the right hand side of a nested-loop join will be executed multiple times, so they will have busy periods and idle periods during a SQL execution. The timeline column shows the first time an operation was active and the total duration of that activity, including idle periods, until the last time the operation was active. Finally, no timeline is shown when the operation executes for less than one second.

The **Executions** column displays the number of times each line of the plan has been executed. In our example you can see that the PX Coordinator operation was executed 33 times. This is because every process participating in the parallel execution executes that operation. The total number of process executing the

parallel query is DOP x 2 because we have two sets of parallel servers and one for the query coordinator process. If the DOP is 16 as in our example, 33 processes are executing that particular operation. Other operations in the plan are executed in parallel by either one of the two sets of servers, hence the column executions shows 16, which is equal to the DOP of our query. The Table Access operation is the exception here since it was executed 1296 times. This is explained by the fact that this operation is re-started by the PX BLOCK ITERATOR operation each time a parallel server scans a different range of blocks in the LINMEITEM table. Indeed, every time a range of blocks has been scanned, the PX BLOCK ITERATOR operation would request the next range of blocks to scan from the parallel coordinator process. Upon reception of that next range, the TABLE ACCESS operation would get once again executed with that new range of blocks. A simple calculation allows us to see the granularity of that TABLE ACCESS operation. We had a total of 1296 executions, and all of them combined scanned 247 gigabytes of data. Thus each execution of the TABLE ACCESS operation scanned a little less than 200 megabytes of data, on average. Also, on average, each parallel execution server executed the TABLE ACCESS operation 81 times (i.e. 1296 / 16). The table is divided into a large number of block ranges, many more than the total number of server processes, to enable dynamic load balancing among the parallel servers involved in the scan operation. This scheme guarantees that each server will finish at about the same time, even if some parallel servers go faster than others.

Operation	Executions
SELECT STATEMENT	33
PX COORDINATOR	33
PX SEND QC (ORDER)	16
SORT GROUP BY	16
PX RECEIVE	16
PX SEND RANGE	16
HASH GROUP BY	16
PX BLOCK ITERATOR	16
TABLE ACCESS	1296

Figure 15: number of executions column.

The **Actual Rows** column displays the actual number of rows that the plan operation produced. The number of rows is accumulated across all parallel servers working on the corresponding plan operation. This number is also cumulated, over all executions of the operation. Note that the **Actual Rows** and **Estimated Rows** column values can be different for two reasons: one reason is because the optimizer only makes an estimate, which can differ from the actual run-time value. The other reason for a mismatch is that the estimate made by the optimizer is sometimes based on one execution of the operation and is not cumulative. A good example of this is a nested-loop join operation where the right side is re-executed as many times as the number of rows on the left. The optimizer estimates the rows only for one execution of the right tree while SQL monitoring will show the cumulative number. Hence, the two numbers will differ. Saying that, comparing the actual and estimated number of rows can greatly help to uncover large variances in optimizer estimates. Such variances can cause the optimizer to select a sub-optimal plan. Bad

estimates can be corrected by making sure that statistics for the underlying database objects accessed by the query are up-to-date. Other bad estimates are often addressed by running the SQL Tuning Advisor, assuming that these bad estimates are causing the optimizer to select a sub-optimal execution plan. SQL Tuning Advisor can correct those by recommending a SQL profile.

5	90K	16	54
5	90K	16	54
1737M	72K	16	1775M
1737M	72K	1296	1775M

Figure 16: correlating estimated and actual rows.

As you can see in our case optimizer’s estimates are not too far off.

The **Memory (Max)** column displays the sum of the maximum amount of memory that was used in all execution of the specific plan operation. In our example the HASH GROUP BY operation consumed an aggregated 13 megabytes of memory highlighted by red box in Figure 17: Max memory column.

HASH GROUP BY	5	90K	16	54	13M
---------------	---	-----	----	----	-----

Figure 17: Max memory column.

Note that this operation has been executed 16 times as seen in Figure 17: Max memory column. In this example, the aggregation was performed in memory and overall used 13 megabytes. This is a little bit less than 1 megabyte per parallel server. The reason we did not have any memory consumed at earlier levels of the plan is because many operations are processing data one row at a time. Hence, they need very little run-time memory and are not reported in SQL monitoring. SQL monitoring only reports the PGA memory consumed by work areas: hash join, hash group-by, sort, bitmap merge, and bitmap create index.

Finally, note that the column Memory (Max) is only named “Memory” when the query is executing. In that case, this column reports how much memory the operation is consuming at that particular point in time. Once the query completes the maximum value of consumed memory will be reported.

The **Temp (Max)** column displays the amount of temp space consumed by its associated plan operation. A non-null value indicates that the operation has “spilled” to disk. In some case, spilling can be avoided by tuning the value of the parameter “pga_aggregate_target”. Please refer to the Oracle tuning guide for more information on how to tune this parameter.

Finally, note that the column Temp (Max) is only named “Temp” when the query is executing. In that case, this column reports how much temp space the operation is consuming at that particular point in time.

The next column **IO Requests** shows the number of read-and-write requests per plan operation. In our example, we can see that all IOs have been performed by the TABLE ACCESS operation, to scan the LINEITEM table. Once we place the mouse cursor over the IO bar, we get additional information about the number of requests and the actual bytes read as well as the average size of the IO operation, which in our case is 863 bytes.

