solarwinds

Everything You Need to Know About Oracle 12c Indexes

Janis Griffin

Senior DBA / Performance Evangelist

Who Am I

solarwinds

- Senior DBA / Performance Evangelist for SolarWinds
  - Janis.Griffin@solarwinds.com
  - Twitter® - @DoBoutAnything
  - Current – 25+ Years in Oracle®, DB2®, ASE, SQL Server®, MySQL®
  - DBA and Developer
- Specialize in Performance Tuning
- Review Database Performance for Customers and Prospects
- Common Question – How do I tune it?

ORACLE ACE

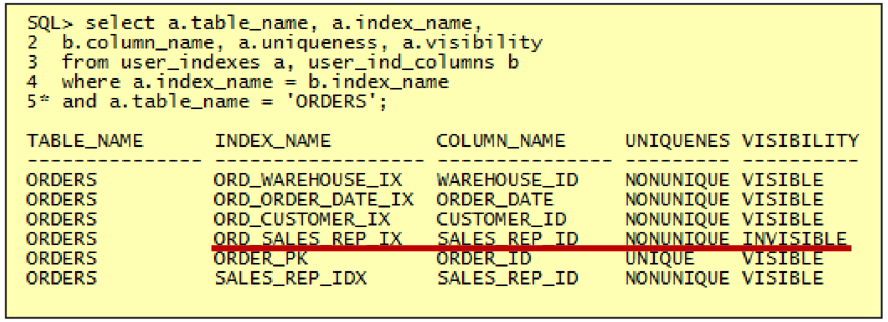
Agenda

- Oracle Index Structures and Options
    - Understanding B-Tree Indexes
    - When to use Bitmap Indexes
- Indexes on Referential Constraints
    - Differences on unique and not unique indexes
    - Foreign keys vs. primary keys
    - Nullable columns and indexes
- New 12.2 Index Features
    - Partial indexes
    - Advanced index compression
- Index Statistics
    - Dictionary views – DBA_INDEXES, INDEX_STATS, V$SEGMENT_STATISTICS
    - Collection strategies

Index Overview

- Optional structure associate with a table or table cluster
  - Can be on one or more columns of a table
    - Can be unique or non-unique values
  - Can speed up data retrieval
  - Reduces disk I/O
- Two types of indexes
  - B-Tree indexes
    - The default when using ‘create index’ clause
  - Bitmap indexes
- Index states
  - Default is Usable
    - Can make unusable so optimizer won’t use or maintain
      - Takes no physical space
  - Default is Visible
    - Can make invisible so optimizer will maintain but won’t use it

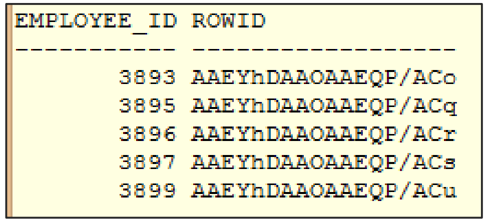
ALTER SESSION/SYSTEM SET
optimizer_use_invisible_indexes=false;



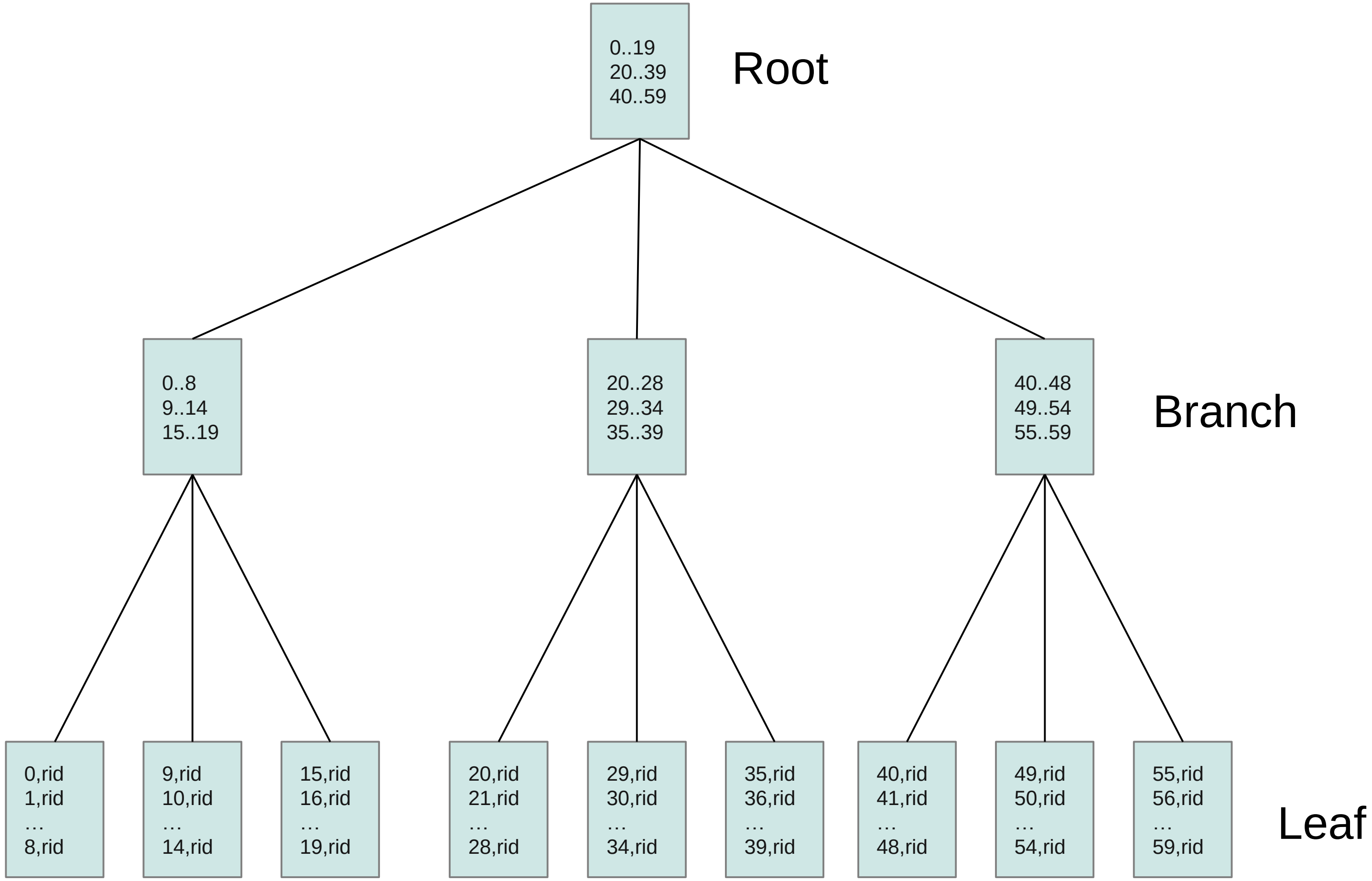
```
SQL> select a.table_name, a.index_name,
2  b.column_name, a.uniqueness, a.visibility
3  from user_indexes a, user_ind_columns b
4  where a.index_name = b.index_name
5* and a.table_name = 'ORDERS';

TABLE_NAME       INDEX_NAME          COLUMN_NAME      UNIQUENES VISIBILITY
---------------- ------------------- ---------------- --------- ----------
ORDERS           ORD_WAREHOUSE_IX    WAREHOUSE_ID     NONUNIQUE VISIBLE
ORDERS           ORD_ORDER_DATE_IX   ORDER_DATE       NONUNIQUE VISIBLE
ORDERS           ORD_CUSTOMER_IX     CUSTOMER_ID      NONUNIQUE VISIBLE
ORDERS           ORD_SALES_REP_IX    SALES_REP_ID     NONUNIQUE INVISIBLE
ORDERS           ORDER_PK            ORDER_ID         UNIQUE    VISIBLE
ORDERS           SALES_REP_IDX       SALES_REP_ID     NONUNIQUE VISIBLE
```

B-Tree Index Overview

- B-Tree Indexes (Default)
  - Two types of blocks
    - Branch
    - Leaf
  - Root points to branch
  - Branch points to leaf
  - Leaf points to rowid
    - In table



```
EMPLOYEE_ID ROWID
----------- ------------------
       3893 AAEYhDAAOAAEQP/ACo
       3895 AAEYhDAAOAAEQP/ACq
       3896 AAEYhDAAOAAEQP/ACr
       3897 AAEYhDAAOAAEQP/ACs
       3899 AAEYhDAAOAAEQP/ACu
```

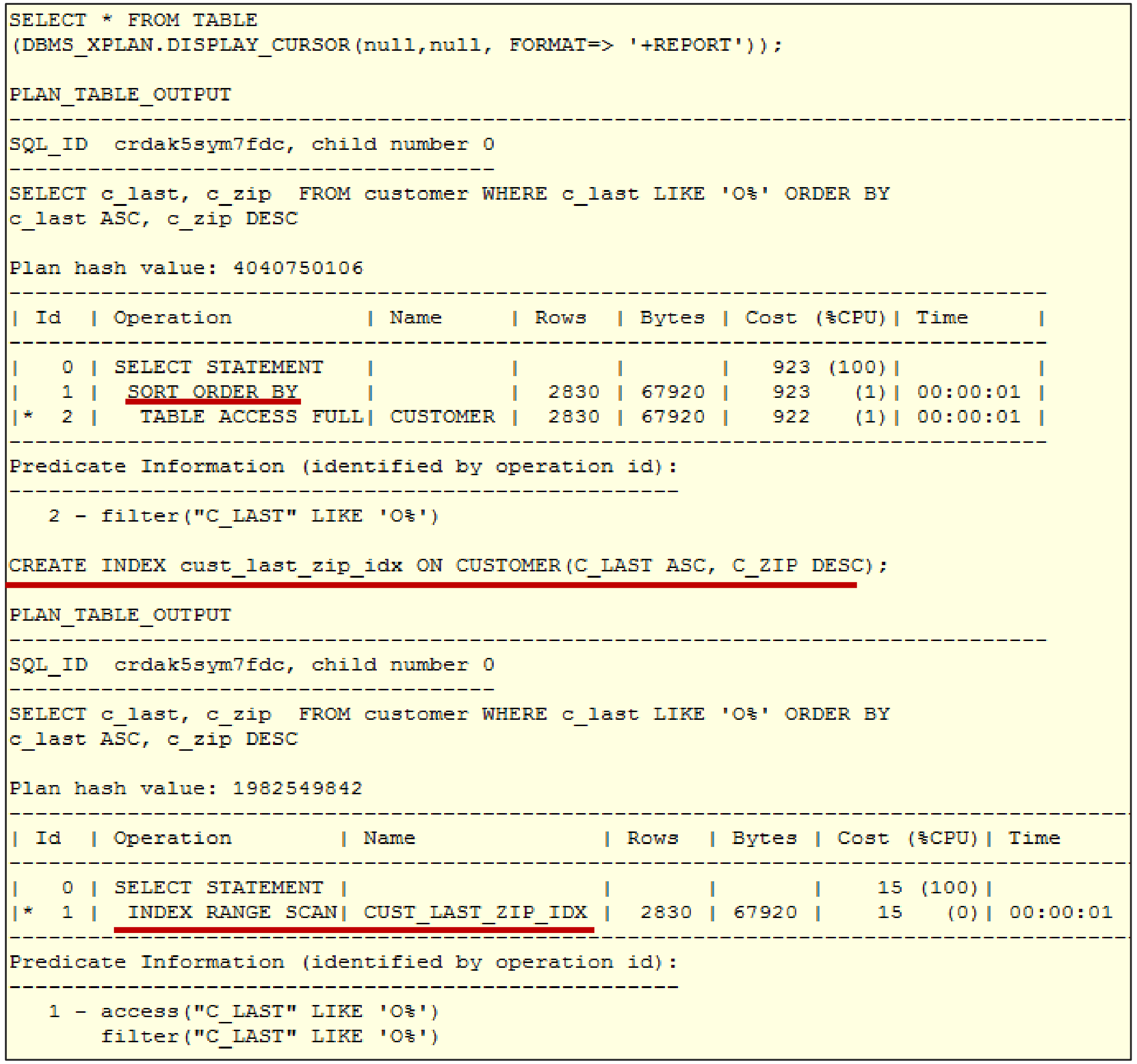


B-Tree Index Sub-types

solarwinds

- Descending indexes
  - Physically stores data in descending order
    - FUNCTION-BASED NORMAL index type
      - Can take up more space
  - Default is stored in ascending order
  - Can reduce query sorts

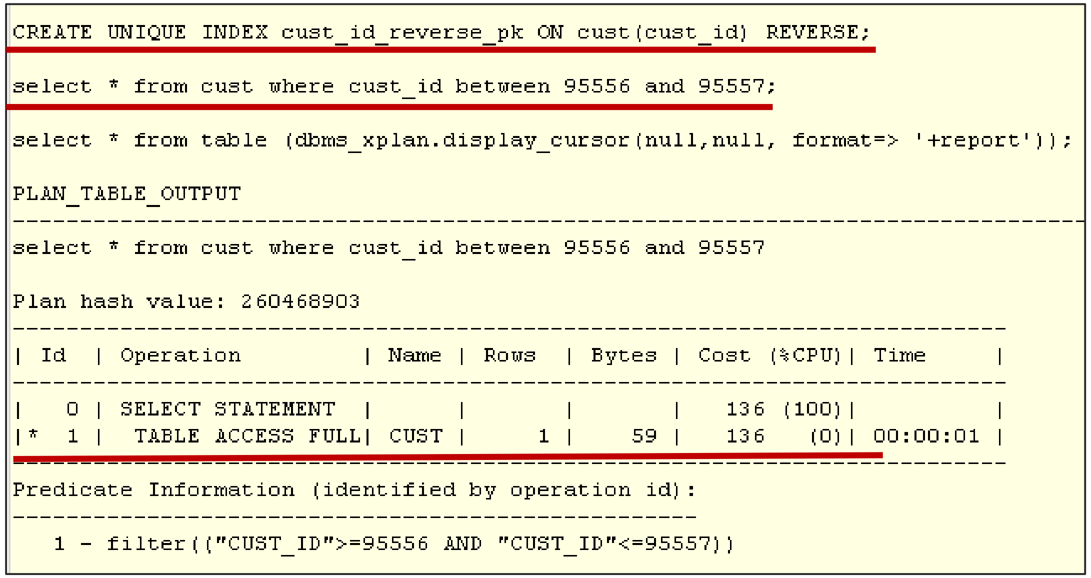
SELECT c_last, c_zip FROM customer
WHERE c_last LIKE 'O%'
ORDER BY c_last ASC, c_zip DESC;



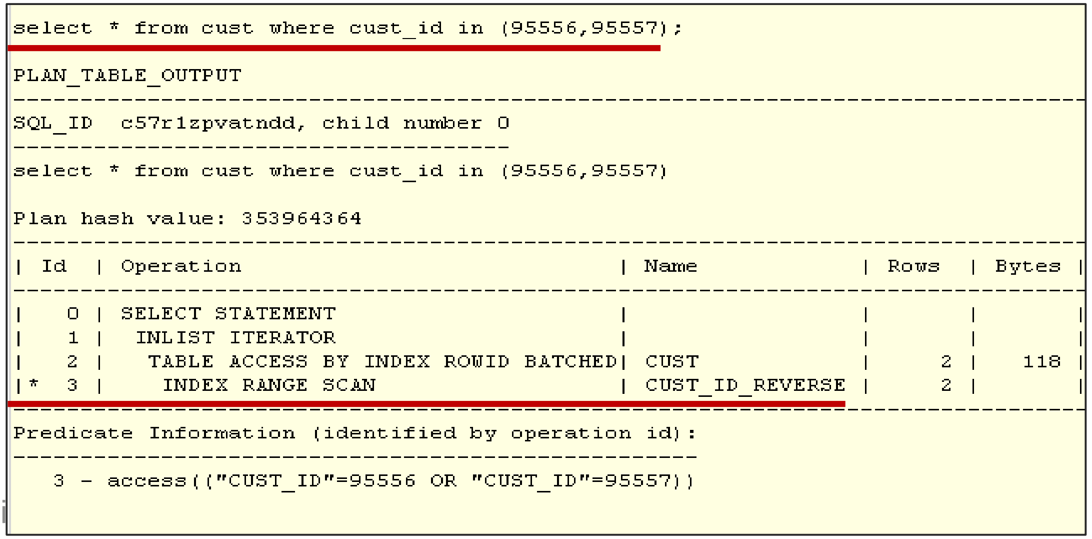
```
SELECT * FROM TABLE
(DBMS_XPLAN.DISPLAY_CURSOR(null,null, FORMAT=> '+REPORT'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------------
SQL_ID  crdak5sym7fdc, child number 0
-------------------------------------
SELECT c_last, c_zip  FROM customer WHERE c_last LIKE 'O%' ORDER BY
c_last ASC, c_zip DESC

Plan hash value: 4040750106

-------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |       |       |   923 (100)|          |
|   1 |  SORT ORDER BY     |          |  2830 | 67920 |   923   (1)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| CUSTOMER |  2830 | 67920 |   922   (1)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("C_LAST" LIKE 'O%')

CREATE INDEX cust_last_zip_idx ON CUSTOMER(C_LAST ASC, C_ZIP DESC);

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
SQL_ID  crdak5sym7fdc, child number 0
-------------------------------------
SELECT c_last, c_zip  FROM customer WHERE c_last LIKE 'O%' ORDER BY
c_last ASC, c_zip DESC

Plan hash value: 1982549842

--------------------------------------------------------------------------------------
| Id  | Operation        | Name              | Rows  | Bytes | Cost (%CPU)| Time
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |                   |       |       |    15 (100)|
|*  1 |  INDEX RANGE SCAN| CUST_LAST_ZIP_IDX |  2830 | 67920 |    15   (0)| 00:00:01
--------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("C_LAST" LIKE 'O%')
       filter("C_LAST" LIKE 'O%')
```

B-Tree Index Sub-types - cont.

- Reverse key indexes
  - Helps with index block contention
  - Physically reverses the bytes of index key
    - To spread sequential inserts over many blocks
    - Example: 123, 124, 125
      - stored as 321, 421, 521 respectively
  - Reduces high waits on index segments
    - Look for “buffer busy waits” wait event
      - Or “read by other session” wait event
  - May be useful in RAC environments
    - Many nodes inserting into same hot index block
  - Great if needing insert performance
  - Optimizer may not use it for index range scans
    - Be careful of using ‘between’ or ‘like’
    - Might use it with ‘in’, ‘=‘, and ‘or’

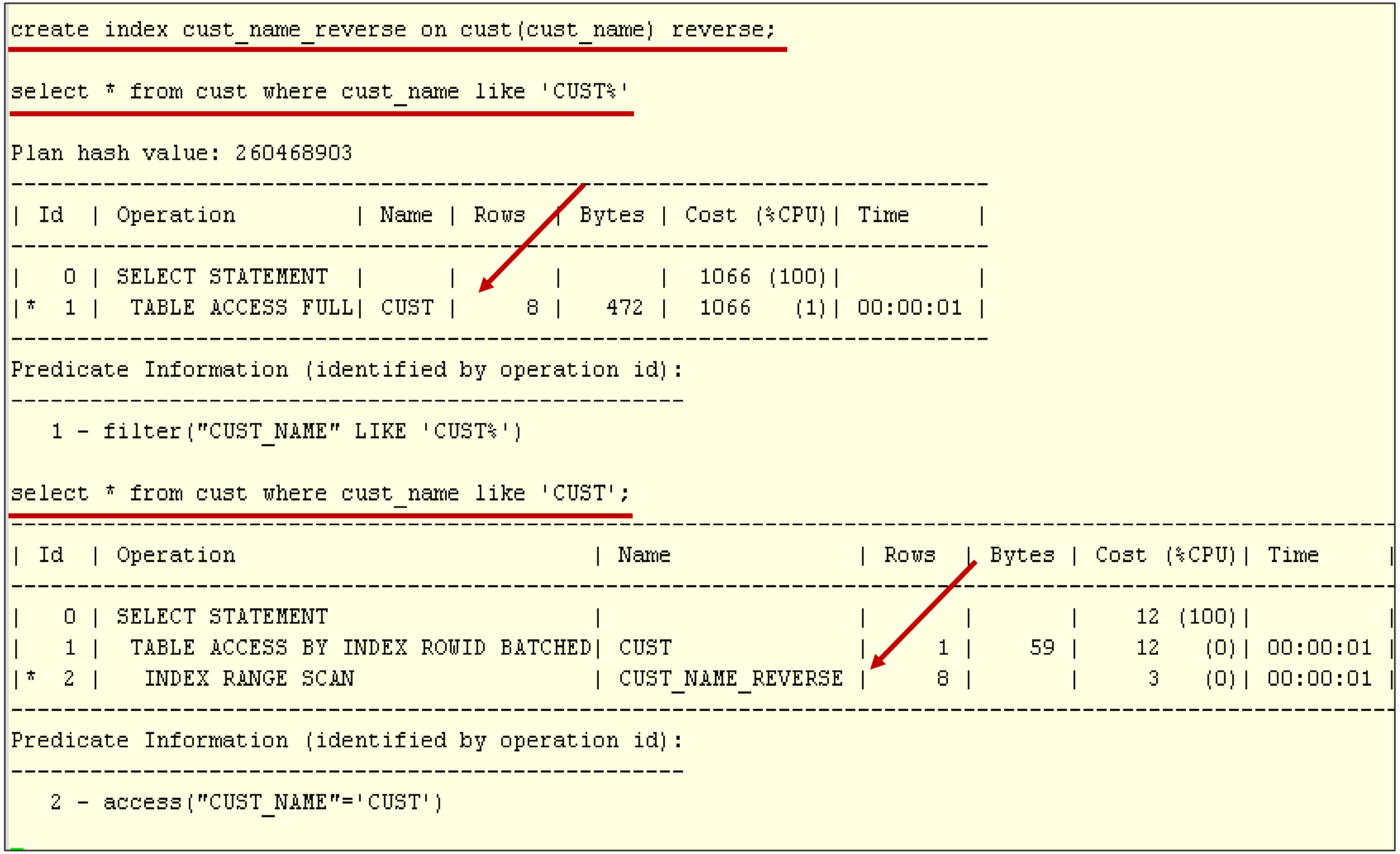


```
CREATE UNIQUE INDEX cust_id_reverse_pk ON cust(cust_id) REVERSE;

select * from cust where cust_id between 95556 and 95557;

select * from table (dbms_xplan.display_cursor(null,null, format=> '+report'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
select * from cust where cust_id between 95556 and 95557

Plan hash value: 260468903
---------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |       |       |   136 (100)|          |
|*  1 |  TABLE ACCESS FULL| CUST |     1 |    59 |   136   (0)| 00:00:01 |
---------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(("CUST_ID">=95556 AND "CUST_ID"<=95557))
```



```
select * from cust where cust_id in (95556,95557);

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
SQL_ID  c57r1zpvatndd, child number 0
-------------------------------------
select * from cust where cust_id in (95556,95557)

Plan hash value: 353964364
--------------------------------------------------------------------------------
| Id  | Operation                            | Name              | Rows  | Bytes |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |                   |       |       |
|   1 |  INLIST ITERATOR                     |                   |       |       |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| CUST              |     2 |   118 |
|*  3 |    INDEX RANGE SCAN                  | CUST_ID_REVERSE   |     2 |       |
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access(("CUST_ID"=95556 OR "CUST_ID"=95557))
```

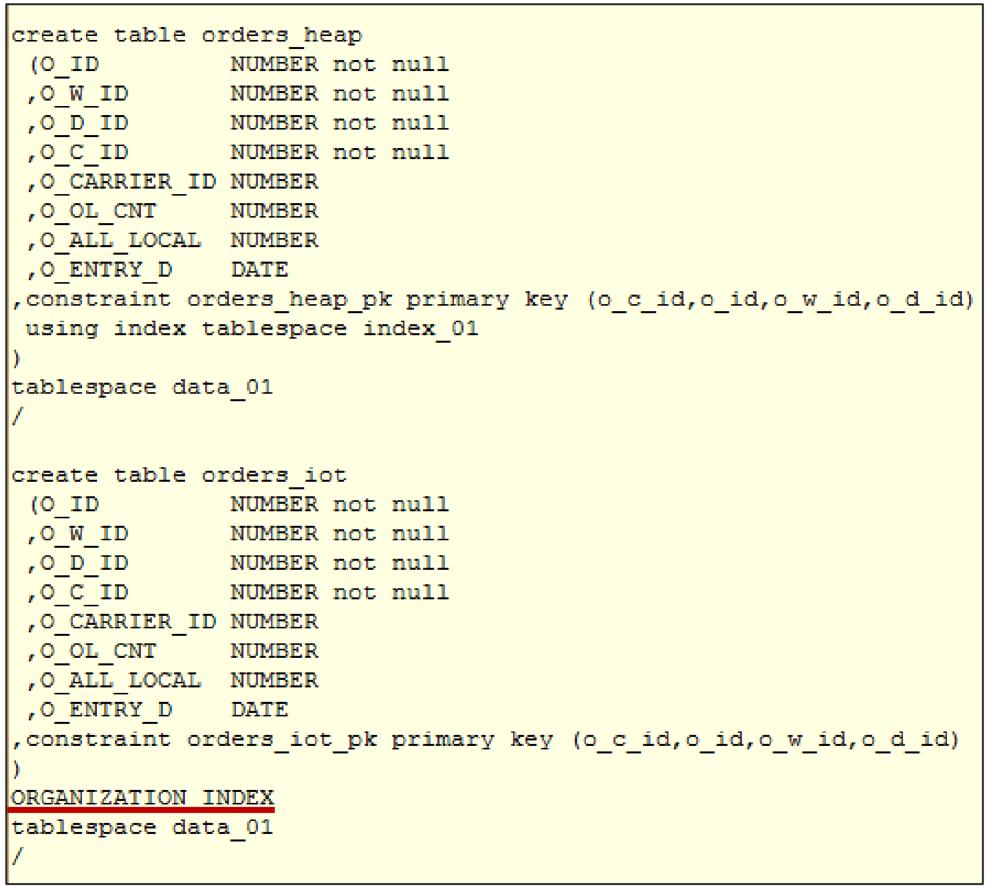
Another Reverse Key Index Example



```
create index cust_name_reverse on cust(cust_name) reverse;

select * from cust where cust_name like 'CUST%'

Plan hash value: 260468903
---------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |       |       |  1066 (100)|          |
|*  1 |  TABLE ACCESS FULL| CUST |     8 |   472 |  1066   (1)| 00:00:01 |
---------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("CUST_NAME" LIKE 'CUST%')

select * from cust where cust_name like 'CUST';
-------------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |                   |       |       |    12 (100)|          |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| CUST              |     1 |    59 |    12   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | CUST_NAME_REVERSE |     8 |       |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("CUST_NAME"='CUST')
```

B-Tree Index Sub-types - cont.

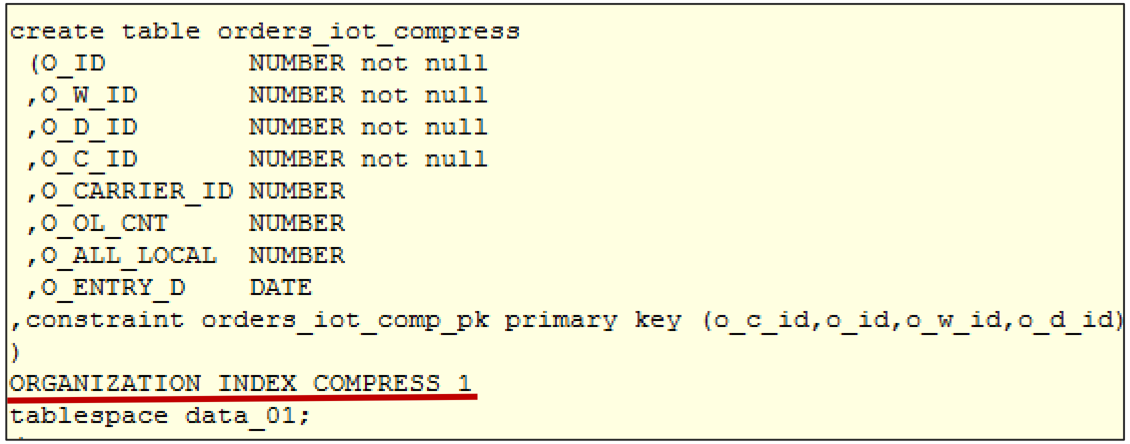
- Index-organized tables (IOT)
  - Rows are physically sorted and stored by primary key
    - The index is the data
    - Requires less storage space
      - Can further be reduced by using key compression
  - Can significantly reduce IO
    - When accessing a range of primary key values
    - No table access as data is in the index leaf
  - Rows are accessed via a logical rowid
    - Not a physical rowid like in heap-organized tables
  - Disadvantages
    - IOT must have a primary key
    - IOT can’t be in a cluster
    - IOT can’t have LONG data type columns
      - Or virtual columns
    - Inserts and updates may be much slower



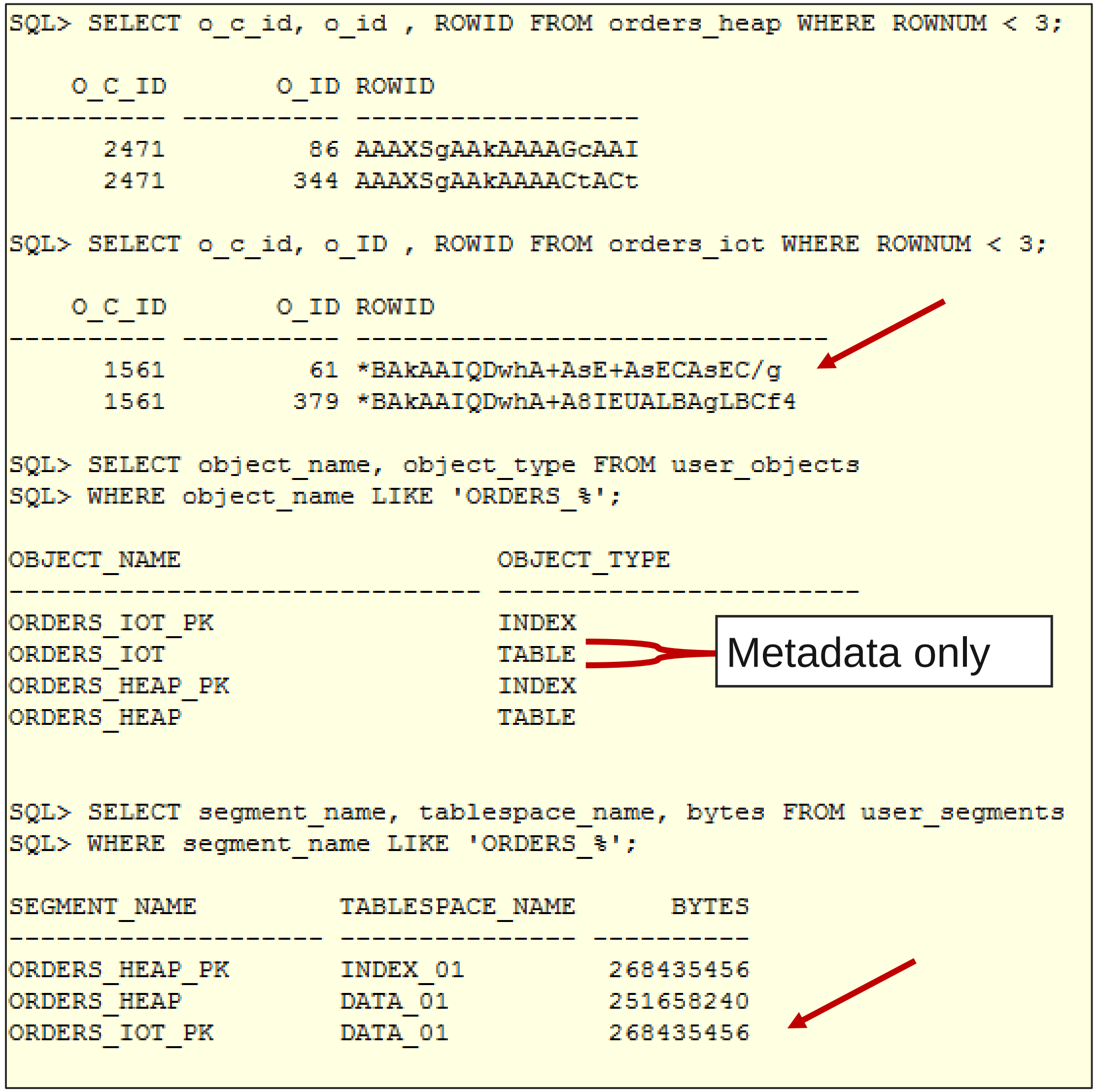
```
create table orders_heap
 (O_ID         NUMBER not null
 ,O_W_ID       NUMBER not null
 ,O_D_ID       NUMBER not null
 ,O_C_ID       NUMBER not null
 ,O_CARRIER_ID NUMBER
 ,O_OL_CNT     NUMBER
 ,O_ALL_LOCAL  NUMBER
 ,O_ENTRY_D    DATE
,constraint orders_heap_pk primary key (o_c_id,o_id,o_w_id,o_d_id)
 using index tablespace index_01
)
tablespace data_01
/

create table orders_iot
 (O_ID         NUMBER not null
 ,O_W_ID       NUMBER not null
 ,O_D_ID       NUMBER not null
 ,O_C_ID       NUMBER not null
 ,O_CARRIER_ID NUMBER
 ,O_OL_CNT     NUMBER
 ,O_ALL_LOCAL  NUMBER
 ,O_ENTRY_D    DATE
,constraint orders_iot_pk primary key (o_c_id,o_id,o_w_id,o_d_id)
)
ORGANIZATION INDEX
tablespace data_01
/
```

