

PERFORMANCE FEATURES IN 11G

Introduction

This paper is to explore various new features specific to performance introduced Oracle version 11g. Scripts are provided, inline, wherever possible for reproducing test cases.

This paper is NOT designed as a step by step approach, rather designed as a guideline. Every effort has been taken to reproduce the test results. It is possible for the test results to differ slightly due to version/platform differences.

1. (True) Online index rebuild

Non-online rebuild locks the table causing application downtime. Online index rebuild was introduced to reduce downtime during index rebuild and to alleviate this issue. But, still online index rebuild, until version 11g, did not completely eliminate application downtime. Let's consider a simple test case in version 10g.

Version 10g:

REM First creating a table and inserting 100K rows and adding an index on column n1.

```
Create table t1 (n1 number, v1 varchar2(1024) );
Insert into t1 select n1 , lpad (n1, 1000,'x') from
(select level n1 from dual connect by level <=100001);
Create index t1_n1 on t1(n1);
```

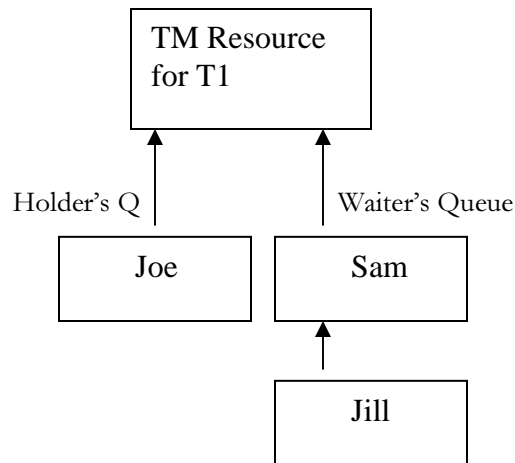
timeline	Joe	Jill	Sam (DBA)
T	insert into t1 values (100010,'x'); 1 row created.		
t + t1			alter index t1_n1 rebuild online; [This session waits for TM lock in Share mode].
t + t2		insert into t1 values (100011,'x'); [This session waits for TM lock in row-share mode]	
t + t3	insert into t1 values (100011,'x'); 1 row created.	Session still waiting For lock.	Session still waiting for table level lock in Share mode.
t + t4	Commit;		Command successful
t + t5		1 row created.	

In the table above, Joe and Jill are application users. These sessions are performing DML activity on table t1. Sam is the DBA and trying to perform online index rebuild operation.

Between time lines, $t + t2$ and $t + t4$ Jill is in a waiting state and can not proceed. Users are stuck.

Due to queueing mechanism employed in a typical locking scenario, Sam's session is waiting to acquire table level lock in share mode. But, Table level lock is held in row-share mode by Joe's session and so Sam must wait for Joe to commit. Jill's session meanwhile tried to insert a row in to that table t1. But, insert needs table level lock (TM type) in row-share mode. So, Jill joins the waiter queue behind Sam.

At this time, Jill can not proceed until Sam completes rebuild. Sam can not proceed until Joe commits her session. Essentially, application downtime has been introduced.



Sessions are waiting for TM locks on that table. ID1 is object_id 69663, which is table T1.

SESS	ID1	ID2	LMODE	REQUEST	TY	
Holder: 170	69663	0	3	0	TM	<Joe>
waiter: 542	69663	0	2	4	TM	<Sam>
waiter: 1262	69663	0	0	3	TM	<Jill>

Version 11g:

Version 11g introduces true online index rebuild. Index rebuild waits for transaction(s) to complete, instead of, transactions queuing behind session creating index. A true online rebuild as application does not wait.

But, all pending transactions on that table must complete before rebuild can be successful.

