Exadata Performance Tuning – SmartScans and Parallelism

Mark Smith, Database Specialists

ABSTRACT

We all have to do "more with less" and focus on "immediate ROI" - as a DBA, it can be difficult to allocate time and/or money on anything other than "firefighting"/"keeping the lights on".

Managing Exadata appropriately can realize exponential performance improvements to key IT infrastructure facilitate better business decisions and actually reduce costs.

The customer has bought a sports car - but sometimes doesn't realize that they haven't taken it out of second gear. Yet.

COLLABORATE Abstract
DBA 3.0 – How to Become a Real-World Exadata DBA (Presentation)
White Paper
This paper focuses on the most important feature of Exadata – the SmartScan. The paper explains how SmartScans work, how to make your queries eligible for SmartScans and how to manage indexes and parallelism on an Exadata database.

TARGET AUDIENCE

Database administrators / Exadata administrators / Exadata data machine administrators / developers on Exadata databases.

EXECUTIVE SUMMARY

After attending the presentation, the attendee should have gained an insight into how "modern" DBAs support Exadata and be able to take a "deep-dive" into the most important Exadata performance feature - SmartScans.

After attending the presentation and reading the white paper, the learner will be able to:

- Make appropriate configuration decisions during the initial Exadata installation.
- Maximize and monitor Exadata's performance features such as SmartScan and EHCC.
- Troubleshoot / performance-tune the use of Exadata SmartScans in their database.

EXADATA PERFORMANCE AND SMARTSCANS

Without specific tuning and administration, it can be very difficult to justify the additional expense of an Exadata machine. If treated like a "normal" Oracle database - in configuration and/or support - the customer will not be able to maximize (or, perhaps, use at all) the specific features which allow Exadata to achieve very high levels of performance.

On average, Oracle's benchmarks indicate that an Exadata machine - without tuning - provides a performance improvement of **3x** over a comparable non-Exadata system thanks to the hardware and software integration. Such customers are, informally, considered to be in "*The 3x Club*".

However, with the right configuration and support, a well-tuned Exadata system should realize a **8-10x performance improvement**, with some systems benefiting from up to a **15x improvement.**

This paper describes how SmartScans work, how to make your queries eligible for SmartScans and how to manage parallelism on an Exadata database.

How SmartScans Work

The most important Exadata-specific feature - often called the "*Exadata secret sauce*" - is **storage cell offloading**, also known as **Exadata SmartScans**.

Exadata is able to offload data-intensive queries to the storage cells instead of the compute nodes for processing. Each storage cell has its own memory and CPU and makes use of storage indexes to filter the data by eliminating entire regions of data from consideration.

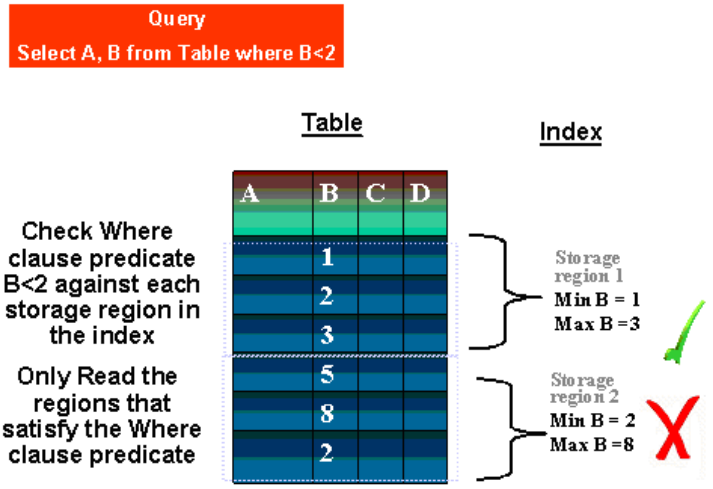
Instead of loading huge amounts of unnecessary data into the database instance's memory, **SmartScans will only bring back the data needed for processing**. This massively reduces the burden of the database instance and frees resources which can allow it to perform complex database processing such as joins and aggregation.

Exadata Storage Indexes

Storage indexes reside on the storage cells and maintain summary information about data in memory. Each disk in the storage cell is divided up into 1Mb "*regions*", each with an index and the MIN and MAX column values for up to 8 columns.

The storage index eliminates entire regions which don't match the WHERE clause of a query, thus **eliminating unnecessary I/O**. They are automatically maintained based on the SQL predicates executed by the database and which are passed down from the compute nodes to the storage cells.