Index-Organized Tables (IOT) Example

- Uses logical rowids
  - Not physical rowids
  - Contains a physical guess of data block
    - Used by secondary indexes
- Key Compression on IOTs
  - Eliminates repeated key values
    - E.g. Keys 1,2,3 and 1,2,4
      - Values 1,2 are compressed



```
create table orders_iot_compress
 (O_ID          NUMBER not null
 ,O_W_ID        NUMBER not null
 ,O_D_ID        NUMBER not null
 ,O_C_ID        NUMBER not null
 ,O_CARRIER_ID  NUMBER
 ,O_OL_CNT      NUMBER
 ,O_ALL_LOCAL   NUMBER
 ,O_ENTRY_D     DATE
,constraint orders_iot_comp_pk primary key (o_c_id,o_id,o_w_id,o_d_id)
)
ORGANIZATION INDEX COMPRESS 1
tablespace data_01;
```

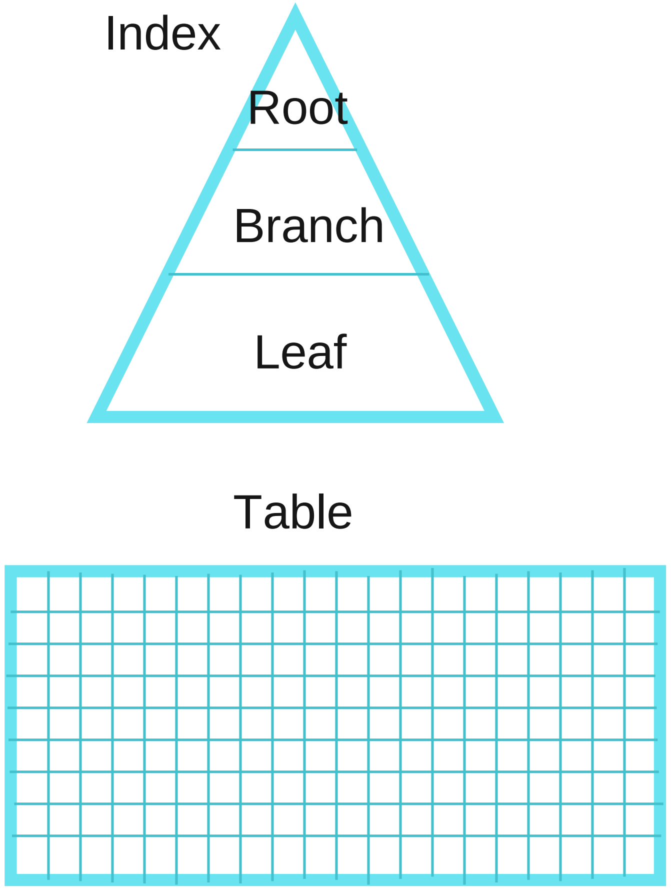


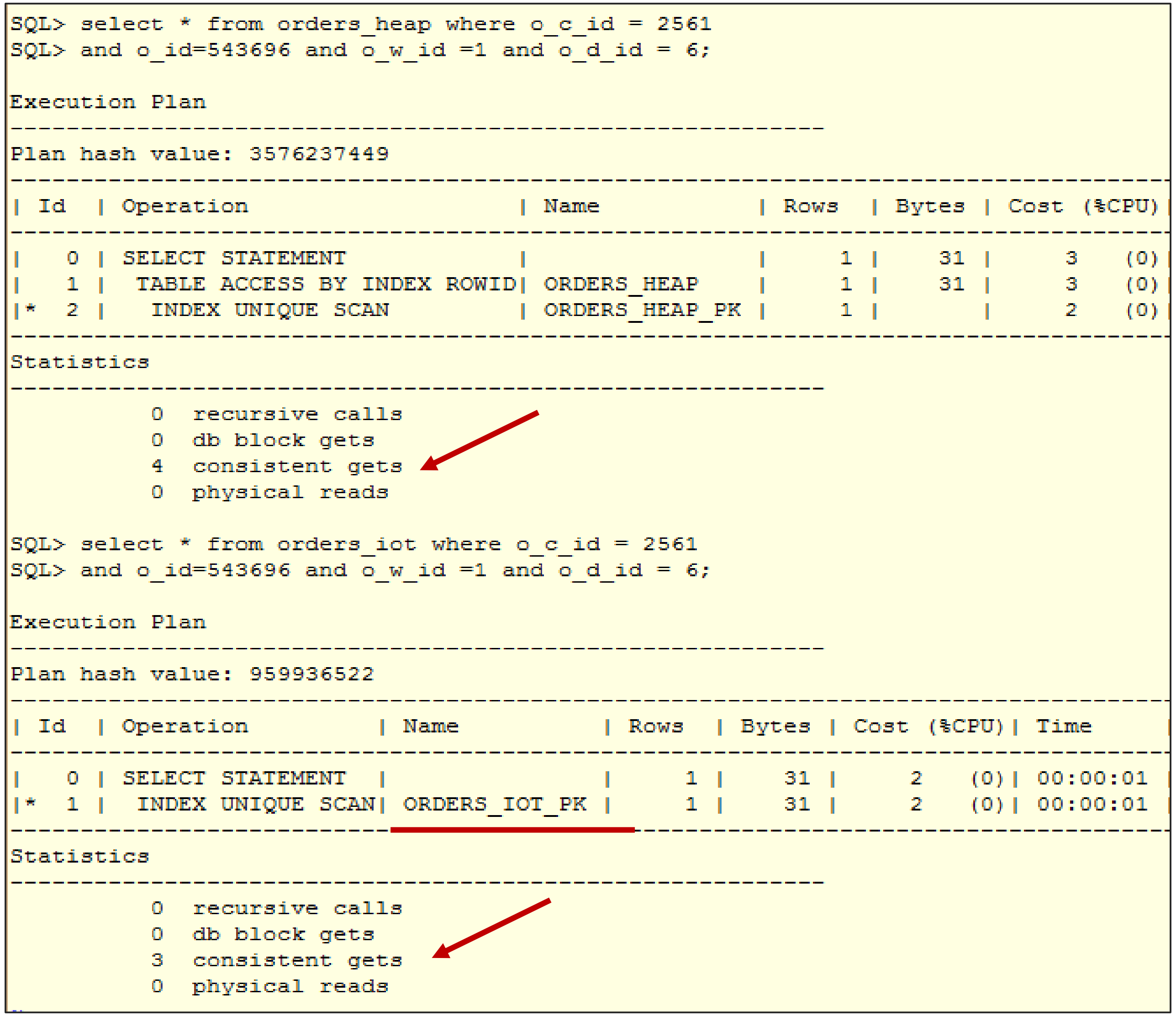
```
SQL> SELECT o_c_id, o_id , ROWID FROM orders_heap WHERE ROWNUM < 3;

    O_C_ID       O_ID ROWID
---------- ---------- ------------------
      2471         86 AAAXSgAAkAAAAGcAAI
      2471        344 AAAXSgAAkAAAACtACt

SQL> SELECT o_c_id, o_ID , ROWID FROM orders_iot WHERE ROWNUM < 3;

    O_C_ID       O_ID ROWID
---------- ---------- ------------------------------
      1561         61 *BAkAAIQDwhA+AsE+AsECAsEC/g
      1561        379 *BAkAAIQDwhA+A8IEUALBAgLBCf4

SQL> SELECT object_name, object_type FROM user_objects
SQL> WHERE object_name LIKE 'ORDERS_%';

OBJECT_NAME                    OBJECT_TYPE
------------------------------ -----------------------
ORDERS_IOT_PK                  INDEX
ORDERS_IOT                     TABLE
ORDERS_HEAP_PK                 INDEX
ORDERS_HEAP                    TABLE


SQL> SELECT segment_name, tablespace_name, bytes FROM user_segments
SQL> WHERE segment_name LIKE 'ORDERS_%';

SEGMENT_NAME         TABLESPACE_NAME      BYTES
-------------------- --------------- ----------
ORDERS_HEAP_PK       INDEX_01         268435456
ORDERS_HEAP          DATA_01          251658240
ORDERS_IOT_PK        DATA_01          268435456
```

Metadata only

Index-Organized Tables (IOT) Example

- Single row lookup

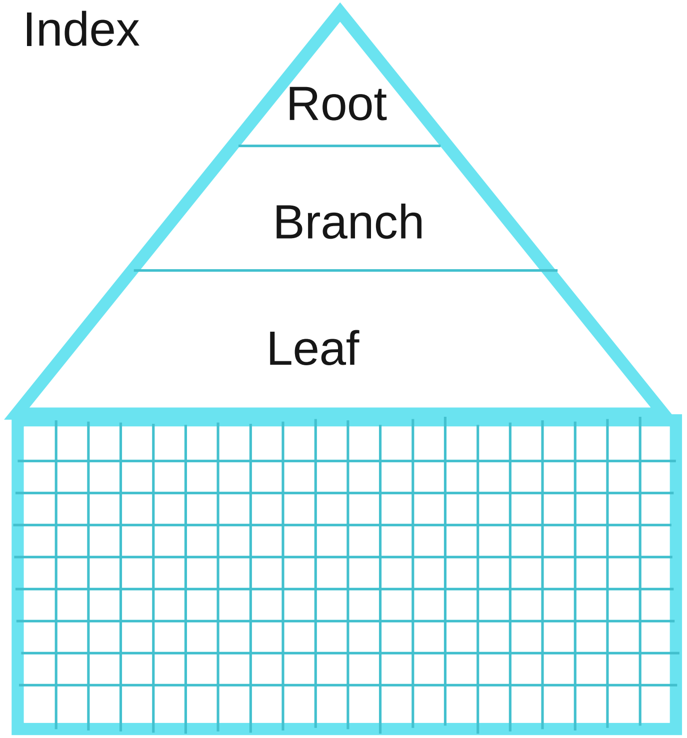


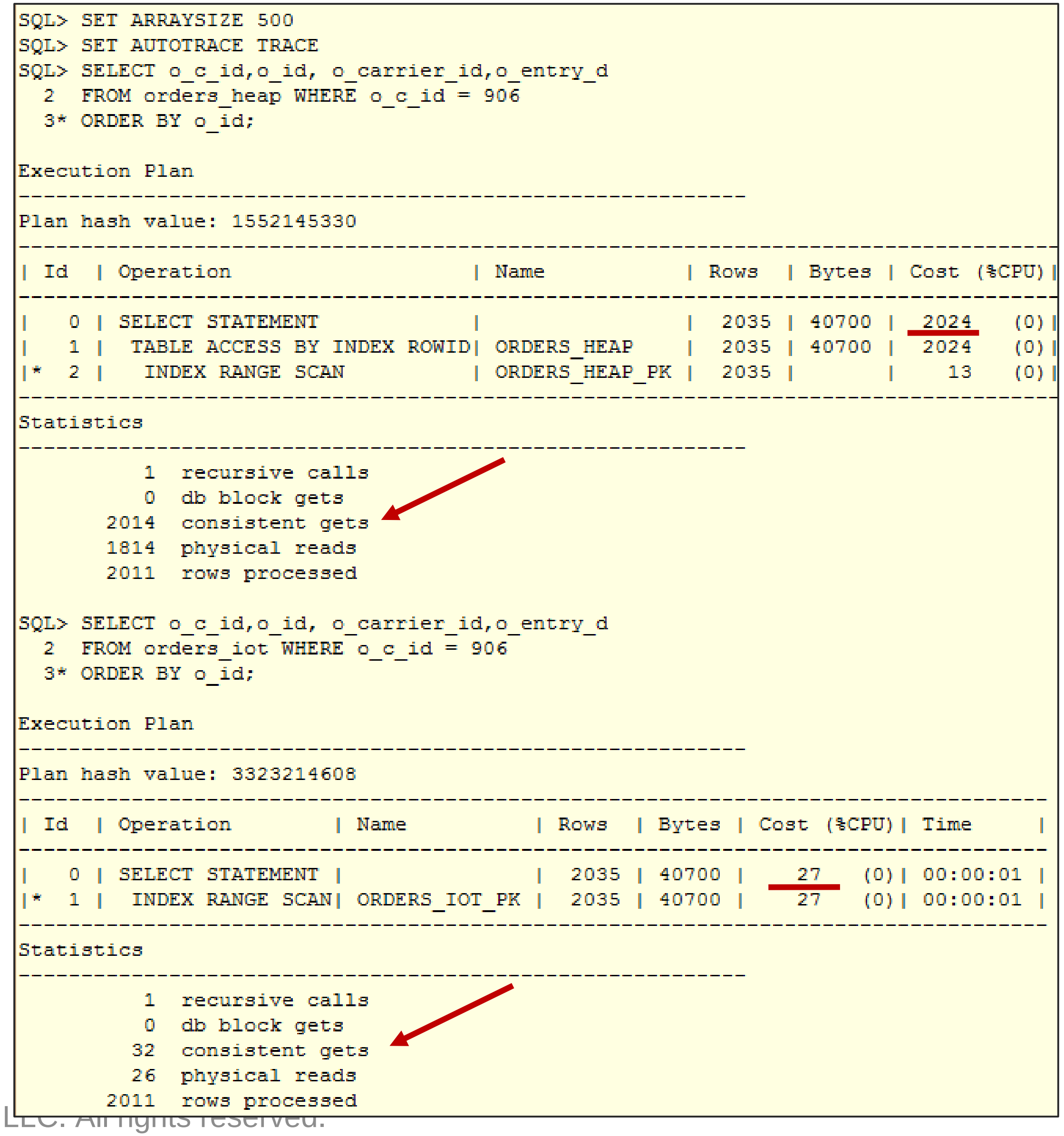


```
SQL> select * from orders_heap where o_c_id = 2561
SQL> and o_id=543696 and o_w_id =1 and o_d_id = 6;

Execution Plan
----------------------------------------------------------
Plan hash value: 3576237449

----------------------------------------------------------------------------------
| Id  | Operation                   | Name            | Rows  | Bytes | Cost (%CPU)|
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                 |     1 |    31 |     3   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| ORDERS_HEAP     |     1 |    31 |     3   (0)|
|*  2 |   INDEX UNIQUE SCAN         | ORDERS_HEAP_PK  |     1 |       |     2   (0)|
----------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads

SQL> select * from orders_iot where o_c_id = 2561
SQL> and o_id=543696 and o_w_id =1 and o_d_id = 6;

Execution Plan
----------------------------------------------------------
Plan hash value: 959936522

----------------------------------------------------------------------------------
| Id  | Operation         | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |               |     1 |    31 |     2   (0)| 00:00:01 |
|*  1 |  INDEX UNIQUE SCAN| ORDERS_IOT_PK |     1 |    31 |     2   (0)| 00:00:01 |
----------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          3  consistent gets
          0  physical reads
```

Index-Organized Tables (IOT) Example

- Index Range Scan
  - Significant performance gain

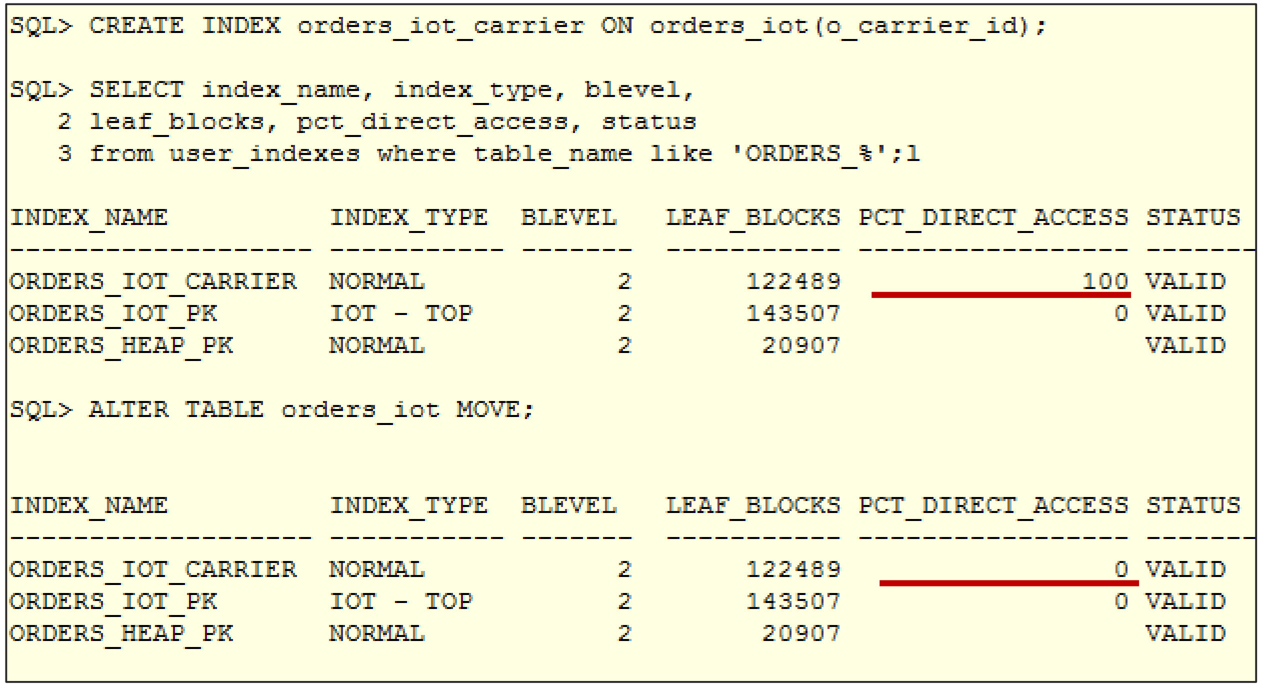




```
SQL> SET ARRAYSIZE 500
SQL> SET AUTOTRACE TRACE
SQL> SELECT o_c_id,o_id, o_carrier_id,o_entry_d
  2  FROM orders_heap WHERE o_c_id = 906
  3* ORDER BY o_id;

Execution Plan
----------------------------------------------------------
Plan hash value: 1552145330

-----------------------------------------------------------------------------------
| Id  | Operation                    | Name             | Rows  | Bytes | Cost (%CPU)|
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                  |  2035 | 40700 |  2024   (0)|
|   1 |  TABLE ACCESS BY INDEX ROWID| ORDERS_HEAP      |  2035 | 40700 |  2024   (0)|
|*  2 |   INDEX RANGE SCAN           | ORDERS_HEAP_PK |  2035 |       |    13   (0)|
-----------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
       2014  consistent gets
       1814  physical reads
       2011  rows processed

SQL> SELECT o_c_id,o_id, o_carrier_id,o_entry_d
  2  FROM orders_iot WHERE o_c_id = 906
  3* ORDER BY o_id;

Execution Plan
----------------------------------------------------------
Plan hash value: 3323214608

----------------------------------------------------------------------------------
| Id  | Operation          | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |                | 2035 | 40700 |    27   (0)| 00:00:01 |
|*  1 |  INDEX RANGE SCAN| ORDERS_IOT_PK | 2035 | 40700 |    27   (0)| 00:00:01 |
----------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
         32  consistent gets
         26  physical reads
       2011  rows processed
```

Index-Organized Tables (IOT) Example

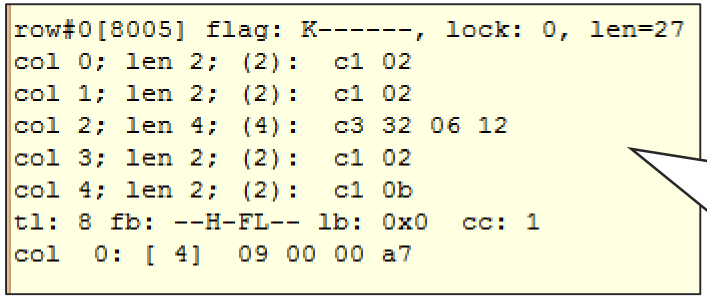
- Secondary Indexes
  - Can be unique or non-unique,
    - function-based, b-tree or bitmap
  - Use physical guess of data block
    - Can become stale overtime
    - PCT_DIRECT_ACCESS (USER_INDEXES)
  - Also contains primary key
    - Used if ‘guess’ becomes stale
  - Are still usable after
    - ‘ALTER TABLE… MOVE’
  - Index structure
    - See Appendix



```
SQL> CREATE INDEX orders_iot_carrier ON orders_iot(o_carrier_id);

SQL> SELECT index_name, index_type, blevel,
   2 leaf_blocks, pct_direct_access, status
   3 from user_indexes where table_name like 'ORDERS_%';l

INDEX_NAME          INDEX_TYPE  BLEVEL   LEAF_BLOCKS PCT_DIRECT_ACCESS STATUS
------------------- ----------- -------  ----------- ----------------- --------
ORDERS_IOT_CARRIER  NORMAL            2       122489               100 VALID
ORDERS_IOT_PK       IOT - TOP         2       143507                 0 VALID
ORDERS_HEAP_PK      NORMAL            2        20907                   VALID

SQL> ALTER TABLE orders_iot MOVE;


INDEX_NAME          INDEX_TYPE  BLEVEL   LEAF_BLOCKS PCT_DIRECT_ACCESS STATUS
------------------- ----------- -------  ----------- ----------------- --------
ORDERS_IOT_CARRIER  NORMAL            2       122489                 0 VALID
ORDERS_IOT_PK       IOT - TOP         2       143507                 0 VALID
ORDERS_HEAP_PK      NORMAL            2        20907                   VALID
```

ALTER SYSTEM DUMP DATAFILE 16 BLOCK 1134035;

Partial dump of orders_iot_carrier index block

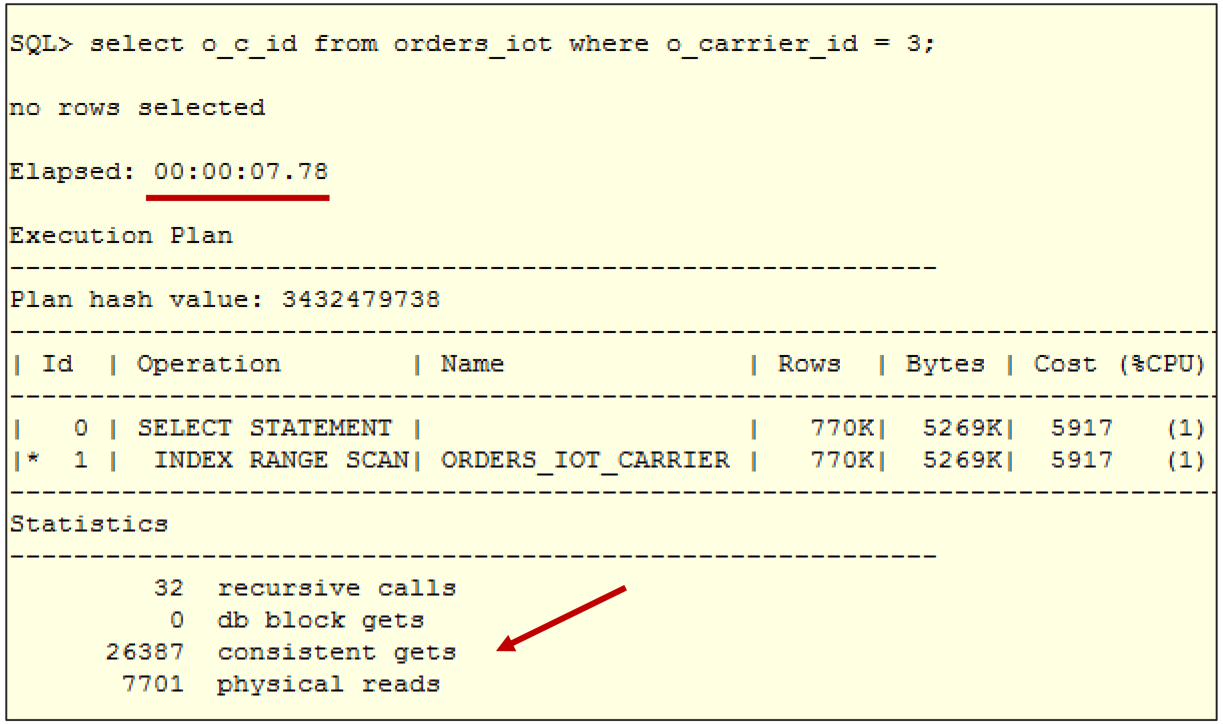


```
row#0[8005] flag: K------, lock: 0, len=27
col 0; len 2; (2):  c1 02
col 1; len 2; (2):  c1 02
col 2; len 4; (4):  c3 32 06 12
col 3; len 2; (2):  c1 02
col 4; len 2; (2):  c1 0b
tl: 8 fb: --H-FL-- lb: 0x0  cc: 1
col  0: [ 4]  09 00 00 a7
```

col 0 = index value
col 1 – col 4 = pk values
tl = table overhead
(needed for guess)
col 0 = 4 byte ‘guess’
(of last known location)

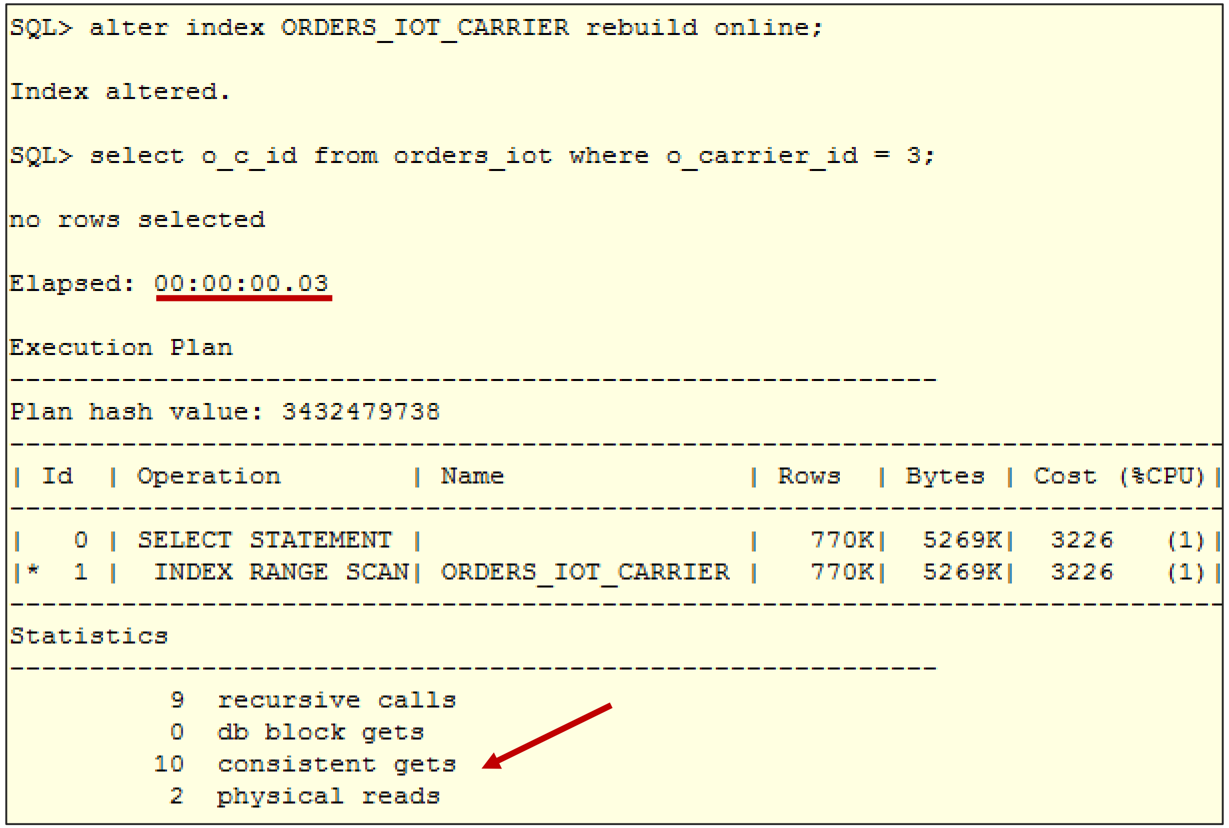
Secondary Indexes on IOTs

- Example of performance when stale
  - PCT_DIRECT_ACCESS = 0



```
SQL> select o_c_id from orders_iot where o_carrier_id = 3;

no rows selected

Elapsed: 00:00:07.78

Execution Plan
----------------------------------------------------------
Plan hash value: 3432479738

-----------------------------------------------------------------------------
| Id  | Operation         | Name                 | Rows  | Bytes | Cost (%CPU)
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT |                      |   770K|  5269K|  5917   (1)
|*  1 |  INDEX RANGE SCAN| ORDERS_IOT_CARRIER |   770K|  5269K|  5917   (1)
-----------------------------------------------------------------------------
Statistics
----------------------------------------------------------
         32  recursive calls
          0  db block gets
      26387  consistent gets
       7701  physical reads
```