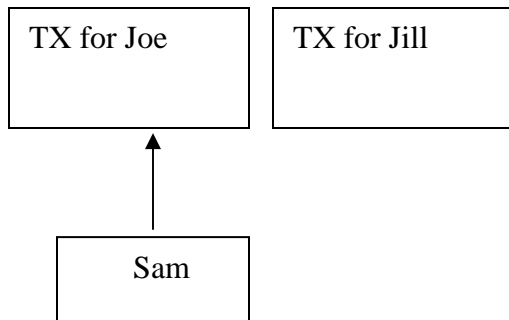
In the table below, Sam session is not trying to acquire a table level lock. That session is waiting to confirm all active transactions on table t1 is committed before completing online rebuild operation. At time $t + t3$, Joe's session committed, but Jill's transaction still active. So, Sam's session will wait for Jill's transaction to complete.

Essentially, Sam's session rebuilding that index did not lock out application users.

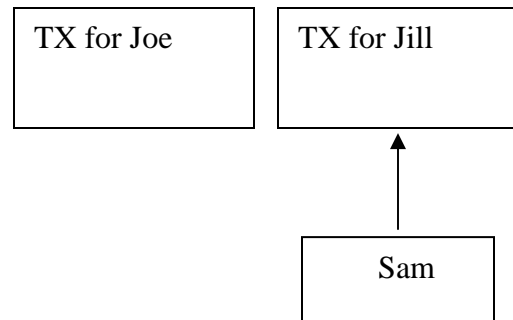
timeline	Joe	Jill	Sam (DBA)
t	insert into t1 values (100010,'x'); 1 row created.		
t + t1			alter index t1_n1 rebuild online; [This session waits for TX lock in Share mode].
t + t2		insert into t1 values (100011,'x'); 1 row created	
t + t3	Commit;		
t + t4			Session waits for Jill transaction to complete.
t + t5		Commit;	Online rebuild complete.

Following pictures depicts that Sam's session is waiting for Transactions, not TM table level lock.

At t + t2:



At t + t4



Even two sessions can concurrently rebuild. In the following locking scenarios sessions 164 and 121 are rebuilding two different indices concurrently. These two sessions are waiting for pending transactions to complete avoiding application down time.

```

SESS          INST          ID1          ID2 LMODE REQUEST TY
-----
Holder: 129   1          589853       3745 6      0 TX
waiter: 164  1          589853       3745 0      4 TX ← rebuild t1_n1
waiter: 121  1          589853       3745 0      4 TX ← rebuild t1_n2

```

Create index:

In addition to, online index rebuild, “create index online” operation does not acquire locks either.

For example, these two commands can be executed concurrently with out affecting DML on t1 table.

```

alter index t1_n1 rebuild online;
create index t1_n2 on t1(n2) online

```

DDL concurrency:

In release 10g, TM level locks were also used to control DDL concurrency. For example, while index rebuild is under way, table shouldn't be dropped. How is that controlled in 11g?

For DDL concurrency, internally, a new type of lock type has been introduced.

```
Sess #1: alter index t1_n1 rebuild online;
```

```
Sess #2: alter index t1_n1 rebuild online;
```

```
ERROR at line 1:
```

```
ORA-08104: this index object 72143 is being online built or rebuilt.
```

This new lock type OD with id1 as object_id has been introduced for DDL concurrency. In the following locking scenario, lock on table is acquired in share mode and an exclusive OD type lock was acquired on index.

```
SQL> select sid, type, id1, id2, lmode, request from v$lock where sid=164;
  SID TY          ID1          ID2 LMODE REQUEST
-----
  164 OD          72143          0    6      0 ← For index exclusive mode
  164 OD          72142          0    4      0 ← For table Shared mode
```

2. Invisible indices

There is an inherent risk in adding a new index to a database. Few issues to consider:

- (i) We can't be sure whether that index will be cheaper so that Cost based optimizer will prefer it, at least for the SQL we are trying to tune.
- (ii) Neither can we be sure that some execution plan can go haywire due to the presence of new index.