Unlike database indexes – which tend to **INCLUDE** regions of data – storage indexes are designed to **EXCLUDE** regions of data from processing. They are maintained by the cell daemon on the storage cells and do not persist through cell restarts as they are memory structures.



Storage indexes are populated during the fetch of data of a SmartScan, so the first queries after a cell restart often process a lot more data; obviously, they can't exclude certain regions as the indexes are in the process of being rebuilt.

On V2 machines, offloaded processing on newly-started storage cells was **noticeably worse** until the indexes had been rebuilt. For X3-2 machines and up, this appears to be less noticeable, though the storage indexes are still transient.

Without the use of "hidden" storage cell parameters, it is not possible to influence the contents of storage indexes as they are automatically managed by the storage cell.

Storage indexes will not work in the following conditions:

- They are not created on CLOBs.
- They do not work with predicates using the != operator or the % wildcard.
- The _smu_debug_mode parameter is set to 134217728.
- The _enable_minscn_cr parameter is set to FALSE.

SmartScan Eligibility

For the optimizer to consider a SmartScan, the query must be "eligible":

- The query must perform full object scans (full-table or full-index scans).
- The query must bypass the database buffers and use direct path reads via the PGA.

Even with a full-table scan, direct path reads will not occur if the query is running serially. Using parallelism will **usually** force direct path reads and make the query eligible for SmartScans, though this is not **always** the case.

Eligible SQL functions can be seen in the V$SQLFN_METADATA view (SmartScan eligibility does not guarantee that SmartScans will happen).

Determine the SmartScan Ratio

There are a number of ways you can measure how well you're using SmartScans, but none of these seem to be the DEFINITIVE method to do so. Instead, it's probably a good idea to more than one formula, if not all, to get a good idea of our SmartScan usage.

Why are there multiple formulas? Because the existing database metrics don't quite capture what we're looking to measure. For instance:

- 'physical read total bytes' – is all the data including compressed data AND SmartScan-ineligible data.
- 'cell physical IO interconnect bytes' – includes the writes (multiplied due to ASM mirroring) AND the reads.
- 'cell IO uncompressed bytes' – is the data volume for predicate offloading AFTER the Storage Index filtering and any decompression
- 'cell physical IO interconnect bytes returned by smart scan' – includes uncompressed data.

"SmartScan Eligibility Ratio"

The percentage of the total sum of the physical reads which are eligible for SmartScan.

= (100 / 'physical read total bytes') * 'cell physical IO bytes eligible for predicate offload'

- 'physical read total bytes' – all non-system data which was read from the storage cells

- 'cell physical IO bytes eligible for predicate offload' – data that is eligible for SmartScan processing (total direct path reads)

"Storage Index Filter Ratio"

The percentage of the SmartScan-eligible data filtered by the Storage Indexes.

= **(100 / 'cell physical IO bytes eligible for predicate offload') * 'cell physical IO bytes saved by storage index'**

- 'cell physical IO bytes saved by storage index' – data that the cells did not need to scan through because the storage indexes told them they were not in that region.

"SmartScan Offloading Ratio"

The percentage of the uncompressed, filtered data returned by the SmartScan.

= **(100 / 'cell IO uncompressed bytes') * 'cell physical IO interconnect bytes returned by smart scan'**

- 'cell IO uncompressed bytes' – data read from the storage cells for predicate offloading, after the storage index filtering and data decompression. N.B. compressed data has to be uncompressed first before applying the predicates on the storage cells.

"SmartScan Efficiency Ratio"

The percentage of the SmartScan eligible data returned by the SmartScan.

This is probably the most-used / "best" ratio to use, if you could only use one.

= **100 – ((100 / 'cell physical IO bytes eligible for predicate offload') * 'cell physical IO interconnect bytes returned by smart scan')**

- 'cell physical IO interconnect bytes returned by smart scan' – data returned to the database from the SmartScan.

Subtracting the "**cell physical IO interconnect bytes returned by smart scan**" from the "**cell physical IO bytes eligible for predicate offload**" shows the amount of I/O AVOIDED by using SmartScans.