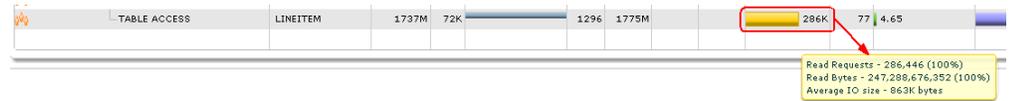


Figure 18: IO Requests column.

It is also possible to toggle the IO requests and bytes by selecting the top item in the context menu. The context menu is activated by the right mouse click.

The **Cell Offload Efficiency** column has to do with the IO efficiency factor when using Exadata storage. In our example the improvement of 77% is achieved because, of the 247 gigabytes of data read from disks, only 58 gigabytes is shipped through the IO interconnect. This corresponds to a reduction factor of 77%. This efficiency is typically achieved because projections and selections are offloaded and evaluated directly by the Exadata storage cells. Hence, only the relevant rows and columns are shipped back to the database server, saving IO bandwidth. The offloading optimization is only performed for full table scans and index fast full scans.

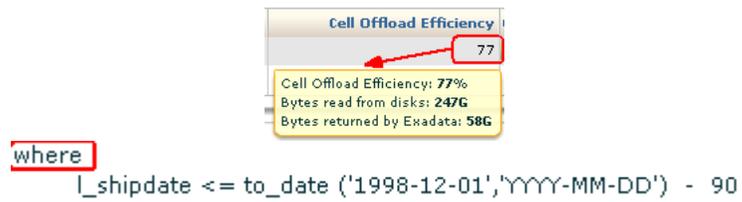


Figure 19: Cell Offload Efficiency column.

Note, the column will only be rendered if the workload is run against the Oracle Exadata V1 system.

The next column, **CPU Activity %**, indicates the share of CPU resources consumed by each operation. This percentage is derived by querying GV\$ACTIVE_SESSION_HISTORY (or ASH for short) and by counting how many CPU samples are found for each operation relatively to all CPU samples. In our example, the table access consumed only 4.65% of the CPU resources while the hash group by consumed almost all the rest (i.e. 95%). Additional information about the CPU activity is displayed when you mouseover it. It shows how many ASH samples for CPU were captured for the selected operation. Keep in mind that in the parallel query case, each parallel server is sampled every second, assuming it

is not on an idle wait. This explains the large number of samples here, 2912 samples for a query that executed for less than 240s.

Note that the accuracy of ASH estimation depends on the total number of samples observed for the statement, which is proportional to the total runtime. Thus, long running queries will have better estimates of resource breakdown by plan line. Also note that a CPU sample in ASH means that the operation was on CPU or was waiting for CPU (i.e. placed by the OS scheduler in the run queue).



Figure 20: CPU Activity column.

In our example almost all of the CPU activity happened at the HASH GROUP BY aggregation step.

The last column, **Wait Activity %**, is showing us the wait activity distribution for a query. For example, the TABLE ACCESS operation needs to wait for data coming from the Exadata cells prior to process this data, hence causing some user I/O waits.

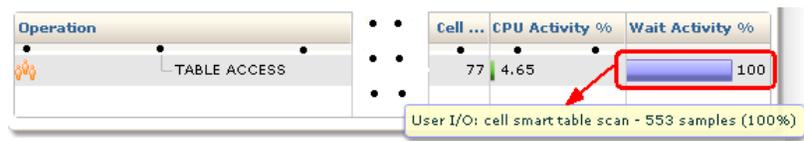


Figure 21: Wait Activity column.

In our example all the wait activity happened at the table scan level and the first instinct is to tune that large wait activity bar. However in order to correctly concentrate one's tuning efforts, a larger picture has to be considered. The question that needs to be answered before embarking on a mission of tuning User I/O waits is, "How do CPU and User I/O waits measure up against each other?" The right way to do this is to examine the DB time breakdown. If most of that time is CPU, then it makes sense to optimize for CPU.

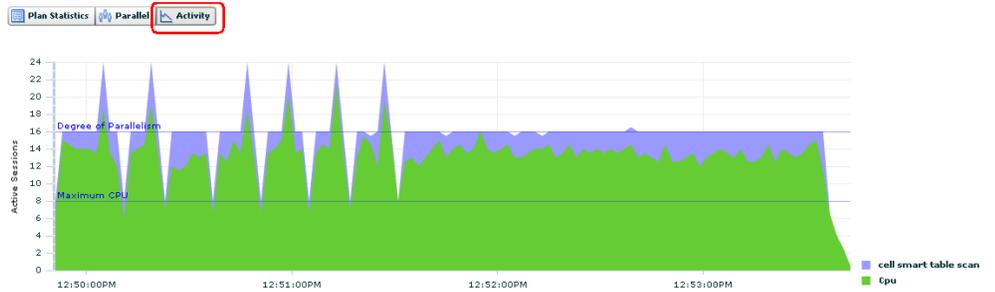


Figure 22: Activity tab on the Monitored SQL Execution Details page.

The activity tab is the 3rd tab on the Monitored SQL Execution Details page and it visualizes ASH data on the Active Sessions vs. time graph. Here we can see exactly same colors as on the Plan Statistics tab, however the colors (i.e., Waits) are presented along a different dimension.

The activity tab is useful to see how query activity spreads over time. Also in the case of Parallel Query it is useful to look for intervals where the SQL was not fully parallelized.

The first main observation from this tab is that CPU time exceed I/O waits by a wide margin, thus our tuning efforts should be concentrated on the most significant wait class rather than I/O.