Oracle version 11g introduces a new feature known as invisible index. A new index can be created with less risk. These indices are not visible to the optimizer and optimizer can not select these invisible indices in the execution plan.

This new feature enables us to add an invisible index, test to see whether SQLs are using this index and also can be tested to see if any other SQL will be negatively affected by testing execution plan.

In the example below, a new index t1_n1 is created with 'invisible' property and subsequent plan does not use the index as shown in the execution plan.

```
SQL> create index t1_n1 on t1(n1) invisible;
Index created
```

```
SQL> explain plan for select count(*) from t1 where n1=:b1;
Explained.
```

```
SQL> select * from table(dbms_xplan.display);
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	5	4020 (1)	00:00:49
1	SORT AGGREGATE		1	5		
* 2	TABLE ACCESS FULL	T1	1	5	4020 (1)	00:00:49

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

```
2 - filter("N1"=TO_NUMBER(:B1))
```

Following SQL changes property of this index to visible state and optimizer chooses that index in the execution plan.

```
alter index t1_n1 visible;
Index altered
```

```
SQL> explain plan for select count(*) from t1 where n1=:b1;
Explained.
SQL> select * from table(dbms_xplan.display);
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	5	1 (0)	00:00:01
1	SORT AGGREGATE		1	5		
* 2	INDEX RANGE SCAN	T1_N1	1	5	1 (0)	00:00:01

```
Predicate Information (identified by operation id):
  2 - access("N1"=TO_NUMBER(:B1))
```

Optimizer trace 10053 shows that index is marked as unusable. This does not mean that index is unusable, it just that Optimizer considers that index as unusable and can not choose that index.

```
*****
BASE STATISTICAL INFORMATION
*****
Table Stats::
  Table: T1 Alias: T1 (Using composite stats)
  #Rows: 100001 #Blks: 1461 AvgRowLen: 511.00
Index Stats::
  Index: T1_N1 Col#: 1
  USING COMPOSITE STATS
  LVLS: 1 #LB: 225 #DK: 100001 LB/K: 1.00 DB/K: 1.00 CLUF: 7145.00
  UNUSABLE
  Index: T1_N2 Col#: 2
  USING COMPOSITE STATS
  LVLS: 1 #LB: 196 #DK: 100 LB/K: 1.00 DB/K: 1000.00 CLUF: 100001.00
  Index: T1_N3 Col#: 4
  USING COMPOSITE STATS
  LVLS: 1 #LB: 34 #DK: 10 LB/K: 3.00 DB/K: 10.00 CLUF: 100.00
```

Parameter optimizer_use_invisible_indexes can be used to control this behaviour at session level. So, this can be set at session level and various SQLs can be tested to see behaviour of those SQL with new index. SQL access advisor uses this method to measure the impact of new invisible indices if enabled.

3. Virtual columns

Usual way to tune this query of this form is to create function based indices.

```
Select * from emp where upper(employee_name) = :b1
create index fb_i1 on emp (upper (employee_number));
```

Oracle version 11g introduces virtual column

```
create table emp (
  emp_id number,
  emp_name varchar2(30),
  emp_name_upper varchar2(30)
  generated always as ( upper(emp_name) )
);
```

Emp_name_upper in the table emp is a virtual column. Virtual columns are not stored in the table explicitly though. Virtual columns are “calculated”, every time column is accessed. An index can be created on virtual column, which almost acts like a function based index. And, importantly, values are stored in the index.

```
create index emp_i1 on emp (emp_name_upper);
```

Expression for that index definition behaves like a function based index too.

```
select column_expression from user_ind_expressions
where table_name='EMP';
```

```

COLUMN_EXPRESSION
-----
"CBQT"."F_UPPER"("EMP_NAME")

```

Test case:

Let's create a function that consumes 5 seconds of CPU for each call.

```

CREATE OR REPLACE function f_upper(v_emp_name in varchar2)
return varchar2 deterministic
is
  v1 number;
  v2 char(32);
begin
  select count(*) into v1 from kill_cpu
  connect by n > prior n
  start with n = 1;
  v2:= upper(v_emp_name);
  return (v2);
end;
/