For instance:

cell physical IO bytes eligible for predicate offload = 1,511,049,773,060
MINUS
cell physical IO interconnect bytes returned by smart scan = 110,441,960,872
EQUALS
I/O in bytes avoided by using SmartScans = 1,400,607,812,188

In this case:

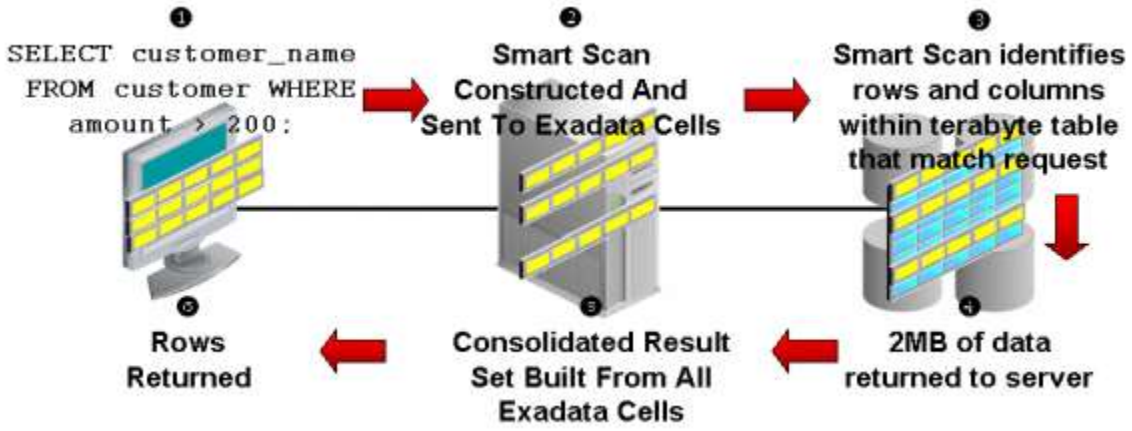
cell physical IO bytes eligible for predicate offload = 1,511,049,773,060
of which
I/O in bytes avoided by using SmartScans = 1,400,607,812,188
EQUALS
SmartScan Ratio = 92.69%

A SmartScan Example

As an example, let's say we wanted to run the following query:

```
SELECT customer_name
FROM customer
WHERE nonuniquecol > 200;
```

Let's also say that the CUSTOMER table is 1Tb in size and our query will bring back a total of 2Mb of data.



With SmartScans, we offload processing to the storage cells so they do the "*heavy-lifting*" of figuring out which data to bring back. The storage cell **only returns the 2Mb of data to the database instance** as it has already filtered out data which is not required.

To prove that we're using SmartScans, we should see a **STORAGE** clause in the operation name in the explain plan:

```
EXPLAIN PLAN FOR
SELECT customer_name
FROM customer
WHERE nonuniquecol > 200;

SELECT *
FROM TABLE(DBMS_XPLAN.DISPLAY);
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 87389835

-------------------------------------------------------------------------------
| Id  | Operation                 | Name     | Rows    | Bytes     | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          | customer | 2000000 | 2097152 | 635342 (1)| 00:00:09 |
```

| 1 | TABLE ACCESS STORAGE FULL | customer | 2000000 | 2097152 | 635342 (1)| 00:00:09 |*

Predicate Information (identified by operation id):
---

1 - storage("NONUNIQUECOL"=200)
filter("NONUNIQUECOL "=200)

Let's also check the statistics (after we've run the query):

SELECT customer_name
FROM customer
WHERE nonuniquecol > 200;

2000000 rows returned.

COL value FORMAT 99999999999999999999999999999

SELECT vsys.name,
vmys.value
FROM v$sysstat vsys,
v$mystat vmys
WHERE vsys.statistic# = vmys.statistic#
AND vsys.name IN (
'cell physical IO bytes eligible for predicate offload',
'cell physical IO interconnect bytes returned by smart scan'
)
ORDER BY vsys.name;

NAME	*VALUE*
cell physical IO bytes eligible for predicate offload	*1801489640*
cell physical IO interconnect bytes returned by smart scan	*20971520*

This tells us that without a SmartScan, we would have pulled back 1801489640 bytes from the storage, but with SmartScan, we reduced that to just 20971520 bytes – giving us a **SmartScan ratio of 98.86%.**

Next, let's check what the V$SQL view shows. We know our explain plan is 87389835 and let's assume that the SQL_ID is 5t34tq47tz3p5:

SELECT sql_id,
hash_value,
ROUND ((io_cell_offload_eligible_bytes/1024/1024),2) AS elig_mb,
ROUND ((io_cell_offload_returned_bytes/1024/1024),2) AS return_mb,

```
ROUND (((io_cell_offload_eligible_bytes - io_cell_offload_returned_bytes)/1024/1024),2) AS skip_mb,
100 - (ROUND (((100 / io_cell_offload_eligible_bytes ) * (io_cell_offload_returned_bytes)),2)) AS ratio
FROM v$sql
WHERE sql_id = '5t34tq47tz3p5'
AND hash_value = 87389835;
```

