Are you getting the most out of your Exadata?

Part 1: Basic Smart Scans - version 1.02
By Tanel Poder (tanel@tanelpoder.com)

Welcome to reading my first whitepaper in the Exadata Performance series. I will spare you from yet another echo of the marketing material and copy of the Exadata spec-sheets. Instead I will explain a few scenarios here, where you might not be getting the most out of your Exadata investment and how to detect that yourself. We'll look into Exadata performance in a Data Warehousing workload context.

Whenever I have been involved in a migration from "old" platform to Exadata, the applications usually run much faster on the Exadata platform – as expected. This can happen thanks to all the fundamental improvements of Exadata (the marketing stuff you constantly hear about), but some of the extra performance may just come from the fact that you migrated to Oracle 11.2 from your old database version. And some of the performance may come from running on *way* faster CPUs than your 5-year-old big iron box had in it. Yeah, that's a wild generalization here, but my point is that even if you are happy with *your* Exadata experience so far, you might not actually be fully using all the benefits that Exadata can offer!

Checking Whether Smart Scanning Is Used

Let's first see how to identify whether your queries are taking full advantage of the Exadata "secret sauce" – Smart Scans. Here's a simple execution plan, which has some Exadata-specific elements in it:

```
-------------------------------------------------------------------------
| Id  | Operation                         | Name        | Pstart| Pstop |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |             |       |       |
|*  1 |  FILTER                           |             |       |       |
|   2 |   HASH GROUP BY                   |             |       |       |
|*  3 |    HASH JOIN                      |             |       |       |
|   4 |     PART JOIN FILTER CREATE       | :BF0000     |       |       |
|   5 |      PARTITION HASH ALL           |             |     1 |    16 |
|*  6 |       HASH JOIN                   |             |       |       |
|*  7 |        TABLE ACCESS STORAGE FULL| ORDERS      |     1 |    16 |
|   8 |        TABLE ACCESS STORAGE FULL| ORDER_ITEMS |     1 |    16 |
|   9 |     PARTITION HASH JOIN-FILTER    |             |:BF0000|:BF0000|
|* 10 |      TABLE ACCESS STORAGE FULL    | CUSTOMERS   |:BF0000|:BF0000|
-------------------------------------------------------------------------

   1 - filter("C"."CREDIT_LIMIT"<MAX("OI"."UNIT_PRICE"*"OI"."QUANTITY"))
   3 - access("C"."CUSTOMER_ID"="O"."CUSTOMER_ID")
   6 - access("O"."ORDER_ID"="OI"."ORDER_ID")
   7 - storage(("O"."ORDER_STATUS"=5 AND "O"."ORDER_MODE"='online'))
       filter(("O"."ORDER_STATUS"=5 AND "O"."ORDER_MODE"='online'))
  10 - storage("C"."NLS_TERRITORY"='AMERICA')
       filter("C"."NLS_TERRITORY"='AMERICA')
```

All the TABLE ACCESS FULL row sources do have the **STORAGE** keyword in them. It is important to know that this does *not* mean that Exadata smart scans are used. The STORAGE keyword in row sources means that this execution plan is using an Exadata-storage aware and smart scan capable row source, but it doesn't tell us whether a smart scan actually happened for given execution. Also, in the bottom you see two ***storage()*** predicates in addition to the usual *filter()* and *access()* ones. This shows that *if* a smart scan were chosen for scanning some segments during given SQL execution, then Oracle would

attempt to offload the predicate conditions into the cells. If the smart scan was not used, then predicate offloading did not happen either (as the filter predicate offloading is part of smart scan).

Important:
The key thing to remember is that Exadata smart scan is not only a property of an execution plan - it's actually a runtime decision done separately for each table/index/partition segment accessed with a full scan. It is not possible to determine whether a smart scan happened just by looking into the execution plan, you should measure execution metrics from V$SQL or V$SESSION to be sure.

It is also possible that a smart scan is attempted against a segment, but during the smart scan execution, the cells have to fall back to regular block IO mode for some blocks and ship the blocks back to database for processing - instead of extracting rows from them inside the cell. For example, this happens for blocks for which the consistent reads require access to undo data or there are chained rows in a block. There are more reasons and we explain these in detail in the upcoming Expert Oracle Exadata book.

This should also explain why is there a *filter* predicate (executed in the database layer) in addition to every *storage* predicate in the plan, because sometimes the filtering cannot be entirely offloaded to the storage cells and the database layer has to perform the final filtering.