```

Let's create a table with virtual column calling that function.

```

create table emp (
  emp_id number, emp_name varchar2(32),
  emp_name_upper generated always as
  (f_upper ( emp_name ) ) );

```

Describing that table and emp_name_upper is of size varchar2 (4000).

```

SQL> desc emp
Name                               Null?    Type
-----
EMP_ID                              NUMBER
EMP_NAME                            VARCHAR2(32)
EMP_NAME_UPPER                      VARCHAR2(4000)

```

Let's create an index

```
create index emp_f1 on emp (emp_name_upper);
```

Now, predicates specifying virtual column will use index. Function calls avoided at run time.

```

select * from emp e where e.emp_name_upper like 'I_%'

```

EMP_ID	EMP_NAME	EMP_NAME_UPPER
20	ICOL\$	ICOL\$
46	I_USER1	I_USER1

```

6 rows selected.
Elapsed: 00:00:00.00

```

Virtual columns and partitioning

One of the advantages of virtual column is that table can be partitioned using a virtual column.

```
create table emp (  
    emp_id number, emp_name varchar2(32),  
    emp_name_upper generated always as  
        (upper ( emp_name) )  
)  
partition by range ( emp_name_upper)  
( partition p1 values less than ('C'),  
  partition p2 values less than ('G'),  
  partition p3 values less than ('J'),  
  partition p4 values less than ('N'),  
  partition p5 values less than ('Q'),  
  partition p6 values less than ('W'),  
  partition pmax values less than (maxvalue)  
)  
/
```

Partition pruning can be done on virtual column. For emp_name_upper column predicate 'Adam' applied and pruning took place, only partition 1 is accessed as indicated by pstart and pstop columns in the plan below.

Explain plan for select * from ep where emp_name_upper='Adam';

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Pstart	Pstop
0	SELECT STATEMENT		1	14	3 (0)		
1	PARTITION RANGE SINGLE		1	14	3 (0)	1	1
* 2	TABLE ACCESS FULL	EMP	1	14	3 (0)	1	1

Predicate Information (identified by operation id):

```
2 - filter("EMP_NAME_UPPER"='Adam')
```

Parameter _replace_virtual_columns parameter controls this behavior. By setting this parameter to false, partition pruning did not take place.

```
alter session set "_replace_virtual_columns"=false;
```

```
explain plan for select * from emp where upper(emp_name) ='Adam';
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Pstart	Pstop
0	SELECT STATEMENT		1	15	6 (0)		
1	PARTITION RANGE ALL		1	15	6 (0)	1	7
* 2	TABLE ACCESS FULL	EMP	1	15	6 (0)	1	7

Predicate Information (identified by operation id):

```
2 - filter(UPPER("EMP_NAME")='Adam')
```

Parameter _trace_virtual_columns parameter can be used to enable trace of virtual columns.

```
alter session set "_trace_virtual_columns"=true;
```

```
***** BEGIN FIRST PHASE VC REPLACEMENT (kkmpqag) *****  
***** Before transformation heap size 204 *****  
***** BEGIN : Final replacement chain *****
```

¹ Of course, underscore parameters are unsupported by Oracle. Before setting these parameters in production databases, Oracle support need to be contacted.

