Drilling Deep Into Exadata Performance

With ASH, SQL Monitoring and Exadata Snapper

Tanel Põder
Enkitec

http://www.enkitec.com
http://blog.tanelpoder.com

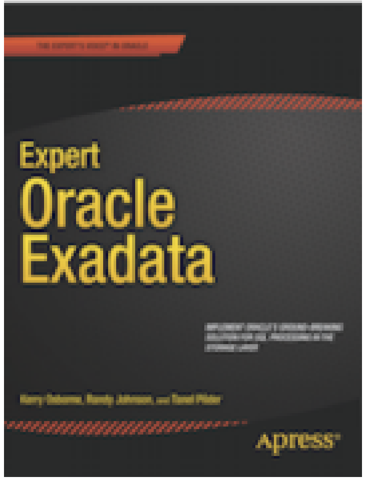
Intro: About me

OakTable.net

ORACLE ACE Director

ORACLE | CERTIFIED MASTER

- Tanel Põder
  - Former Oracle Database Performance geek
  - Present Exadata Performance geek ;-)
- My Exadata experience
  - I have had the luck to work with all possible Exadata configurations out there
  - Exadata V1 … X3
  - Multi-rack Exadatas …
  - Even a mixed rack Exadata (V2 <-> X2-2 :)
- Enkitec Exadata experience
  - Over 100 implementations!



Expert Oracle Exadata book
(with Kerry Osborne and Randy Johnson of Enkitec)

About Enkitec

- Enkitec LP
  - North America
- Enkitec Europe
  - EMEA
- 90+ staff
  - In US, Europe
  - Consultants with Oracle experience of 15+ years on average
- What makes us unique
  - **100+ Exadata implementations to date**
    - 100% customer satisfaction rate
  - Exadata-specific services
    - Exadata Quarterly Patching Service
  - Enkitec Exadata lab
    - We have 2 Exadatas for thorough testing and PoCs

Platinum Partner

PARTNER NETWORK
TITAN AWARD WINNER

Everything Exadata
Planning/PoC
Implementation
Consolidation
Migration
Backup/Recovery
Patching
Troubleshooting
Performance
Capacity
Training

Agenda

1. Finding non-Exadata friendly SQL
2. A little bit of Smart Scan internals
3. Reading a SQL Monitoring report on Exadata
   - … and where it falls short
4. Using advanced Exadata performance metrics
   - Exadata Snapper (ExaSnap)!!!

Exadata's "secret sauce" for different workloads

- **"DW / Reporting"**
  - Long running SQL statements
  - Executed less frequently
  - Secret Sauce: Smart Scans + Offloading + Storage Indexes
  - SQL statements should use full table scans + hash joins

- **"OLTP"**
  - Short SQL statements
  - Executed very frequently
  - Secret Sauce: Flash Cache
  - SQL & performance tuning is the same as usual!

Finding non-Exadata-friendly SQL

(non-smart-scan-friendly SQL...)

1) Finding top non-Exadata-friendly SQLs

- Option 1:
  - Find SQLs which **wait** for non-smart scan IO operations the most
  - **ASH!**

- Option 2:
  - Find SQLs doing **the most disk reads** without Smart Scans
  - Highest MB read or highest IOPS

    a) ASH!
      - SUM(DELTA_READ_IO_REQUESTS)
      - SUM(DELTA_READ_IO_BYTES)

    b) ..or join to V$SQLSTAT (or DBA_HIST_SQLSTAT)
      - SUM(PHYSICAL_READ_REQUESTS_DELTA)
      - SUM(PHYSICAL_READ_BYTES_DELTA)

2) Are these long-running or frequently executed short queries?

- Exadata Smart scans are not optimized for *ultra*-frequent execution
  - Smart Scanning 1000s of times per second isn't good
  - That's why Smart Scans shouldn't be used for your OLTP-queries

- Here's an analogy:
  1. Want to deliver a single parcel to a destination
     - Use a Ferrari
  2. Want to deliver 10 000 parcels to a destination
     - Use a truck
  3. Want to deliver 10 000 000 parcels to a destination
     - Use a freight train

  - **Smart Scan is the Exadata's freight train**
    - Brute force scanning with relatively high inertia to get started, not a few quick (buffered) I/O operations done with surgical precision

Demo – exafriendly.sql

- Drill down into ASH wait data:

```
SQL> @exadata/exafriendly.sql gv$active_session_history

SESSION WAIT_CLASS          EVENT                                       SECONDS    PCT
------- ------------------- ---------------------------------------- ---------- ------
ON CPU                                                                   192356   34.6
WAITING User I/O            cell single block physical read              191838   34.5
WAITING User I/O            db file parallel read                         40577    7.3
WAITING User I/O            cell smart table scan                         28593    5.1
WAITING User I/O            cell multiblock physical read                 19424    3.5
WAITING User I/O            direct path read temp                         18398    3.3
WAITING Application         enq: RO - fast object reuse                    8690    1.6
WAITING User I/O            direct path read                               5812    1.0
...

PLAN_LINE                            USERNAME     EVENT                               SECONDS    PCT
------------------------------------ ------------ ---------------------------------- --------- ------
TABLE ACCESS BY LOCAL INDEX ROWID    USER_104     cell single block physical read        40954   21.3
TABLE ACCESS BY INDEX ROWID          USER_779     cell single block physical read        32129   16.7
INDEX RANGE SCAN                     USER_104     cell single block physical read        25272   13.2
TABLE ACCESS STORAGE FULL            USER_49      cell single block physical read         9258    4.8
INDEX RANGE SCAN                     USER_779     cell single block physical read         4118    2.1
TABLE ACCESS STORAGE FULL            USER_783     cell single block physical read         3641    1.9
UPDATE                               USER_104     cell single block physical read         3509    1.8
TABLE ACCESS BY INDEX ROWID          USER_420     cell single block physical read         3341    1.7
MERGE                                USER_962     cell single block physical read         2910    1.5
```

Demo – mon_topsql.sql

- TOP Time-consuming SQLs with IO & execution counts
  - That way we'll separate the Ferraris from Freight Trains!
  - The "Ferraris" aren't Exadata *smart scan* friendly

DAY	PCT	OWNER	OBJECT_NAME	PROCEDURE_NAME	SQL_ID	TOTAL_HOURS	TOTAL_SECONDS	EXECUTIONS	SECONDS_PER_EXEC	IO_PCT	CPU_PCT
9/4/11	4.10%				32vkfmvcdgfkp	40.3	145200	0		65.8	19.1
	3.20%				5qjq8ckgsu054	31.7	114050	0		98.7	1.3
	3.20%				8b1bta0wk04s9	31.7	114010	1159739	0.1	0	98
	3.20%					31.7	113980			2.5	34.1
	3.10%				cwyy1g22pyf60	30.3	109180	0		0	99.5
	2.80%				4suwa3nur4d2q	27.6	99440	0		90.4	6.2
	2.30%				1bnn9jw75t95c	23.1	83130	0		99.3	0.7
	2.10%				gsv18p3ykvv34	20.3	73230	0		0.1	99.9
	1.80%	APPX_MOD_RMS	MAP_PACKAGE	STATE_MAPPINGS	gbmmdsp0ra23n	17.8	63990	7	9141.43	93.3	6
	1.60%				5vgtm5hfhw1wa	15.5	55890	0		98.8	0.8
	1.50%				5xdxcr7pk0cn5	14.7	52810	37985	1.39	98.5	1.5
	1.40%				5fpy9xtn5tv2g	13.6	49130	0		58.8	41.2
	1.20%				3zzcx99ufryt1	12.1	43420	52004462	0	86.9	13.1
9/5/11	2.40%					58.6	211110			7	35.6
	2.40%				fw0tar4rfa39x	58.6	210880	28666	7.36	92.8	5.9
	2.30%				1xxqkv6n9bgu7	58.5	210660	19295	10.92	92.6	6.5
	2.30%				a615cdv2xn65v	58.3	210010	24302	8.64	94	5
	2.30%				fqh4bksfg44bw	58.1	209000	17863	11.7	93.7	5.2
	2.30%				10jbqndnjwuvk	57.5	206980	30286	6.83	86	12.5
	2.00%				cwyy1g22pyf60	49.6	178530	0		0	99.7
	1.90%				4suwa3nur4d2q	47.9	172480	0		94.9	2.4
	1.60%				3zzcx99ufryt1	40	143960	61655359	0	95.6	4.4