SQL_ID	*HASH_VALUE*	*ELIG_MB*	*RETURN_MB*	*SKIP_MB*	*RATIO*
------	------	------	------	------	------
5t34tq47tz3p5	*87389835*	*1718.03*	*20*	*1716.03*	*98.86*

SmartScans and (Database) Indexes

Obviously, **some indexes still have their place on Exadata** - a single-block lookup from memory is (as of 2014) faster than from disk - but for a lot of intensive queries, such as analytical processing, SmartScans bring significant performance improvements over indexes.

Though this might be counter-intuitive, using full-table scans on larger tables on Exadata is usually beneficial to performance.

To test whether a SmartScan IS better than using an index, use the FULL and PARALLEL hints to tell the optimizer to use a full-table scan and parallelism to query a very large table. Compare this performance and resource consumption to using a (database) index.

SmartScans are faster than a lot of the larger, non-unique indexes and **index usage is the most common reason why SmartScans are not considered by the optimizer**: remember, if a full-object scan is NOT being used, the object will not be SmartScan-eligible.

Therefore, it is important to review index usage in an Exadata database, especially on large tables.

Index Monitoring, Review and Deletion

First, **enable index monitoring** on the NONUNIQUE application indexes and, after a full batch cycle, review all indexes which have not been used and consider them for deletion.

N.B. as of 11.2.0.3, the **V$OBJECT_USAGE** view only displays LOCAL indexes instead of ALL indexes, similar to the **USER_INDEXES** view. To save on unnecessary wear-and-tear on keyboards, consider creating a customized view such as:

```
/*      -- The DSI001.INDEX_MONITOR view
        -- Use this view instead of the V$OBJECT_USAGE view because of a bug in the view
        -- For some reason, V$OBJECT_USAGE only shows local indexes, not all indexes
        -- Excludes any indexes which are NONUNIQUE.
        -- In this example, all application indexes were owned by DW* or DS* users.
        -- The query checks for all indexes in D* schemas, excluding DIP, DSI001 and DBSNMP.
*/

CREATE OR REPLACE FORCE VIEW DSI001.INDEX_MONITOR ("Owner", "Table Name", "Index Name",
"Monitored?", "Used?", "Monitoring Started", "Monitoring Ended", "Unique?", "Index Size (Mb)") AS
SELECT "Owner",
"Table Name",
"Index Name",
"Monitored?",
"Used?",
"Monitoring Started",
"Monitoring Ended",
"Unique?",
"Index Size (Mb)"
FROM
(
```

```
SELECT /*+ PARALLEL(dba_indexes 4) */ u.name AS "Owner",
t.name AS "Table Name",
io.name AS "Index Name",
DECODE(BITAND(i.flags, 65536), 0, 'NO', 'YES') AS "Monitored?",
DECODE(BITAND(ou.flags, 1), 0, 'NO', 'YES') AS "Used?",
ou.start_monitoring AS "Monitoring Started",
ou.end_monitoring AS "Monitoring Ended",
dbi.uniqueness AS "Unique?",
dbi.visibility AS "Visible?"
FROM sys.obj$ io,
sys.obj$ t,
sys.ind$ i,
sys.object_usage ou,
sys.user$ u,
sys.dba_indexes dbi
WHERE i.obj# = ou.obj#
AND io.obj# = ou.obj#
AND t.obj# = i.bo#
AND u.user# = io.owner#
AND u.name = dbi.owner
AND io.name = dbi.index_name
AND dbi.uniqueness = 'NONUNIQUE' x,
(
SELECT /*+ PARALLEL(dba_segments 4) */ owner,
segment_name,
SUM(bytes)/1024/1024 as "Index Size (Mb)"
FROM dba_segments
WHERE owner LIKE 'D%'
AND owner NOT IN ('DBSNMP', 'DSI001', 'DIP')
AND segment_type = 'INDEX'
GROUP BY owner, segment_name
) s
WHERE x."Owner" = s.owner(+)
AND x."Index Name" = s.segment_name(+)
ORDER BY 1,2,3;
```

Indexes considered for deletion should first be marked invisible to the optimizer so that they can only be used if an explicit hint is requested in the query itself and to ensure that the indexes will continue to be updated by DML operations on their tables.