Advantage is that you can quickly rebuild index online

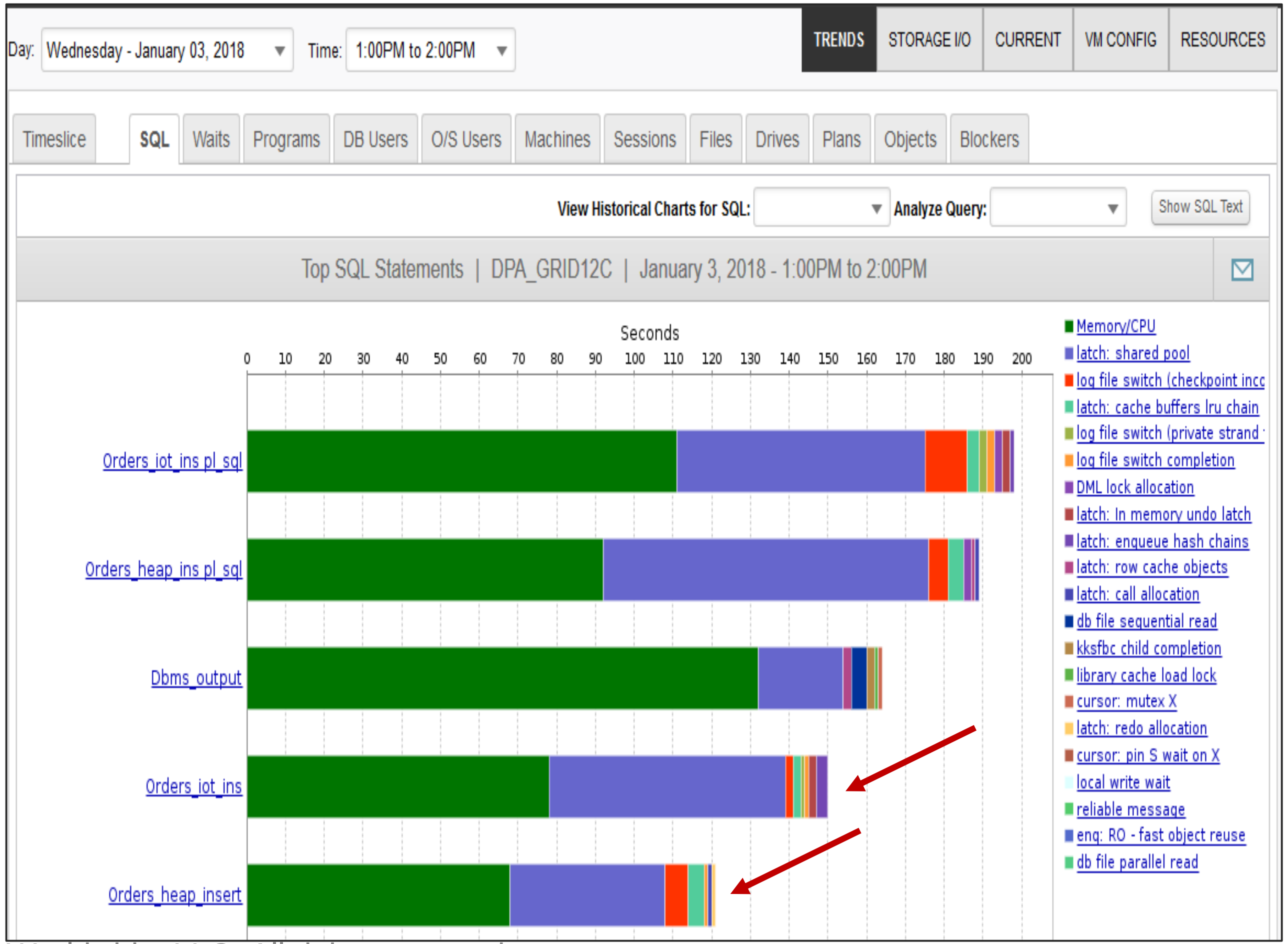


```
SQL> alter index ORDERS_IOT_CARRIER rebuild online;

Index altered.

SQL> select o_c_id from orders_iot where o_carrier_id = 3;

no rows selected

Elapsed: 00:00:00.03

Execution Plan
----------------------------------------------------------
Plan hash value: 3432479738

------------------------------------------------------------------------------
| Id  | Operation         | Name                 | Rows  | Bytes | Cost (%CPU)|
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT |                      |   770K|  5269K|  3226   (1)|
|*  1 |  INDEX RANGE SCAN| ORDERS_IOT_CARRIER |   770K|  5269K|  3226   (1)|
------------------------------------------------------------------------------
Statistics
----------------------------------------------------------
          9  recursive calls
          0  db block gets
         10  consistent gets
          2  physical reads
```

DML Performance on IOTs

- Insert 10000 random records
    - In both ORDERS_HEAP_INS and ORDERS_IOT_INS
    - See Appendix for scripts
  - No significant difference

```
HEAP           IOT
--------       --------
3.232323       2.452359
5.254339       2.992034
2.512612       2.549905
2.446256       7.430977
8.574844       7.846481
6.486806       2.385829
8.486466       3.863179
2.689221       8.623375
... +100 times

HEAP
----
       AVG        MIN        MAX
---------- ---------- ----------
3.48107223   2.253257  11.321287

IOT
       AVG        MIN        MAX
---------- ---------- ----------
3.83098879   2.215589  10.667338
```



Day: Wednesday - January 03, 2018   Time: 1:00PM to 2:00PM

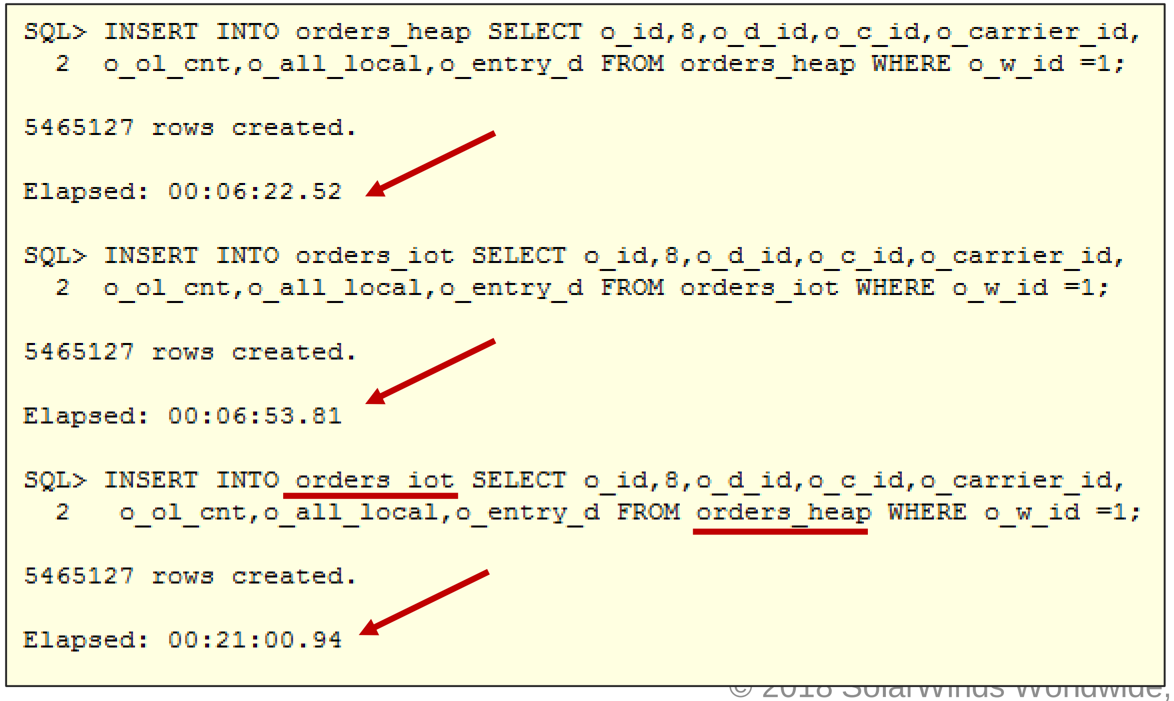
Top SQL Statements | DPA_GRID12C | January 3, 2018 - 1:00PM to 2:00PM

- ■ Memory/CPU
- ■ latch: shared pool
- ■ log file switch (checkpoint incc
- ■ latch: cache buffers lru chain
- ■ log file switch (private strand
- ■ log file switch completion
- ■ DML lock allocation
- ■ latch: In memory undo latch
- ■ latch: enqueue hash chains
- ■ latch: row cache objects
- ■ latch: call allocation
- ■ db file sequential read
- ■ kksfbc child completion
- ■ library cache load lock
- ■ cursor: mutex X
- ■ latch: redo allocation
- ■ cursor: pin S wait on X
- ■ local write wait
- ■ reliable message
- ■ enq: RO - fast object reuse
- ■ db file parallel read

DML Performance on IOTs

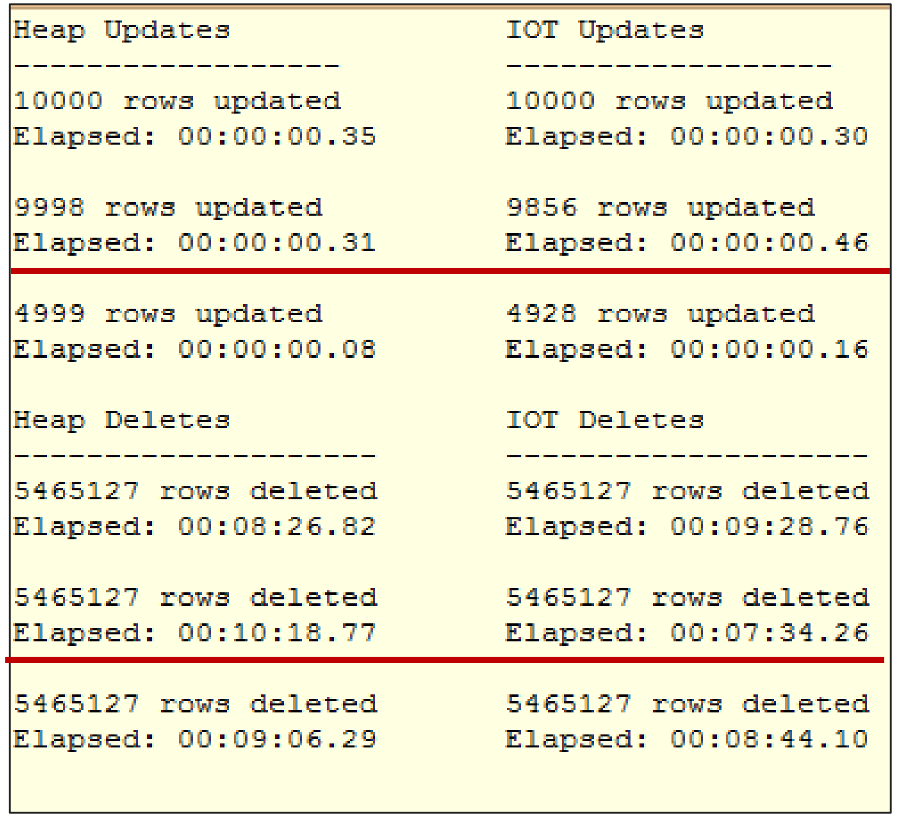
solarwinds

- Bulk inserts
  - New warehouse values added to:
    - Orders_heap from orders_heap
      - Using ‘insert into table select …’
    - Orders_iot from orders_iot
    - Orders_iot from orders_heap
      - After starting over
        - .i.e. Delete, move and rebuild indexes

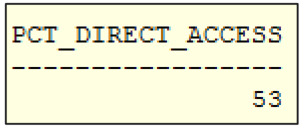


```
SQL> INSERT INTO orders_heap SELECT o_id,8,o_d_id,o_c_id,o_carrier_id,
  2  o_ol_cnt,o_all_local,o_entry_d FROM orders_heap WHERE o_w_id =1;

5465127 rows created.

Elapsed: 00:06:22.52

SQL> INSERT INTO orders_iot SELECT o_id,8,o_d_id,o_c_id,o_carrier_id,
  2  o_ol_cnt,o_all_local,o_entry_d FROM orders_iot WHERE o_w_id =1;

5465127 rows created.

Elapsed: 00:06:53.81

SQL> INSERT INTO orders_iot SELECT o_id,8,o_d_id,o_c_id,o_carrier_id,
  2   o_ol_cnt,o_all_local,o_entry_d FROM orders_heap WHERE o_w_id =1;

5465127 rows created.

Elapsed: 00:21:00.94
```

- Update and Delete Performance



```
Heap Updates                IOT Updates
------------------          ------------------
10000 rows updated          10000 rows updated
Elapsed: 00:00:00.35        Elapsed: 00:00:00.30

9998 rows updated           9856 rows updated
Elapsed: 00:00:00.31        Elapsed: 00:00:00.46

4999 rows updated           4928 rows updated
Elapsed: 00:00:00.08        Elapsed: 00:00:00.16

Heap Deletes                IOT Deletes
--------------------        --------------------
5465127 rows deleted        5465127 rows deleted
Elapsed: 00:08:26.82        Elapsed: 00:09:28.76

5465127 rows deleted        5465127 rows deleted
Elapsed: 00:10:18.77        Elapsed: 00:07:34.26

5465127 rows deleted        5465127 rows deleted
Elapsed: 00:09:06.29        Elapsed: 00:08:44.10
```

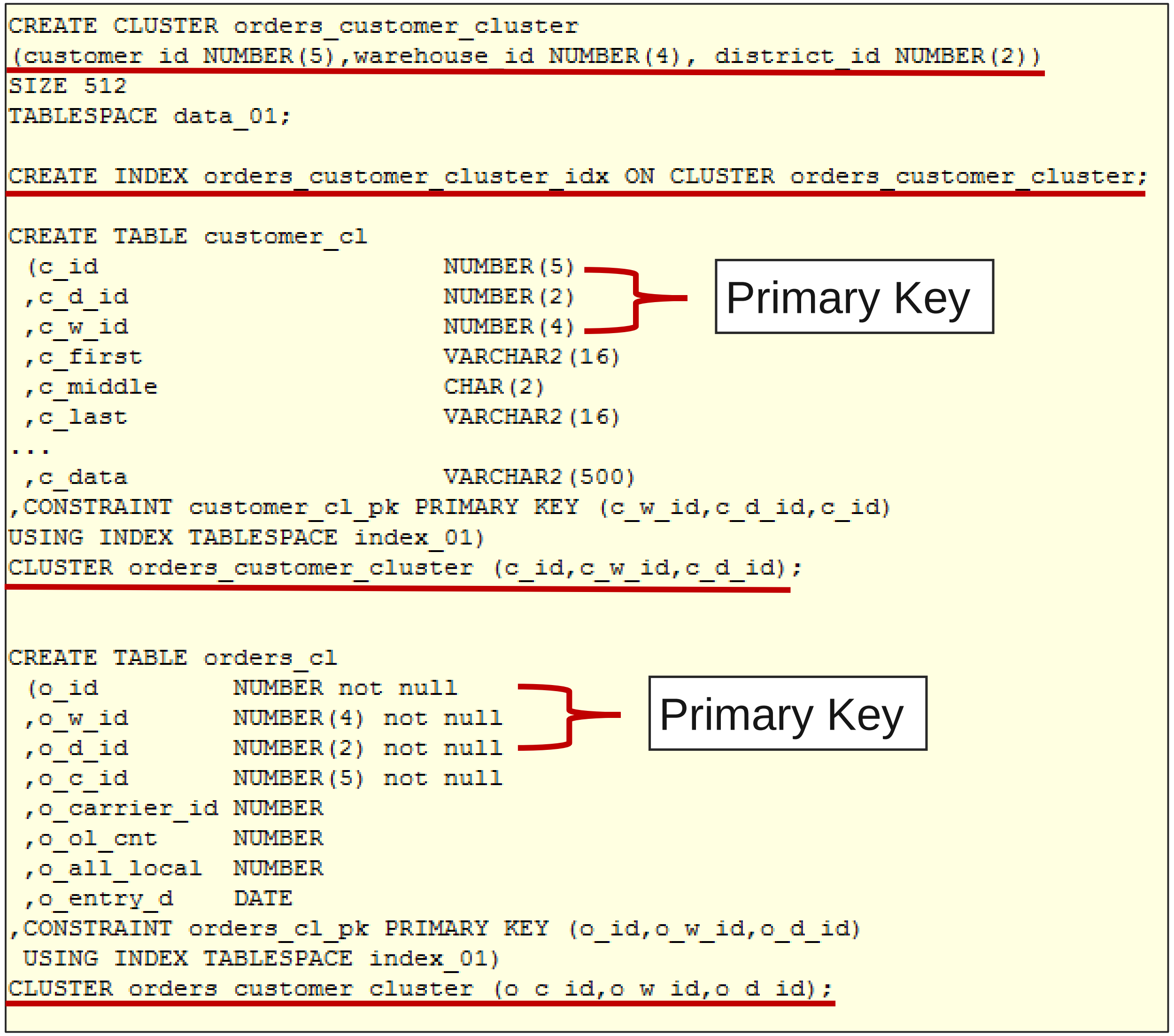


```
PCT_DIRECT_ACCESS
-----------------
               53
```

Final B-Tree Index Sub-type – Cluster Indexes

- Need to first understand Oracle clusters
  - A method for storing more then one related table in the same block
    - E.g. EMP and DEPT tables could be clustered on DEPTNO
      - Data for both tables stored in same block
  - Related tables benefit from:
    - Less disk I/O for joins
    - Less storage as cluster key values only stored once
    - Faster access for related tables
  - Clusters aren’t good if:
    - Tables are updated frequently
      - Especially if the updates occur on cluster key as data must move
    - Data takes up more than one or two blocks
      - Cluster key points to first cluster block
    - Tables need to be truncated
    - Full single table scans happen frequently

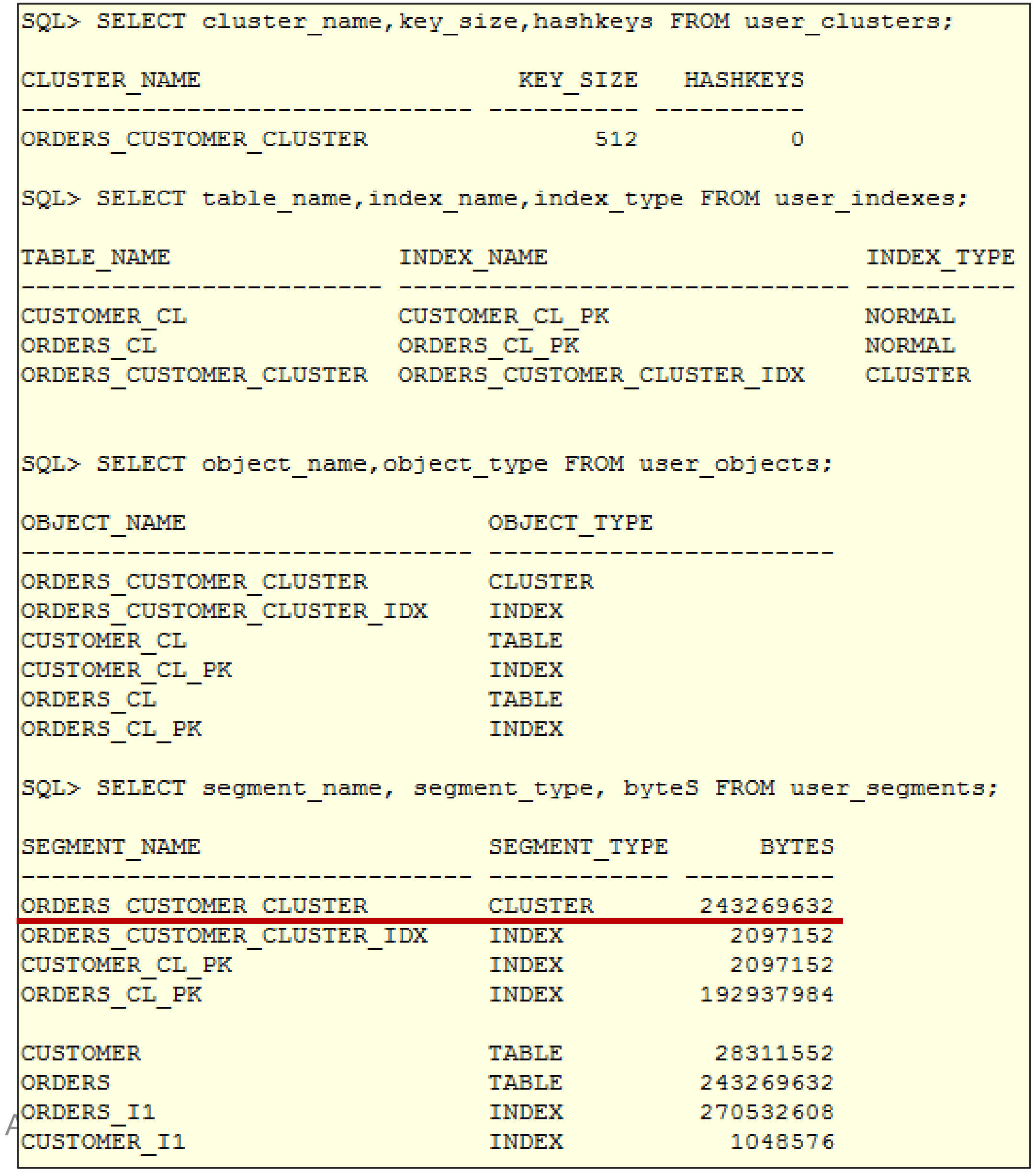
B-Tree Cluster Index Example



```
CREATE CLUSTER orders_customer_cluster
(customer id NUMBER(5),warehouse id NUMBER(4), district_id NUMBER(2))
SIZE 512
TABLESPACE data_01;

CREATE INDEX orders_customer_cluster_idx ON CLUSTER orders_customer_cluster;

CREATE TABLE customer_cl
 (c_id                         NUMBER(5)
 ,c_d_id                       NUMBER(2)
 ,c_w_id                       NUMBER(4)
 ,c_first                      VARCHAR2(16)
 ,c_middle                     CHAR(2)
 ,c_last                       VARCHAR2(16)
...
 ,c_data                       VARCHAR2(500)
,CONSTRAINT customer_cl_pk PRIMARY KEY (c_w_id,c_d_id,c_id)
USING INDEX TABLESPACE index_01)
CLUSTER orders_customer_cluster (c_id,c_w_id,c_d_id);


CREATE TABLE orders_cl
 (o_id          NUMBER not null
 ,o_w_id        NUMBER(4) not null
 ,o_d_id        NUMBER(2) not null
 ,o_c_id        NUMBER(5) not null
 ,o_carrier_id NUMBER
 ,o_ol_cnt      NUMBER
 ,o_all_local   NUMBER
 ,o_entry_d     DATE
,CONSTRAINT orders_cl_pk PRIMARY KEY (o_id,o_w_id,o_d_id)
 USING INDEX TABLESPACE index_01)
CLUSTER orders customer cluster (o c id,o w id,o d id);
```

Primary Key

Primary Key



```
SQL> SELECT cluster_name,key_size,hashkeys FROM user_clusters;

CLUSTER_NAME                     KEY_SIZE   HASHKEYS
------------------------------ ---------- ----------
ORDERS_CUSTOMER_CLUSTER               512          0

SQL> SELECT table_name,index_name,index_type FROM user_indexes;

TABLE_NAME                 INDEX_NAME                      INDEX_TYPE
-------------------------- ------------------------------ ----------
CUSTOMER_CL                CUSTOMER_CL_PK                  NORMAL
ORDERS_CL                  ORDERS_CL_PK                    NORMAL
ORDERS_CUSTOMER_CLUSTER    ORDERS_CUSTOMER_CLUSTER_IDX     CLUSTER


SQL> SELECT object_name,object_type FROM user_objects;

OBJECT_NAME                     OBJECT_TYPE
------------------------------ -----------------------
ORDERS_CUSTOMER_CLUSTER         CLUSTER
ORDERS_CUSTOMER_CLUSTER_IDX     INDEX
CUSTOMER_CL                     TABLE
CUSTOMER_CL_PK                  INDEX
ORDERS_CL                       TABLE
ORDERS_CL_PK                    INDEX

SQL> SELECT segment_name, segment_type, byteS FROM user_segments;

SEGMENT_NAME                    SEGMENT_TYPE      BYTES
------------------------------ ------------ ----------
ORDERS CUSTOMER CLUSTER         CLUSTER       243269632
ORDERS_CUSTOMER_CLUSTER_IDX     INDEX           2097152
CUSTOMER_CL_PK                  INDEX           2097152
ORDERS_CL_PK                    INDEX         192937984

CUSTOMER                        TABLE          28311552
ORDERS                          TABLE         243269632
ORDERS_I1                       INDEX         270532608
CUSTOMER_I1                     INDEX           1048576
```

B-Tree Cluster Index Example

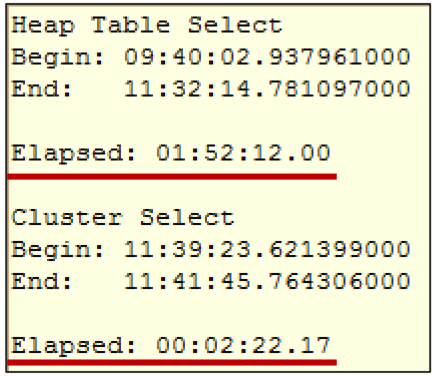
Select Customer Order Summary by State

- Heap table with PK indexes

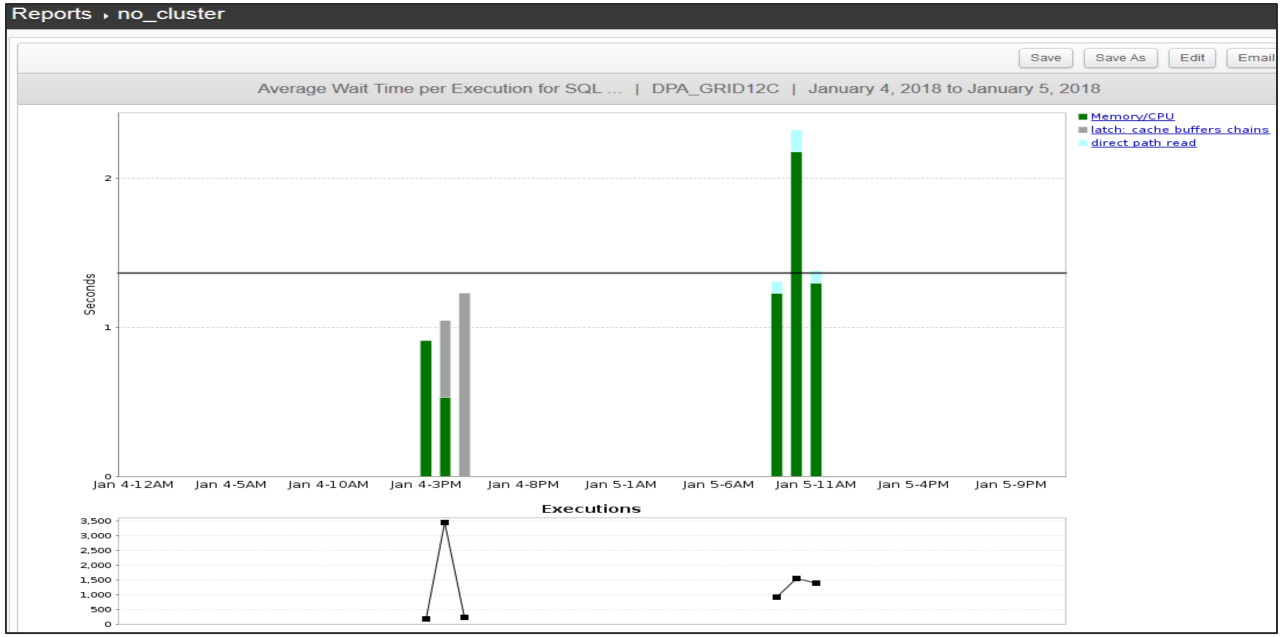
```
SELECT c_first, c_last, c_phone, c_balance,
               o_id, o_entry_d, o_ol_cnt
FROM customer, orders
WHERE c_id = o_c_id
AND c_w_id = o_w_id
AND c_d_id = o_d_id
AND c_state = :b1;
```

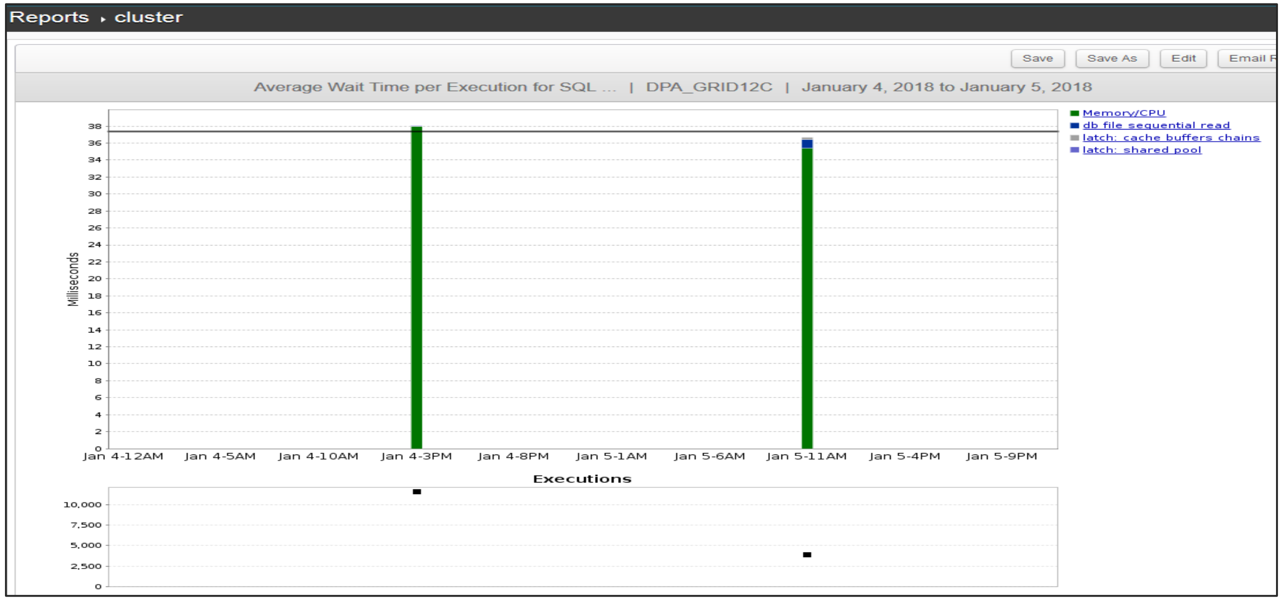
- Cluster with PKs and cluster index

```
SELECT c_first, c_last, c_phone, c_balance,
               o_id, o_entry_d, o_ol_cnt
FROM customer_cl, orders_cl
WHERE c_id = o_c_id
AND c_w_id = o_w_id
AND c_d_id = o_d_id
AND c_state = :b1;
```



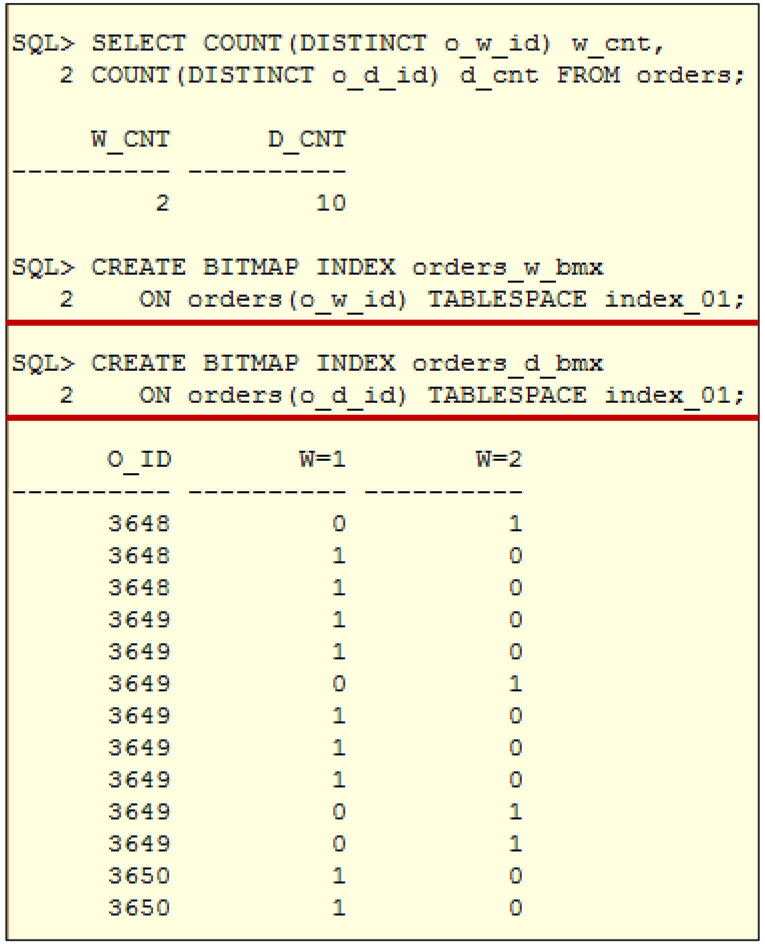
```
Heap Table Select
Begin:  09:40:02.937961000
End:    11:32:14.781097000

Elapsed: 01:52:12.00

Cluster Select
Begin:  11:39:23.621399000
End:    11:41:45.764306000

Elapsed: 00:02:22.17
```