Other sources for overview information

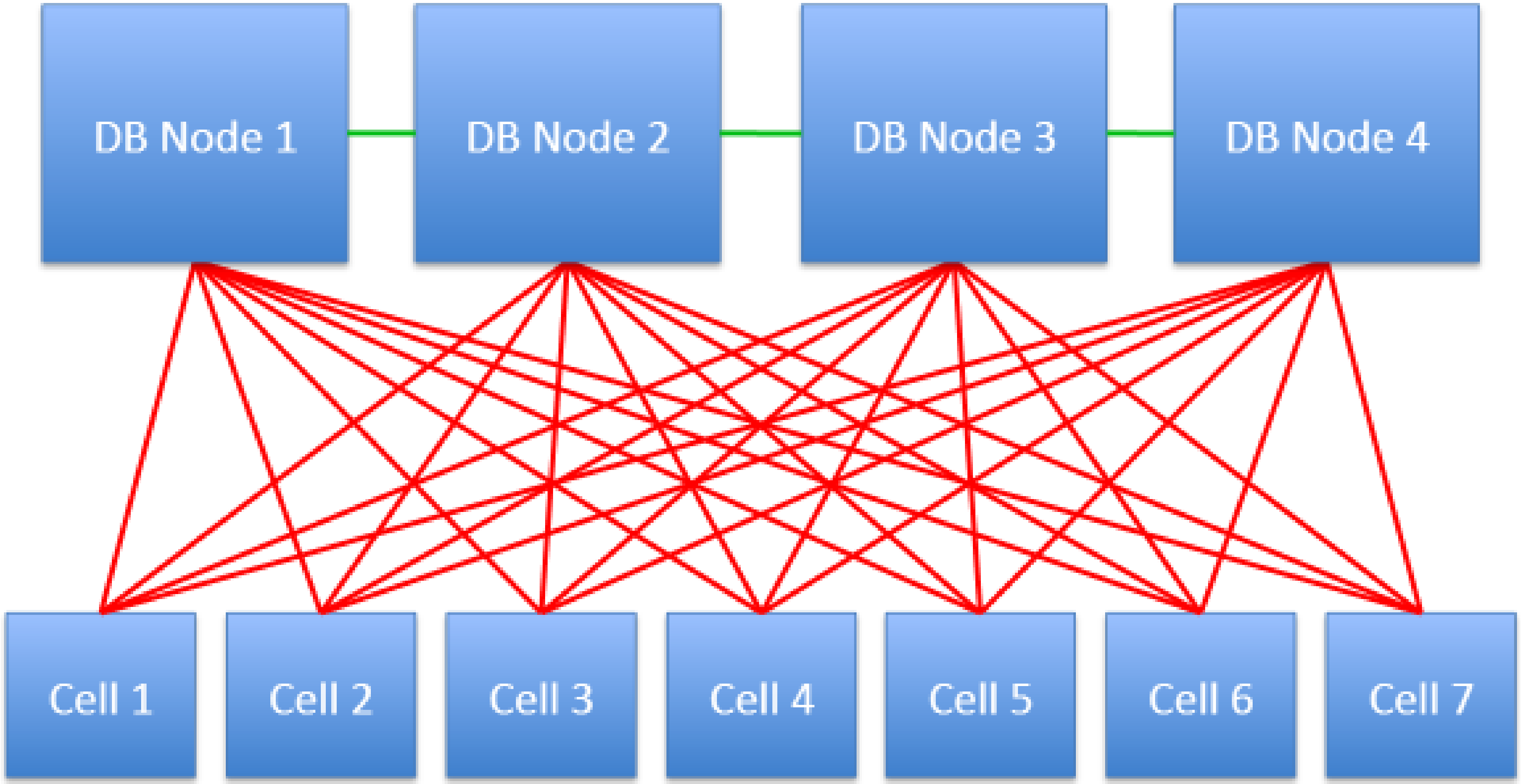
- ASH reports (based on ASH data which I've used)
- EM 12c ASH analytics

Top SQL with Top Row Sources

SQL ID	PlanHash	Sampled # of Executions	% Activity	Row Source	% RwSrc	Top Event	% Event
91atnkya3uq3u	1846793290	1	6.99	TABLE ACCESS - STORAGE FULL	6.99	cell smart table scan	6.91
a7p9s9nark2aj	256013542	3171	6.89	TABLE ACCESS - STORAGE FULL	6.79	CPU + Wait for CPU	6.79
gctaxcyk0dt67	2150706944	1	2.33	SELECT STATEMENT	2.33	SQL*Net break/reset to client	2.33
7mh3k1p8ht1sy	720331710	1	2.15	TABLE ACCESS - STORAGE FULL FIRST ROWS	2.01	cell multiblock physical read	1.68
1u0xfr31yrh2u	785828209	1	1.76	LOAD AS SELECT	0.60	CPU + Wait for CPU	0.47

Exadata Architecture

- All DB nodes talk to all (configured) cells - ASM striped data
- **A cell never talks to another cell !!!**



Smart Scans: Asynchronous, independent prefetching

```
PARSING IN CURSOR #47233445473208 len=38 dep=0 uid=93 oct=3 lid=93 tim=1303079537221426
select * from t3 where owner like 'S%'
END OF STMT
PARSE #47233445473208:c=1000,e=8964,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,plh=4161002650,ti
EXEC #47233445473208:c=0,e=21,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=4161002650,tim=1303
WAIT #47233445473208: nam='SQL*Net message to client' ela= 4 driver id=1413697536 #bytes
WAIT #47233445473208: nam='SQL*Net more data to client' ela= 16 driver id=1413697536 #by
WAIT #47233445473208: nam='reliable message' ela= 1337 channel context=11888341784 chann
WAIT #47233445473208: nam='enq: KO - fast object checkpoint' ela= 143 name|mode=12634685
WAIT #47233445473208: nam='enq: KO - fast object checkpoint' ela= 130 name|mode=12634685
WAIT #47176529789912: nam='cell smart table scan' ela= 25 cellhash#=3176594409 p2=0 p3=0
WAIT #47176529789912: nam='cell smart table scan' ela= 882 cellhash#=2133459483 p2=0 p3=
WAIT #47176529789912: nam='cell smart table scan' ela= 34 cellhash#=3176594409 p2=0 p3=0
WAIT #47176529789912: nam='cell smart table scan' ela= 837 cellhash#=2133459483 p2=0 p3=
WAIT #47176529789912: nam='cell smart table scan' ela= 26 cellhash#=2133459483 p2=0 p3=0
WAIT #47176529789912: nam='cell smart table scan' ela= 22 cellhash#=379339958 p2=0 p3=0
```

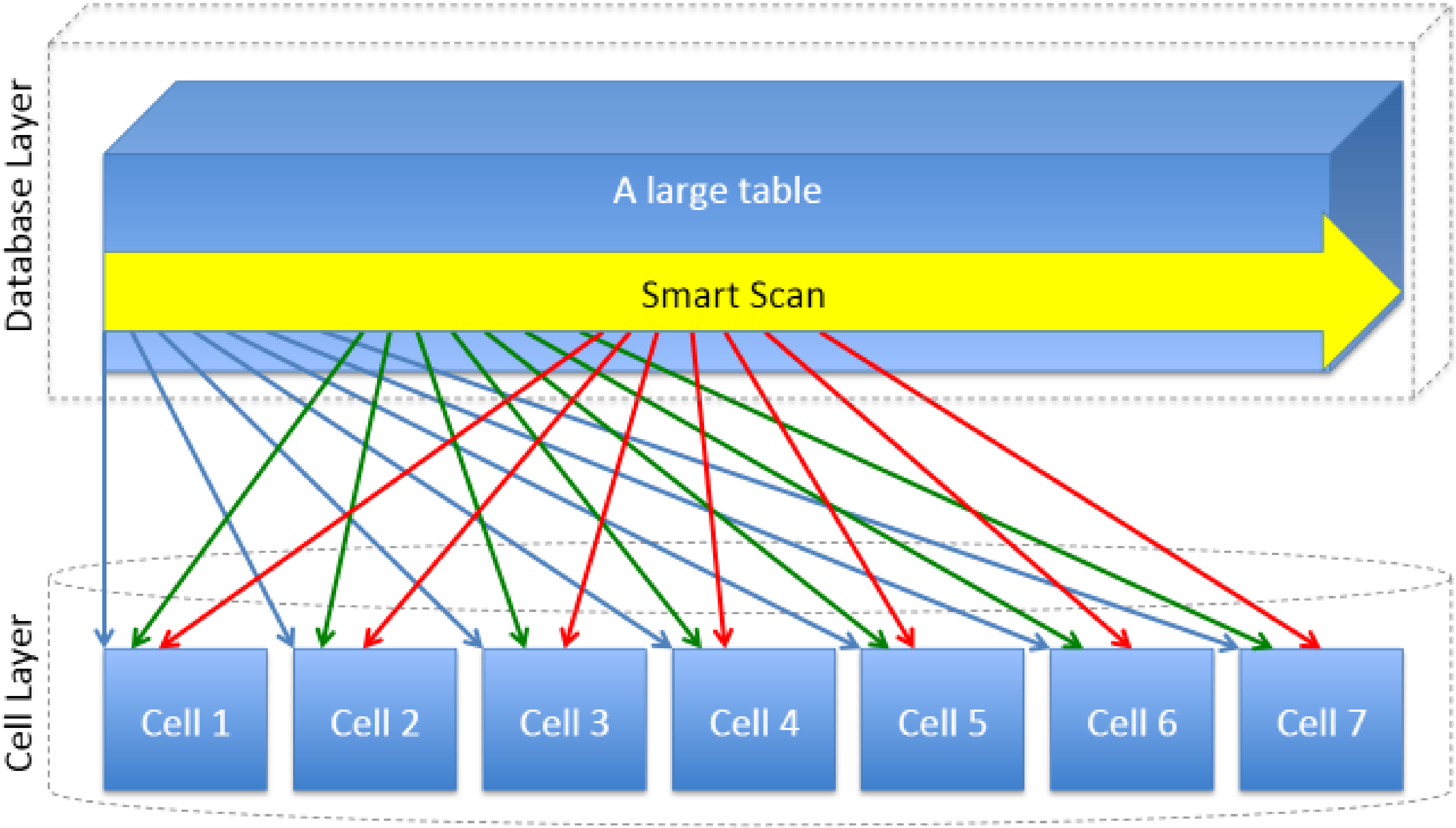
If cell ***smart table/ index scan*** waits show up, then a smart scan is attempted