Run an entire batch cycle with the indexes marked as invisible to confirm whether they are required or not. If they ARE needed, simply mark them as visible. **If they are not, export the index DDL and drop the index** – saving on storage, I/O during DML operations AND improving performance via the SmartScans.

SmartScans, Direct Reads and Parallelism

Parallelism will usually force direct path reads, which are required for SmartScans.

Parallelism itself is not strictly required for SmartScans, but **direct path reads** are.

Not all direct path reads will result in SmartScans but no SmartScans are possible without direct path reads.

The Database Buffer Cache and the _small_table_threshold Parameter

The **_small_table_threshold** is a parameter whose value is calculated based on the size of the database buffer cache. If the buffer cache is "too large" and the optimizer considers your "table" segment to be "small" (relative to the cache), a SmartScan will not be used.

By default, the **_small_table_threshold** is 2% of the size of the buffer cache.

Because of this, in EDW workloads, it can be detrimental to use a very large SGA (30-40Gb) when it is not necessary to do so. It is better to oversize the PGA instead as SmartScans make use of PGA memory.

The **_small_table_threshold** parameter should be not be set without guidance from Oracle Support. To verify:

```
SELECT name, value
FROM v$parameter
WHERE name LIKE '\_%' escape '\'
AND ISDEFAULT = FALSE;
```

Assuming the **_small_table_threshold** parameter value is NOT returned by the previous query, determine its size based on 2% of the database buffer cache:

```
SELECT component,
current_size/1024/1024 AS current_mb,
max_size/1024/1024 AS max_mb,
current_size/1024/1024/50 AS current_small,
max_size/1024/1024/50 AS max_small
FROM v$sga_dynamic_components
WHERE component = 'DEFAULT buffer cache';
```

COMPONENT	*CURRENT_MB*	*MAX_MB*	*CURRENT_SMALL*	*MAX_SMALL*
DEFAULT buffer cache	*10368*	*11520*	*207.36*	*230.4*

The _serial_direct_read Parameter

To run a query without parallelism but with direct path reads, the "hidden" (but well-documented) parameter **_serial_direct_read** can be set to ALWAYS for individual sessions. **DO NOT SET THIS AT THE DATABASE LEVEL!**

For more information about this parameter, raise a Service Request with Oracle Support.

Running Queries in Parallel

The more typical way to make a query eligible for SmartScan is to **run it in parallel**. Parallel full segment scans will use direct path reads UNLESS you are using the "*automatic parallelism*" feature and your parameter **parallel_degree_policy** is set to **auto** (the default setting is **manual**).

When this parameter is set, buffered reads **MAY** result because of in-memory dynamic execution. This feature improves with each release, but in early 11.2 releases, this had the potential to avoid using SmartScans as it did not bypass the buffer cache and, therefore, could result in inconsistent execution plans.

If you are using this feature, ensure that you have run **DBMS_RESOURCE_MANAGER.CALIBRATE_IO** on the database.

Object Level Parallelism

Do not over-allocate too many parallel slaves for processing - not only does a SmartScan parallelize the I/O across the storage cells, but a typical Exadata machine has a relatively low number of CPUs (when compared to a non-Exadata data warehouse, at least) and **excessive parallelism risks system instability.**

Ensure that tables or indexes **do not use DEFAULT parallelism.** This value is derived from the **cpu_count** parameter value and often causes queries to request excessive degrees of parallelism unnecessarily, usually without the user's knowledge. Theoretically, one query joining 3-4 tables with **DEFAULT** parallelism could bring down an entire instance if there are no measures in place to prevent it.

At the object-level, ensure that the DEGREE of parallelism is set as low as possible. As mentioned above, you almost always need some parallelism to use SmartScans, but you should limit the object-level DEGREE to the smallest possible value – 2 – and review if required.

It's far better for additional parallelism to be requested at the query-level via a PARALLEL hint than at the object-level, where it's invisible to the end user, who may end up using significant parallelism unwittingly.