Another important factor to consider is the *Maximum CPU* line, which indicates how many CPUs are available on the host where the database instance(s) run. In our case we have an 8 CPU system, hence the maximum CPU line is at 8 active sessions. Any light green area above this line will be waiting for CPU resources to become available for processing. Regardless of the fact that we have allocated 16 parallel processes to work on the query, only 8 of them can consume CPU simultaneously and the other half will wait for spare CPU resources. In our case we are completely bottlenecked on CPU. Hence, the only way to speedup this query is probably to modify the SQL execution plan or to add more CPUs resources.

To correlate these observations with the Plan Statistics tab, we come to the conclusion that most of time spent by this query is in CPU activity at the partial (i.e., lower-level) aggregation level. There are 1.7 billion rows retrieved from the table scan and only 54 came out of the pre-aggregation operation. So that query is inherently expensive (i.e. aggregating 1.7 billion rows takes a lot of CPU resources) and speeding up the execution would probably require more or faster CPUs. Alternatively, a materialized view could be creating to pre-compute that aggregation. Creating this new access path will obviously also greatly improve the execution time.

Operation	Name	Estimated Rows	Cost	Timeli...	E...	Actual Rows
HASH GROUP BY		5	90K		16	54
PX BLOCK ITERATOR		1737M	72K		16	1775M
TABLE ACCESS	LINEIT	1737M	72K		12	1775M

Figure 23: correlating actual and estimated rows.

Figure 24: long SQL plan. presents a difficult scenario for a DBA who, upon doing an explain plan on the SQL statement, needs to detect potentials problems with it. It is obvious that with such a large plan, a DBA would need some serious help!

Real-Time SQL Monitoring makes it possible to see what part of the plan is important; or in other words, which plan operations consume the most resources. From **Figure 24: long SQL plan.** it is possible to see that only a few plan operations have a significant color band next to them. Those operations are the ones that require closest attention because they are where execution time is being spent. To make things even more visually friendly, it is possible to condense some steps of the plan into one line (see SORT AGGREGATE operation in Figure 25). With one mouse click it was possible to reduce the plan from **Figure 24: long SQL plan.** to its essential part in Figure 25. Now we can see that the TABLE ACCESS FULL operation is taking most of the CPU resources, at first for the ORDERS table and then for the LINEITEM table. Thus, subsequent tuning efforts should be concentrated on making that full table scan faster and less resource intensive, or avoiding it by rewriting the SQL or creating better access structures.

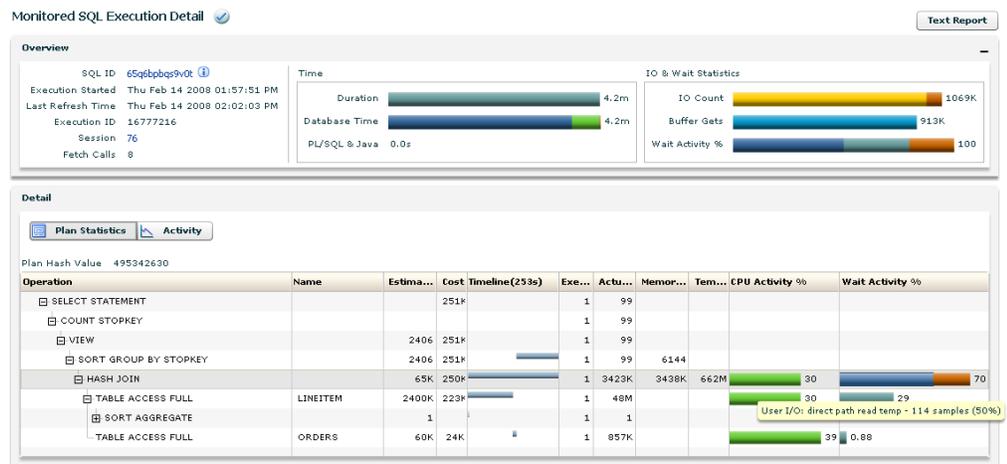


Figure 25: same big plan condensed by SQL Monitoring.

Poor index utilization strategies

The second example of using the Real-Time SQL Monitoring allows us to diagnose a case of a poor indexing strategy. At the first glance it might seem the execution plan may benefit from an index, if one exists. However, when the runtime statistics for the SQL statement are examined, the choice of an index for faster table access presents itself in a different light – see Figure 26.

Looking at the resource consumption part of the Monitored SQL Execution Detail allows us to see that most CPU resources, namely 55%, are being consumed by the INDEX RANGE SCAN operation. Looking closely at this operation, notice that the index is only doing a small amount of the filtering needed by the query - 880K rows are pulled from the index from which only 60K rows are selected. So the DBA should just notice that the query is mostly consuming CPU and most of it is

in the INDEX RANGE SCAN operation as well as most of the filtering is only done after doing a random I/O to get the row from the table.