```

** Mark NOFETCH [2023e228] column EMP_NAME flg[4000020] fl2[1000]
fl3[1080] **
***** Address replaced [0x2023e120] newcolP [0x202312ec]
flg[0x4020000]

Source Operand [WHERE CLAUSE EXPRESSION] [0x2023e1b4] ---->
UPPER("EMP"."EMP_NAME")

Target Operand ---->
"EMP"."EMP_NAME_UPPER"
...

```

There are couple of advantages with virtual columns compared to older function based indices.

1. Design is cleaner as this is a column.
2. Column can be used as a partitioning key.
3. Histograms can be generated on virtual columns increasing usability of these columns.
4. Virtual columns are not explicitly stored reducing storage requirements.

Further, few new features seems to be based on virtual columns and one such feature is 'extended statistics'.

4. LOB enhancements

Oracle version 11g introduces new type of LOB columns known as securefile lobs. Older type of lob is known as basicfile lobs. Internal implementation of securefile is much different from basicfile LOBs.

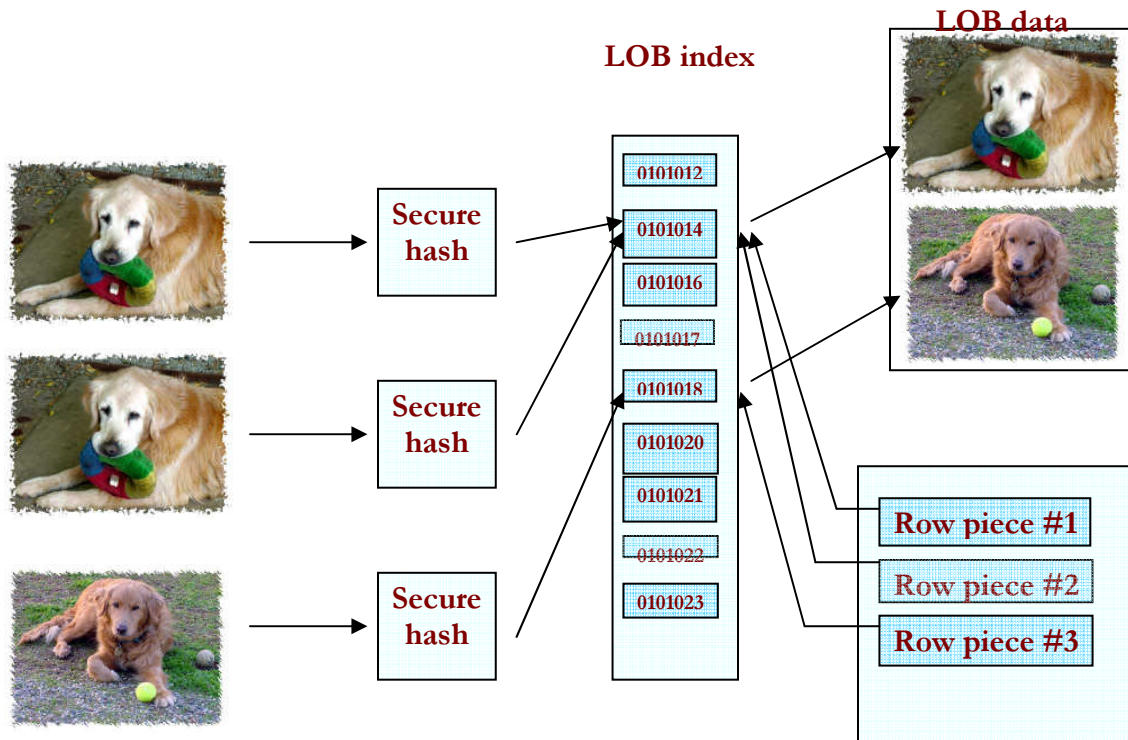
This new type of LOB supports important new features:

1. Deduplication
2. Encrypt.
3. Compression

Deduplication:

In a social networking websites, I had an opportunity to consult with, default images for various portals are stored explicitly wasting precious storage space. These images are stored as LOB columns and can be customizable by portal user. But, initially, default images are stored explicitly. This unnecessarily introduces additional processing and also wastes valuable space.

New LOB column type, securefile lobs, resolves part of this problem. Duplicate values for LOB columns are not stored explicitly using this feature. In the social networking website discussed earlier, we don't need to store default images explicitly and better yet, there is no code change at all to achieve reduction in storage space.



Let's look at operational implementation of this feature. Refer to the picture depicted above. Images are hashed using secure hash algorithm converting the image to a hash key. This hash key value is looked up in LOB index for that LOB column. If that hash key exists already, then images need not be stored explicitly and new row piece will be updated referring to the existing images. Of course, reference count seems to be maintained too.

Let's consider three insert statements inserting three puppy images above. Two images are exactly the same.

1. First insert statement inserts an image in to the table, with image of the puppy #1.
2. Internal algorithm converts that image in to a hash key using a secure hashing algorithm.
3. This hash key is looked up in the LOB index to find if that hash key already exists. If it exists, then this image does not need to be stored again (explicitly).
4. In this specific case, this is the first image and so image is stored explicitly.
5. Next insert statement inserts same image in to that table, with the image of puppy #1.
6. Hashing generates hash key and LOB index lookup shows that this hash key already exists. So, this image need not be stored explicitly as it is already in the database. Row piece #2 is inserted with pointer to this existing image.
7. Next insert statement inserts a different image generating a different hash key. So, this puppy #2 image must be stored explicitly.'

Test case (with unique images):

In this test case, every CLOB column value is unique [a boundary condition really] and so deduplication didn't have much effect. Size of that LOB segment is 88MB.