Serial Execution Without Smart Scan

Let's look into an example query now, executed in serial mode at first. Basically it's returning all customers who have ever made orders where the total order amount exceeds their customer-specific credit limit. All the tables are partitioned and their total size is around 11GB. Note that this query is written against a regular normalized OLTP-style schema, not a star- or snowflake schema which is you are likely using in your DW databases. But for purpose of this demo it should be enough.

```
SELECT
    c.customer_id
  , c.cust_first_name ||' '||c.cust_last_name
  , c.credit_limit
  , MAX(oi.unit_price * oi.quantity) max_order_total
FROM
    soe.orders o
  , soe.order_items oi
  , soe.customers c
WHERE
-- join conditions
    c.customer_id = o.customer_id
AND o.order_id    = oi.order_id
-- constant filter conditions
AND c.nls_territory = 'AMERICA'
AND o.order_mode = 'online'
AND o.order_status = 5
GROUP BY
    c.customer_id
  , c.cust_first_name ||' '||c.cust_last_name
  , c.credit_limit
HAVING
    MAX(oi.unit_price * oi.quantity) > c.credit_limit;
```

When executed on an otherwise idle quarter rack Exadata V2 installation, it took 151 seconds to run, which seems too much, knowing that the smart scans on even a quarter rack

can scan data on hard disks multiple gigabytes per second (and even faster from flash cache).

So, I will identify the SQL ID, child cursor number and execution ID of this current ongoing SQL execution first:

```
SQL> SELECT sql_id, sql_child_number, sql_exec_id
  2> FROM v$session WHERE sid=200;

SQL_ID        SQL_CHILD_NUMBER SQL_EXEC_ID
------------- ---------------- -----------
9n2fg7abbcfyx                0    16777224
```

Now, let's look into V$SQL first, using the SQL ID and child cursor number:

```
SQL> SELECT
  2      ROUND(physical_read_bytes/1048576)    phyrd_mb
  3    , ROUND(io_cell_offload_eligible_bytes/1048576) elig_mb
  4    , ROUND(io_interconnect_bytes/1048576)  ret_mb
  5    , (1-(io_interconnect_bytes/NULLIF(physical_read_bytes,0)))*100 "SAVING%"
  6  FROM
  7      v$sql
  8  WHERE
  9      sql_id = '9n2fg7abbcfyx'
 10  AND child_number = 0;

  PHYRD_MB    ELIG_MB     RET_MB    SAVING%
---------- ---------- ---------- ----------
     10833          0      10833          0
```

The *physical_read_bytes* metric (phyrd_mb) shows that Oracle database layer has issued 10833 MB worth of IO calls for this SQL. And the *io_interconnect_bytes* (ret_mb) shows that this query has also used 10833 MB worth of IO interconnect traffic (between database host and storage cells). So, the smart scans were not able to reduce the cell-database IO traffic at all. In fact, when looking into *io_cell_offload_eligible_bytes* (elig_mb), it's zero. This means that the database has not even tried to do smart scan offloading for this statement. If 10GB worth of segments would be read via smart scans, then the "eligible bytes for offload" would also show 10GB. So, this *io_cell_offload_eligible_bytes* metric is a key for determining whether any offloading has been attempted for a query. Note that V$SQL accumulates statistics over multiple executions of the same query, so if this cursor has already been executed before (and is still in cache) you should not look into the absolute values in the V$SQL columns, but rather by how much they increase (calculate deltas from before- and after-test values).

Another option is to look into what the session(s) executing this SQL are doing and waiting for. You can use SQL trace or ASH for this, or any other tool, which is capable of extracting the wait information from Oracle. Here's a wait profile example from the problem session, taken with Oracle Session Snapper (a free Oracle troubleshooting tool downloadable from my blog):

```
-------------------------------------------------------------------------------
Active% | SQL_ID        | EVENT                          | WAIT_CLASS
-------------------------------------------------------------------------------
    49% | 9n2fg7abbcfyx | cell multiblock physical read | User I/O
    40% | 9n2fg7abbcfyx | ON CPU                         | ON CPU
    10% | 9n2fg7abbcfyx | gc cr multi block request      | Cluster
```

Indeed we aren't seeing the *cell smart table scan* (or *cell smart index scan*) wait events, but a *cell multiblock physical read* wait event as the top one – which shows that half of the query time is spent doing regular multiblock IOs from storage to database. Note that there's still a chance that smart scans are happening for some tables involved in the query (but they don't show up in the top waits thanks to their fast, asynchronous nature), but the regular multiblock reads show up thanks to rest of the tables not using smart scan. Luckily it's possible to check exactly which execution plan row sources do use the smart scan and how much do they benefit from it.