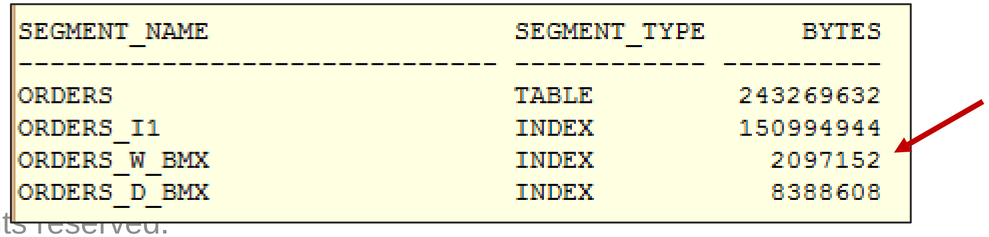


Bitmap and Bitmap Join Indexes

- A bitmap index points to multiple rows
  - Where a B-Tree index points to a single row
  - Bitmap join index is a bitmap index for joining tables
- Good for:
  - Data warehouse applications
    - Where queries access many columns in ad hoc fashion
  - Indexed columns which often have low cardinality
    - E.g. cust_gender column contains ‘M’ or ‘F’
  - Tables that are read-only or not significantly modified
    - Usually aren’t used in OLTP applications
      - Due to locking
- Easier to drop and recreate than maintain
- Can’t be used as a primary key



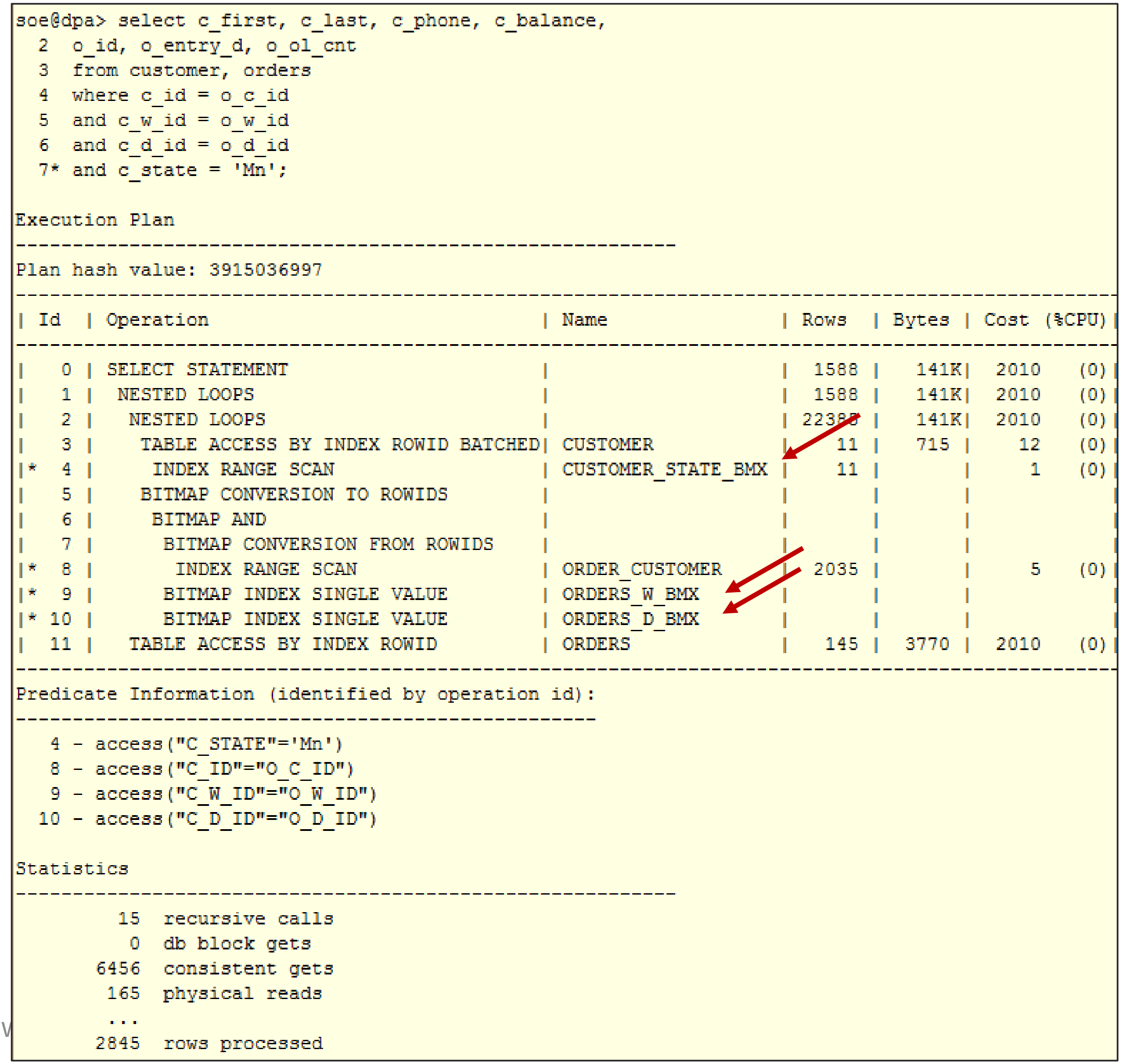
```
SQL> SELECT COUNT(DISTINCT o_w_id) w_cnt,
  2 COUNT(DISTINCT o_d_id) d_cnt FROM orders;

     W_CNT      D_CNT
---------- ----------
         2         10

SQL> CREATE BITMAP INDEX orders_w_bmx
  2     ON orders(o_w_id) TABLESPACE index_01;

SQL> CREATE BITMAP INDEX orders_d_bmx
  2     ON orders(o_d_id) TABLESPACE index_01;

      O_ID        W=1        W=2
---------- ---------- ----------
      3648          0          1
      3648          1          0
      3648          1          0
      3649          1          0
      3649          1          0
      3649          0          1
      3649          1          0
      3649          1          0
      3649          1          0
      3649          0          1
      3649          0          1
      3650          1          0
      3650          1          0
```



```
SEGMENT_NAME                   SEGMENT_TYPE       BYTES
------------------------------ ------------ ----------
ORDERS                         TABLE         243269632
ORDERS_I1                      INDEX         150994944
ORDERS_W_BMX                   INDEX           2097152
ORDERS_D_BMX                   INDEX           8388608
```

Bitmap Indexes

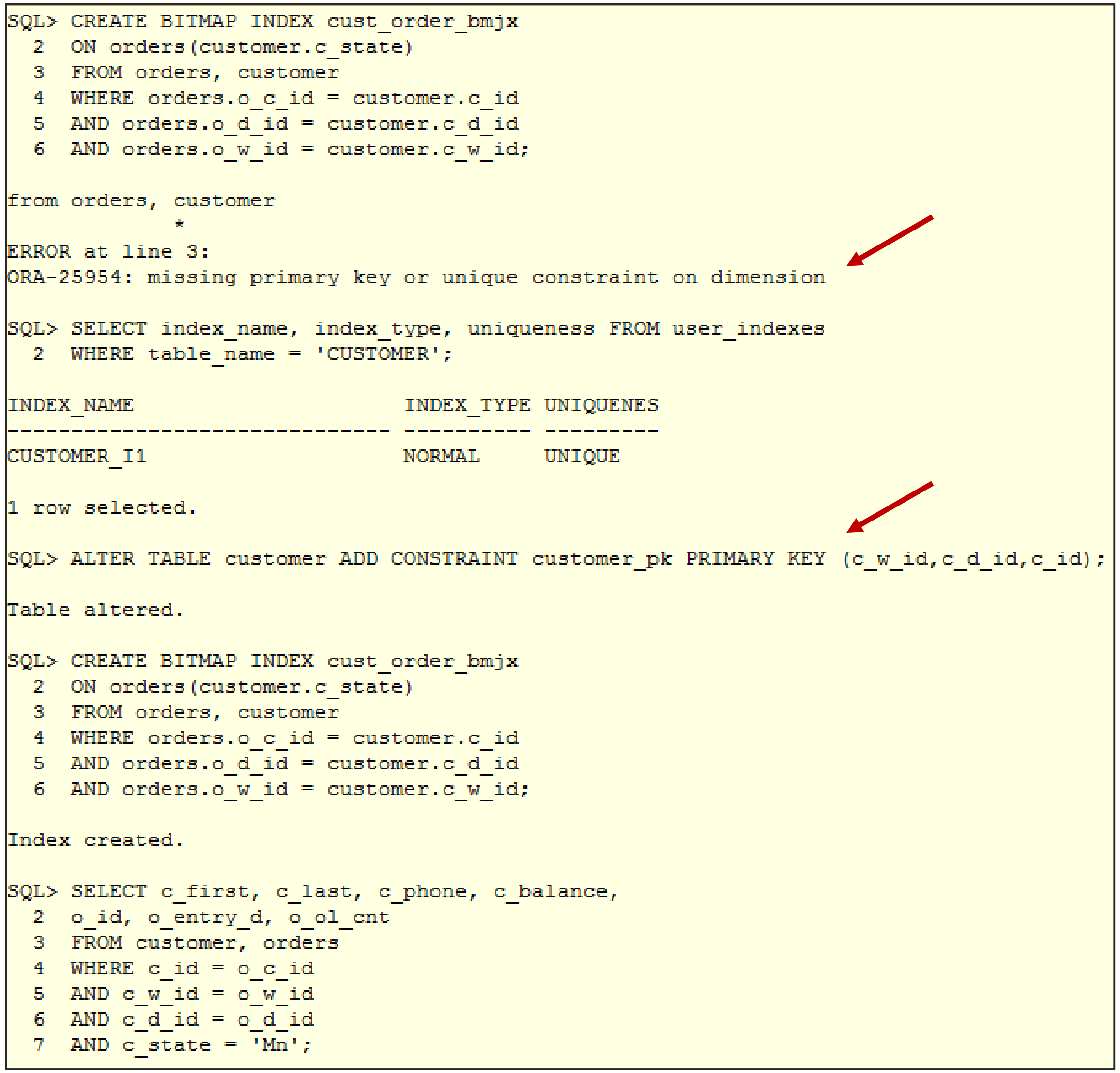
- Stored in B-Tree format
  - Same branch pointing to leaf
  - Leaf block contains
    - Column value, starting and ending rowids
    - Plus a series of bits
      - If ‘1’ the row contains the value,
      - If ‘0’ the row doesn’t have the value
- Can store null values
  - Unlike B-Tree indexes
    - Where null aren’t allowed
  - Useful with count operations



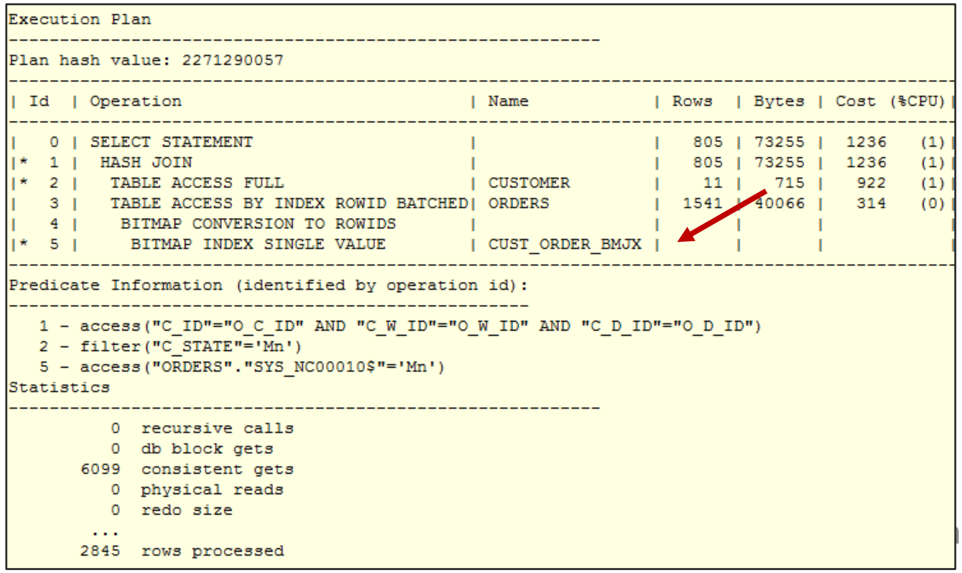
```
soe@dpa> select c_first, c_last, c_phone, c_balance,
  2  o_id, o_entry_d, o_ol_cnt
  3  from customer, orders
  4  where c_id = o_c_id
  5  and c_w_id = o_w_id
  6  and c_d_id = o_d_id
  7* and c_state = 'Mn';

Execution Plan
----------------------------------------------------------
Plan hash value: 3915036997

-------------------------------------------------------------------------------------------------
| Id  | Operation                              | Name                | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                       |                     |  1588 |   141K|  2010   (0)|
|   1 |  NESTED LOOPS                          |                     |  1588 |   141K|  2010   (0)|
|   2 |   NESTED LOOPS                         |                     | 22385 |   141K|  2010   (0)|
|   3 |    TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMER            |    11 |   715 |    12   (0)|
|*  4 |     INDEX RANGE SCAN                   | CUSTOMER_STATE_BMX  |    11 |       |     1   (0)|
|   5 |    BITMAP CONVERSION TO ROWIDS         |                     |       |       |            |
|   6 |     BITMAP AND                         |                     |       |       |            |
|   7 |      BITMAP CONVERSION FROM ROWIDS     |                     |       |       |            |
|*  8 |       INDEX RANGE SCAN                 | ORDER_CUSTOMER      |  2035 |       |     5   (0)|
|*  9 |      BITMAP INDEX SINGLE VALUE         | ORDERS_W_BMX        |       |       |            |
|* 10 |      BITMAP INDEX SINGLE VALUE         | ORDERS_D_BMX        |       |       |            |
|  11 |   TABLE ACCESS BY INDEX ROWID          | ORDERS              |   145 |  3770 |  2010   (0)|
-------------------------------------------------------------------------------------------------

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

   4 - access("C_STATE"='Mn')
   8 - access("C_ID"="O_C_ID")
   9 - access("C_W_ID"="O_W_ID")
  10 - access("C_D_ID"="O_D_ID")

Statistics
----------------------------------------------------------
         15  recursive calls
          0  db block gets
       6456  consistent gets
        165  physical reads
        ...
       2845  rows processed
```

Bitmap Join Indexes

- Used in joining two or more tables
  - Equi-inner join
    - Between primary key and foreign key
    - Constraint must exist – index won't do
  - Columns of dimension tables and fact table
    - I.e. Star model
  - Alternative to materialized join views
    - Bitmap join indexes take up less space

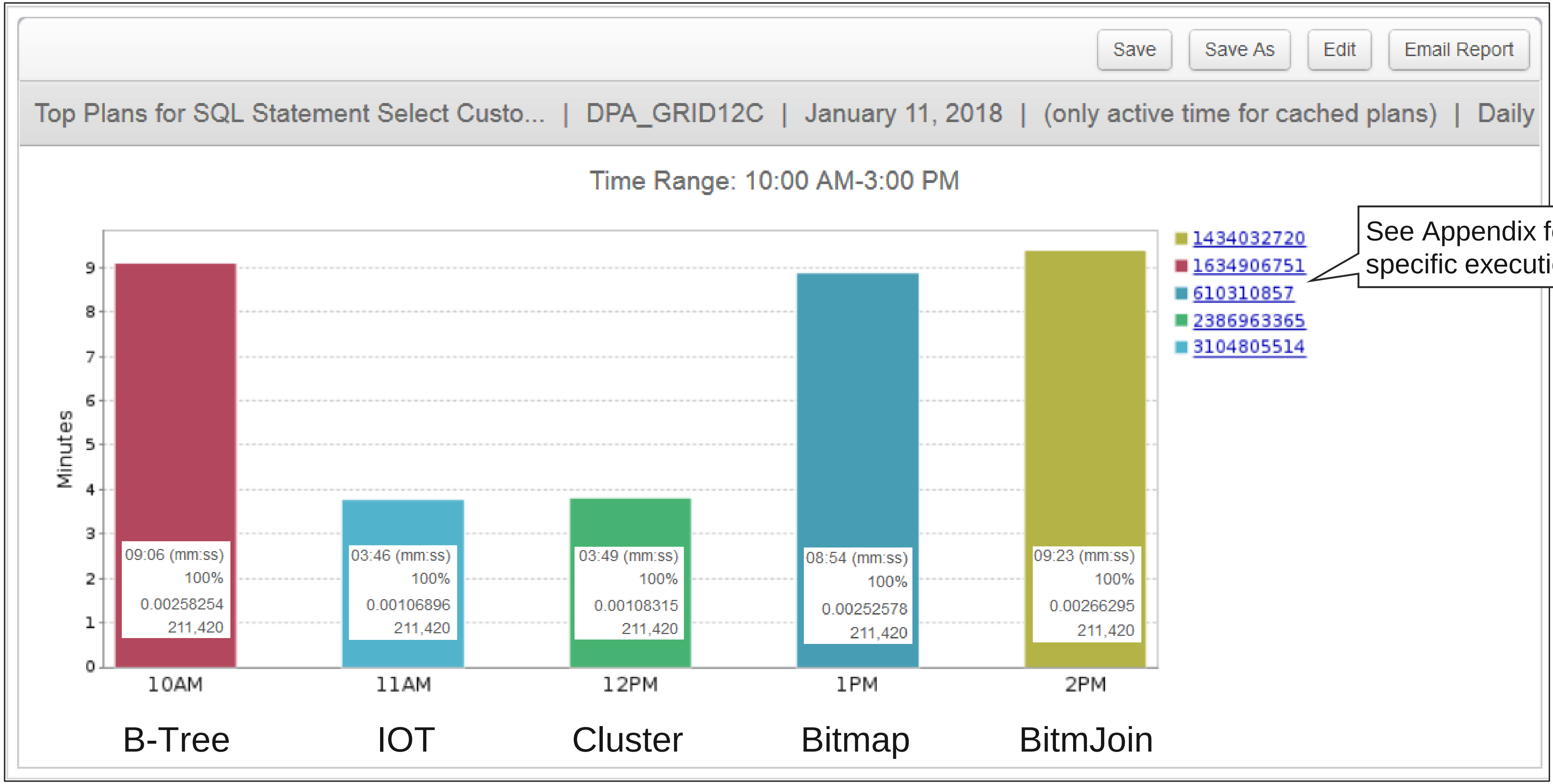


```
SQL> CREATE BITMAP INDEX cust_order_bmjx
  2  ON orders(customer.c_state)
  3  FROM orders, customer
  4  WHERE orders.o_c_id = customer.c_id
  5  AND orders.o_d_id = customer.c_d_id
  6  AND orders.o_w_id = customer.c_w_id;

from orders, customer
             *
ERROR at line 3:
ORA-25954: missing primary key or unique constraint on dimension

SQL> SELECT index_name, index_type, uniqueness FROM user_indexes
  2  WHERE table_name = 'CUSTOMER';

INDEX_NAME                     INDEX_TYPE UNIQUENES
------------------------------ ---------- ---------
CUSTOMER_I1                    NORMAL     UNIQUE

1 row selected.

SQL> ALTER TABLE customer ADD CONSTRAINT customer_pk PRIMARY KEY (c_w_id,c_d_id,c_id);

Table altered.

SQL> CREATE BITMAP INDEX cust_order_bmjx
  2  ON orders(customer.c_state)
  3  FROM orders, customer
  4  WHERE orders.o_c_id = customer.c_id
  5  AND orders.o_d_id = customer.c_d_id
  6  AND orders.o_w_id = customer.c_w_id;

Index created.

SQL> SELECT c_first, c_last, c_phone, c_balance,
  2  o_id, o_entry_d, o_ol_cnt
  3  FROM customer, orders
  4  WHERE c_id = o_c_id
  5  AND c_w_id = o_w_id
  6  AND c_d_id = o_d_id
  7  AND c_state = 'Mn';
```



```
Execution Plan
----------------------------------------------------------
Plan hash value: 2271290057

---------------------------------------------------------------------------------------------
| Id  | Operation                            | Name            | Rows  | Bytes | Cost (%CPU)|
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |                 |   805 | 73255 |  1236   (1)|
|*  1 |  HASH JOIN                           |                 |   805 | 73255 |  1236   (1)|
|*  2 |   TABLE ACCESS FULL                  | CUSTOMER        |    11 |   715 |   922   (1)|
|   3 |   TABLE ACCESS BY INDEX ROWID BATCHED| ORDERS          |  1541 | 40066 |   314   (0)|
|   4 |    BITMAP CONVERSION TO ROWIDS       |                 |       |       |            |
|*  5 |     BITMAP INDEX SINGLE VALUE        | CUST_ORDER_BMJX |       |       |            |
---------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("C_ID"="O_C_ID" AND "C_W_ID"="O_W_ID" AND "C_D_ID"="O_D_ID")
   2 - filter("C_STATE"='Mn')
   5 - access("ORDERS"."SYS_NC00010$"='Mn')
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       6099  consistent gets
          0  physical reads
          0  redo size
        ...
       2845  rows processed
```

Case Study – Select Customer Order Summary by State



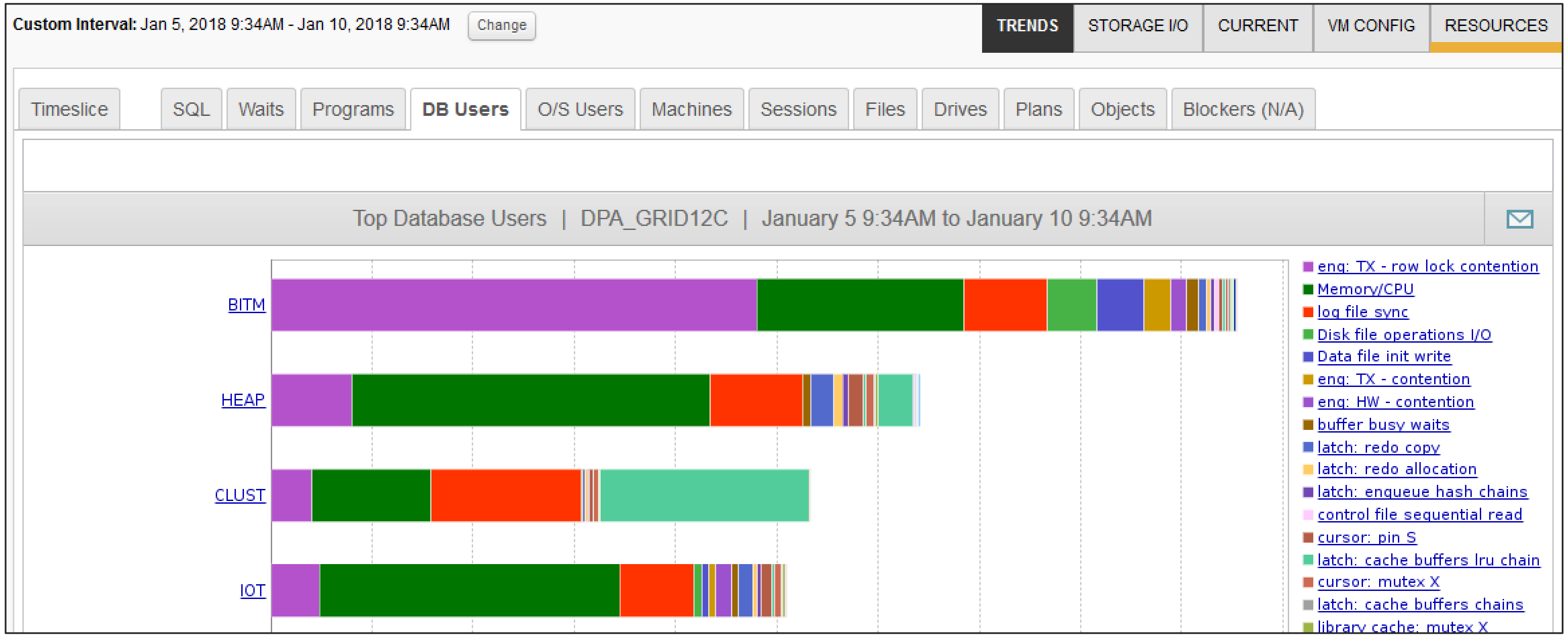
Top Plans for SQL Statement Select Custo... | DPA_GRID12C | January 11, 2018 | (only active time for cached plans) | Daily

Time Range: 10:00 AM-3:00 PM

See Appendix for specific execution plans

Case Study

- HammerDB testing B-Tree, IOT, table cluster, and bitmap indexes
  - Configured two warehouses in 4 different schemas
    - Five virtual users
      - One million transactions each
      - Mix of inserts, updates and selects



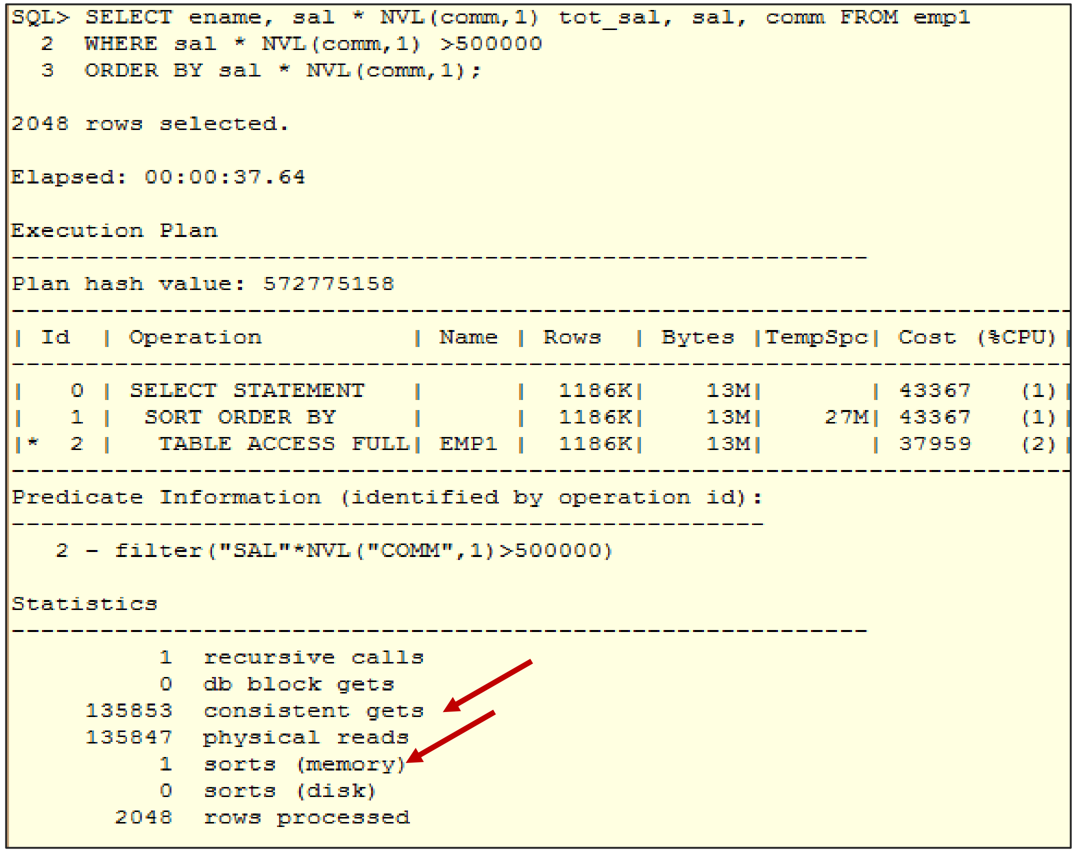
Custom Interval: Jan 5, 2018 9:34AM - Jan 10, 2018 9:34AM

IOT	= 28:23
CLUST	= 29:40
B-TREE	= 35:24
BITMAP	= 53:09

Function-Based Indexes

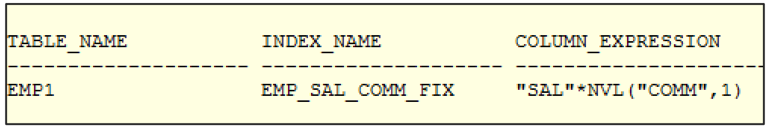
- Can be created for columns using function or expressions
  - E.g. UPPER(last_name)
    - Will turn off an index on last_name column
    - Need to create index on UPPER(last_name)
  - Created as a virtual column
- Can be a B-Tree or bitmap index
  - Unique or non-unique
- Index must return not null values
  - Columns need ‘not null’ constraint or use NVL when creating index
- Useful when sorting by function or expression
- Can’t use with ‘or’ expressions or aggregate functions
- Other restrictions for PL/SQL
  - https://docs.oracle.com/cd/E11882_01/appdev.112/e41502/adfns_indexes.htm#ADFNS257

Function-Based Indexes Example

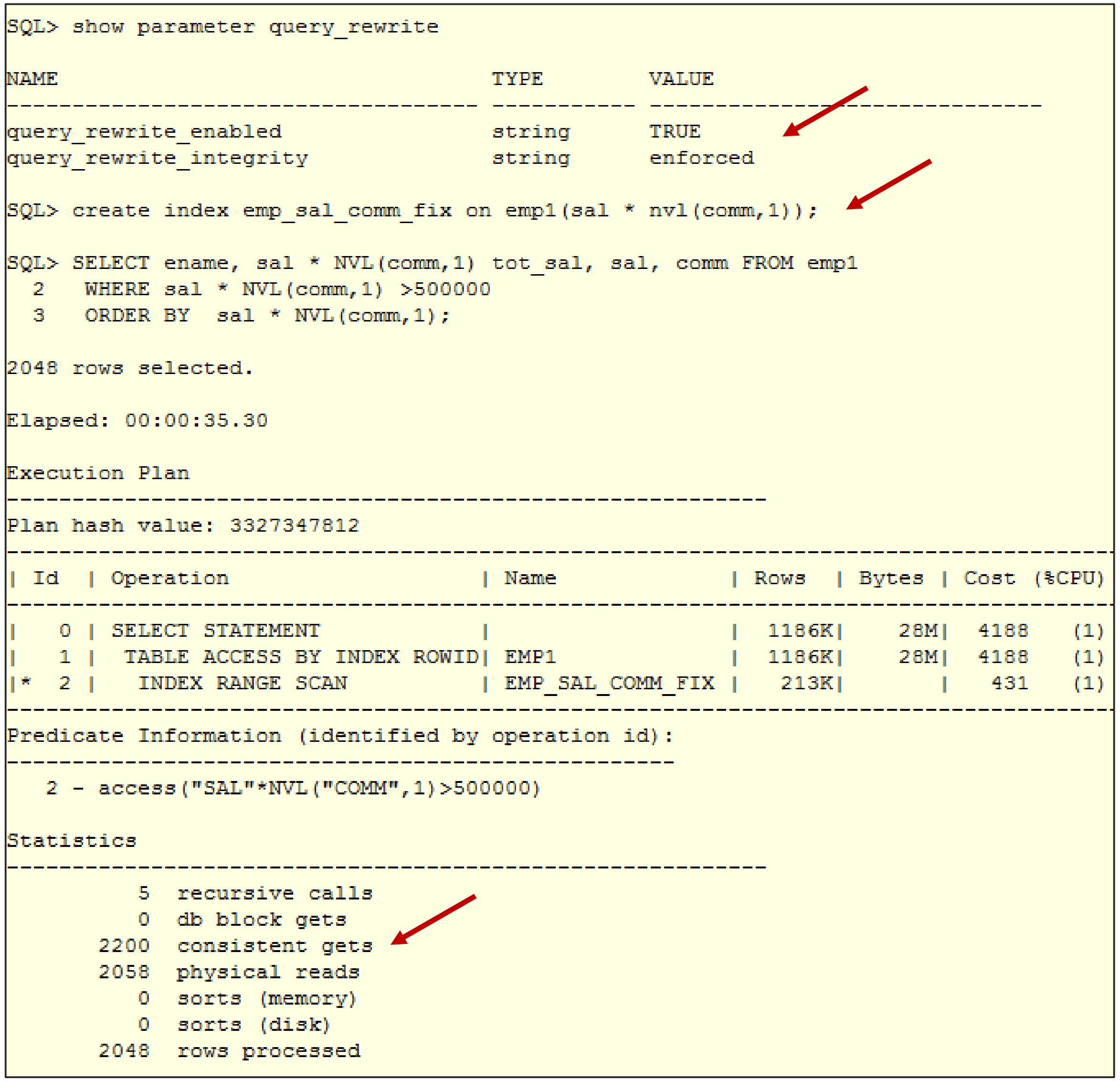


```
SQL> SELECT ename, sal * NVL(comm,1) tot_sal, sal, comm FROM emp1
  2  WHERE sal * NVL(comm,1) >500000
  3  ORDER BY sal * NVL(comm,1);

2048 rows selected.

Elapsed: 00:00:37.64

Execution Plan
----------------------------------------------------------
Plan hash value: 572775158

------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes |TempSpc| Cost (%CPU)|
------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |  1186K|    13M|       | 43367   (1)|
|   1 |  SORT ORDER BY     |      |  1186K|    13M|    27M| 43367   (1)|
|*  2 |   TABLE ACCESS FULL| EMP1 |  1186K|    13M|       | 37959   (2)|
------------------------------------------------------------------------

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

   2 - filter("SAL"*NVL("COMM",1)>500000)

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
     135853  consistent gets
     135847  physical reads
          1  sorts (memory)
          0  sorts (disk)
       2048  rows processed
```