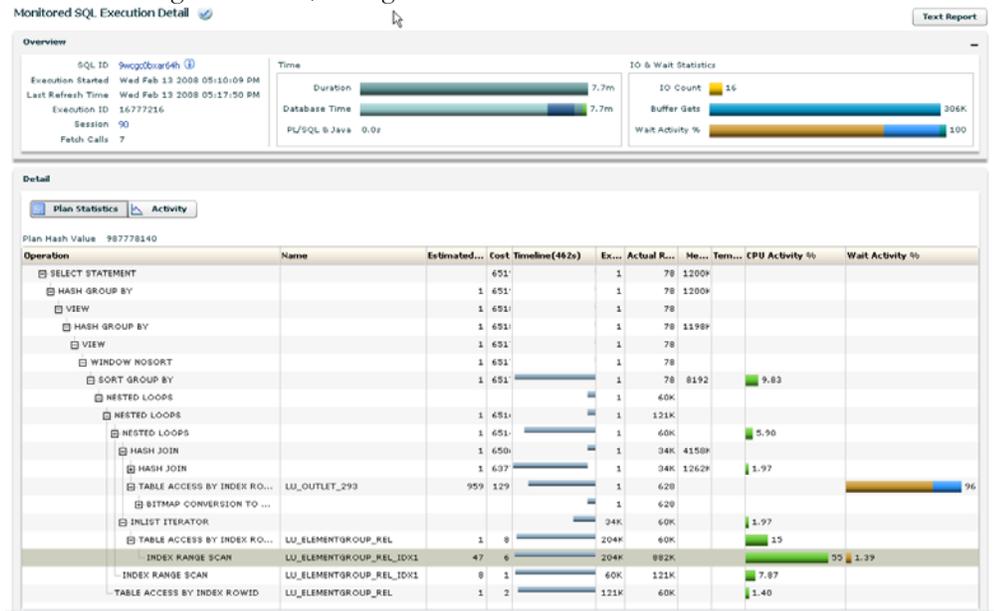


Figure 26: poor indexing strategy.

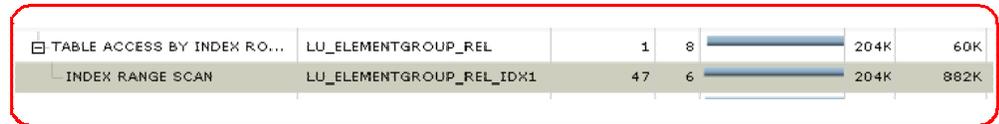


Figure 27: close up of indexing plan operation for the query from the previous figure.

Without having to spend much time it was possible to find out that the problem with the SQL statement from Figure 26 and Figure 27 had to do with the poor index utilization strategy by the optimizer (also known as, using the “unselective index”). Knowing that an index is actually harming the performance of this query, the person trying to tune this SQL query can focus the tuning efforts on the right part of the plan.

Detecting partially parallelized SQL

To reiterate, the first class of problematic SQL queries that SQL Monitoring was intended for are long-running SQL statements. The second class – are parallel SQL queries. The Real-Time SQL Monitoring allows examining how parallel queries run and especially how the workload gets distributed among Parallel Servers running that query.

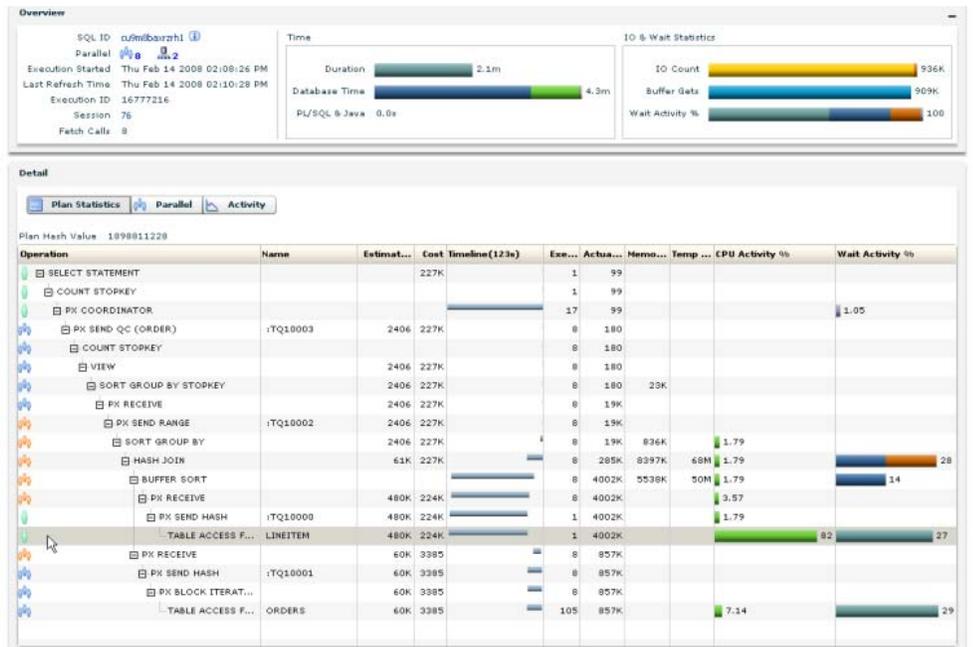


Figure 28: Partially parallelized query.

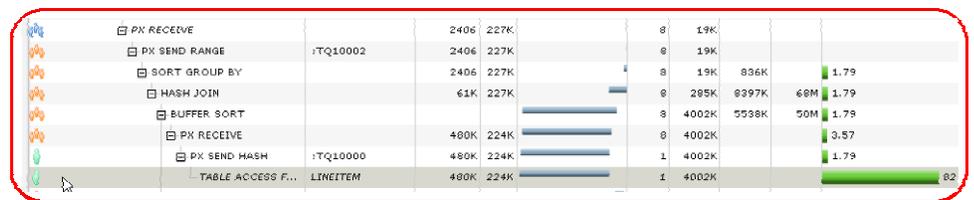


Figure 29: close up of the partially parallelized query.