The waits are so short due to the asynchronous nature of smart scans. Cellsrvs in each cell process the blocks independently to fill their send buffers and the DB just pulls the results from there

The object checkpoint-related wait events ***reliable message*** and ***enq: KO – fast object checkpoint*** always precede every direct path scan (thus smart scan too)

Storage cells are “shared nothing”

- And they don’t see what’s happening in the database layer…



Physical disks - Simple math

- Sequential "brute force" scan rate **150 MB/sec** per disk

or

- **200 random IOPS** per disk

- **12 disks** in a storage cell
- **14 storage cells** in a full rack

I'm leaving flash cache out for simplicity for now

- 150 * 12 * 14 = **25200 MB/sec** physical disk scanning rate
  - If doing only sequential brute force scanning

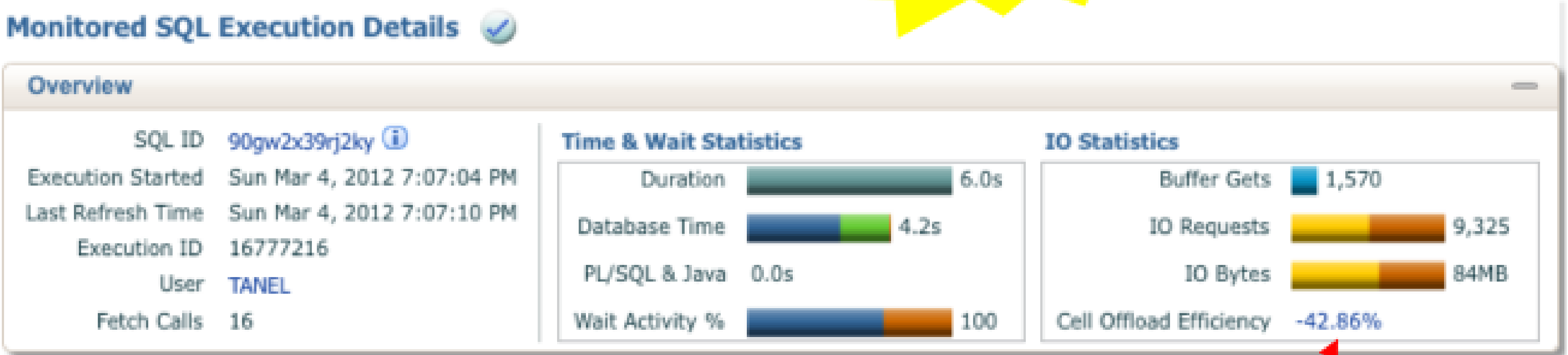
- 200 * 12 * 14 * 8kB = **262.5 MB/sec**
  - Random physical read rate (index scan!) around 100x slower!

However, Index scans can read only a subset of data

The Motivation for Writing Exadata Snapper

```
[celladmin@enkcel01 ~]$ cellcli

CellCLI: Release 11.2.2.4.0 - Production on Mon Mar 05 08:19:41 CST 2012

Copyright (c) 2007, 2011, Oracle.  All rights reserved.
Cell Efficiency Ratio: 699

CellCLI>
```

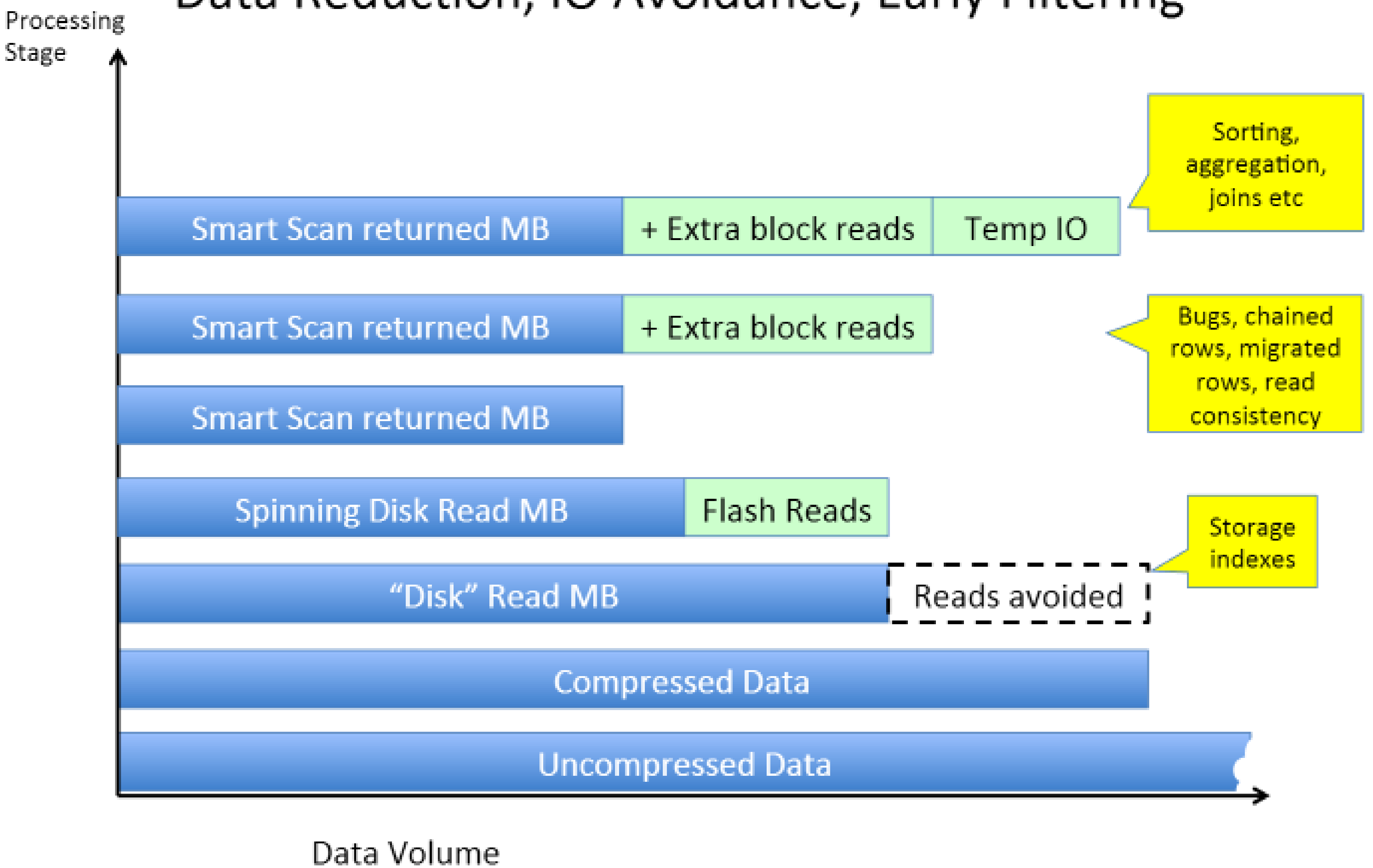
????



?

Following just the **Cell Efficiency Ratio** can be as misleading as tuning by **Buffer Cache Hit Ratio**!