SELECT table_name, index_name, column_expression
FROM dba_ind_expressions
WHERE table_name = 'EMP1'
AND table_owner = 'SOE'
ORDER BY index_name, column_position;



```
TABLE_NAME          INDEX_NAME          COLUMN_EXPRESSION
------------------- ------------------- -------------------
EMP1                EMP_SAL_COMM_FIX    "SAL"*NVL("COMM",1)
```



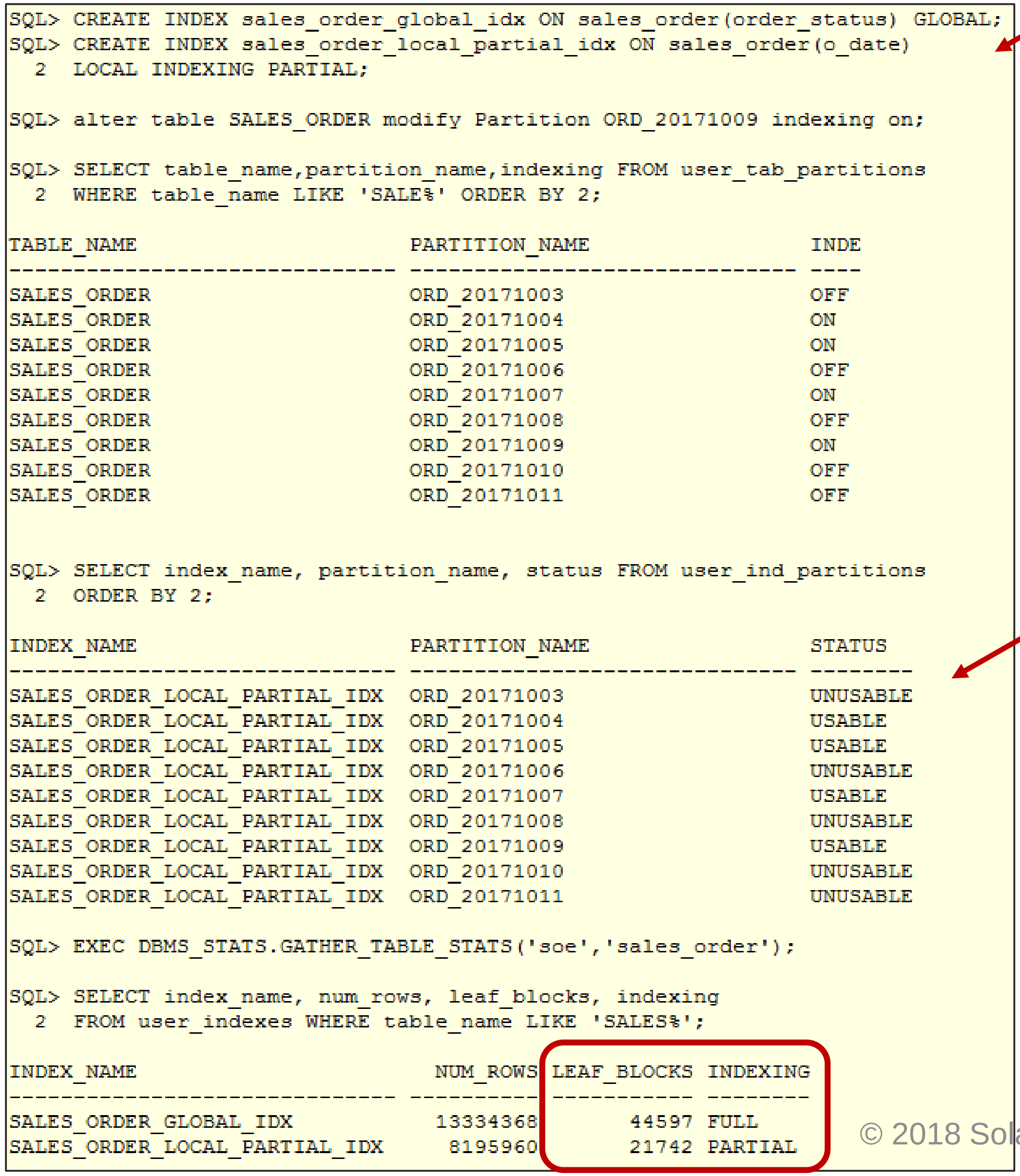
```
SQL> show parameter query_rewrite

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
query_rewrite_enabled                string      TRUE
query_rewrite_integrity              string      enforced

SQL> create index emp_sal_comm_fix on emp1(sal * nvl(comm,1));

SQL> SELECT ename, sal * NVL(comm,1) tot_sal, sal, comm FROM emp1
  2   WHERE sal * NVL(comm,1) >500000
  3   ORDER BY  sal * NVL(comm,1);

2048 rows selected.

Elapsed: 00:00:35.30

Execution Plan
----------------------------------------------------------
Plan hash value: 3327347812

-------------------------------------------------------------------------------------------
| Id  | Operation                   | Name             | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |                  |  1186K|    28M|  4188   (1)|
|   1 |  TABLE ACCESS BY INDEX ROWID| EMP1             |  1186K|    28M|  4188   (1)|
|*  2 |   INDEX RANGE SCAN          | EMP_SAL_COMM_FIX |   213K|       |   431   (1)|
-------------------------------------------------------------------------------------------

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

   2 - access("SAL"*NVL("COMM",1)>500000)

Statistics
----------------------------------------------------------
          5  recursive calls
          0  db block gets
       2200  consistent gets
       2058  physical reads
          0  sorts (memory)
          0  sorts (disk)
       2048  rows processed
```

New 12.2 Index Features

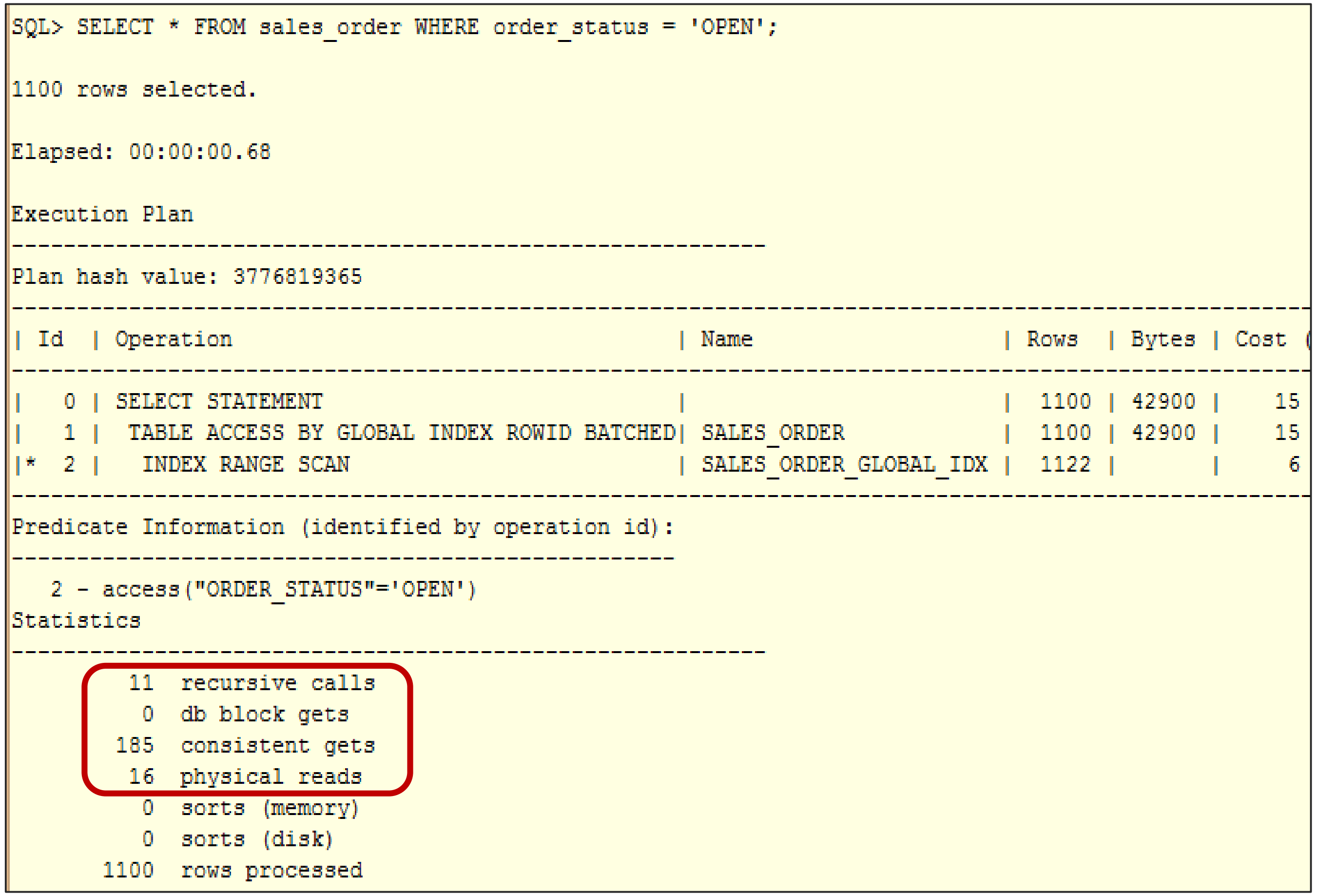
- Partial Indexes for Partitioned Tables
    - Local indexes – index partition is created usable or unusable
    - Global indexes – include only those partitions where indexing is turned on
    - Unique index can’t be a partial index
    - The ‘INDEXING’ clause determines how the partition is to be indexed
        - It can be set at table level , individual partition and subpartition level
        - Default setting is at table level
            - Partition level overrides table level

```
CREATE TABLE sales_order (o_id number, o_w_id number, o_d_id number, o_c_id number, o_carrier_id number, o_ol_cnt
number, o_date date, order_status varchar2(10))
INDEXING OFF
PARTITION BY RANGE (o_date)
(PARTITION ord_20181003 VALUES LESS THAN (TO_DATE('03-OCT-2018','DD-MON-YYYY')),
PARTITION ord_20181004 VALUES LESS THAN (TO_DATE('04-OCT-2018','DD-MON-YYYY')) INDEXING ON,
PARTITION ord_20181005 VALUES LESS THAN (TO_DATE('05-OCT-2018','DD-MON-YYYY')) INDEXING ON,
PARTITION ord_20181006 VALUES LESS THAN (TO_DATE('06-OCT-2018','DD-MON-YYYY')) INDEXING OFF,
PARTITION ord_20181007 VALUES LESS THAN (TO_DATE('07-OCT-2018','DD-MON-YYYY')) INDEXING ON,
PARTITION ord_20181008 VALUES LESS THAN (TO_DATE('08-OCT-2018','DD-MON-YYYY')));
```

Partial Indexes for Partitioned Tables

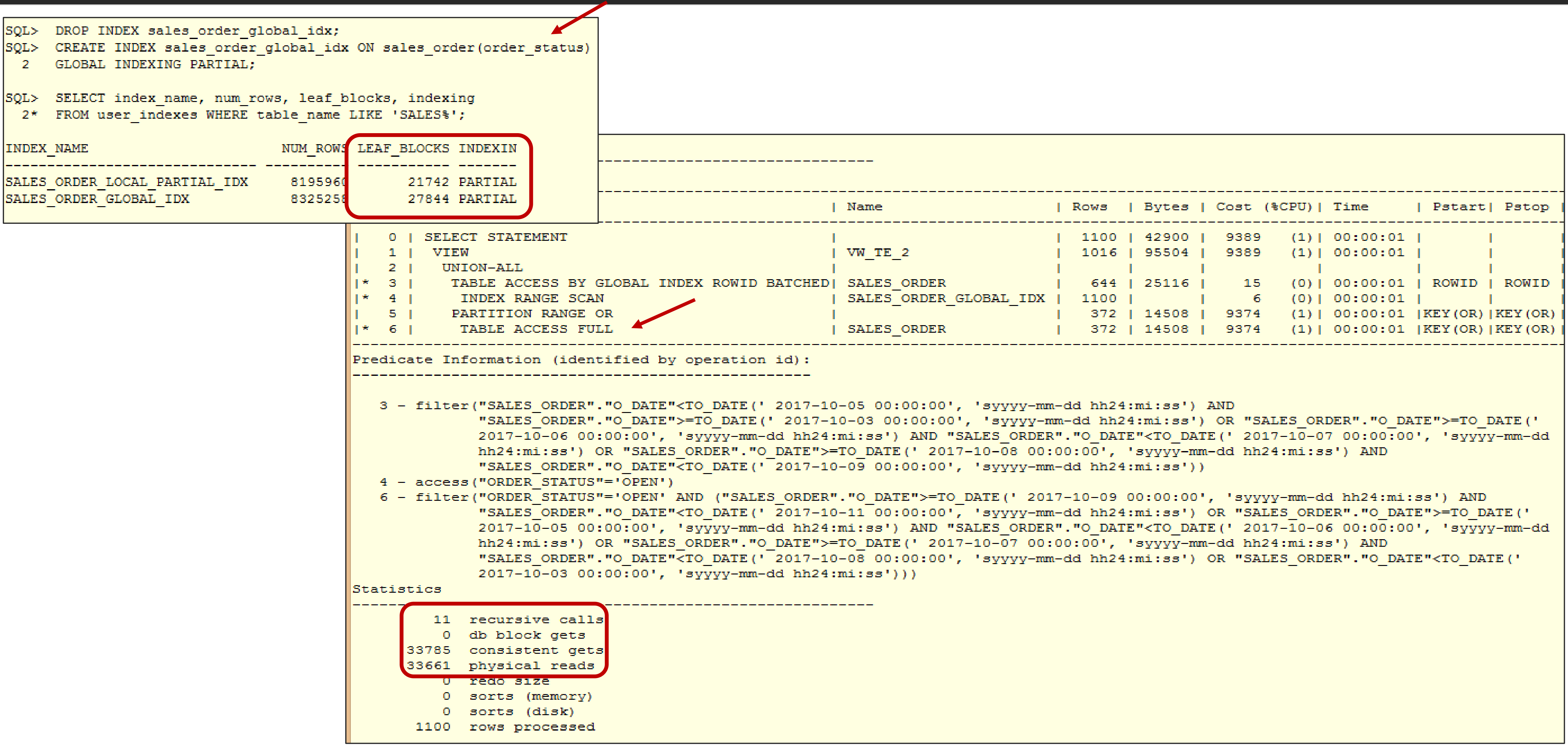


```
SQL> CREATE INDEX sales_order_global_idx ON sales_order(order_status) GLOBAL;
SQL> CREATE INDEX sales_order_local_partial_idx ON sales_order(o_date)
  2  LOCAL INDEXING PARTIAL;

SQL> alter table SALES_ORDER modify Partition ORD_20171009 indexing on;

SQL> SELECT table_name,partition_name,indexing FROM user_tab_partitions
  2  WHERE table_name LIKE 'SALE%' ORDER BY 2;

TABLE_NAME                     PARTITION_NAME                 INDE
------------------------------ ------------------------------ ----
SALES_ORDER                    ORD_20171003                   OFF
SALES_ORDER                    ORD_20171004                   ON
SALES_ORDER                    ORD_20171005                   ON
SALES_ORDER                    ORD_20171006                   OFF
SALES_ORDER                    ORD_20171007                   ON
SALES_ORDER                    ORD_20171008                   OFF
SALES_ORDER                    ORD_20171009                   ON
SALES_ORDER                    ORD_20171010                   OFF
SALES_ORDER                    ORD_20171011                   OFF


SQL> SELECT index_name, partition_name, status FROM user_ind_partitions
  2  ORDER BY 2;

INDEX_NAME                     PARTITION_NAME                 STATUS
------------------------------ ------------------------------ --------
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171003                   UNUSABLE
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171004                   USABLE
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171005                   USABLE
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171006                   UNUSABLE
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171007                   USABLE
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171008                   UNUSABLE
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171009                   USABLE
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171010                   UNUSABLE
SALES_ORDER_LOCAL_PARTIAL_IDX  ORD_20171011                   UNUSABLE

SQL> EXEC DBMS_STATS.GATHER_TABLE_STATS('soe','sales_order');

SQL> SELECT index_name, num_rows, leaf_blocks, indexing
  2  FROM user_indexes WHERE table_name LIKE 'SALES%';

INDEX_NAME                       NUM_ROWS LEAF_BLOCKS INDEXING
------------------------------ ---------- ----------- --------
SALES_ORDER_GLOBAL_IDX           13334368       44597 FULL
SALES_ORDER_LOCAL_PARTIAL_IDX     8195960       21742 PARTIAL
```



```
SQL> SELECT * FROM sales_order WHERE order_status = 'OPEN';

1100 rows selected.

Elapsed: 00:00:00.68

Execution Plan
----------------------------------------------------------
Plan hash value: 3776819365

------------------------------------------------------------------------------------------------
| Id  | Operation                            | Name                   | Rows  | Bytes | Cost (
------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |                        |  1100 | 42900 |    15
|   1 |  TABLE ACCESS BY GLOBAL INDEX ROWID BATCHED| SALES_ORDER      |  1100 | 42900 |    15
|*  2 |   INDEX RANGE SCAN                   | SALES_ORDER_GLOBAL_IDX |  1122 |       |     6
------------------------------------------------------------------------------------------------

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

   2 - access("ORDER_STATUS"='OPEN')
Statistics
----------------------------------------------------------
         11  recursive calls
          0  db block gets
        185  consistent gets
         16  physical reads
          0  sorts (memory)
          0  sorts (disk)
       1100  rows processed
```

Partial Indexes for Partitioned Tables – Cont.

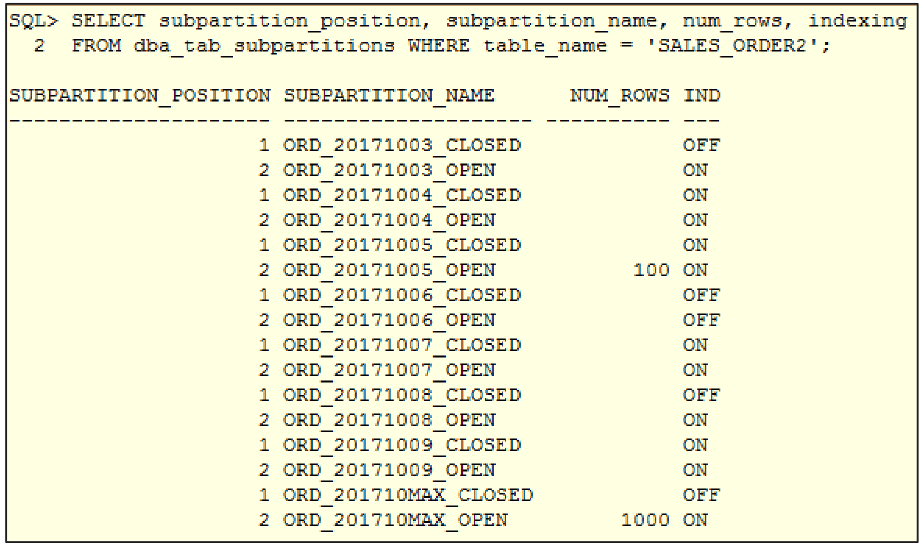


```
SQL>  DROP INDEX sales_order_global_idx;
SQL>  CREATE INDEX sales_order_global_idx ON sales_order(order_status)
  2   GLOBAL INDEXING PARTIAL;

SQL>  SELECT index_name, num_rows, leaf_blocks, indexing
  2*  FROM user_indexes WHERE table_name LIKE 'SALES%';

INDEX_NAME                        NUM_ROWS LEAF_BLOCKS INDEXIN
------------------------------ ---------- ----------- -------
SALES_ORDER_LOCAL_PARTIAL_IDX      8195960       21742 PARTIAL
SALES_ORDER_GLOBAL_IDX             8325258       27844 PARTIAL
```

```
                                              ---------------------------------

                                              -------------------------------------------------------------------------------------------------------------
                                                                  | Name                      | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
------------------------------------------------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                                          |                           |  1100 | 42900 |  9389   (1)| 00:00:01 |       |       |
|   1 |  VIEW                                                     | VW_TE_2                   |  1016 | 95504 |  9389   (1)| 00:00:01 |       |       |
|   2 |   UNION-ALL                                               |                           |       |       |            |          |       |       |
|*  3 |    TABLE ACCESS BY GLOBAL INDEX ROWID BATCHED|               SALES_ORDER               |   644 | 25116 |    15   (0)| 00:00:01 | ROWID | ROWID |
|*  4 |     INDEX RANGE SCAN                                      | SALES_ORDER_GLOBAL_IDX    |  1100 |       |     6   (0)| 00:00:01 |       |       |
|   5 |    PARTITION RANGE OR                                     |                           |   372 | 14508 |  9374   (1)| 00:00:01 |KEY(OR)|KEY(OR)|
|*  6 |     TABLE ACCESS FULL                                     | SALES_ORDER               |   372 | 14508 |  9374   (1)| 00:00:01 |KEY(OR)|KEY(OR)|
------------------------------------------------------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("SALES_ORDER"."O_DATE"<TO_DATE(' 2017-10-05 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              "SALES_ORDER"."O_DATE">=TO_DATE(' 2017-10-03 00:00:00', 'syyyy-mm-dd hh24:mi:ss') OR "SALES_ORDER"."O_DATE">=TO_DATE('
              2017-10-06 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "SALES_ORDER"."O_DATE"<TO_DATE(' 2017-10-07 00:00:00', 'syyyy-mm-dd
              hh24:mi:ss') OR "SALES_ORDER"."O_DATE">=TO_DATE(' 2017-10-08 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              "SALES_ORDER"."O_DATE"<TO_DATE(' 2017-10-09 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
   4 - access("ORDER_STATUS"='OPEN')
   6 - filter("ORDER_STATUS"='OPEN' AND ("SALES_ORDER"."O_DATE">=TO_DATE(' 2017-10-09 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              "SALES_ORDER"."O_DATE"<TO_DATE(' 2017-10-11 00:00:00', 'syyyy-mm-dd hh24:mi:ss') OR "SALES_ORDER"."O_DATE">=TO_DATE('
              2017-10-05 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "SALES_ORDER"."O_DATE"<TO_DATE(' 2017-10-06 00:00:00', 'syyyy-mm-dd
              hh24:mi:ss') OR "SALES_ORDER"."O_DATE">=TO_DATE(' 2017-10-07 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
              "SALES_ORDER"."O_DATE"<TO_DATE(' 2017-10-08 00:00:00', 'syyyy-mm-dd hh24:mi:ss') OR "SALES_ORDER"."O_DATE"<TO_DATE('
              2017-10-03 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))
Statistics
----------------------------------------------------------
         11  recursive calls
          0  db block gets
      33785  consistent gets
      33661  physical reads
          0  redo size
          0  sorts (memory)
          0  sorts (disk)
       1100  rows processed
```

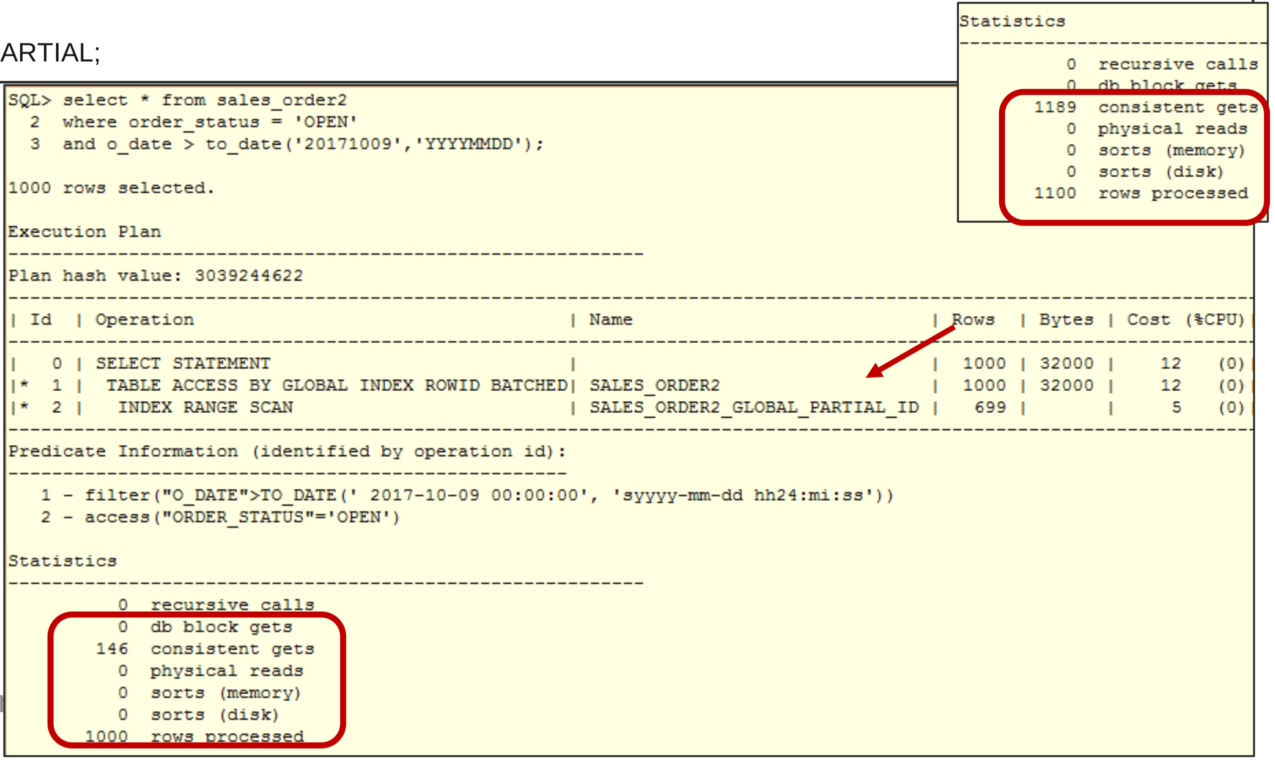
Even Better Partial Index Using Subpartition

```
CREATE TABLE sales_order2 (o_id number, o_w_id number, o_d_id number, o_c_id number, o_carrier_id number, o_ol_cnt number, o_date date, order_status varchar2(10))
INDEXING OFF
PARTITION BY RANGE (o_date) subpartition by list(order_status)
subpartition template (subpartition closed values ('COMPLETE') indexing off, subpartition open values ('OPEN') indexing on)
(PARTITION ord_20181003 VALUES LESS THAN (TO_DATE('03-OCT-2018','DD-MON-YYYY')),
PARTITION ord_20181004 VALUES LESS THAN (TO_DATE('04-OCT-2018','DD-MON-YYYY')) INDEXING ON,
PARTITION ord_20181005 VALUES LESS THAN (TO_DATE('05-OCT-2018','DD-MON-YYYY')) INDEXING ON,
PARTITION ord_20181006 VALUES LESS THAN (TO_DATE('06-OCT-2018','DD-MON-YYYY')) INDEXING OFF,
PARTITION ord_20181007 VALUES LESS THAN (TO_DATE('07-OCT-2018','DD-MON-YYYY')) INDEXING ON,
PARTITION ord_20181008 VALUES LESS THAN (TO_DATE('08-OCT-2018','DD-MON-YYYY')),
PARTITION ord_20181009 VALUES LESS THAN (TO_DATE('09-OCT-2018','DD-MON-YYYY')) INDEXING ON,
PARTITION ord_201810max VALUES LESS THAN (maxvalue))
enable row movement;

CREATE INDEX sales_order2_local_partial_idx ON sales_order2(o_date) LOCAL INDEXING PARTIAL;
CREATE INDEX sales_order2_global_partial_idx ON sales_order2(order_status) GLOBAL INDEXING PARTIAL;
```



```
SQL> SELECT subpartition_position, subpartition_name, num_rows, indexing
  2  FROM dba_tab_subpartitions WHERE table_name = 'SALES_ORDER2';

SUBPARTITION_POSITION SUBPARTITION_NAME      NUM_ROWS IND
--------------------- -------------------- ---------- ---
                    1 ORD_20171003_CLOSED             OFF
                    2 ORD_20171003_OPEN               ON
                    1 ORD_20171004_CLOSED             ON
                    2 ORD_20171004_OPEN               ON
                    1 ORD_20171005_CLOSED             ON
                    2 ORD_20171005_OPEN           100 ON
                    1 ORD_20171006_CLOSED             OFF
                    2 ORD_20171006_OPEN               OFF
                    1 ORD_20171007_CLOSED             ON
                    2 ORD_20171007_OPEN               ON
                    1 ORD_20171008_CLOSED             OFF
                    2 ORD_20171008_OPEN               ON
                    1 ORD_20171009_CLOSED             ON
                    2 ORD_20171009_OPEN               ON
                    1 ORD_201710MAX_CLOSED            OFF
                    2 ORD_201710MAX_OPEN         1000 ON
```

All partitons

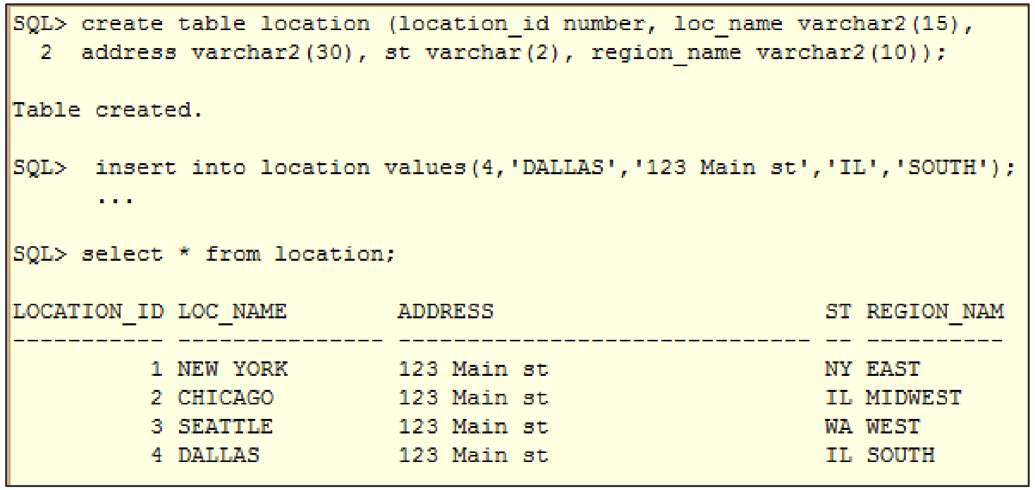


```
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       1189  consistent gets
          0  physical reads
          0  sorts (memory)
          0  sorts (disk)
       1100  rows processed
```

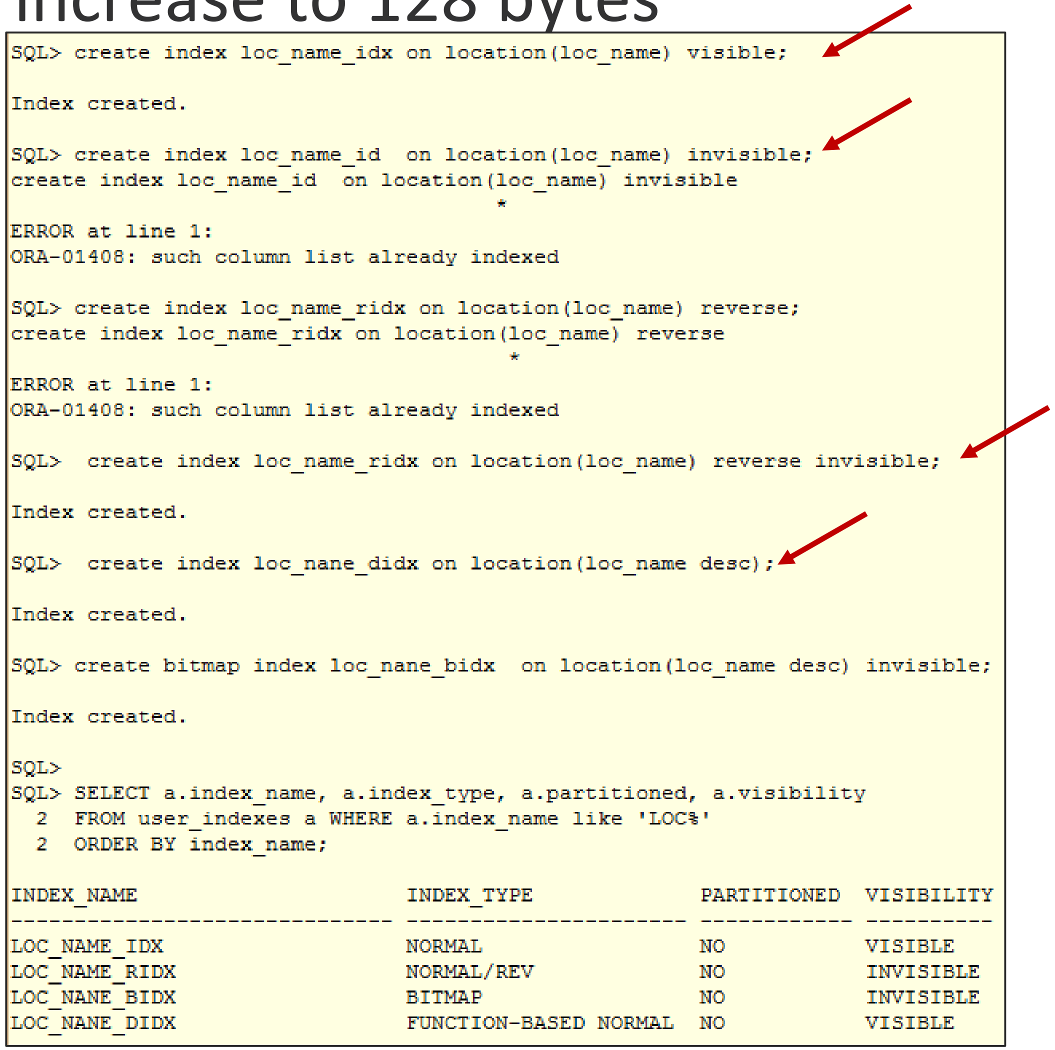
```
SQL> select * from sales_order2
  2  where order_status = 'OPEN'
  3  and o_date > to_date('20171009','YYYYMMDD');

1000 rows selected.

Execution Plan
----------------------------------------------------------
Plan hash value: 3039244622

------------------------------------------------------------------------------------------------------
| Id  | Operation                             | Name                           | Rows  | Bytes | Cost (%CPU)|
------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |                                |  1000 | 32000 |    12   (0)|
|*  1 |  TABLE ACCESS BY GLOBAL INDEX ROWID BATCHED| SALES_ORDER2              |  1000 | 32000 |    12   (0)|
|*  2 |   INDEX RANGE SCAN                    | SALES_ORDER2_GLOBAL_PARTIAL_ID |   699 |       |     5   (0)|
------------------------------------------------------------------------------------------------------

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

   1 - filter("O_DATE">TO_DATE(' 2017-10-09 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
   2 - access("ORDER_STATUS"='OPEN')

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
        146  consistent gets
          0  physical reads
          0  sorts (memory)
          0  sorts (disk)
       1000  rows processed
```

New 12.2 Index Features – Cont.