The example captured in Figure 28 demonstrates a query that runs with DOP of 8 on 2 instances. Looking from top down the familiar pattern emerges that starts with the query coordinator (the light green icons) and then one set of parallel servers (the blue icons) gets alternated by the other (the orange icons). On the resource consumptions side we see one operation of the plan took 82% of all CPU consumed by the query. This operation is TABLE ACCESS FULL that returned 4002K rows and was run serially. That is an obvious case of partially parallelized query since this full table scan would have completed much faster if executed in the parallel manner.

The Parallel tab on the Monitored SQL Execution Detail (see Figure 30 and Figure 31) shows resource consumption by different participants of the parallel execution. We can immediately see how much more work the parallel query coordinator had to perform when compared to the aggregated work for each of the parallel sets. This situation is obviously abnormal. Instead of doing most of the work, the query coordinator is supposed to coordinate workload distribution between parallel execution servers, hence do very little work. Once this situation is detected the DBA responsible for SQL tuning is going to implement an action plan that will

fully parallelize the query. Here, the DBA simply need to alter the table LINEITEM parallel (i.e., alter table LINEITEM parallel).

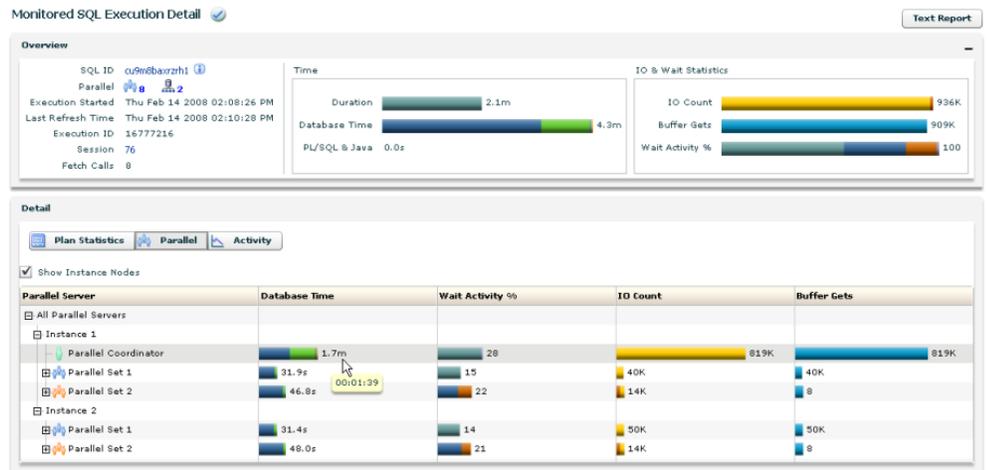


Figure 30: resource consumption across parallel servers for the partially parallelized query.

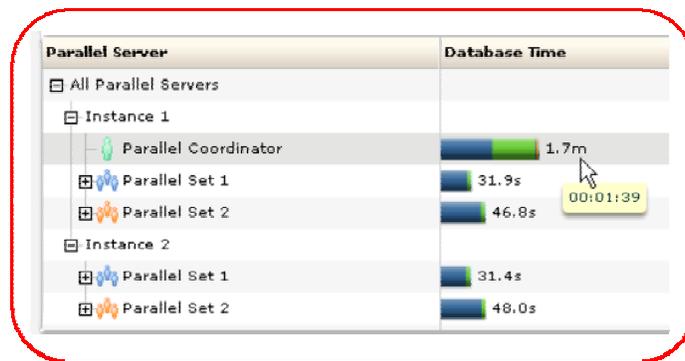


Figure 31: Close up: resource consumption across parallel servers for the partially parallelized query.

Enabling Parallel DML

Continuing on parallel execution, next is an example of a SQL statement that a user expects to run parallel but in fact it runs almost serial, using only one set of parallel servers and a query coordinator. In this use case for whatever reason the DOP for queries was forced to 4 (FORCE PARALLEL QUERY PARALLEL 4) so the expectations is to see all statements hitting the system to run 4-way parallel. However, the Insert statement on the LINEITEM table is a DML statement and DML statement are not running parallel by default, hence the load operation is the bottleneck here.

Starting with the Plan Statistics tab (see Figure 32) it is possible to see the uneven distribution of work between the parallel coordinator and the parallel server set. Most of CPU activity happens in LOAD AS SELECT and PX COORDINATOR

operation for the COUNT STOPKEY. The query coordinator, creating an obvious bottleneck, performs both of these operations.

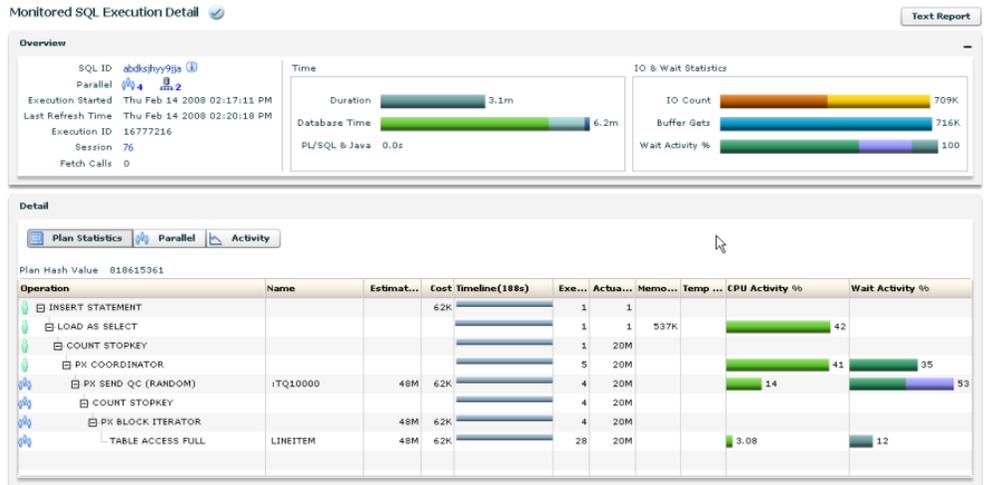


Figure 32: DML statement with forced DOP for queries.