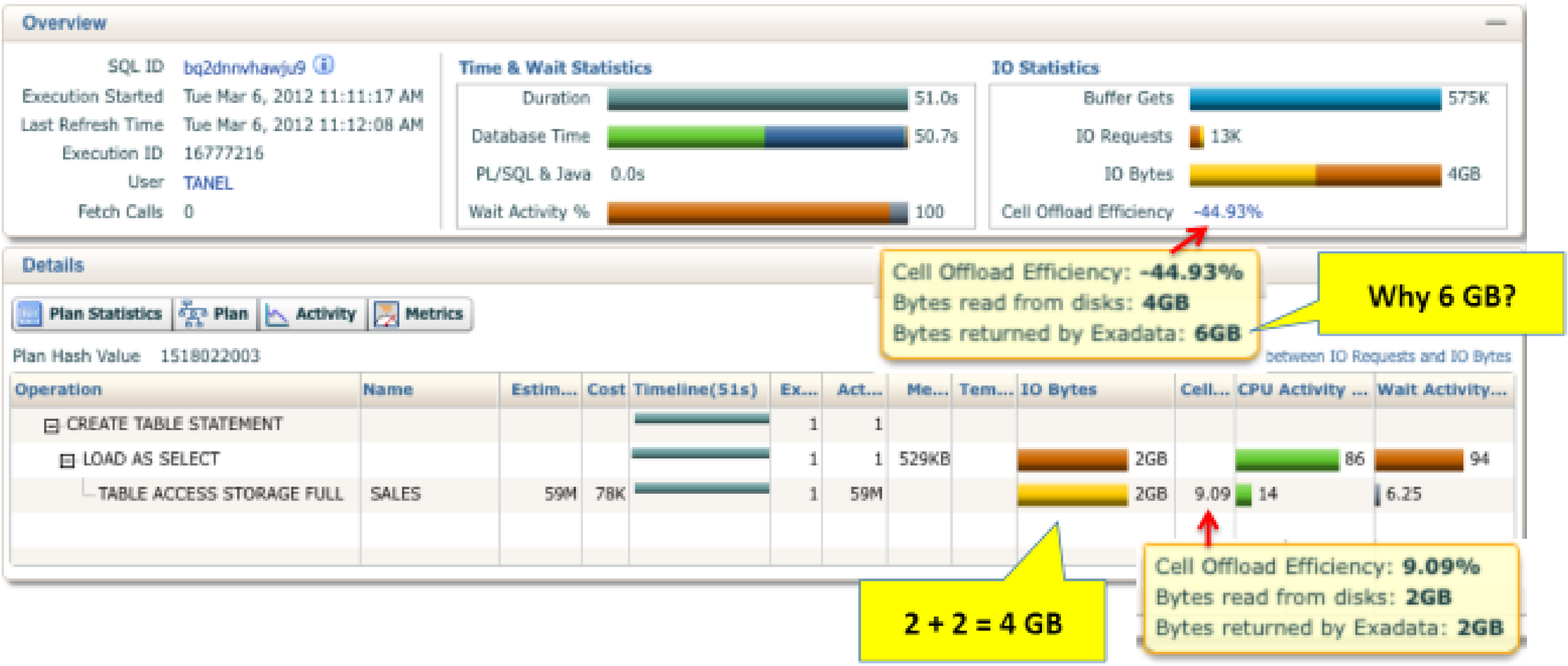
Data Reduction, IO Avoidance, Early Filtering



Negative Cell Offload Efficiency ratio?

- Must understand where does this ratio come from
  - Ratio of which Exact metrics?
  - And use those metrics in the future

Negative Cell Offload Efficiency: Data Load Example



The "Bytes returned by Exadata" metric actually uses the "**cell physical IO interconnect bytes**" metric internally, which includes all traffic, not just rows returned from smart scan.

So, write IOs also influence cell offload efficiency calculation (data loads, sort/join/aggregate operation TEMP usage).
Write IOs are double/triple mirrored by ASM!

Interpreting the raw metrics with Exadata Snapper

- IO Efficiency breakdown
  - How much physical IO did we really do?
  - How much reads / how much writes?
  - How much disk IO did we avoid doing thanks to Flash Cache?
  - How much disk IO did we avoid doing thanks to Storage Indexes?
  - What was the total interconnect traffic?
  - How much data was fully processed in the cell (and not shipped back in blocks due to fallback or problems in the cell?)
  - How much data was returned by a Smart Scan (as rows)?

- @exadata/exasnap.sql
  - Beta quality
  - Meant to *complement* ASH and SQL Monitoring reports, not replace them

ExaSnap example: A CTAS statement

Operation	Name	Estim...	Cost	Timeline(51s)	Ex...	Act...	Me...	Tem...	IO Bytes	Cell...	CPU Activity ...	Wait Activity...
CREATE TABLE STATEMENT					1	1						
LOAD AS SELECT					1	1	529KB		2GB		86	94
TABLE ACCESS STORAGE FULL	SALES	59M	78K		1	59M			2GB	9.09	14	6.25

Cell Offload Efficiency: **-44.93%**
Bytes read from disks: **4GB**
Bytes returned by Exadata: **6GB**

```
SQL> @exadata/exasnap % 123 124
--------------------------------------------------------------------------------------------------------------
-- Exadata Snapper v0.5 BETA by Tanel Poder @ Enkitec - The Exadata Experts (
--------------------------------------------------------------------------------------------------------------

   SID CATEGORY             METRIC                  IOEFF_PERCENTAGE                                               MB
------ -------------------- ----------------------- -------------------------------------------------------- ----------
  1280 DB_IO                DB_PHYSIO_MB            |##################################                   |       4435
       DB_IO                DB_PHYSRD_MB            |#################                                    |       2219
       DB_IO                DB_PHYSWR_MB            |#################                                    |       2216

       AVOID_DISK_IO        PHYRD_FLASH_RD_MB       |                                                     |          2
       AVOID_DISK_IO        PHYRD_STORIDX_SAVED_MB  |                                                     |          0

       DISK_IO              SPIN_DISK_IO_MB         |#####################################################|       6649
       DISK_IO              SPIN_DISK_RD_MB         |#################                                    |       2217
       DISK_IO              SPIN_DISK_WR_MB         |##################################                   |       4432

       REDUCE_INTERCONNECT  PRED_OFFLOAD_MB         |#################                                    |       2216
       REDUCE_INTERCONNECT  TOTAL_IC_MB             |#####################################################|       6444
       REDUCE_INTERCONNECT  SMART_SCAN_RET_MB       |################                                     |       2009
       REDUCE_INTERCONNECT  NON_SMART_SCAN_MB       |##################################                   |       4435

       CELL_PROC_DEPTH      CELL_PROC_DATA_MB       |#################                                    |       2232
       CELL_PROC_DEPTH      CELL_PROC_INDEX_MB      |                                                     |          0
```

The *real* disk writes are doubled due to ASM double-mirroring

ExaSnap example 2: Storage Index Savings

The real (spinning) disk reads IO was only **1078 MB** thanks to **1138 MB** of disk IO avoided due to storage indexes:
(2216 – 1138 = 1078)

```
SQL> @exadata/exasnap basic 90 91
-------------------------------------------------------------------------------
-- Exadata Snapper v0.5 BETA by Tanel Poder @ Enkitec - The Exada
-------------------------------------------------------------------------------

CATEGORY             METRIC                   IOEFF_PERCENTAGE                                          MB
-------------------- ------------------------ ------------------------------------------------------ ---------- ----
DB_LAYER_IO          DB_PHYSIO_MB             |####################################################|       2216
DB_LAYER_IO          DB_PHYSRD_MB             |####################################################|       2216
DB_LAYER_IO          DB_PHYSWR_MB             |                                                    |          0

AVOID_DISK_IO        PHYRD_FLASH_RD_MB        |                                                    |          0
AVOID_DISK_IO        PHYRD_STORIDX_SAVED_MB   |###########################                         |       1138

REAL_DISK_IO         SPIN_DISK_IO_MB          |#########################                           |       1079
REAL_DISK_IO         SPIN_DISK_RD_MB          |#########################                           |       1078
REAL_DISK_IO         SPIN_DISK_WR_MB          |                                                    |          1

REDUCE_INTERCONNECT  PRED_OFFLOADABLE_MB      |####################################################|       2216
REDUCE_INTERCONNECT  TOTAL_IC_MB              |                                                    |          2
REDUCE_INTERCONNECT  SMART_SCAN_RET_MB        |                                                    |          2
REDUCE_INTERCONNECT  NON_SMART_SCAN_MB        |                                                    |          0

CELL_PROC_DEPTH      CELL_PROC_DATA_MB        |#########################                           |       1078
CELL_PROC_DEPTH      CELL_PROC_INDEX_MB       |                                                    |          0
```

All 1078 MB worth of blocks got offloaded: they were opened and processed inside the cells (data layer)

So, why isn't my query Exadata-friendly?