We can get this information from the Real Time SQL Monitoring views, either by manually querying V$SQL_PLAN_MONITOR or by Grid Control/Database Control UI tools. I like to always know where the data is coming from, so let's start from a manual query. Note that the sql_id and sql_exec_id values are taken from my previous query against V$SESSION above:

```
SQL> SELECT
  2      plan_line_id id
  3    , LPAD(' ',plan_depth) || plan_operation
  4        ||' '||plan_options||' '
  5        ||plan_object_name operation
  6    , ROUND(physical_read_bytes   /1048576) phyrd_mb
  7    , ROUND(io_interconnect_bytes /1048576) ret_mb
  8    , (1-(io_interconnect_bytes/NULLIF(physical_read_bytes,0)))*100 "SAVING%"
  9  FROM
 10      v$sql_plan_monitor
 11  WHERE
 12      sql_id = '9n2fg7abbcfyx'
 13  AND sql_exec_id = 16777224;


 ID OPERATION                                      PHYRD_MB     RET_MB  SAVING%
--- ---------------------------------------------- ---------- ---------- --------
  0 SELECT STATEMENT                                       0          0
  1  FILTER                                                0          0
  2   HASH GROUP BY                                        0          0
  3    HASH JOIN                                           0          0
  4     PART JOIN FILTER CREATE :BF0000                    0          0
  5      HASH JOIN                                         0          0
  6       PART JOIN FILTER CREATE :BF0001                  0          0
  7        PARTITION HASH ALL                              0          0
  8         TABLE ACCESS STORAGE FULL ORDERS            2038       2038          0
  9       PARTITION HASH JOIN-FILTER                       0          0
 10        TABLE ACCESS STORAGE FULL CUSTOMERS         3943       3943          0
 11     PARTITION HASH JOIN-FILTER                       0          0
 12      TABLE ACCESS STORAGE FULL ORDER_ITEMS          4834       4834          0
```

The V$SQL_PLAN_MONITOR doesn't have the io_offload_eligible_bytes column showing how many bytes worth of segments were scanned using smart scans, but nevertheless the *io_interconnect_bytes* column tells us how many bytes of IO traffic between the database host and cells were done for the given access path. In the above example, exactly the same amount of data was returned by cells (RET_MB) as requested by database (PHYRD_MB), so there was no interconnect traffic saving at all. When these numbers exactly match, this is a good indication that no offloading was performed (as all the IO requested by database was returned in untouched blocks to the database and no data reduction due to filtering and projection offloading was performed in the cells). So, this is additional confirmation that none of the tables were scanned with smart scan.

I caused the above problem deliberately. I just set the undocumented parameter *_small_table_threshhold* to a big value (1000000 blocks) in my testing session. This made Oracle table scanning function think that all the scanned partitions were "small" segments,

which should be scanned using regular buffered reads as opposed to direct path reads, which are a pre-requisite for smart scans. Note that the above query was executed in serial mode, not parallel. Starting from Oracle 11g, Oracle can decide to use direct path reads (and as a result smart scans) even for serial sessions. The direct path read decision is done during runtime, separately for each table/index/partition segment accessed – and it's dependent on how many blocks this segment has under its HWM and what's the buffer cache size etc. So, you might encounter such an issue of Oracle deciding to not use direct path reads (and thus smart scan) in real life too – and as it's an automatic decision, it may change unexpectedly.

Serial Execution With Smart Scan

Let's run the same query without my trick to disable smart scans, I'm using default session parameters now. The query, still executed in serial, tool 26 seconds (as opposed to 151 seconds previously). The wait profile looks different, the *cell smart table scan* wait is present and there are no regular block IO related waits. The CPU usage percentage is higher, but remember that this query completed over 5 times faster than previously. Higher CPU usage with lower response time is a good thing, as high CPU usage means that your query spent less time waiting instead of working and wasn't throttled by IO and other issues.

```
--------------------------------------------------------------------------------
Active% | SQL_ID          | EVENT                                | WAIT_CLASS
--------------------------------------------------------------------------------
    73% | 9n2fg7abbcfyx   | ON CPU                               | ON CPU
    28% | 9n2fg7abbcfyx   | cell smart table scan                | User I/O
```

For simple enough queries you might actually see that only a couple of percent of response time is spent waiting for smart scans and all the rest is CPU usage. This happens thanks to the asynchronous nature of smart scans, where cells work for the database sessions independently and may be able to constantly have some data ready for the DB session to consume.