The following Figure 33 and Figure 34 only reinforce the view that the parallel coordinator was overwhelmed and performed roughly 3 times more work than the Parallel Set 1. At this point it is clear that the DBA should be concentrating the tuning efforts on getting this DML statement run parallel. This is simply achieved by enabling parallel DML for the session (*alter session enable parallel DML*) and making sure that the LINEITEM table is parallel (*alter table lineitem parallel*).

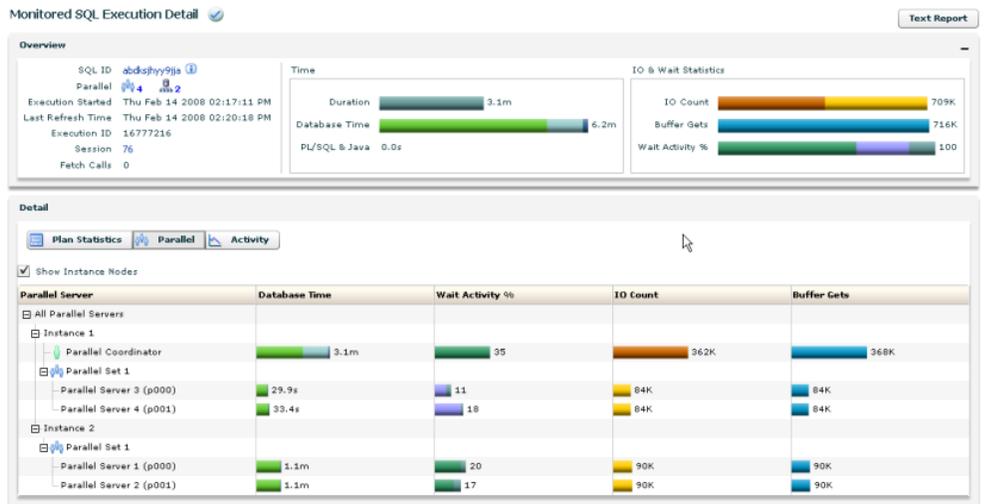


Figure 33: parallel tab for the DML statement with forced DOP for queries.

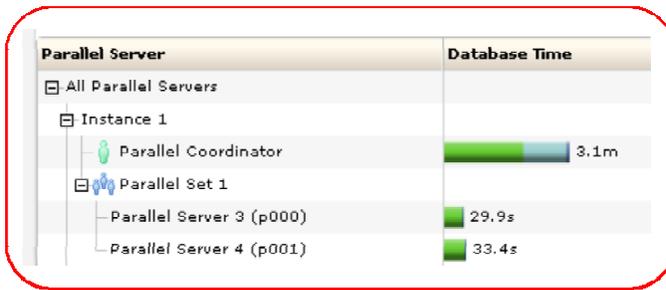


Figure 34: Close up: parallel tab for the DML statement with forced DOP for queries.

Detecting skews within PQ server groups

The final example covered in this paper has also to do with parallel queries. In fact, SQL Monitoring has provided an unprecedented way of monitoring parallel SQL statements as they are running. Making their tuning efforts a snap compared to the equivalent task in previous releases of the Oracle Database.

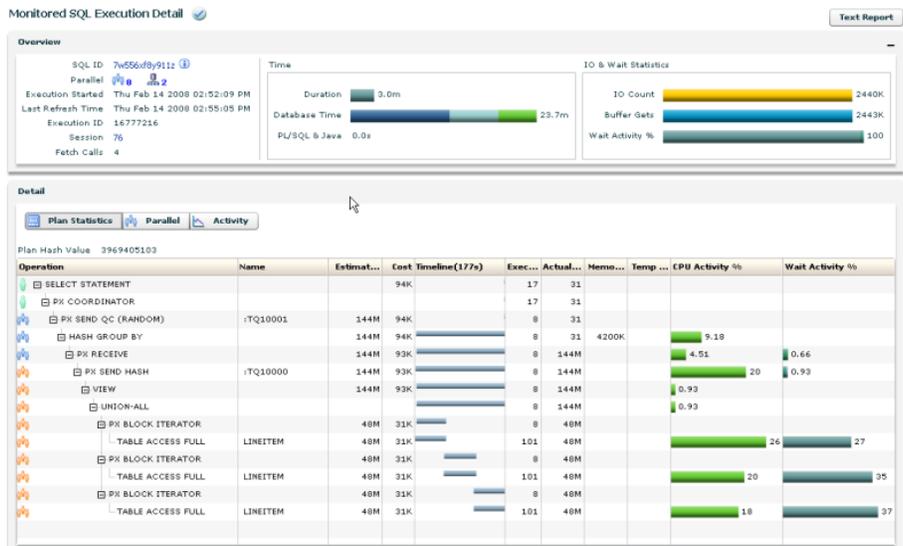


Figure 35: query with advanced PQ skews.