- Object identifiers including index names increase to 128 bytes
  - Used to be 30 bytes
- Multiple indexes on same columns
  - Only one index must be visible
  - All indexes are different in some way



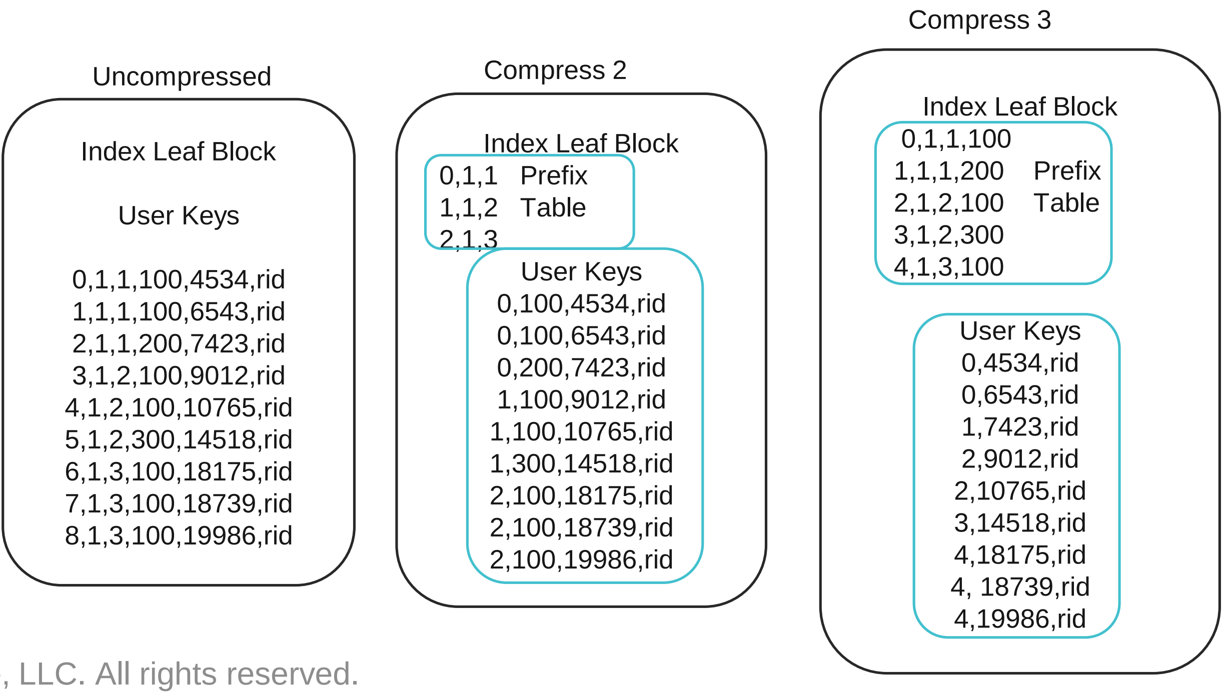
```
SQL> create table location (location_id number, loc_name varchar2(15),
  2  address varchar2(30), st varchar(2), region_name varchar2(10));

Table created.

SQL>  insert into location values(4,'DALLAS','123 Main st','IL','SOUTH');
      ...

SQL> select * from location;

LOCATION_ID LOC_NAME        ADDRESS                        ST REGION_NAM
----------- --------------- ------------------------------ -- ----------
          1 NEW YORK        123 Main st                    NY EAST
          2 CHICAGO         123 Main st                    IL MIDWEST
          3 SEATTLE         123 Main st                    WA WEST
          4 DALLAS          123 Main st                    IL SOUTH
```



```
SQL> create index loc_name_idx on location(loc_name) visible;

Index created.

SQL> create index loc_name_id  on location(loc_name) invisible;
create index loc_name_id  on location(loc_name) invisible
                                         *
ERROR at line 1:
ORA-01408: such column list already indexed

SQL> create index loc_name_ridx on location(loc_name) reverse;
create index loc_name_ridx on location(loc_name) reverse
                                          *
ERROR at line 1:
ORA-01408: such column list already indexed

SQL>  create index loc_name_ridx on location(loc_name) reverse invisible;

Index created.

SQL>  create index loc_nane_didx on location(loc_name desc);

Index created.

SQL> create bitmap index loc_nane_bidx  on location(loc_name desc) invisible;

Index created.

SQL>
SQL> SELECT a.index_name, a.index_type, a.partitioned, a.visibility
  2  FROM user_indexes a WHERE a.index_name like 'LOC%'
  2  ORDER BY index_name;

INDEX_NAME                     INDEX_TYPE             PARTITIONED  VISIBILITY
------------------------------ ---------------------- ------------ ----------
LOC_NAME_IDX                   NORMAL                 NO           VISIBLE
LOC_NAME_RIDX                  NORMAL/REV             NO           INVISIBLE
LOC_NANE_BIDX                  BITMAP                 NO           INVISIBLE
LOC_NANE_DIDX                  FUNCTION-BASED NORMAL  NO           VISIBLE
```

New 12.2 Index Features – Cont.

- Basic index key compression (v 8.1.3+)
  - CREATE INDEX idx ON tbl (col1,col2,col3) COMPRESS;
    - Must know prefix column count
      - Compress 1, compress 2, etc…
  - DBA needs to know selectivity of columns
    - Can make performance worse
      - Can take more space than uncompressed
- Advanced index compression
  - 12.1 – COMPRESS ADVANCED LOW
    - Oracle figures out the prefix column count
    - Prevents making index performance worse
  - 12.2 – COMPRESS ADVANCED HIGH
    - More complex compression algorithms
    - Stores the index in a Compression Unit
      - Similar to Hybrid Columnar Compression



Uncompressed

Index Leaf Block

User Keys

0,1,1,100,4534,rid
1,1,1,100,6543,rid
2,1,1,200,7423,rid
3,1,2,100,9012,rid
4,1,2,100,10765,rid
5,1,2,300,14518,rid
6,1,3,100,18175,rid
7,1,3,100,18739,rid
8,1,3,100,19986,rid

Compress 2

Index Leaf Block

0,1,1 Prefix
1,1,2 Table
2,1,3

User Keys
0,100,4534,rid
0,100,6543,rid
0,200,7423,rid
1,100,9012,rid
1,100,10765,rid
1,300,14518,rid
2,100,18175,rid
2,100,18739,rid
2,100,19986,rid

Compress 3

Index Leaf Block

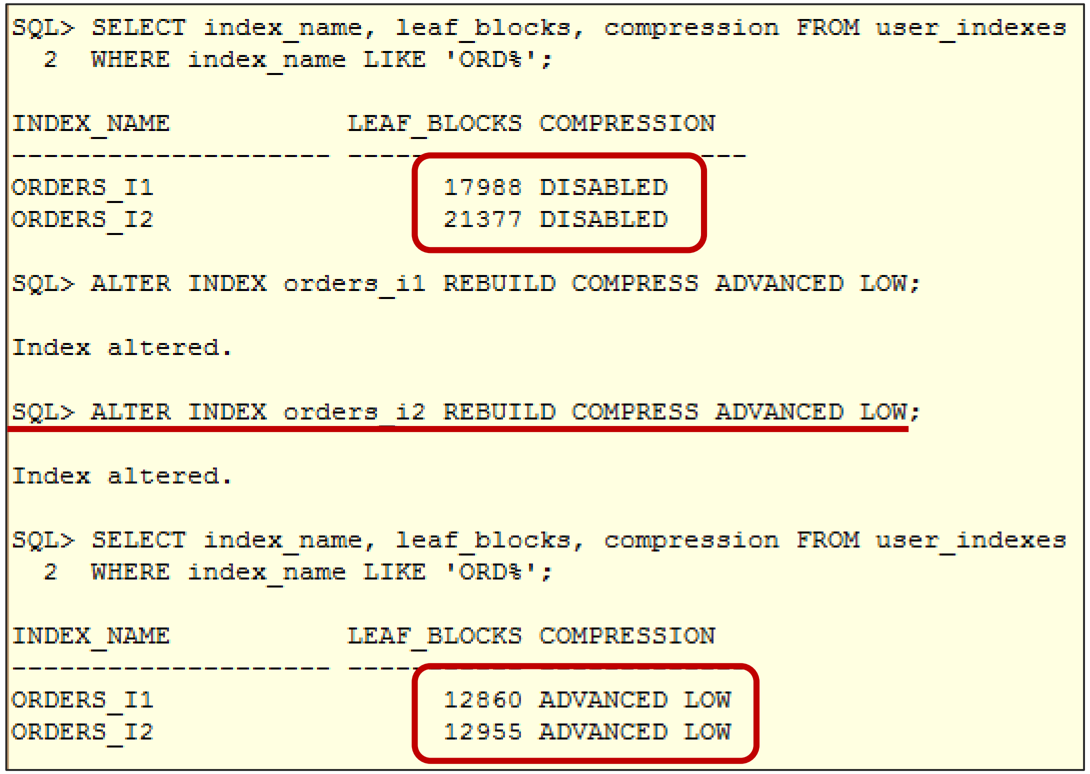
0,1,1,100
1,1,1,200 Prefix
2,1,2,100 Table
3,1,2,300
4,1,3,100

User Keys
0,4534,rid
0,6543,rid
1,7423,rid
2,9012,rid
2,10765,rid
3,14518,rid
4,18175,rid
4, 18739,rid
4,19986,rid

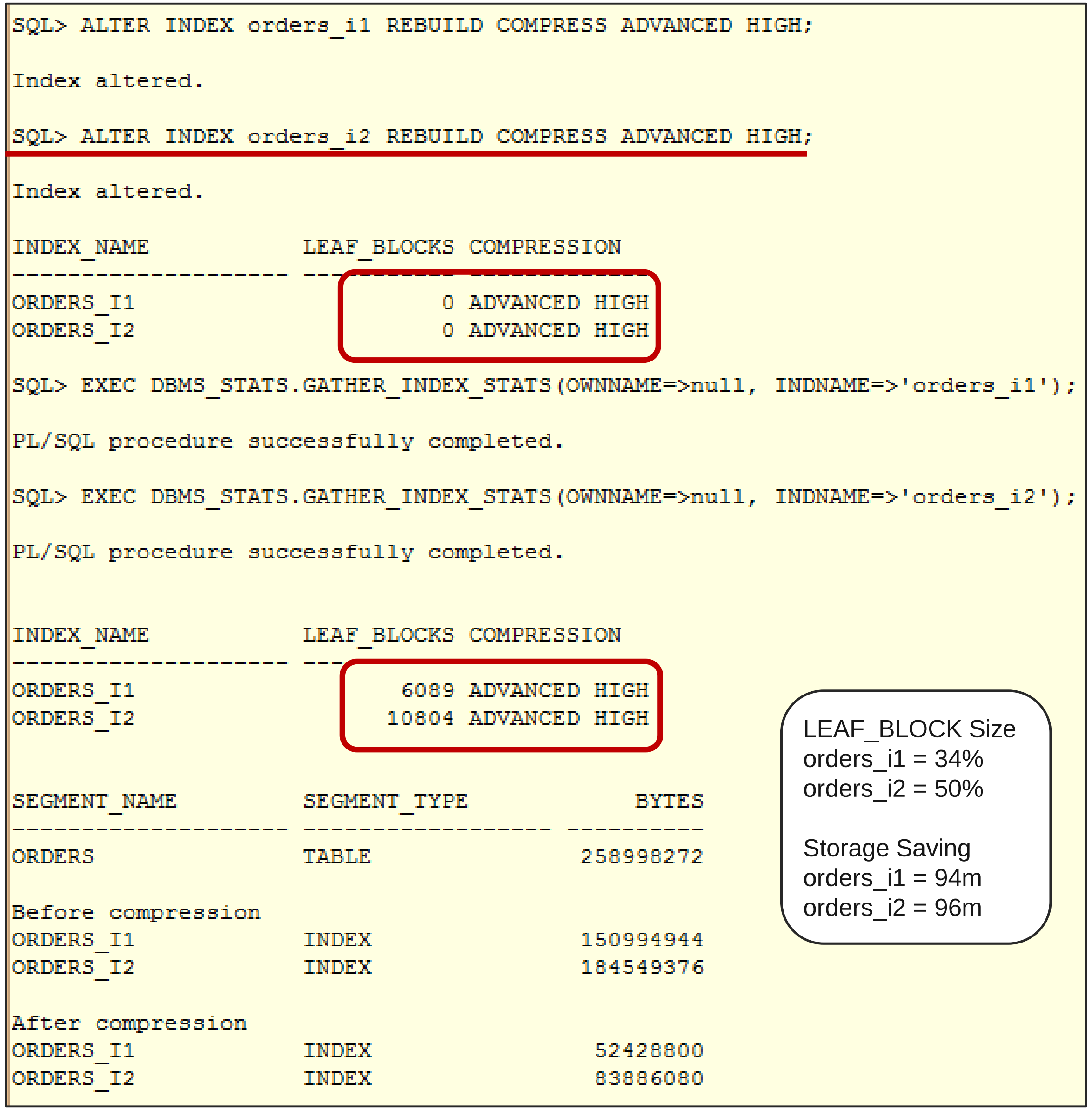
12.2 – Advanced Compression HIGH

solarwinds

- More than just deduping index values
  - Stores index entries in Compression Units
    - Similar Hybrid Columnar Compression
    - DBMS_COMPRESS (See Appendix)



```
SQL> SELECT index_name, leaf_blocks, compression FROM user_indexes
  2  WHERE index_name LIKE 'ORD%';

INDEX_NAME           LEAF_BLOCKS COMPRESSION
-------------------- ----------- -------------
ORDERS_I1                  17988 DISABLED
ORDERS_I2                  21377 DISABLED

SQL> ALTER INDEX orders_i1 REBUILD COMPRESS ADVANCED LOW;

Index altered.

SQL> ALTER INDEX orders_i2 REBUILD COMPRESS ADVANCED LOW;

Index altered.

SQL> SELECT index_name, leaf_blocks, compression FROM user_indexes
  2  WHERE index_name LIKE 'ORD%';

INDEX_NAME           LEAF_BLOCKS COMPRESSION
-------------------- ----------- -------------
ORDERS_I1                  12860 ADVANCED LOW
ORDERS_I2                  12955 ADVANCED LOW
```



```
SQL> ALTER INDEX orders_i1 REBUILD COMPRESS ADVANCED HIGH;

Index altered.

SQL> ALTER INDEX orders_i2 REBUILD COMPRESS ADVANCED HIGH;

Index altered.

INDEX_NAME           LEAF_BLOCKS COMPRESSION
-------------------- ----------- -------------
ORDERS_I1                      0 ADVANCED HIGH
ORDERS_I2                      0 ADVANCED HIGH

SQL> EXEC DBMS_STATS.GATHER_INDEX_STATS(OWNNAME=>null, INDNAME=>'orders_i1');

PL/SQL procedure successfully completed.

SQL> EXEC DBMS_STATS.GATHER_INDEX_STATS(OWNNAME=>null, INDNAME=>'orders_i2');

PL/SQL procedure successfully completed.


INDEX_NAME           LEAF_BLOCKS COMPRESSION
-------------------- ----------- -------------
ORDERS_I1                   6089 ADVANCED HIGH
ORDERS_I2                  10804 ADVANCED HIGH


SEGMENT_NAME         SEGMENT_TYPE            BYTES
-------------------- ------------------ ----------
ORDERS               TABLE               258998272

Before compression
ORDERS_I1            INDEX               150994944
ORDERS_I2            INDEX               184549376

After compression
ORDERS_I1            INDEX                52428800
ORDERS_I2            INDEX                83886080
```

LEAF_BLOCK Size
orders_i1 = 34%
orders_i2 = 50%

Storage Saving
orders_i1 = 94m
orders_i2 = 96m

Understand Statistics Gathering

solarwinds

- GATHER_*_STATS procedures have many parameters
  - Consider taking the default values
  - exec dbms_stats.gather_schema_stats('SOE');
- New 12.2 optimizer statistics advisor
  - Based on 23 predefined rules
    - V$stats_advisor_rules
  - Makes recommendations on collecting stats
  - Can generate scripts for statistics gathering
    - Uses statistic gathering best practices
  - More details on how it works in my session tomorrow

DBMS_STATS package

- Rewritten in 11g
  - A Faster & better AUTO_SAMPLE_SIZE
  - 100% in less time & more accurate than 10% estimate
- Avoid using ESTIMATE_PERCENT

Getting the most out of your Oracle 12.2 Optimizer (i.e. The Brain)

Thursday, May 17, 2018
LL10AB, 11:15 am - 12:15 pm

Index Statistics

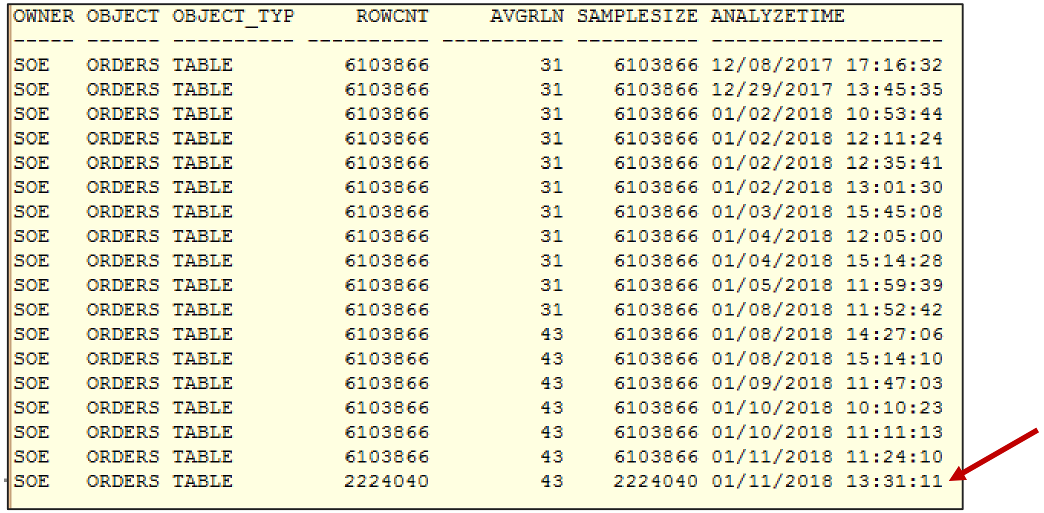
- Useful index views and tables
  - DBA_INDEXES and DBA_IND_COLUMNS

    select index_name, num_rows, blevel, leaf_blocks, distinct_keys
    from dba_indexes where index_name = '&index‘;
  - INDEX_STATS

    analyze index &index_name validate structure;
  - V$SEGMENT_STATISTICS for runtime stats

    select object_name, statistic_name, value
    from V$SEGMENT_STATISTICS where object_name = '&index_name‘;
  - sys.WRI$_OPTSTAT_TAB_HISTORY
    - Shows historical statistics

```
SELECT ob.owner, ob.object_name, ob.object_type,
rowcnt, avgrln ,samplesize, analyzetime
FROM sys.WRI$_OPTSTAT_TAB_HISTORY, dba_objects ob
WHERE owner=upper('&OWNER')
AND object_name=upper('&TABLE')
AND object_type in ('TABLE')
AND object_id=obj#
ORDER BY savtime ASC;
```



```
OWNER OBJECT OBJECT_TYP     ROWCNT     AVGRLN SAMPLESIZE ANALYZETIME
----- ------ ---------- ---------- ---------- ---------- -------------------
SOE   ORDERS TABLE         6103866         31    6103866 12/08/2017 17:16:32
SOE   ORDERS TABLE         6103866         31    6103866 12/29/2017 13:45:35
SOE   ORDERS TABLE         6103866         31    6103866 01/02/2018 10:53:44
SOE   ORDERS TABLE         6103866         31    6103866 01/02/2018 12:11:24
SOE   ORDERS TABLE         6103866         31    6103866 01/02/2018 12:35:41
SOE   ORDERS TABLE         6103866         31    6103866 01/02/2018 13:01:30
SOE   ORDERS TABLE         6103866         31    6103866 01/03/2018 15:45:08
SOE   ORDERS TABLE         6103866         31    6103866 01/04/2018 12:05:00
SOE   ORDERS TABLE         6103866         31    6103866 01/04/2018 15:14:28
SOE   ORDERS TABLE         6103866         31    6103866 01/05/2018 11:59:39
SOE   ORDERS TABLE         6103866         31    6103866 01/08/2018 11:52:42
SOE   ORDERS TABLE         6103866         43    6103866 01/08/2018 14:27:06
SOE   ORDERS TABLE         6103866         43    6103866 01/08/2018 15:14:10
SOE   ORDERS TABLE         6103866         43    6103866 01/09/2018 11:47:03
SOE   ORDERS TABLE         6103866         43    6103866 01/10/2018 10:10:23
SOE   ORDERS TABLE         6103866         43    6103866 01/10/2018 11:11:13
SOE   ORDERS TABLE         6103866         43    6103866 01/11/2018 11:24:10
SOE   ORDERS TABLE         2224040         43    2224040 01/11/2018 13:31:11
```

Summary

- Indexes are optional structures that can speed up performance
- B-Tree sub-types can be descending, reverse key, IOT, or cluster indexes
  - Default index type is B-Tree ascending
- Bitmap and Bitmap Join Indexes
  - Useful in data warehouse or OLAP queries
  - Star schemas
- Function-Based Indexes are useful when sorting by function or expression
  - Can be a B-Tree or Bitmap index
- Partial indexes for partitioned tables in 12c
  - Useful when partitions contain rows that are rarely accessed
- Consider advanced index compression
  - Save space and increases performance
- Index statistics gathering is important

Thank You!!!

Resolve Performance Issues quickly—Free Trial

- Try ***Database Performance Analyzer*** FREE for 14 days
- Improve root cause of slow performance
    - Quickly identify root cause of issues that impact end-user response time
    - See historical trends over days, months, and years
    - Understand impact of VMware® performance
    - Agentless architecture with no dependence on Oracle Packs, installs in minutes

www.solarwinds.com/dpa-download/

Appendix

- Index Structure (index_dump.sql)

```
accept sowner prompt 'Enter Schema Name: '
accept index_name prompt 'Enter Index Name: '

col header_file for 9999999  new_value header_file_no
col root for 999999999 new_value root_block
col rdba for 999999999 new_value rdb_addr
col object_id for 999999999 new_value obj_id
col data_object_id for 999999999 new_value data_obj_id
col relative_fno for 999999999 new_value rfno

SELECT header_file, relative_fno, header_block+1 root
FROM dba_segments
WHERE segment_name = UPPER('&&index_name')
AND owner = UPPER('&&sowner');

-- get relative data block address
SELECT DBMS_UTILITY.MAKE_DATA_BLOCK_ADDRESS(&&rfno, &&root_block) rdba
FROM dual;

SELECT object_id, data_object_id FROM dba_objects
WHERE object_name = UPPER('&index_name');

ALTER SYSTEM DUMP DATAFILE &header_file_no BLOCK &root_block;

SELECT DBMS_UTILITY.DATA_BLOCK_ADDRESS_FILE(&rfno),
    DBMS_UTILITY.DATA_BLOCK_ADDRESS_BLOCK(&&rdb_addr)
FROM dual;

-- get a treedump of the index
ALTER SESSION SET EVENTS 'immediate trace name treedump level &&obj_id';
```

An example of the index dump is on the next slides.