To check parallelism set at the table level (don't forget to check table partitions in **DBA_TAB_PARTITIONS**):

```
SELECT owner, table_name, degree
WHERE owner IN ('APP_OWNER','MYSCHEMA')
ORDER BY degree, owner, table_name;
```

OWNER	*TABLE_NAME*	*DEGREE*
APP_OWNER	*TAB1*	*DEFAULT*
APP_OWNER	*TAB2*	*20*
MYSCHEMA	*TAB3*	*1*

To check parallelism set at the index level (don't forget to check index partitions in **DBA_IND_PARTITIONS**):

```
SELECT owner, index_name, index_type, uniqueness, degree
WHERE owner IN ('APP_OWNER','MYSCHEMA')
AND uniqueness = 'NONUNIQUE'
ORDER BY degree, owner, index_name;
```

OWNER	*INDEX_NAME*	*INDEX_TYPE*	*UNIQUENESS*	*DEGREE*
APP_OWNER	*IDX1*	*NORMAL*	*NONUNIQUE*	*DEFAULT*
APP_OWNER	*IDX2*	*BITMAP*	*NONUNIQUE*	*20*
MYSCHEMA	*IDX1*	*NORMAL*	*NONUNIQUE*	*1*

Inheriting Parallelism

DON'T FORGET that **an object will inherit an operation's degree of parallelism**. If you perform an *ALTER INDEX … REBUILD …. PARALLEL 20*; statement, **this will set the object-level parallelism of the index to 20.**

```
SELECT owner,  index_name, index_type, uniqueness, degree
WHERE index_name = 'MYVLTABLE_IDX_BIG'
AND owner = 'MYSCHEMA';
```

OWNER	*INDEX_NAME*	*INDEX_TYPE*	*UNIQUENESS*	*DEGREE*
MYSCHEMA	*MYVLTABLE_IDX_BIG*	*NORMAL*	*UNIQUE*	*2*

```
ALTER INDEX REBUILD PARALLEL 20;
```

Index altered.

```
SELECT owner,  index_name, index_type, uniqueness, degree
WHERE index_name = 'MYVLTABLE_IDX_BIG'
AND owner = 'MYSCHEMA';
```

OWNER	*INDEX_NAME*	*INDEX_TYPE*	*UNIQUENESS*	*DEGREE*
MYSCHEMA	*MYVLTABLE_IDX_BIG*	*NORMAL*	*UNIQUE*	*20*

Restricting Parallelism with the Database Resource Manager (DBRM)

Manage the consumption of parallel slaves by restricting a query's maximum degree of parallelism in the Database Resource Manager plan.

For example, if you do not want the ADHOC_GROUP consumer group users to consume excessive parallelism with their queries, **set their maximum degree of parallelism per query to 4 and limit the number of active sessions per instance to 4**. This will restrict these users to a maximum of 16 parallel slaves per user per instance.

Obviously, you should inform the users prior to this change, but expect to receive calls nonetheless after the change with users complaining that their connections are "*spinning*" or "*hourglassing*" – likely because they have already hit their limit of active sessions per instance and they are attempting to open a new connection.

If Resource Manager is limiting user connections, you will see a "*resmgr*" wait event on the session that is attempting to connect.

In the author's experience, no user has ever complained about their maximum degree of parallelism per query being limited, whether it by the Resource Manager or by changing the object level parallelism. Indeed, it is very likely that they will see a performance improvement if we are able to use SmartScans.

Implementing Resource Manager is a useful – if incidental - way **to identify the real power users of the system**, **to discover any examples of user accounts being shared amongst users and to identify any clients which are not closing connections properly** (i.e. older versions of the TOAD application)

References

My Oracle Support: Exadata Smart Scan FAQ (Doc ID 1927934.1)
My Oracle Support: Smart scan : Find the statistics related to cell offload (Doc ID 1438173.1)
My Oracle Support: Oracle Sun Database Machine Application Best Practices for Data Warehousing [Doc ID 1094934.1]
Uwe Hesse: http://uhesse.com/2011/07/06/important-statistics-wait-events-on-exadata/

About Database Specialists

Database Specialists, Inc. is an Oracle Certified Solution Partner headquartered in the San Francisco Bay Area with consultants based throughout the United States.

We provide expert Oracle consultancy, remote DBA and 24/7 pro-active monitoring via our DBRx platform to clients from a wide-range of industries such as ecommerce, retail, marketing, health care, financial, news media, telecoms, film and television, automotive and aviation.

We work exclusively on Oracle technology and each member of our team is an expert Oracle DBA with at least ten years of experience. We frequently contribute to industry publications and conferences such as Oracle OpenWorld and IOUG and enthusiastically support regional and national user groups.

For more details about how we can help, please visit our web site at www.dbspecialists.com where we regularly publish white papers, post to our blog and make some of our more useful scripts available for use.

If you have any questions, comments or suggestions about this white paper (or presentation), please email me at msmith@dbspecialists.com.

Expert Oracle Consulting and
Remote Database Administration

www.dbspecialists.com
1-888-648-0500