Now, let's look into V$SQL entries of that cursor (a new child cursor 1 was parsed thanks to me changing the *_small_table_threshold* parameter value back):

```
SQL> SELECT
  2      ROUND(physical_read_bytes/1048576)    phyrd_mb
  3    , ROUND(io_cell_offload_eligible_bytes/1048576) elig_mb
  4    , ROUND(io_interconnect_bytes/1048576)  ret_mb
  5    , (1-(io_interconnect_bytes/NULLIF(physical_read_bytes,0)))*100 "SAVING%"
  6  FROM
  7      v$sql
  8  WHERE
  9      sql_id = '9n2fg7abbcfyx'
 10  AND child_number = 1;

  PHYRD_MB    ELIG_MB     RET_MB    SAVING%
---------- ---------- ---------- ----------
     10815      10815       3328       69.2%
```

Apparently all the physical IOs requested were requested using the smart scanning method, as ELIG_MB is the same as PHYRD_MB. Apparently some filtering and projection was done in the cell as the interconnect traffic (RET_MB) of that statement is about 69.2% less than the scanned segments sizes on disk.

Let's drill down into individual execution plan lines, which allows us to look into the efficiency of accessing individual tables. In real life DWs you are more likely dealing with 10..20 table joins, not 3-table joins. Note that the sql_exec_id has increased as I've re-executed the SQL statement:

```
SQL> SELECT
  2      plan_line_id id
  3    , LPAD(' ',plan_depth) || plan_operation
  4        ||' '||plan_options||' '
  5        ||plan_object_name operation
  6    , ROUND(physical_read_bytes   /1048576) phyrd_mb
  7    , ROUND(io_interconnect_bytes /1048576) ret_mb
  8    , (1-(io_interconnect_bytes/NULLIF(physical_read_bytes,0)))*100 "SAVING%"
  9  FROM
 10      v$sql_plan_monitor
 11  WHERE
 12      sql_id = '9n2fg7abbcfyx'
 13  AND sql_exec_id = 16777225;

 ID OPERATION                                      PHYRD_MB     RET_MB SAVING%
--- ------------------------------------------- ---------- ---------- -------
  0 SELECT STATEMENT                                    0          0
  1  FILTER                                             0          0
  2   HASH GROUP BY                                     0          0
  3    HASH JOIN                                        0          0
  4     PART JOIN FILTER CREATE :BF0000                 0          0
  5      PARTITION HASH ALL                             0          0
  6       HASH JOIN                                     0          0
  7        TABLE ACCESS STORAGE FULL ORDERS          2038          2   99.9
  8        TABLE ACCESS STORAGE FULL ORDER_ITEMS     4834       3125   35.3
  9     PARTITION HASH JOIN-FILTER                      0          0
 10      TABLE ACCESS STORAGE FULL CUSTOMERS          3943        201   94.9
```

The above output clearly shows that the ORDERS and CUSTOMERS tables benefit from smart scan the most.

Note that the execution plan join order has changed too, this is thanks to the automatic cursor re-optimization using *cardinality feedback* from previous child cursor's execution statistics.

Summary

In order to get the most out of your Exadata performance for your DW application, you'll have to use smart scans. Otherwise you won't be able to take advantage of the cells computing power and its IO reduction features. Full segment scans (like full table/partition scans and fast full index scans) are a pre-requisite for smart scans. Additionally, direct path reads have to be used in order to the smart scans to kick in. For serial sessions, the direct path read decision is done based on the scanned segment size, buffer cache size and some other factors. For parallel execution, direct path access is always used for full scans, unless you use the new *parallel_degree_policy = AUTO* feature, in which case the decision would again be dynamic.

In this article we only managed to touch the surface of all the optimization and benefits that Exadata gives us. We didn't even look into parallel execution yet, although from smart scanning perspective it's not too different from serial execution. Also, I ran out of space before I got to explain the bloom filter pushdown to the cells, which would allow to push the ":BF000x" filters that you may see in the execution plans with hash joins, all the way to the cells - to reduce the amount of returned data even more. Well, hopefully in a future article ☺