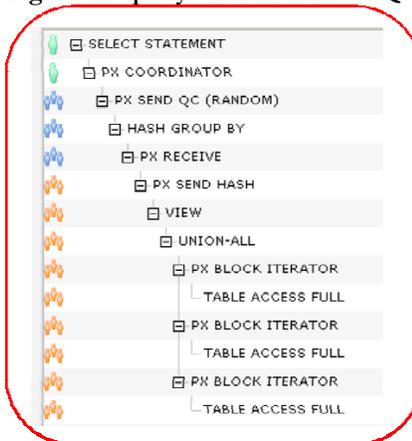


Figure 36: Close up: query with advanced PQ skews.

Once the DBA examines the plan and notices the query ran in parallel, it is natural to switch to the Parallel tab and further investigate various aspects of parallelism for the query. Here, once all the parallel sets are expanded, it is possible to see the workload distribution among parallel servers in each of the sets. On average the distribution looks even; however, if Parallel Set 1 is examined, it is possible to see that Parallel Server 3 is doing twice the amount of work of Parallel Server 2 and 8 times the amount of work of Parallel Server 4. This distribution of work is uneven, so the tuning efforts should be centered on making it more even.

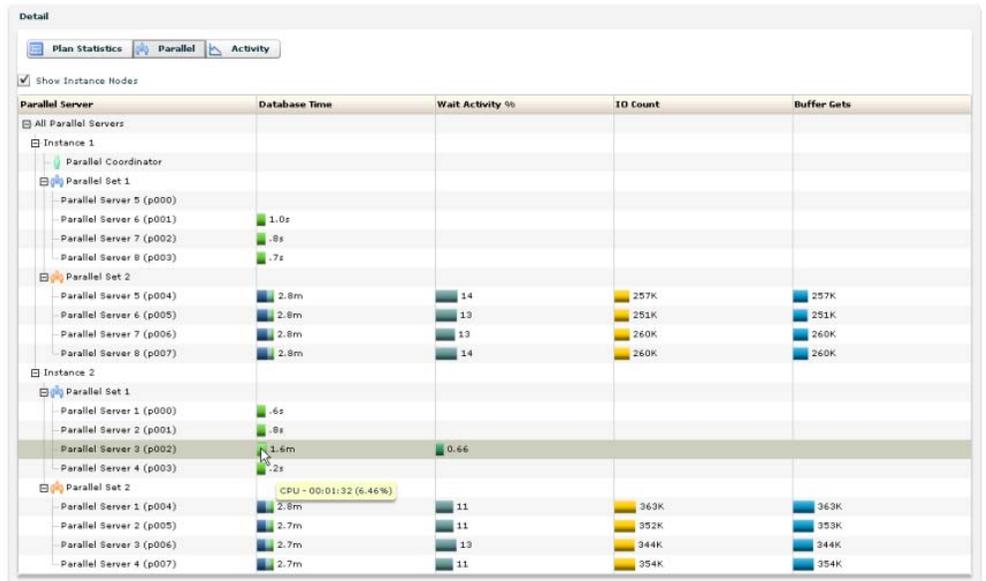


Figure 37: Parallel tab: query with advanced PQ skews.

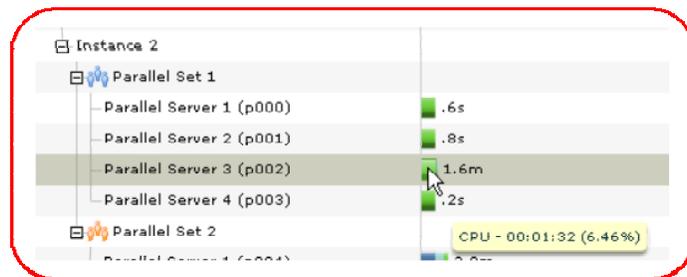


Figure 38: Close up: Parallel tab: query with advanced PQ skews.

Here is an interesting insight into how the Activity and Parallel tabs can be used in tuning efforts. Notice, the graph on the Activity tab seems to be even except for two gaps where the system seems to be idle for some time.

There are two things one needs to notice to resolve this problem. First, the Parallel tab to the Plan Statistics tab need to be correlated, which is how the user can

diagnose the issue. On the Parallel tab, one server set has a skew (some server doing more work than 3 others, see Figure 37), then from the color of that set (blue) one has to look at the plan to find out what step in the plan is being executed by that parallel set. Secondly, from the Activity tab one can find gaps when the skew was occurring, and, then, based on when the gaps occurred during the query's timeline, and the timeline information on the Plan Statistics tab, one can figure out what specific row source encountered the issue.

In this case, the skew happens at the GROUP BY level because the values of the group-by keys are skewed (one value of the group-by has most of the row). Since only one execution server handles each specific value, this server receives most of the rows to aggregate, causing this particular skew.

Tuning this query seems a little more involved in terms of tuning efforts. However, not having SQL Monitoring in the DBA's toolbox would make this tuning task next to impossible. At least SQL monitor is able to pinpoint the root cause of the problem even if finding a simple solution can be hard in that case since it involves rewriting the query to push the aggregation also inside the union all views.



Figure 39: Activity tab: Query with advanced PQ skews.

CONCLUSION

There is a wealth of runtime statistics being interactively exposed by the Real-Time SQL Monitoring. This data can be used for monitoring of long-running and parallel SQL statements. This paper covered the core concepts of the feature, went over the graphical interface provided by Enterprise Manager and illustrated a number of use cases in which Real-Time SQL Monitoring could be used to help solve important and difficult SQL tuning tasks. Hopefully this material will be helpful in getting the reader started on using this exciting new feature of Oracle Database 11g.