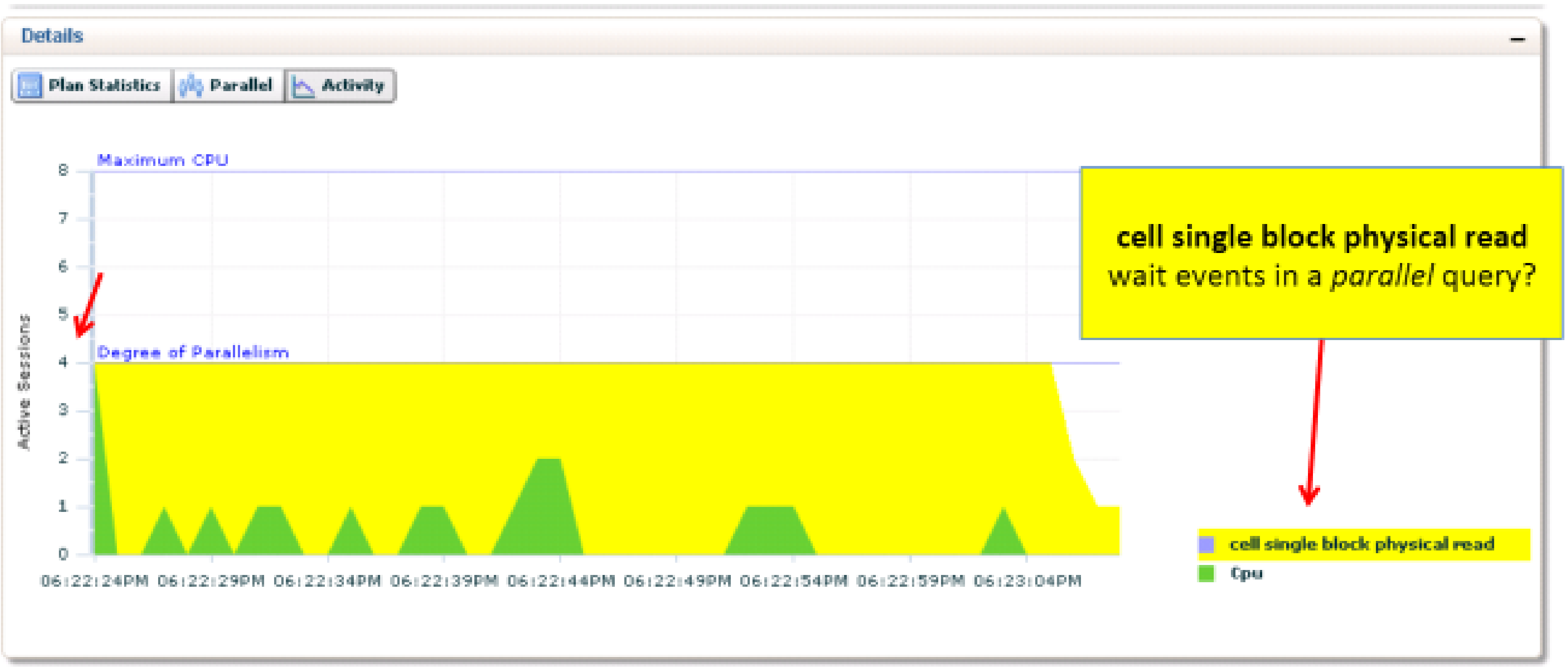
Drilling down into a SQL execution

1. SQL Monitoring report
   - Execution plan!
   - Where is most of the response time spent (retrieval vs. subsequent processing)
   - Are smart scans used for data retrieval?
   - IO MB read from disks vs data returned from the cells
     - (also called the offload efficiency ratio but knowing the underlying numbers is way better)

2. ExaSnapper
   - Or read the raw v$sesstat metrics if you dare ;-)

Warm-up case study

- Testing after migration, a query is slow:



Warm-up case study: check the execution plan

- Parallel execution plan does *not* force full table scans...

```
---------------------------------------------------------------------------------------------------
| Id  | Operation                                 | Name            |    TQ  |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                          |                 |        |      |            |
|   1 |  PX COORDINATOR                           |                 |        |      |            |
|   2 |   PX SEND QC (RANDOM)                     | :TQ10002        |  Q1,02 | P->S | QC (RAND)  |
|*  3 |    FILTER                                 |                 |  Q1,02 | PCWC |            |
|   4 |     HASH GROUP BY                         |                 |  Q1,02 | PCWP |            |
|   5 |      PX RECEIVE                           |                 |  Q1,02 | PCWP |            |
|   6 |       PX SEND HASH                        | :TQ10001        |  Q1,01 | P->P | HASH       |
|   7 |        NESTED LOOPS                       |                 |  Q1,01 | PCWP |            |
|   8 |         NESTED LOOPS                      |                 |  Q1,01 | PCWP |            |
|   9 |          NESTED LOOPS                     |                 |  Q1,01 | PCWP |            |
|  10 |           BUFFER SORT                     |                 |  Q1,01 | PCWC |            |
|  11 |            PX RECEIVE                     |                 |  Q1,01 | PCWP |            |
|  12 |             PX SEND ROUND-ROBIN           | :TQ10000        |        | S->P | RND-ROBIN  |
|* 13 |              TABLE ACCESS BY GLOBAL INDEX ROWID| ORDERS     |        |      |            |
|* 14 |               INDEX RANGE SCAN            | ORD_STATUS_IX   |        |      |            |
|  15 |           PARTITION HASH ITERATOR         |                 |  Q1,01 | PCWP |            |
|* 16 |            TABLE ACCESS BY LOCAL INDEX ROWID | CUSTOMERS    |  Q1,01 | PCWP |            |
|* 17 |             INDEX UNIQUE SCAN             | CUSTOMERS_PK    |  Q1,01 | PCWP |            |
|  18 |          PARTITION HASH ITERATOR          |                 |  Q1,01 | PCWP |            |
|* 19 |           INDEX RANGE SCAN                | ORDER_ITEMS_PK  |  Q1,01 | PCWP |            |
|  20 |         TABLE ACCESS BY LOCAL INDEX ROWID | ORDER_ITEMS     |  Q1,01 | PCWP |            |
---------------------------------------------------------------------------------------------------
```

Serial stage in a parallel plan

Parallel index scans (on different partitions)

Warm-up case study – adjusted execution plan

- After forcing full table scans:

```
------------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name          | Pstart| Pstop |    TQ  |IN-OUT| PQ Di
------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |               |       |       |        |      |
|   1 |  PX COORDINATOR                    |               |       |       |        |      |
|   2 |   PX SEND QC (RANDOM)              | :TQ10003      |       |       |  Q1,03 | P->S | QC (RAND)  |
|*  3 |    FILTER                          |               |       |       |  Q1,03 | PCWC |            |
|   4 |     HASH GROUP BY                  |               |       |       |  Q1,03 | PCWP |            |
|   5 |      PX RECEIVE                    |               |       |       |  Q1,03 | PCWP |            |
|   6 |       PX SEND HASH                 | :TQ10002      |       |       |  Q1,02 | P->P | HASH       |
|*  7 |        HASH JOIN                   |               |       |       |  Q1,02 | PCWP |            |
|   8 |         PART JOIN FILTER CREATE    | :BF0000       |       |       |  Q1,02 | PCWP |            |
|   9 |          PX RECEIVE                |               |       |       |  Q1,02 | PCWP |            |
|  10 |           PX SEND BROADCAST        | :TQ10001      |       |       |  Q1,01 | P->P | BROADCAST  |
|* 11 |            HASH JOIN               |               |       |       |  Q1,01 | PCWP |            |
|  12 |             PART JOIN FILTER CREATE| :BF0001       |       |       |  Q1,01 | PCWP |            |
|  13 |              PX RECEIVE            |               |       |       |  Q1,01 | PCWP |            |
|  14 |               PX SEND BROADCAST    | :TQ10000      |       |       |  Q1,00 | P->P | BROADCAST  |
|  15 |                PX BLOCK ITERATOR   |               |     1 |    16 |  Q1,00 | PCWC |            |
|* 16 |                 TABLE ACCESS STORAGE FULL| ORDERS  |     1 |    16 |  Q1,00 | PCWP |            |
|  17 |             PX BLOCK ITERATOR      |               |:BF0001|:BF0001|  Q1,01 | PCWC |
|* 18 |              TABLE ACCESS STORAGE FULL | CUSTOMERS |:BF0001|:BF0001|  Q1,01 | PCWP |
|  19 |         PX BLOCK ITERATOR          |               |:BF0000|:BF0000|  Q1,02 | PCWC |
|* 20 |          TABLE ACCESS STORAGE FULL | ORDER_ITEMS   |:BF0000|:BF0000|  Q1,02 | PCWP |
------------------------------------------------------------------------------------------------------

  18 - storage(:Z>=:Z AND :Z<=:Z AND ("C"."NLS_TERRITORY"='AMERICA' AND
              SYS_OP_BLOOM_FILTER(:BF0000,"C"."CUSTOMER_ID")))
```