```

create table t1 (n1 number, c1 clob)
      lob(c1) store as securefile (deduplicate );
set timing on

insert into t1 select n1 , lpad(n1, 8192, 'x') from
      (select level n1 from dual connect by level <=10000);

10000 rows created.
Elapsed: 00:01:44.40

select segment_name, bytes/1024/1024 from user_segments where
      segment_name in
      (select segment_name from user_lobs where table_name='T1' );

```

SEGMENT_NAME	BYTES/1024/1024
SYS_LOB0000072244C00002\$\$	88

In the following test case every LOB column value is exactly the same, another boundary condition.

Test case (Same image):

```

create table t1 (n1 number, c1 clob)
      lob(c1) store as securefile (deduplicate );

set timing on
insert into t1 select n1 , lpad(1, 8192, 'x') from
      (select level n1 from dual connect by level <=10000);
10000 rows created.
Elapsed: 00:00:25.10

select segment_name, bytes/1024/1024 from user_segments where
      segment_name in
      (select segment_name from user_lobs where table_name='T1' );

```

SEGMENT_NAME	BYTES/1024/1024
SYS_LOB0000072244C00002\$\$.3125

From the test case above, when all LOB columns values are exactly the same size of LOB columns reduced to 0.3125MB. This is because Duplicate values are not stored explicitly.

Compression:

11g uses industry standard algorithm to compress LOB values. If images are already compressed or if compression doesn't provide any size reduction, then code defaults column value back to nocompress [12].

Test Case

```

create table t1 (n1 number, c1 clob)
      lob(c1) store as securefile (compress);
set timing on

insert into t1 select n1 , lpad(n1, 8192, 'x') from
      (select level n1 from dual connect by level <=10000);
10000 rows created.
Elapsed: 00:00:09.70