Appendix

- Dump of B-Tree index (from .trc file)

```
Start dump data blocks tsn: 3 file#:16 minblk 1133899 maxblk 1133899
Block dump from cache:
Dump of buffer cache at level 4 for pdb=5 tsn=3 rdba=43076939
Block dump from disk:
buffer tsn: 3 rdba: 0x02914d4b (10/1133899)
scn: 0x0.1b416abe seq: 0x01 flg: 0x04 tail: 0x6abe0601
frmt: 0x02 chkval: 0x88ab type: 0x06=trans data
Hex dump of block: st=0, typ_found=1
Dump of memory from 0x00007F07B8D30E00 to 0x00007F07B8D32E00
7F07B8D30E00 0000A206 02914D4B 1B416ABE 04010000  [....KM...jA.....]
7F07B8D32DF0 80010BC1 01757807 20081411 6ABE0601  [.....xu.... ...j]
...memory dump cut..
Block header dump:  0x02914d4b
 Object id on Block? Y
 seg/obj: 0x174f9  csc: 0x00.1b416abe  itc: 1  flg: E  typ: 2 - INDEX
     brn: 0  bdba: 0x2914d48 ver: 0x01 opc: 0
     inc: 0  exflg: 0

 Itl           Xid                  Uba         Flag  Lck        Scn/Fsc
0x01   0x0008.002.0005b24a  0x010262e8.597c.02  C---    0  scn 0x0000.1b416a14
```

```
Branch block dump
=================
header address 139671142338124=0x7f07b8d30e4c
kdxcolev 2
KDXCOLEV Flags = - - -
kdxcolok 0
kdxcoopc 0x80: opcode=0: iot flags=--- is converted=Y
kdxconco 2
kdxcosdc 0
kdxconro 123
kdxcofbo 274=0x112
kdxcofeo 6474=0x194a
kdxcoavs 6200
kdxbrlmc 43077516=0x2914f8c
kdxbrsno 70
kdxbrbksz 8056
kdxbr2urrc 0
row#0[8042] dba: 43078030=0x291518e
col 0; len 2; (2):  c1 02
col 1; len 6; (6):  09 01 1e 4c 00 1d
row#1[8028] dba: 43078545=0x2915391
col 0; len 2; (2):  c1 02
...cut..
col 1; len 6; (6):  09 01 50 61 00 0c
row#122[7184] dba: 43109596=0x291ccdc
col 0; len 2; (2):  c1 0b
----- end of branch block dump -----
End dump data blocks tsn: 3 file#: 16 minblk 1133899 maxblk 1133899
```

Appendix

- Tree dump of B-Tree index (from .trc file)

```
----- begin tree dump
branch: 0x2a7ed23 44559651 (0: nrow: 169, level: 1)
   leaf: 0x2a7ed24 44559652 (-1: row:361.361 avs:830)
   leaf: 0x2a7ed25 44559653 (0: row:356.356 avs:832)
   leaf: 0x2a7ed26 44559654 (1: row:356.356 avs:831)
   leaf: 0x2a7ed27 44559655 (2: row:356.356 avs:832)
   leaf: 0x2a8c8b0 44615856 (3: row:356.356 avs:831)
   leaf: 0x2a8c8b1 44615857 (4: row:356.356 avs:832)
   leaf: 0x2a8c8b2 44615858 (5: row:356.356 avs:831)
   leaf: 0x2a8c8b3 44615859 (6: row:356.356 avs:832)
   leaf: 0x2a8c8b4 44615860 (7: row:361.361 avs:831)
   leaf: 0x2a8c8b5 44615861 (8: row:356.356 avs:831)
…
   leaf: 0x2a8c8b6 44615862 (9: row:356.356 avs:832)
   leaf: 0x2a920b3 44638387 (165: row:356.356 avs:832)
   leaf: 0x2a920b4 44638388 (166: row:356.356 avs:832)
   leaf: 0x2a920b5 44638389 (167: row:91.91 avs:6129)
----- end tree dump
```

Relative Block Address = 44559651
Root block starts with 0
Number of distinct index blocks in level below root = 169
Level: 1 means index has a height of 2 (blocks below root are leaf blocks)

First leaf block always start with -1
Row:361.361 = number of index entries in leaf block
Row:361.361 = number of non-deleted index entries in leaf block
(no rows deleted)
Avs:830 = Available free space in leaf block

Appendix

- Scripts for random IOT inserts

HEAP_INS.sql

```
create table orders_heap_ins
 (O_ID         NUMBER not null
 ,O_W_ID       NUMBER not null
 ,O_D_ID       NUMBER not null
 ,O_C_ID       NUMBER not null
 ,O_CARRIER_ID NUMBER
 ,O_OL_CNT     NUMBER
 ,O_ALL_LOCAL  NUMBER
 ,O_ENTRY_D    DATE
,constraint orders_heap_ins_pk primary key (o_c_id,o_id,o_w_id,o_d_id)
 using index tablespace index_01
) tablespace data_01;
```

```
set serverout on size 1000000
set long 100000
alter system flush buffer_cache;
alter system flush shared_pool;
truncate table orders_heap_ins;
declare
v_num number :=10000; v_offset number :=-4; v_o_id number; v_o_w_id number;
v_o_d_id number; v_o_c_id number; v_o_entry_d date; v_o_carrier_id number;
v_o_all_local number; v_o_ol_cnt number; v_num_1 number(2) :=10;
v_num_2 number(2) :=15;
begin
dbms_output.put_line('HEAP insert');
dbms_output.put_line (to_char(SYSTIMESTAMP,'HH24:MI:SS.FF'));
for i in 1..10000 loop
    v_o_id    := 1000000+i;
    v_o_w_id :=mod(1,3)+1;
    v_o_d_id := 5+(trunc(dbms_random.value(1,3))*5); -- 10 or 15
    v_o_c_id   :=trunc(dbms_random.value(1,v_num/2)); -- 1 to 5000
    v_o_carrier_id :=mod(1,3)+1;
    v_o_ol_cnt :=mod(1,3)+1;
    v_o_all_local   :=trunc(dbms_random.value(1,v_num/2)); -- 1 to 5000
    v_o_entry_d  :=trunc(sysdate-v_offset)+(i/(60*60*24));
    INSERT INTO orders_heap_ins (o_id,o_w_id,o_d_id,o_c_id,o_carrier_id, o_ol_cnt,o_all_local,o_entry_d)
     values
    (v_o_id,v_o_w_id,v_o_d_id,v_o_c_id,v_o_carrier_id, v_o_ol_cnt,v_o_all_local,v_o_entry_d);
    commit;
end loop;
dbms_output.put_line (to_char(SYSTIMESTAMP,'HH24:MI:SS.FF'));
end;
/
```

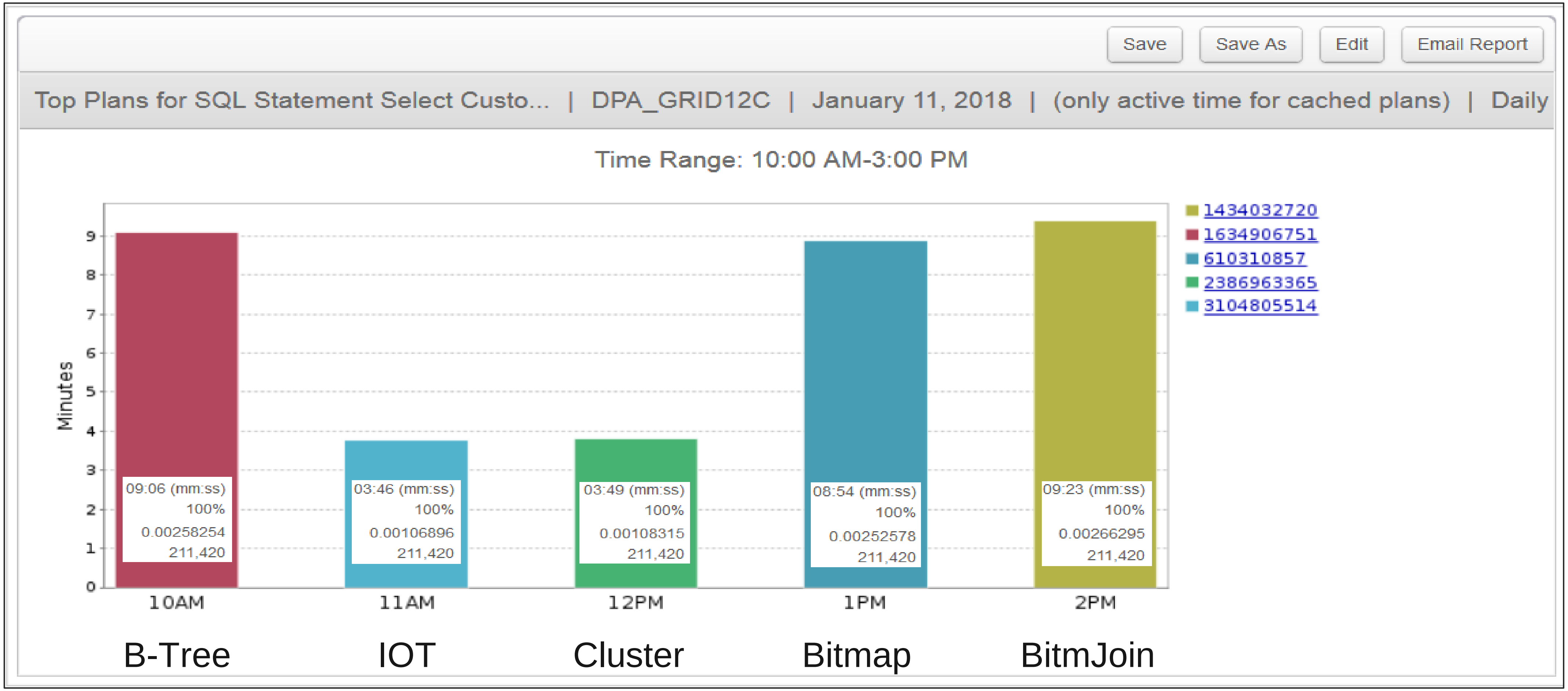
IOT_INS.sql

```
create table orders_iot_ins
 (O_ID         NUMBER not null
 ,O_W_ID       NUMBER not null
 ,O_D_ID       NUMBER not null
 ,O_C_ID       NUMBER not null
 ,O_CARRIER_ID NUMBER
 ,O_OL_CNT     NUMBER
 ,O_ALL_LOCAL  NUMBER
 ,O_ENTRY_D    DATE
,constraint orders_iot_pk primary key (o_w_id,o_d_id,o_c_id,o_id)
)
ORGANIZATION INDEX
tablespace data_01;
```

```
set serverout on size 1000000
set long 100000
alter system flush buffer_cache;
alter system flush shared_pool;
truncate table orders_iot_ins;
declare
v_num number :=10000; v_offset number :=-4; v_o_id number;
v_o_w_id number; v_o_d_id number; v_o_c_id number;
v_o_entry_d date; v_o_carrier_id number; v_o_all_local  number;
v_o_ol_cnt number; v_num_1 number(2) :=10; v_num_2 number(2) :=15;
begin
dbms_output.put_line('IOT insert');
dbms_output.put_line (to_char(SYSTIMESTAMP,'HH24:MI:SS.FF'));
for i in 1..10000 loop
    v_o_id    := 1000000+i;
    v_o_w_id :=mod(1,3)+1;
    v_o_d_id := 5+(trunc(dbms_random.value(1,3))*5); -- 10 or 15
    v_o_c_id   :=trunc(dbms_random.value(1,v_num/2)); -- 1 to 5000
    v_o_carrier_id :=mod(1,3)+1;
    v_o_ol_cnt :=mod(1,3)+1;
    v_o_all_local   :=trunc(dbms_random.value(1,v_num/2)); -- 1 to 5000
    v_o_entry_d  :=trunc(sysdate-v_offset)+(i/(60*60*24));
    INSERT INTO orders_iot_ins (o_id,o_w_id,o_d_id,o_c_id,o_carrier_id, o_ol_cnt,o_all_local,o_entry_d)
     values
    (v_o_id,v_o_w_id,v_o_d_id,v_o_c_id,v_o_carrier_id, v_o_ol_cnt,v_o_all_local,v_o_entry_d);
    commit;
end loop;
dbms_output.put_line (to_char(SYSTIMESTAMP,'HH24:MI:SS.FF'));
end;
/
```

Appendix

- Execution Plans from Case Study

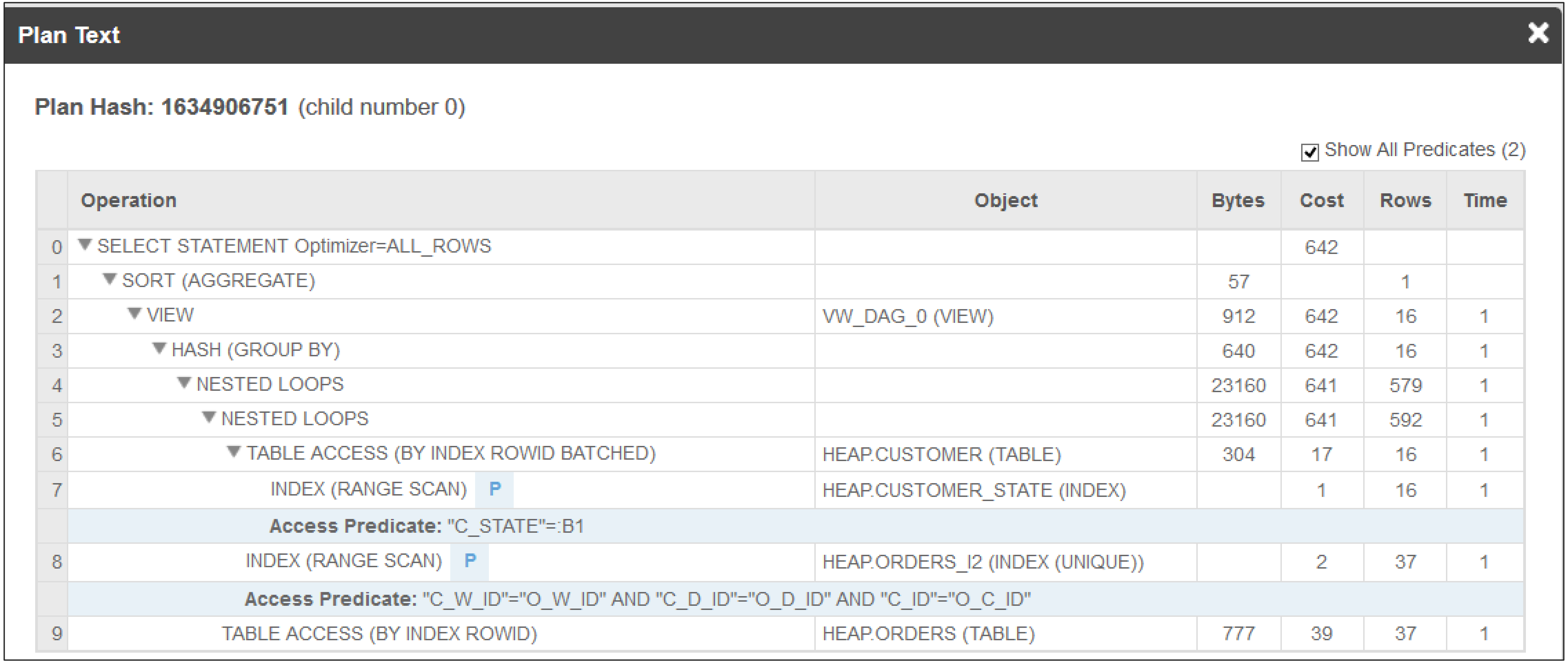


Top Plans for SQL Statement Select Custo... | DPA_GRID12C | January 11, 2018 | (only active time for cached plans) | Daily

Time Range: 10:00 AM-3:00 PM

Appendix

- B-Tree Index



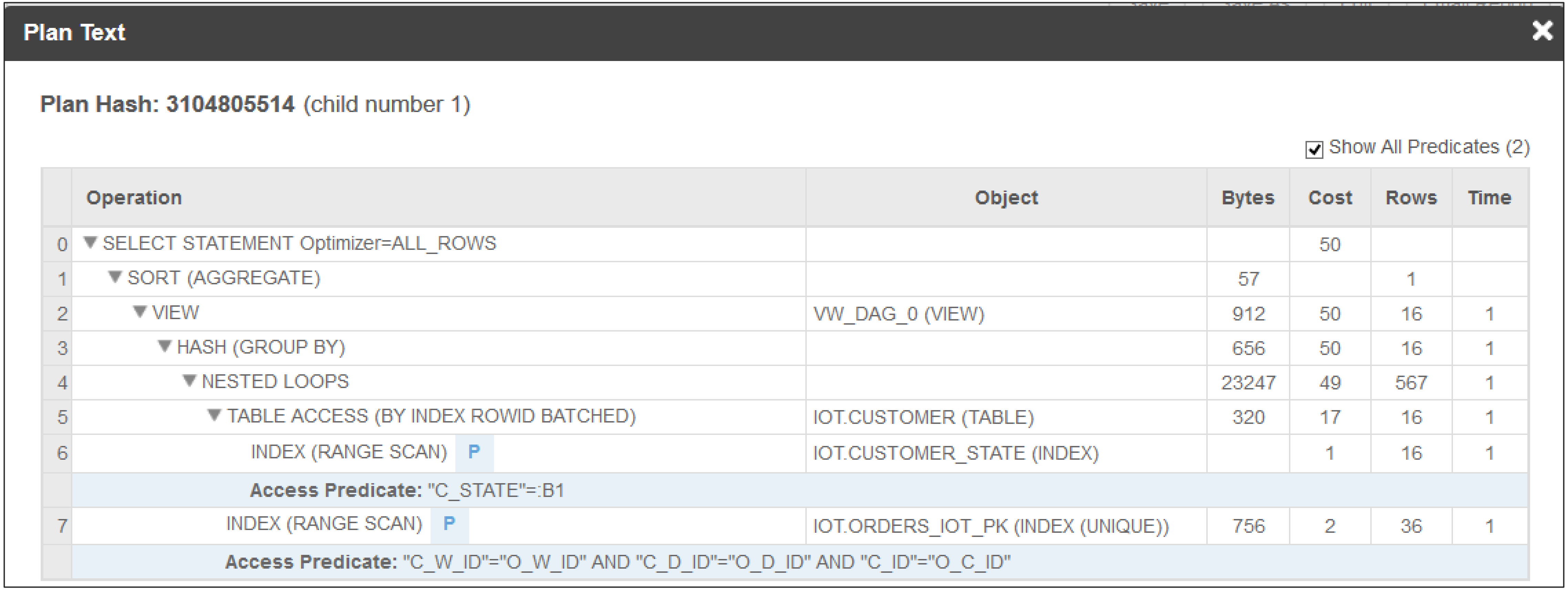
Plan Hash: 1634906751 (child number 0)

☑ Show All Predicates (2)

	Operation	Object	Bytes	Cost	Rows	Time
0	▼ SELECT STATEMENT Optimizer=ALL_ROWS			642		
1	▼ SORT (AGGREGATE)		57		1	
2	▼ VIEW	VW_DAG_0 (VIEW)	912	642	16	1
3	▼ HASH (GROUP BY)		640	642	16	1
4	▼ NESTED LOOPS		23160	641	579	1
5	▼ NESTED LOOPS		23160	641	592	1
6	▼ TABLE ACCESS (BY INDEX ROWID BATCHED)	HEAP.CUSTOMER (TABLE)	304	17	16	1
7	INDEX (RANGE SCAN) P	HEAP.CUSTOMER_STATE (INDEX)		1	16	1
	Access Predicate: "C_STATE"=:B1					
8	INDEX (RANGE SCAN) P	HEAP.ORDERS_I2 (INDEX (UNIQUE))		2	37	1
	Access Predicate: "C_W_ID"="O_W_ID" AND "C_D_ID"="O_D_ID" AND "C_ID"="O_C_ID"					
9	TABLE ACCESS (BY INDEX ROWID)	HEAP.ORDERS (TABLE)	777	39	37	1

Appendix

- IOT



Plan Text ✕

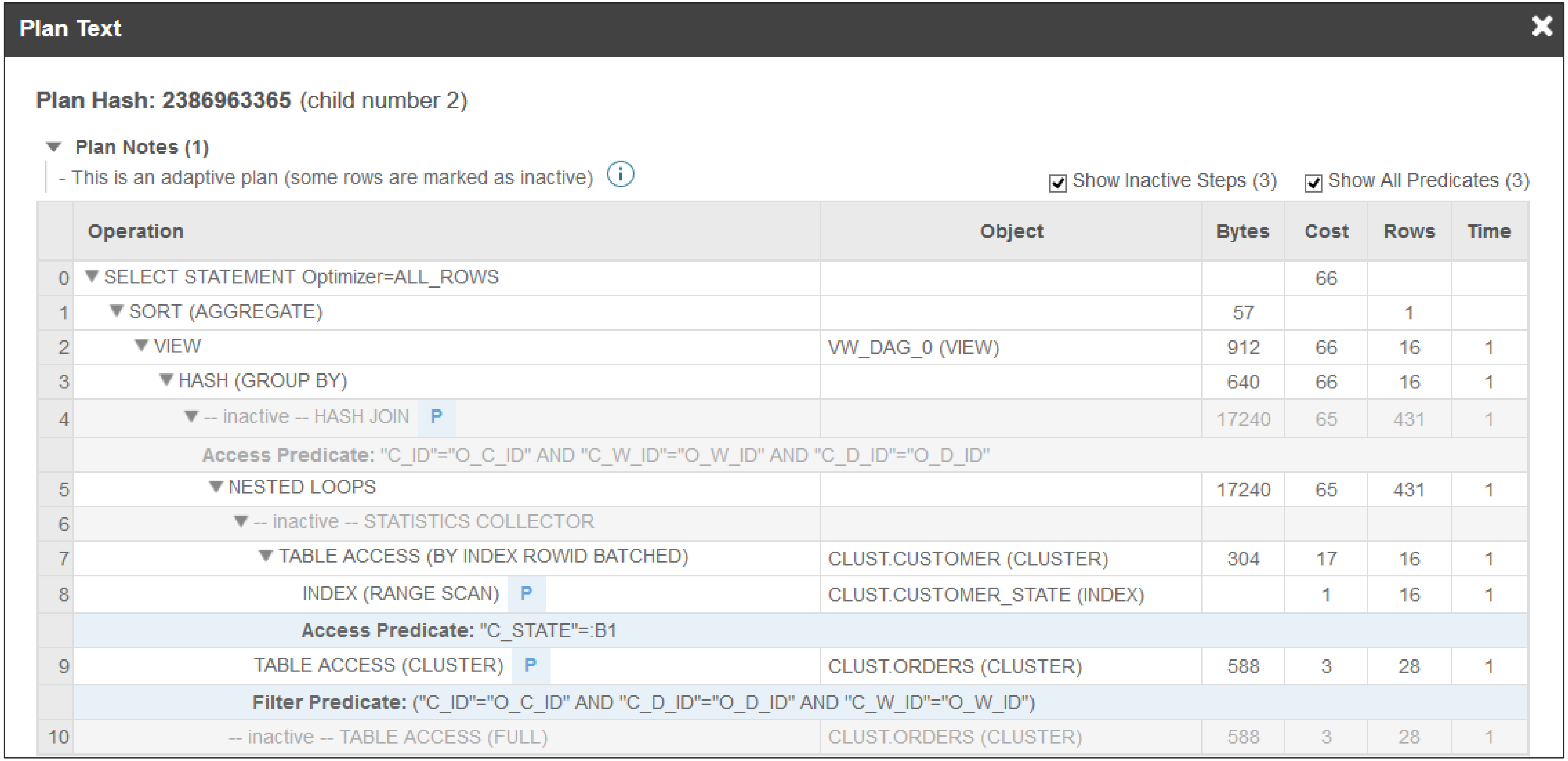
Plan Hash: 3104805514 (child number 1)

☑ Show All Predicates (2)

	Operation	Object	Bytes	Cost	Rows	Time
0	▼ SELECT STATEMENT Optimizer=ALL_ROWS			50		
1	▼ SORT (AGGREGATE)		57		1	
2	▼ VIEW	VW_DAG_0 (VIEW)	912	50	16	1
3	▼ HASH (GROUP BY)		656	50	16	1
4	▼ NESTED LOOPS		23247	49	567	1
5	▼ TABLE ACCESS (BY INDEX ROWID BATCHED)	IOT.CUSTOMER (TABLE)	320	17	16	1
6	INDEX (RANGE SCAN) P	IOT.CUSTOMER_STATE (INDEX)		1	16	1
	Access Predicate: "C_STATE"=:B1					
7	INDEX (RANGE SCAN) P	IOT.ORDERS_IOT_PK (INDEX (UNIQUE))	756	2	36	1
	Access Predicate: "C_W_ID"="O_W_ID" AND "C_D_ID"="O_D_ID" AND "C_ID"="O_C_ID"					

Appendix

- Cluster Index



Plan Hash: 2386963365 (child number 2)

▼ **Plan Notes (1)**

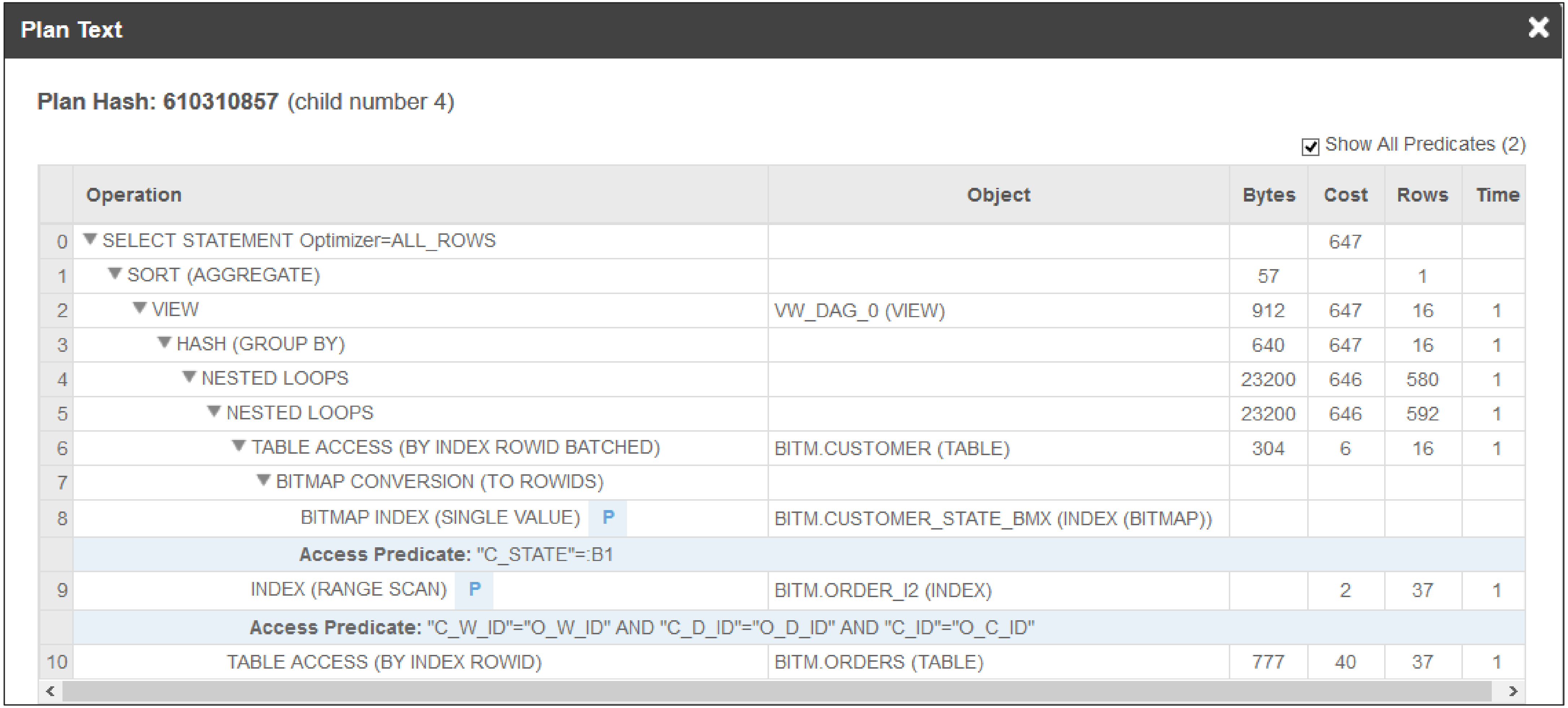
- This is an adaptive plan (some rows are marked as inactive) ⓘ

☑ Show Inactive Steps (3) ☑ Show All Predicates (3)

	Operation	Object	Bytes	Cost	Rows	Time
0	▼ SELECT STATEMENT Optimizer=ALL_ROWS			66		
1	▼ SORT (AGGREGATE)		57		1	
2	▼ VIEW	VW_DAG_0 (VIEW)	912	66	16	1
3	▼ HASH (GROUP BY)		640	66	16	1
4	▼ -- inactive -- HASH JOIN P		17240	65	431	1
	Access Predicate: "C_ID"="O_C_ID" AND "C_W_ID"="O_W_ID" AND "C_D_ID"="O_D_ID"					
5	▼ NESTED LOOPS		17240	65	431	1
6	▼ -- inactive -- STATISTICS COLLECTOR					
7	▼ TABLE ACCESS (BY INDEX ROWID BATCHED)	CLUST.CUSTOMER (CLUSTER)	304	17	16	1
8	INDEX (RANGE SCAN) P	CLUST.CUSTOMER_STATE (INDEX)		1	16	1
	Access Predicate: "C_STATE"=:B1					
9	TABLE ACCESS (CLUSTER) P	CLUST.ORDERS (CLUSTER)	588	3	28	1
	Filter Predicate: ("C_ID"="O_C_ID" AND "C_D_ID"="O_D_ID" AND "C_W_ID"="O_W_ID")					
10	-- inactive -- TABLE ACCESS (FULL)	CLUST.ORDERS (CLUSTER)	588	3	28	1

Appendix

- Bitmap Index



Plan Text ✕

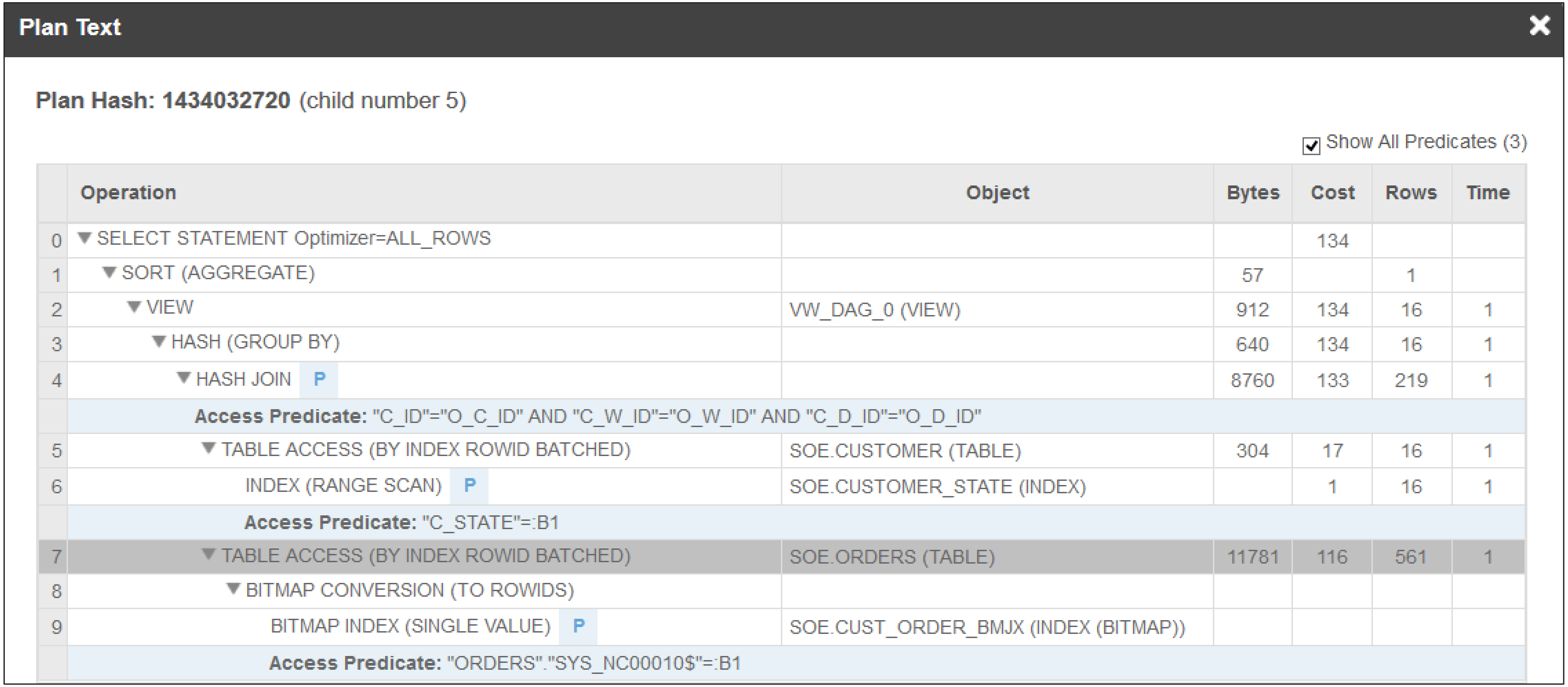
Plan Hash: 610310857 (child number 4)

☑ Show All Predicates (2)

	Operation	Object	Bytes	Cost	Rows	Time
0	▼ SELECT STATEMENT Optimizer=ALL_ROWS			647		
1	▼ SORT (AGGREGATE)		57		1	
2	▼ VIEW	VW_DAG_0 (VIEW)	912	647	16	1
3	▼ HASH (GROUP BY)		640	647	16	1
4	▼ NESTED LOOPS		23200	646	580	1
5	▼ NESTED LOOPS		23200	646	592	1
6	▼ TABLE ACCESS (BY INDEX ROWID BATCHED)	BITM.CUSTOMER (TABLE)	304	6	16	1
7	▼ BITMAP CONVERSION (TO ROWIDS)					
8	BITMAP INDEX (SINGLE VALUE) P	BITM.CUSTOMER_STATE_BMX (INDEX (BITMAP))				
	Access Predicate: "C_STATE"=:B1					
9	INDEX (RANGE SCAN) P	BITM.ORDER_I2 (INDEX)		2	37	1
	Access Predicate: "C_W_ID"="O_W_ID" AND "C_D_ID"="O_D_ID" AND "C_ID"="O_C_ID"					
10	TABLE ACCESS (BY INDEX ROWID)	BITM.ORDERS (TABLE)	777	40	37	1

Appendix

- Bitmap Join



Plan Text ✕

Plan Hash: 1434032720 (child number 5)

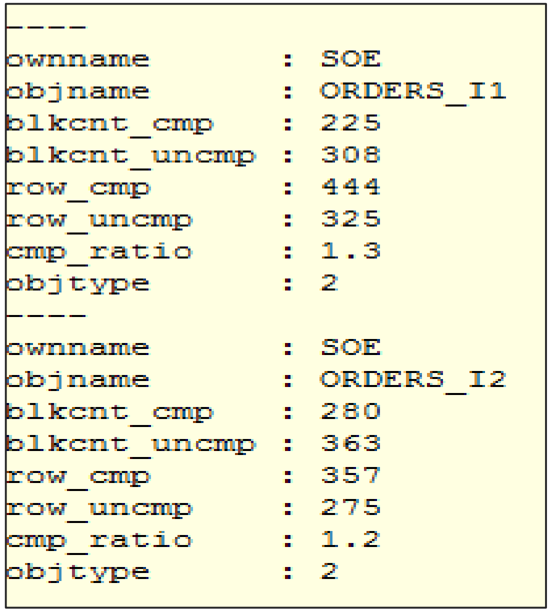
☑ Show All Predicates (3)