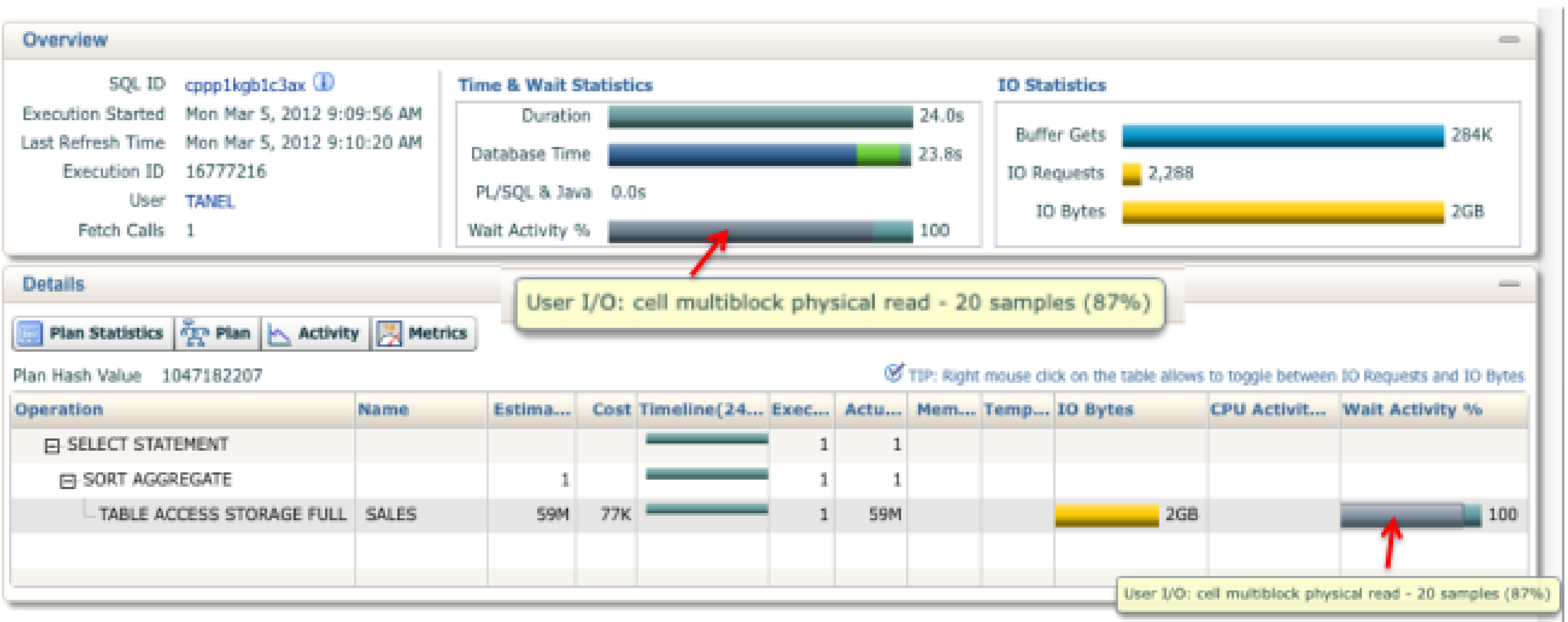
Fully parallel execution plan

Full table (partition) scans with bloom filter offloading

Case 2: Response time 24 seconds – where is it spent?

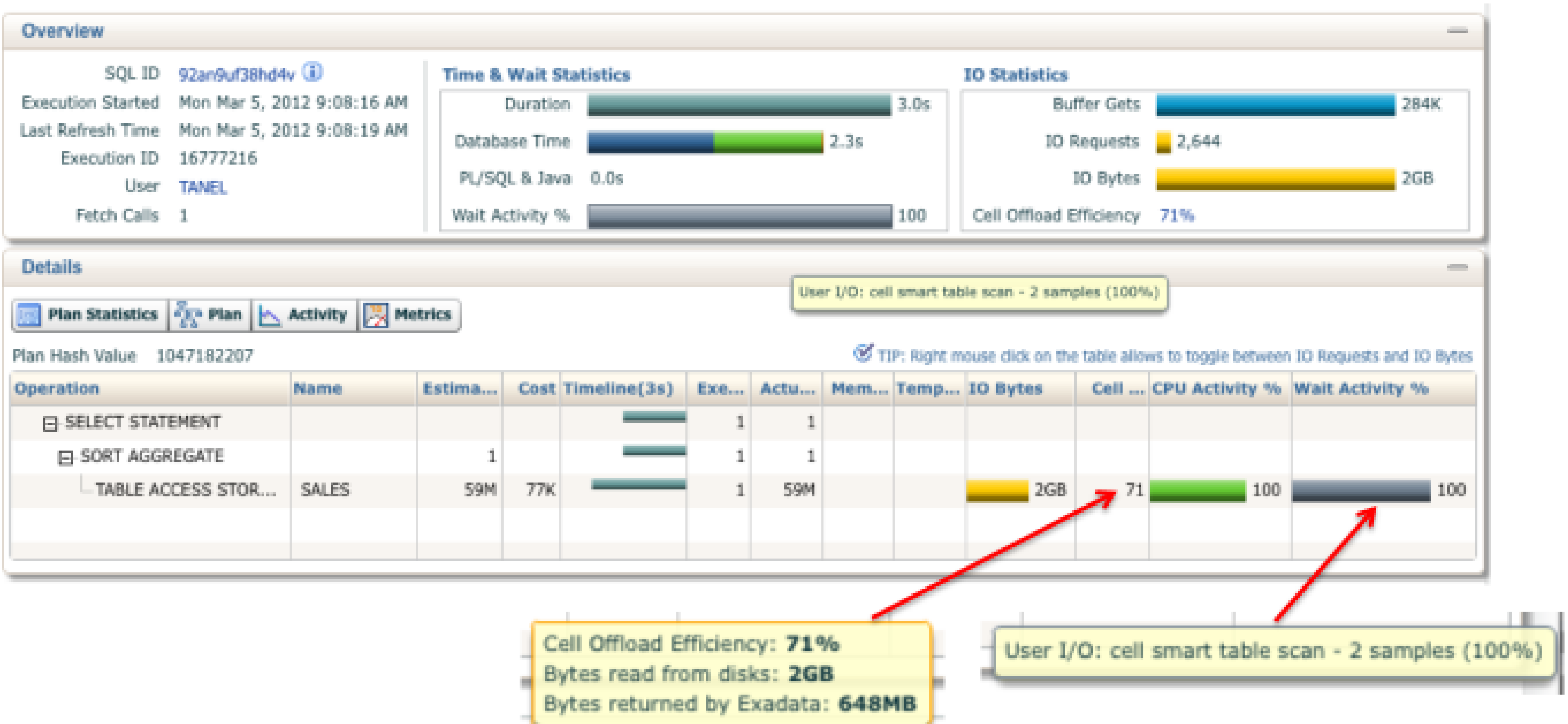
1) Full scan: TABLE ACCESS STORAGE FULL
2) But waiting for a buffered read wait event
   - *cell multiblock physical read* instead of *cell smart table/index scan*

Smart scan will help!