LICENSE INFORMATION

The Real-Time SQL Monitoring feature is licensed with Oracle Tuning Pack.

http://download.oracle.com/docs/cd/B28359_01/license.111/b28287/options.htm - CIHFIHFG)

REFERENCES

[DOCREF] Oracle® Database Reference 11g Release 1 (11.1)

http://download.oracle.com/docs/cd/B28359_01/server.111/b28274/instance_tune.htm - CACGEEIF)

APPENDIX A: COMMAND LINE USE OF SQL MONITORING

The purpose of this Appendix is to shed some light on how to use SQL Monitoring via command line APIs for users who do not have Enterprise Manager user interface available. It is worth noting that all the interactivity of EM SQL Monitoring screens will not be available however the data can still be extracted through the command line interface.

To generate the SQL monitor report, run the REPORT_SQL_MONITOR function in the DBMS_SQLTUNE package:

```
variable my_rept CLOB;
BEGIN
  :my_rept :=DBMS_SQLTUNE.REPORT_SQL_MONITOR();
END;
/
print :my_rept
```

The DBMS_SQLTUNE.REPORT_SQL_MONITOR function accepts several input parameters to specify the execution, the level of detail in the report, and the report type ('TEXT', 'HTML', or 'XML'). By default, a text report is generated for the last execution that was monitored if no parameters are specified as shown in the example.

Here is a sample text-based SQL Monitoring report:

SQL Text

```
-----
select * from (select O_ORDERDATE, sum(O_TOTALPRICE)
                from orders o, lineitem l
                where l.l_orderkey = o.o_orderkey
                group by o_orderdate
                order by o_orderdate) where rownum < 100
-----
```

Global Information

```
Status          : EXECUTING
Instance ID     : 1
Session ID      : 980
SQL ID          : br4m75c20p97h
SQL Execution ID : 16777219
Plan Hash Value : 2992965678
Execution Started : 06/07/2007 08:36:42
First Refresh Time : 06/07/2007 08:36:46
Last Refresh Time : 06/07/2007 08:40:02
-----
```

Elapsed Time(s)	Cpu Time(s)	IO Waits(s)	Application Waits(s)	Other Waits(s)	Buffer Gets	Reads	Writes
198	140	56	0.31	1.44	1195K	1264K	84630

SQL Plan Monitoring Details

Id	Operation	Name	Rows (Estim)	Cost	Time Active(s)
0	SELECT STATEMENT			125K	
1	COUNT STOPKEY				
2	VIEW		2406	125K	
3	SORT GROUP BY STOPKEY		2406	125K	99 +101
-> 4	HASH JOIN		8984K	123K	189 +12
5	INDEX FAST FULL SCAN	I_L_OKEY	8984K	63191	82 +1
6	PARTITION RANGE ALL		44913K	57676	94 +84
7	PARTITION HASH ALL		44913K	57676	94 +84
8	TABLE ACCESS FULL	ORDERS	44913K	57676	95 +84

continuation of above table (additional columns to the right)

Starts Progress	Rows (Actual)	Memory	Temp	Activity (percent)	Activity Detail (sample #)	
1						
1						
1						
1	0			4.02	Cpu (8)	
1	28130K	10000K	724M	25.13	Cpu (48)	87%
					direct path read temp (2)	
1	32734K			34.17	Cpu (58)	100%
					direct path read (10)	
1	45000K					
84	45000K					
672	45000K			36.68	Cpu (28)	
					reliable message (3)	
					direct path read (42)	

There are few things to notice about this sample report. In the Global Information section of the report, the *Status* field shows that this statement is still executing. The *Time Active(s)* column shows how long the operation has been active (the delta in seconds between the first and the last active time). The *Start Active* column shows, in seconds, when the operation in the execution plan started relative to the SQL statement execution start time. In this report, the fast full scan operation at ID 5 was the first to start (+1s Start Active) and ran for the first 82 seconds. The *Starts* column shows the number of times the operation in the execution plan was executed. The *Rows (Actual)* column indicates the number of rows produced, and the *Rows (Estim)* column shows the estimated cardinality from the optimizer. The *Memory* and *Temp* columns indicate the amount of memory and temporary space consumed by each operation of the execution plan. The *Activity (percent)* and *Activity Detail (sample #)* columns are derived by joining the V\$SQL_PLAN_MONITOR and V\$ACTIVE_SESSION_HISTORY views. *Activity (percent)* shows the percentage of database time consumed by each operation of the execution plan. *Activity Detail (sample#)* shows the nature of that activity (i.e., CPU or wait event). In the report, this column shows that most of the database time, 36.68%, is consumed by operation ID 8 (TABLE ACCESS FULL of ORDERS). This activity consists of 73 samples (28+3+42), of which more than half of the activity is attributed to direct path read (42 samples), and a third to CPU (28 samples). The last column, *Progress*, shows progress monitoring information for the operation from the V\$SESSION_LONGOPS view. In this report, it shows that the hash-join operation is 87% complete.



Real-Time SQL Monitoring

December 2009

Author: Sergey Koltakov

Contributing Authors: Benoit Dageville, Peter Belknap

Oracle Corporation

World Headquarters

500 Oracle Parkway

Redwood Shores, CA 94065

U.S.A.

Worldwide Inquiries:

Phone: +1.650.506.7000

Fax: +1.650.506.7200

oracle.com

Copyright © 2009, Oracle and/or its affiliates. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission. Oracle is a registered trademark of Oracle Corporation and/or its affiliates.

Other names may be trademarks of their respective owners. 0408