	Operation	Object	Bytes	Cost	Rows	Time
0	▼ SELECT STATEMENT Optimizer=ALL_ROWS			134		
1	▼ SORT (AGGREGATE)		57		1	
2	▼ VIEW	VW_DAG_0 (VIEW)	912	134	16	1
3	▼ HASH (GROUP BY)		640	134	16	1
4	▼ HASH JOIN P		8760	133	219	1
	Access Predicate: "C_ID"="O_C_ID" AND "C_W_ID"="O_W_ID" AND "C_D_ID"="O_D_ID"					
5	▼ TABLE ACCESS (BY INDEX ROWID BATCHED)	SOE.CUSTOMER (TABLE)	304	17	16	1
6	INDEX (RANGE SCAN) P	SOE.CUSTOMER_STATE (INDEX)		1	16	1
	Access Predicate: "C_STATE"=:B1					
7	▼ TABLE ACCESS (BY INDEX ROWID BATCHED)	SOE.ORDERS (TABLE)	11781	116	561	1
8	▼ BITMAP CONVERSION (TO ROWIDS)					
9	BITMAP INDEX (SINGLE VALUE) P	SOE.CUST_ORDER_BMJX (INDEX (BITMAP))				
	Access Predicate: "ORDERS"."SYS_NC00010$"=:B1					

Appendix

- DBMS_COMPRESSION – list all indexes and estimate of compression ratio

```
SET SERVEROUTPUT ON
DECLARE
  l_index_cr      DBMS_COMPRESSION.compreclist;
  l_comptype_str  VARCHAR2(32767);
BEGIN
  DBMS_COMPRESSION.get_compression_ratio (
    scratchtbsname  => 'USERS',
    ownname         => 'SOE',
    tabname         => 'ORDERS',
    comptype        => DBMS_COMPRESSION.comp_index_advanced_low,
    index_cr        => l_index_cr,
    comptype_str    => l_comptype_str,
    subset_numrows  => DBMS_COMPRESSION.comp_ratio_lob_maxrows
  );

  FOR i IN l_index_cr.FIRST .. l_index_cr.LAST LOOP
    DBMS_OUTPUT.put_line('----');
    DBMS_OUTPUT.put_line('ownname      : ' || l_index_cr(i).ownname);
    DBMS_OUTPUT.put_line('objname      : ' || l_index_cr(i).objname);
    DBMS_OUTPUT.put_line('blkcnt_cmp   : ' || l_index_cr(i).blkcnt_cmp);
    DBMS_OUTPUT.put_line('blkcnt_uncmp : ' || l_index_cr(i).blkcnt_uncmp);
    DBMS_OUTPUT.put_line('row_cmp      : ' || l_index_cr(i).row_cmp);
    DBMS_OUTPUT.put_line('row_uncmp    : ' || l_index_cr(i).row_uncmp);
    DBMS_OUTPUT.put_line('cmp_ratio    : ' || l_index_cr(i).cmp_ratio);
    DBMS_OUTPUT.put_line('objtype      : ' || l_index_cr(i).objtype);
  END LOOP;
END;
```



```
----
ownname       : SOE
objname       : ORDERS_I1
blkcnt_cmp    : 225
blkcnt_uncmp  : 308
row_cmp       : 444
row_uncmp     : 325
cmp_ratio     : 1.3
objtype       : 2
----
ownname       : SOE
objname       : ORDERS_I2
blkcnt_cmp    : 280
blkcnt_uncmp  : 363
row_cmp       : 357
row_uncmp     : 275
cmp_ratio     : 1.2
objtype       : 2
```

Appendix

- Script to Find Missing Foreign Key Indexes (missing_indexes.sql)

```
SELECT dcc.owner,dcc.constraint_name,dcc.column_name,dcc.position
FROM dba_cons_columns dcc, dba_constraints dc
WHERE dc.constraint_name = dcc.constraint_name
AND dc.constraint_type = 'R'
AND (dcc.owner, dcc.table_name, dcc.column_name, dcc.position) IN
   (SELECT dcc.owner, dcc.table_name, dcc.column_name, dcc.position
    FROM dba_cons_columns dcc, dba_constraints dc
    WHERE dc.constraint_name = dcc.constraint_name
    AND dc.constraint_type = 'R'
    MINUS
    SELECT table_owner, table_name, column_name, column_position
    FROM dba_ind_columns)
ORDER BY dcc.owner, dcc.constraint_name, dcc.column_name, dcc.position;
```

Appendix - Optimizer Statistics Advisor

1. Create task

```
EXEC DBMS_STATS.DROP_ADVISOR_TASK('STAT_ADVICE');

DECLARE
  task_name VARCHAR2(100);
  results VARCHAR2(32767);
BEGIN
  task_name := 'STAT_ADVICE';
  results := DBMS_STATS.CREATE_ADVISOR_TASK(task_name);
END;
/

select task_name, advisor_name, created, status from
dba_advisor_tasks where advisor_name = 'Statistics Advisor‘;
```

2. Define filters>

```
  filter1 CLOB;  -- disable advisor on all objects
  filter2 CLOB; -- enable advice on SOE.ORDER_LINE
  filter3 CLOB; --  disable rule AvoidDropRecreate
  filter4 CLOB; -- enable rule UseGatherSchemaStats
BEGIN
filter1 := DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER(
  task_name => 'STAT_ADVICE',
  stats_adv_opr_type => 'EXECUTE',
  rule_name => NULL,
  ownname => NULL,
  tabname => NULL,
  action => 'DISABLE' );

filter2 := DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER(
  task_name => 'STAT_ADVICE',
  stats_adv_opr_type => 'EXECUTE',
  rule_name => NULL,
  ownname => 'SOE',
  tabname => 'ORDER_LINE',
  action => 'ENABLE' );

filter3 := DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER(
  task_name => 'STAT_ADVICE',
  stats_adv_opr_type => 'EXECUTE',
  rule_name => 'AvoidDropRecreate',
  action => 'DISABLE' );

filter4 := DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER(
  task_name => 'STAT_ADVICE',
  stats_adv_opr_type => 'EXECUTE',
  rule_name => 'UseGatherSchemaStats',
  action => 'ENABLE' );
END;
/
```

3. Execute task

```
DECLARE
  task_name VARCHAR2(100);
  results VARCHAR2(32767);
BEGIN
  task_name := 'STAT_ADVICE';
  results := DBMS_STATS.EXECUTE_ADVISOR_TASK(task_name);
END;
/
```

Appendix - Optimizer Statistics Advisor

4. Report task

```
set pagesize 1000
set linesize 132
set long 1000000
select dbms_stats.report_advisor_task('STAT_ADVICE',null,'text','all','all') as report  from dual;
```

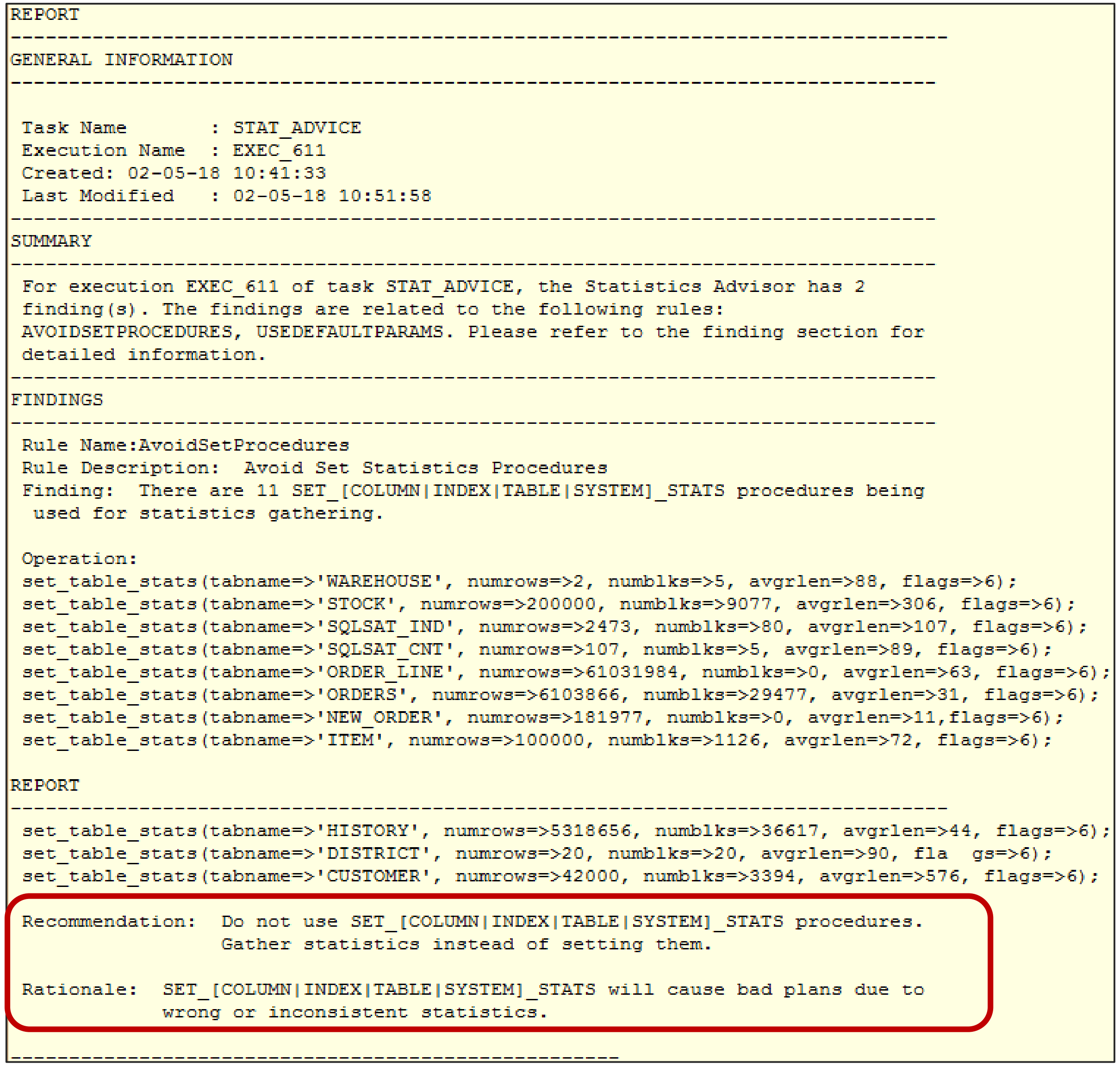
5. Generate script

```
VAR script CLOB
DECLARE
  task_name VARCHAR2(100);
BEGIN
  task_name := 'STAT_ADVICE';
  :script := DBMS_STATS.SCRIPT_ADVISOR_TASK(task_name);
END;
/
```

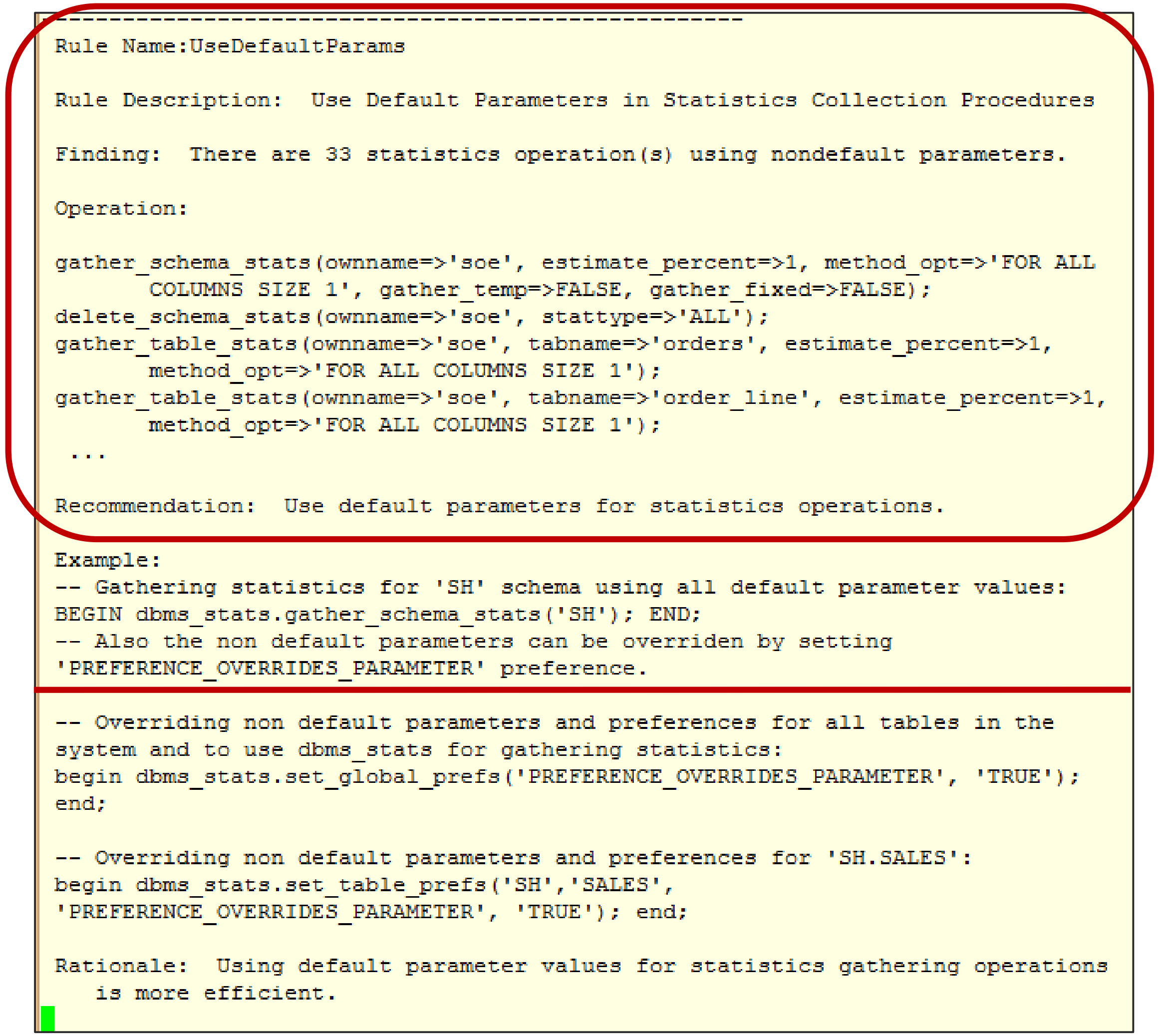
6. Display script>

```
set linesize 132
set long 100000
set pagesize 0
set longchunksize 100000
set serveroutput on

DECLARE
  v_len NUMBER(10);
  v_offset NUMBER(10) :=1;
  v_amount NUMBER(10) :=10000;
BEGIN
  v_len := DBMS_LOB.getlength(:script);
  WHILE (v_offset < v_len)
  LOOP

DBMS_OUTPUT.PUT_LINE(DBMS_LOB.SUBSTR(:script,v_amount,v_offset));
    v_offset := v_offset + v_amount;
  END LOOP;
END;
/
```

Appendix - Optimizer Statistics Advisor Report

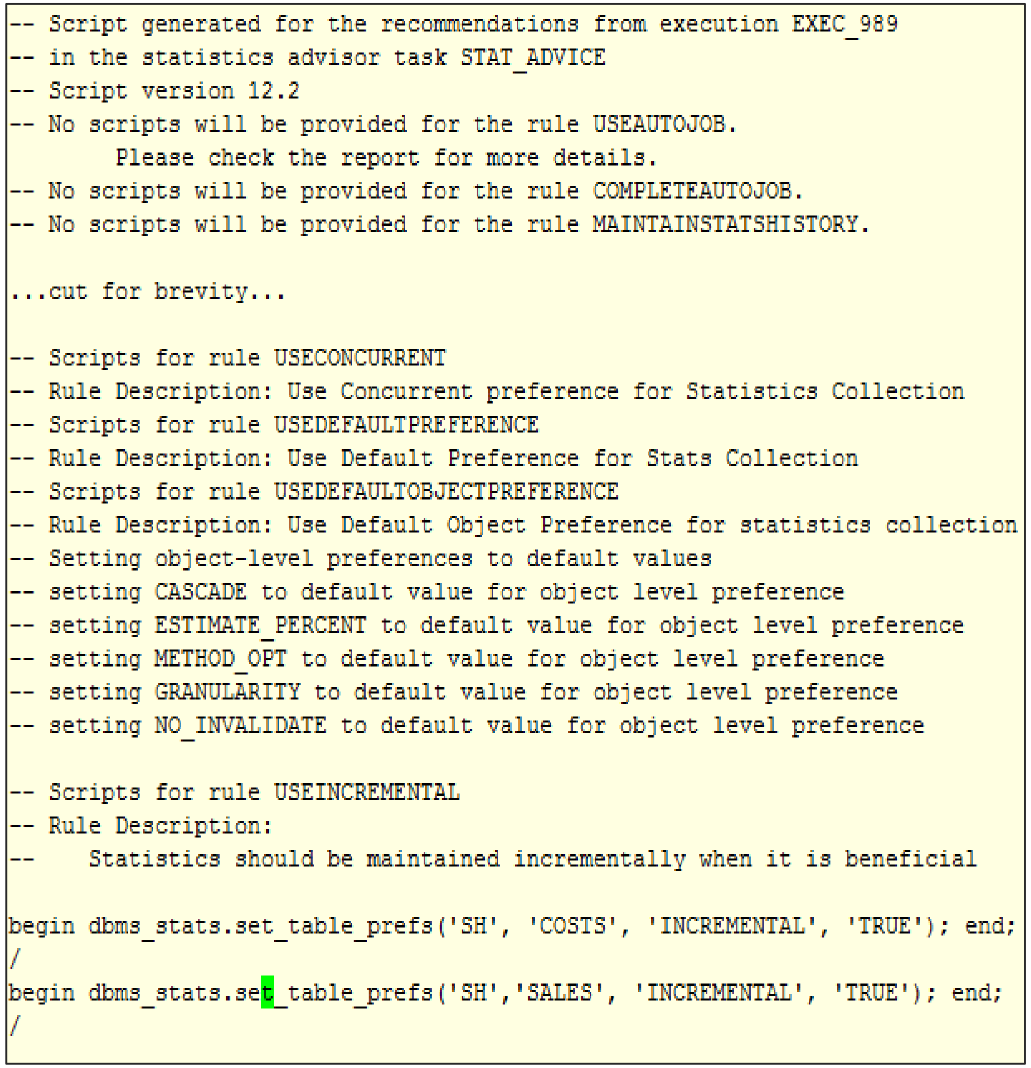


```
REPORT
-------------------------------------------------------------------------------
GENERAL INFORMATION
-------------------------------------------------------------------------------

 Task Name       : STAT_ADVICE
 Execution Name  : EXEC_611
 Created: 02-05-18 10:41:33
 Last Modified   : 02-05-18 10:51:58
-------------------------------------------------------------------------------
SUMMARY
-------------------------------------------------------------------------------
 For execution EXEC_611 of task STAT_ADVICE, the Statistics Advisor has 2
 finding(s). The findings are related to the following rules:
 AVOIDSETPROCEDURES, USEDEFAULTPARAMS. Please refer to the finding section for
 detailed information.
-------------------------------------------------------------------------------
FINDINGS
-------------------------------------------------------------------------------
 Rule Name:AvoidSetProcedures
 Rule Description:  Avoid Set Statistics Procedures
 Finding:  There are 11 SET_[COLUMN|INDEX|TABLE|SYSTEM]_STATS procedures being
  used for statistics gathering.

 Operation:
 set_table_stats(tabname=>'WAREHOUSE', numrows=>2, numblks=>5, avgrlen=>88, flags=>6);
 set_table_stats(tabname=>'STOCK', numrows=>200000, numblks=>9077, avgrlen=>306, flags=>6);
 set_table_stats(tabname=>'SQLSAT_IND', numrows=>2473, numblks=>80, avgrlen=>107, flags=>6);
 set_table_stats(tabname=>'SQLSAT_CNT', numrows=>107, numblks=>5, avgrlen=>89, flags=>6);
 set_table_stats(tabname=>'ORDER_LINE', numrows=>61031984, numblks=>0, avgrlen=>63, flags=>6);
 set_table_stats(tabname=>'ORDERS', numrows=>6103866, numblks=>29477, avgrlen=>31, flags=>6);
 set_table_stats(tabname=>'NEW_ORDER', numrows=>181977, numblks=>0, avgrlen=>11,flags=>6);
 set_table_stats(tabname=>'ITEM', numrows=>100000, numblks=>1126, avgrlen=>72, flags=>6);

REPORT
-------------------------------------------------------------------------------
 set_table_stats(tabname=>'HISTORY', numrows=>5318656, numblks=>36617, avgrlen=>44, flags=>6);
 set_table_stats(tabname=>'DISTRICT', numrows=>20, numblks=>20, avgrlen=>90, fla  gs=>6);
 set_table_stats(tabname=>'CUSTOMER', numrows=>42000, numblks=>3394, avgrlen=>576, flags=>6);

 Recommendation:  Do not use SET_[COLUMN|INDEX|TABLE|SYSTEM]_STATS procedures.
                  Gather statistics instead of setting them.

 Rationale:  SET_[COLUMN|INDEX|TABLE|SYSTEM]_STATS will cause bad plans due to
             wrong or inconsistent statistics.

---------------------------------------------------
```

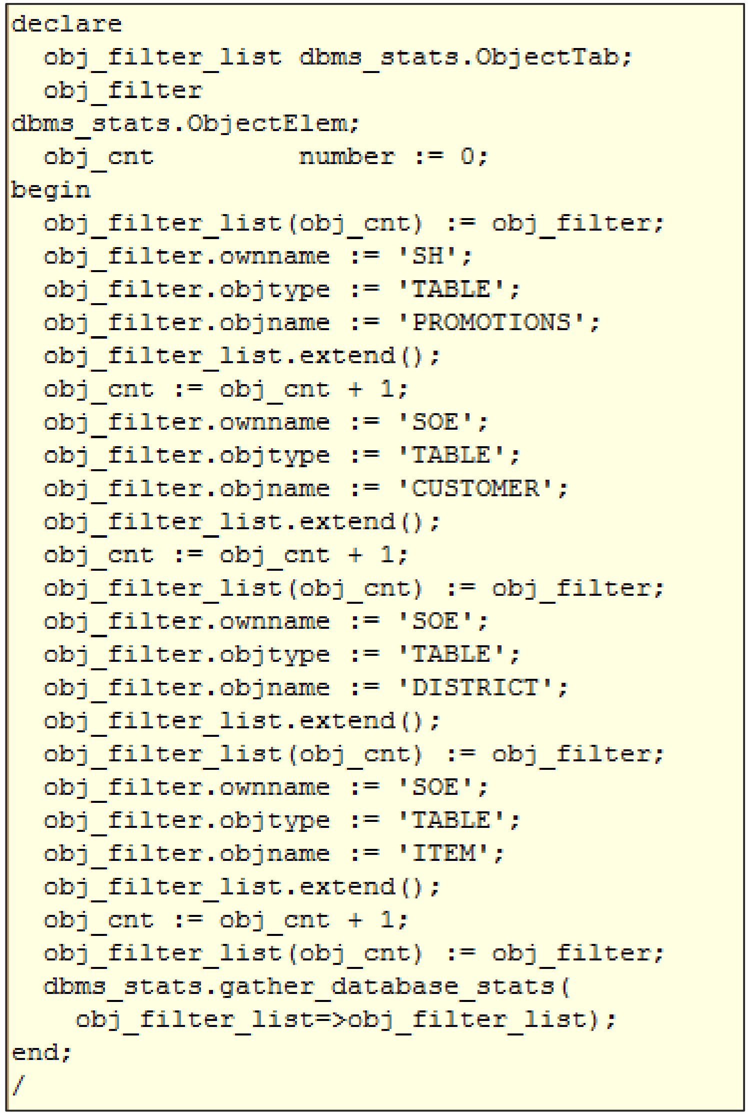


```
----------------------------------------------------
Rule Name:UseDefaultParams

Rule Description:  Use Default Parameters in Statistics Collection Procedures

Finding:  There are 33 statistics operation(s) using nondefault parameters.

Operation:

gather_schema_stats(ownname=>'soe', estimate_percent=>1, method_opt=>'FOR ALL
       COLUMNS SIZE 1', gather_temp=>FALSE, gather_fixed=>FALSE);
delete_schema_stats(ownname=>'soe', stattype=>'ALL');
gather_table_stats(ownname=>'soe', tabname=>'orders', estimate_percent=>1,
       method_opt=>'FOR ALL COLUMNS SIZE 1');
gather_table_stats(ownname=>'soe', tabname=>'order_line', estimate_percent=>1,
       method_opt=>'FOR ALL COLUMNS SIZE 1');
 ...

Recommendation:  Use default parameters for statistics operations.

Example:
-- Gathering statistics for 'SH' schema using all default parameter values:
BEGIN dbms_stats.gather_schema_stats('SH'); END;
-- Also the non default parameters can be overriden by setting
'PREFERENCE_OVERRIDES_PARAMETER' preference.

-- Overriding non default parameters and preferences for all tables in the
system and to use dbms_stats for gathering statistics:
begin dbms_stats.set_global_prefs('PREFERENCE_OVERRIDES_PARAMETER', 'TRUE');
end;

-- Overriding non default parameters and preferences for 'SH.SALES':
begin dbms_stats.set_table_prefs('SH','SALES',
'PREFERENCE_OVERRIDES_PARAMETER', 'TRUE'); end;

Rationale:  Using default parameter values for statistics gathering operations
   is more efficient.
```

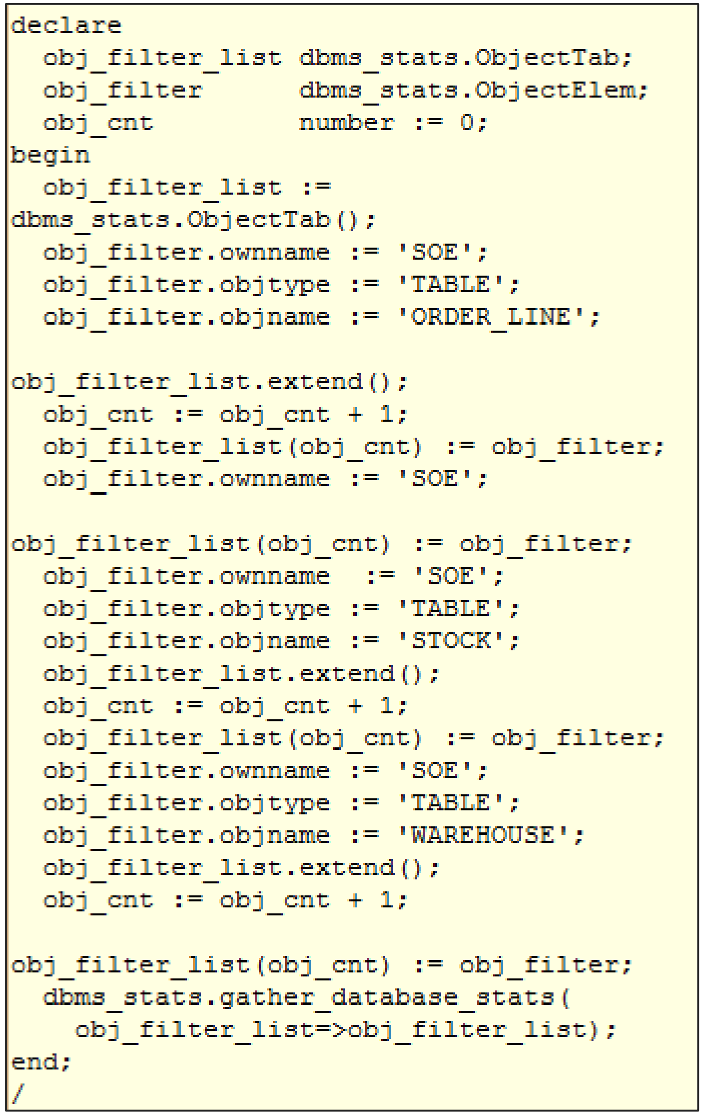
Appendix - Optimizer Statistics Advisor Script



```
-- Script generated for the recommendations from execution EXEC_989
-- in the statistics advisor task STAT_ADVICE
-- Script version 12.2
-- No scripts will be provided for the rule USEAUTOJOB.
       Please check the report for more details.
-- No scripts will be provided for the rule COMPLETEAUTOJOB.
-- No scripts will be provided for the rule MAINTAINSTATSHISTORY.

...cut for brevity...

-- Scripts for rule USECONCURRENT
-- Rule Description: Use Concurrent preference for Statistics Collection
-- Scripts for rule USEDEFAULTPREFERENCE
-- Rule Description: Use Default Preference for Stats Collection
-- Scripts for rule USEDEFAULTOBJECTPREFERENCE
-- Rule Description: Use Default Object Preference for statistics collection
-- Setting object-level preferences to default values
-- setting CASCADE to default value for object level preference
-- setting ESTIMATE_PERCENT to default value for object level preference
-- setting METHOD_OPT to default value for object level preference
-- setting GRANULARITY to default value for object level preference
-- setting NO_INVALIDATE to default value for object level preference

-- Scripts for rule USEINCREMENTAL
-- Rule Description:
--    Statistics should be maintained incrementally when it is beneficial

begin dbms_stats.set_table_prefs('SH', 'COSTS', 'INCREMENTAL', 'TRUE'); end;
/
begin dbms_stats.set_table_prefs('SH','SALES', 'INCREMENTAL', 'TRUE'); end;
/
```



```
declare
  obj_filter_list dbms_stats.ObjectTab;
  obj_filter
dbms_stats.ObjectElem;
  obj_cnt         number := 0;
begin
  obj_filter_list(obj_cnt) := obj_filter;
  obj_filter.ownname := 'SH';
  obj_filter.objtype := 'TABLE';
  obj_filter.objname := 'PROMOTIONS';
  obj_filter_list.extend();
  obj_cnt := obj_cnt + 1;
  obj_filter.ownname := 'SOE';
  obj_filter.objtype := 'TABLE';
  obj_filter.objname := 'CUSTOMER';
  obj_filter_list.extend();
  obj_cnt := obj_cnt + 1;
  obj_filter_list(obj_cnt) := obj_filter;
  obj_filter.ownname := 'SOE';
  obj_filter.objtype := 'TABLE';
  obj_filter.objname := 'DISTRICT';
  obj_filter_list.extend();
  obj_filter_list(obj_cnt) := obj_filter;
  obj_filter.ownname := 'SOE';
  obj_filter.objtype := 'TABLE';
  obj_filter.objname := 'ITEM';
  obj_filter_list.extend();
  obj_cnt := obj_cnt + 1;
  obj_filter_list(obj_cnt) := obj_filter;
  dbms_stats.gather_database_stats(
    obj_filter_list=>obj_filter_list);
end;
/
```



```
declare
  obj_filter_list dbms_stats.ObjectTab;
  obj_filter      dbms_stats.ObjectElem;
  obj_cnt         number := 0;
begin
  obj_filter_list :=
dbms_stats.ObjectTab();
  obj_filter.ownname := 'SOE';
  obj_filter.objtype := 'TABLE';
  obj_filter.objname := 'ORDER_LINE';

obj_filter_list.extend();
  obj_cnt := obj_cnt + 1;
  obj_filter_list(obj_cnt) := obj_filter;
  obj_filter.ownname := 'SOE';

obj_filter_list(obj_cnt) := obj_filter;
  obj_filter.ownname  := 'SOE';
  obj_filter.objtype := 'TABLE';
  obj_filter.objname := 'STOCK';
  obj_filter_list.extend();
  obj_cnt := obj_cnt + 1;
  obj_filter_list(obj_cnt) := obj_filter;
  obj_filter.ownname := 'SOE';
  obj_filter.objtype := 'TABLE';
  obj_filter.objname := 'WAREHOUSE';
  obj_filter_list.extend();
  obj_cnt := obj_cnt + 1;

obj_filter_list(obj_cnt) := obj_filter;
  dbms_stats.gather_database_stats(
    obj_filter_list=>obj_filter_list);
end;
/
```