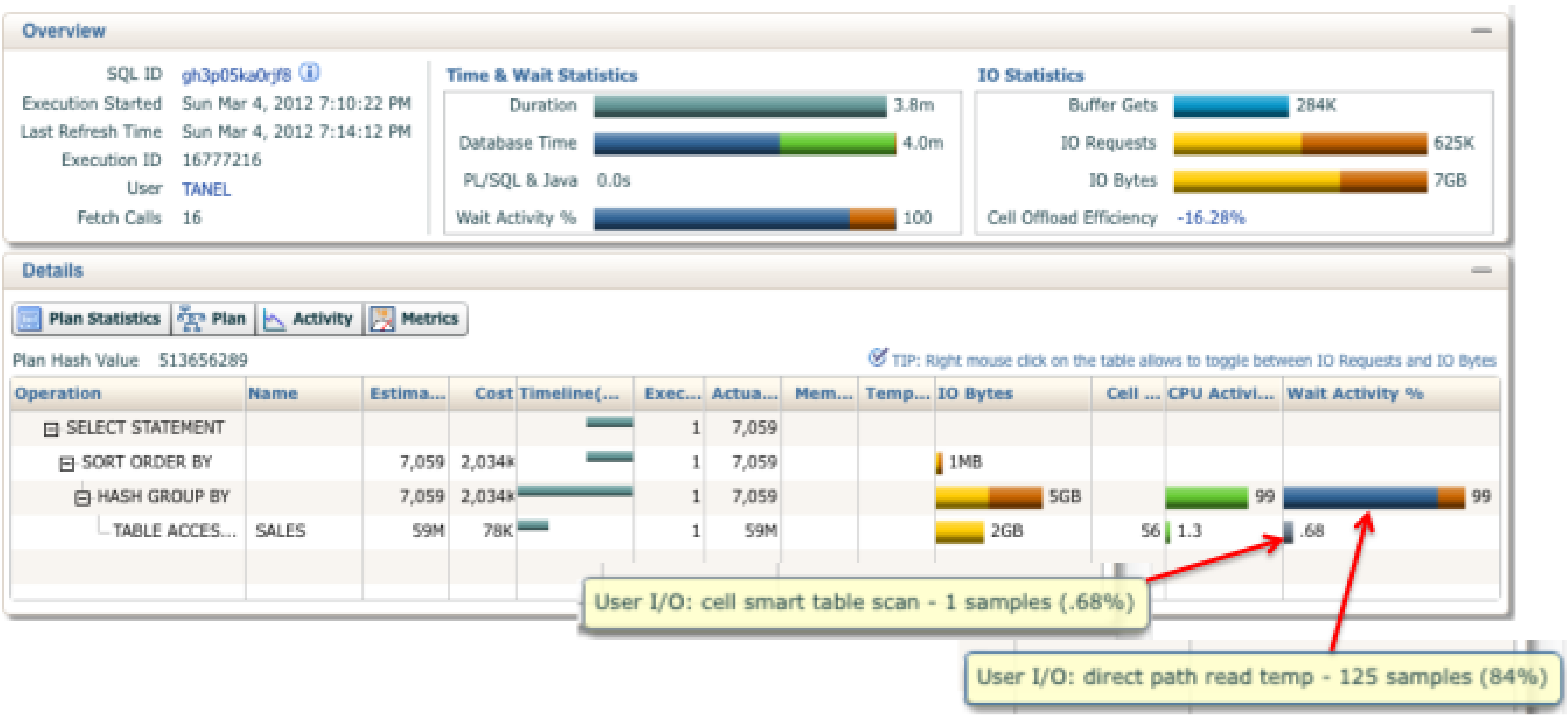
Case 2: The same query runs in 3 seconds with Smart Scan

- So, why is it faster?
  - Data retrieval (ACCESS) from storage is faster



Case 2: Now let's do something with the data…

- Same query, with an extra GROUP BY:
  - Takes 3.8 minutes! (228 seconds)
  - Why? Let's see what's taking the extra time:



Checkpoint

- Smart Scans make the **data retrieval from storage** faster
- Any other problems require the usual SQL & DB optimization

Smart Scans do ***not*** make any of these things faster (by design):

- Index unique/range/skip scans *
- Sorting
- Aggregation (GROUP BY)
- Analytic Functions
- Filters that can't be pushed down to table level:
  - WHERE **t1**.col + **t2**.col = 1
- Any function calls in projection (select list)
- PL/SQL function calls in projection (or WHERE clause)
- Nested loop joins, sort-merge joins and FILTER lookups
  - Hash joins are special though

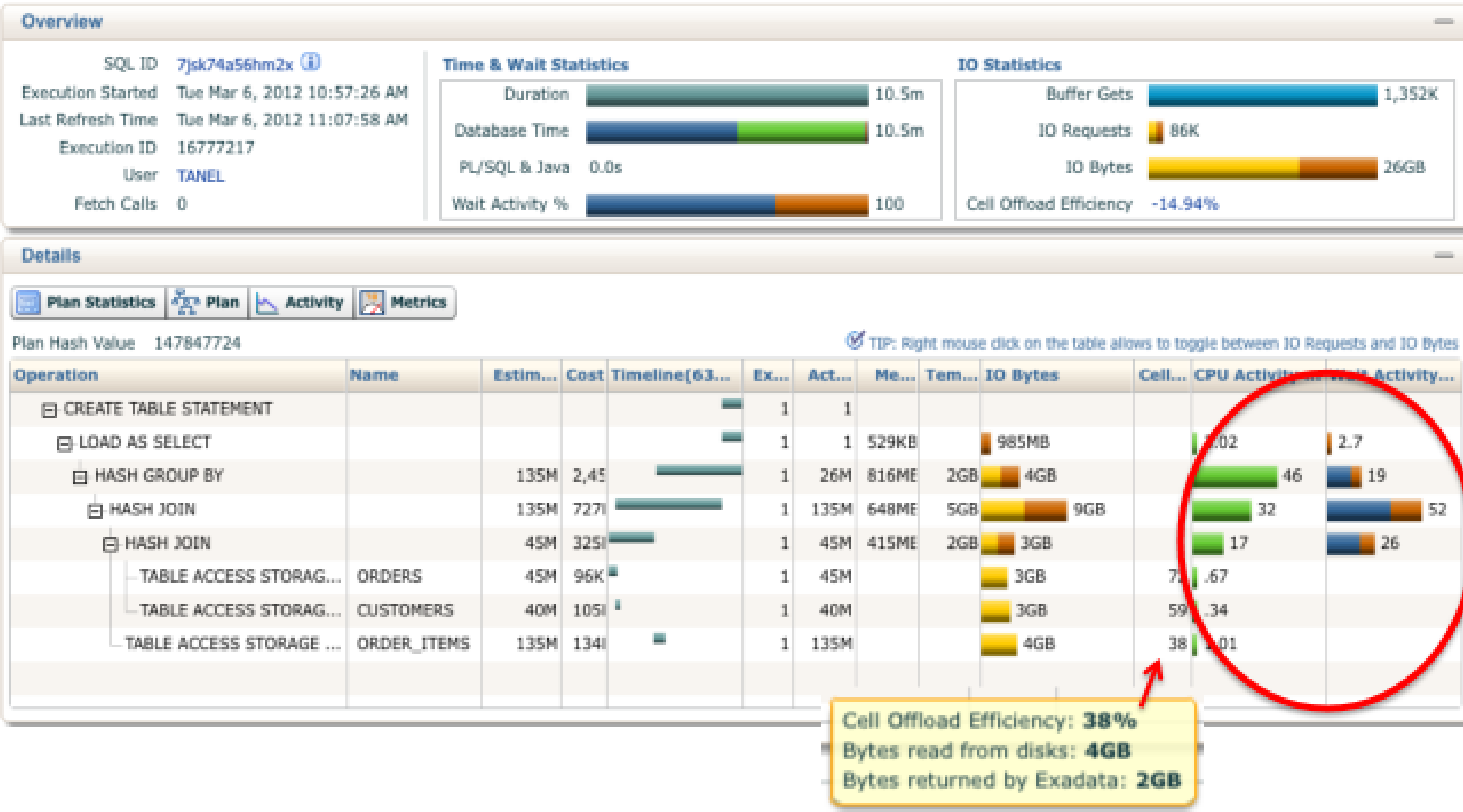
* A prerequisite for smart scans is a *full segment scan* anyway

- So, you'll have to see where is your bottleneck!
  - SQL Monitoring report is a good start

This is not a full list of limitations.

A query bottlenecked by data *processing*, not *retrieval*

- A SQL performing data load and spills to TEMP



Case 3: Case insensitive search

The **plan** hash value does not change as it doesn't take the predicate differences into account!

```
ALTER SESSION SET nls_comp = LINGUISTIC;
ALTER SESSION SET nls_sort = BINARY_CI;
```

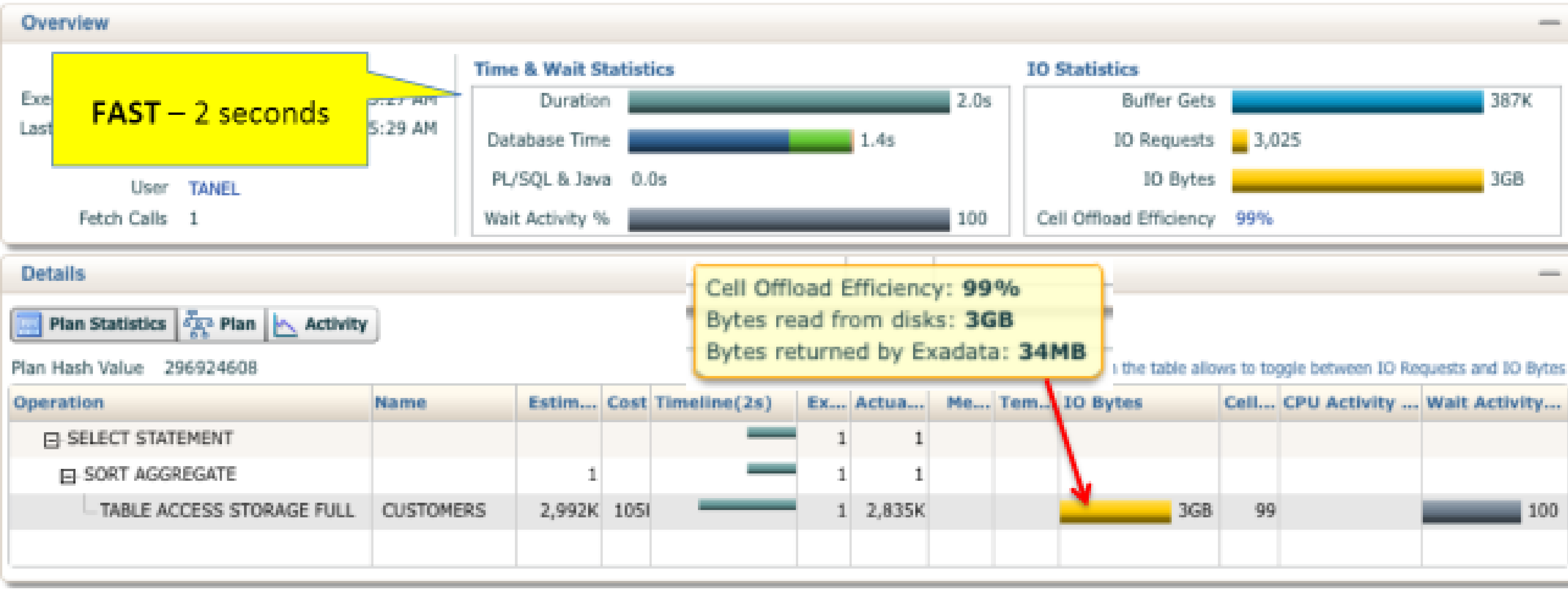
```
SELECT SUM(credit_limit) FROM soe.customers
WHERE cust_first_name LIKE 'j%'

Plan hash value: 296924608

FAST ~ 2 seconds

----------------------------------------------
| Id | Operation                  | Name      |
----------------------------------------------
|  0 | SELECT STATEMENT           |           |
|  1 |  SORT AGGREGATE            |           |
|* 2 |   TABLE ACCESS STORAGE FULL| CUSTOMERS |
----------------------------------------------
```

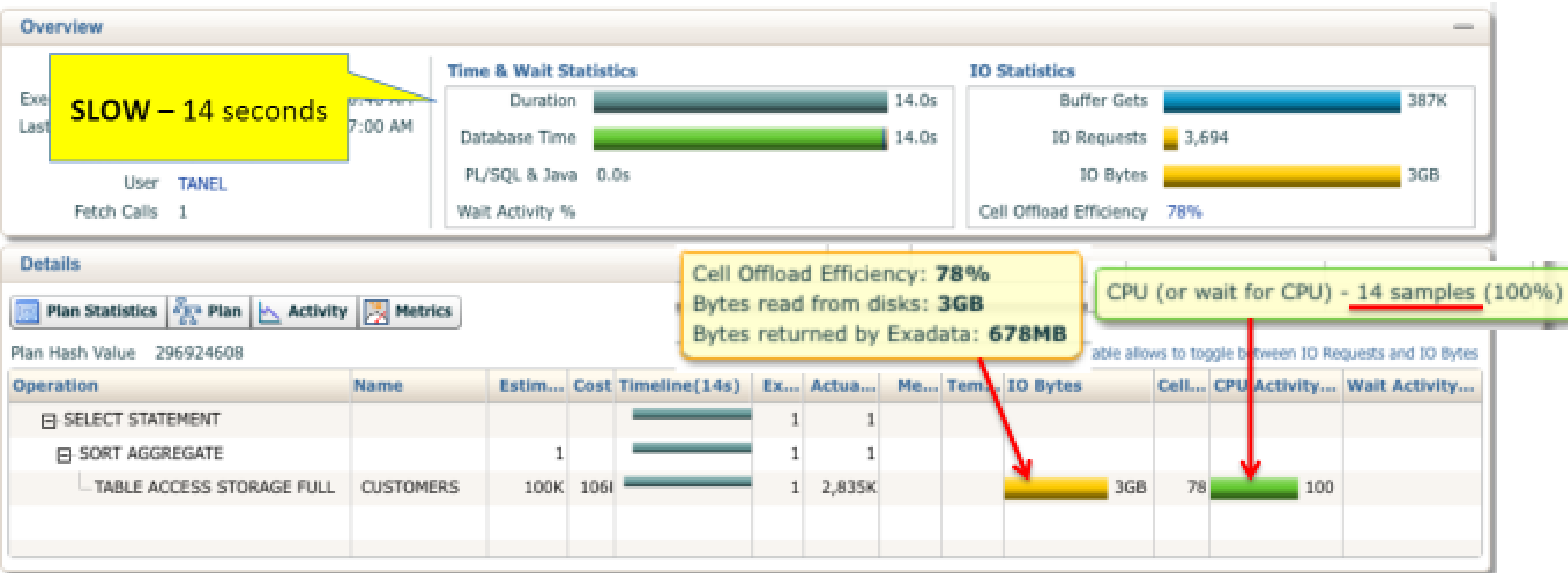
```
SELECT SUM(credit_limit) FROM soe.customers
WHERE cust_first_name LIKE 'j%'

Plan hash value: 296924608

SLOW ~14 seconds

----------------------------------------------
| Id  | Operation                  | Name     |
----------------------------------------------
|   0 | SELECT STATEMENT           |          |
|   1 |  SORT AGGREGATE            |          |
|*  2 |   TABLE ACCESS STORAGE FULL| CUSTOMERS|
----------------------------------------------
```

```
Predicate Information
----------------------------------------------

 2 - storage("CUST_FIRST_NAME" LIKE 'j%')
       filter("CUST_FIRST_NAME" LIKE 'j%')
```

```
Predicate Information
------------------------------------------------

   2 - filter(NLSSORT
       ("OWNER",'nls_sort=''BINARY_CI''')
         >HEXTORAW('7300') )
```

Where's the *storage* predicate?


Overview
FAST – 2 seconds
User TANEL
Fetch Calls 1
Time & Wait Statistics
Duration 2.0s
Database Time 1.4s
PL/SQL & Java 0.0s
Wait Activity % 100
IO Statistics
Buffer Gets 387K
IO Requests 3,025
IO Bytes 3GB
Cell Offload Efficiency 99%
Details
Plan Statistics
Plan
Activity
Plan Hash Value 296924608
Cell Offload Efficiency: 99%
Bytes read from disks: 3GB
Bytes returned by Exadata: 34MB
the table allows to toggle between IO Requests and IO Bytes
Operation
Name
Estim...
Cost
Timeline(2s)
Ex...
Actua...
Me...
Tem...
IO Bytes
Cell...
CPU Activity ...
Wait Activity...
SELECT STATEMENT 1 1
SORT AGGREGATE 1 1 1
TABLE ACCESS STORAGE FULL CUSTOMERS 2,992K 105I 1 2,835K 3GB 99 100


Overview
SLOW – 14 seconds
7:00 AM
User TANEL
Fetch Calls 1
Time & Wait Statistics
Duration 14.0s
Database Time 14.0s
PL/SQL & Java 0.0s
Wait Activity %
IO Statistics
Buffer Gets 387K
IO Requests 3,694
IO Bytes 3GB
Cell Offload Efficiency 78%
Details
Plan Statistics
Plan
Activity
Metrics
Plan Hash Value 296924608
Cell Offload Efficiency: 78%
Bytes read from disks: 3GB
Bytes returned by Exadata: 678MB
CPU (or wait for CPU) - 14 samples (100%)
able allows to toggle between IO Requests and IO Bytes
Operation
Name
Estim...
Cost
Timeline(14s)
Ex...
Actua...
Me...
Tem...
IO Bytes
Cell...
CPU Activity...
Wait Activity...
SELECT STATEMENT 1 1
SORT AGGREGATE 1 1 1
TABLE ACCESS STORAGE FULL CUSTOMERS 100K 106I 1 2,835K 3GB 78 100

Thank you – and oh, wait!

- **Advanced Exadata Performance seminar!**
  - By Tanel Poder
  - Systematic Exadata Performance Troubleshooting and Optimization
  - 2-day seminar:
    - Dallas, TX – 2-3 May 2013
    - Online – 13-16 May 2013
  - http://blog.tanelpoder.com/seminar/
  - We'll go very deep! ☺

- **Enkitec Extreme Exadata Expo (E4)**
  - August 5-6 2013 - lots of great speakers! ->
  - http://enkitec.com/e4