select segment_name, bytes/1024/1024 from user_segments where
      segment_name in
      (select segment_name from user_lobs where table_name='T1' );

```

SEGMENT_NAME	BYTES/1024/1024
SYS_LOB0000072359C00002\$\$.125

In the test case above, 10000 rows were inserted with compress on. With just compression, LOB column values consumed just 0.125MB, compared to 88MB of nocompress case. Combination of compression and de-duplication will reduce this space usage further.

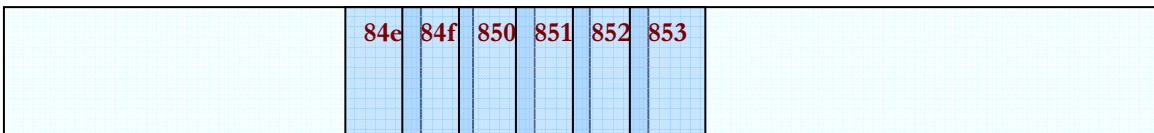
LOB writes:

LOBs are written much differently in 11g compared to 10g. In version 10g, Session inserting rows also writes LOB data block using ‘direct path’ writes, bypassing SGA. But, in version 11g, LOBs are gathered in a memory area of size 4MB (Write Gather Cache WGC) and written by DBWR. This improves performance a) as sessions are freed from waiting for I/O b) DBWR writes are contiguous and reduces disk seek time.

Parameter `_kdlw_enable_write_gathering` enables WGC. One WGC is allocated per transaction² [12]. WGC is flushed when 4MB is full or at the end of a transaction. Parameter `_kdlwp_flush_threshold` seems to control this behavior.

As DBWR writes blocks in a sequential fashion avoiding costly disk seek times. Here is the redo block dumps from redo log files in a test case.

```
CHANGE #1 MEDIA RECOVERY MARKER SCN:0x0000.00000000 SEQ: 0 OP:23.1
Block written - afn: 3 rdba: 0x00c0084e BFT:(1024,12585038) non-BFT:(3,2126)
                  scn: 0x0000.00176a9b seq: 0x01 flg:0x04
Block written - afn: 3 rdba: 0x00c0084f BFT:(1024,12585039) non-BFT:(3,2127)
                  scn: 0x0000.00176aa0 seq: 0x01 flg:0x04
Block written - afn: 3 rdba: 0x00c00850 BFT:(1024,12585040) non-BFT:(3,2128)
                  scn: 0x0000.00176aa1 seq: 0x01 flg:0x04
Block written - afn: 3 rdba: 0x00c00851 BFT:(1024,12585041) non-BFT:(3,2129)
                  scn: 0x0000.00176aa4 seq: 0x05 flg:0x04
Block written - afn: 3 rdba: 0x00c00852 BFT:(1024,12585042) non-BFT:(3,2130)
                  scn: 0x0000.00176aa2 seq: 0x01 flg:0x04
Block written - afn: 3 rdba: 0x00c00853 BFT:(1024,12585043) non-BFT:(3,2131)
                  scn: 0x0000.00176aa5 seq: 0x01 flg:0x04
Block written - afn: 3 rdba: 0x00c00854 BFT:(1024,12585044) non-BFT:(3,2132)
                  scn: 0x0000.00176aa8 seq: 0x05 flg:0x04
```



Following trace lines also shows the differences between 10g and 11g. In 11g, there are no waits for direct path writes. But, there are waits for ‘direct path reads’ since this test case is for deduplication and values had to be read for every column to compare.

basicfile:

Event waited on	Times Waited	Max. wait	Total waited
db file sequential read	24	0.05	0.29
direct path write	12000	0.00	0.66

securefile:

Event waited on	Times Waited	Max. wait	Total waited
db file sequential read	63	0.01	0.11
direct path read	12000	0.06	3.18

² Author is unable to confirm this. This is copied from [12].

5. Extended Statistics

Cost based optimizer assumes no correlation between columns by default and this has the effect of reducing cardinality of a row source, erroneously. Incorrect cardinality estimates are one of the many root causes of SQL performance issues.

Consider the following test case:

REM There are strong correlation between columns n1,n2 and n3.

REM

```
create table t_vc as
  select
    mod(n, 100) n1, mod(n, 100) n2 , /* For 100% of rows n1=n2 */
    mod(n, 50) n3 , /* For 50% of rows n1=n3 */
    mod(n, 20) n4 /* For 20% of rows n1=n4 */
  from
    (select level n from dual
     connect by level <= 10001);
```

REM Collecting stats with histograms for all columns

```
begin
  dbms_stats.gather_Table_stats(
    user, 'T_VC',
    estimate_percent => null,
    method_opt => 'for all columns size 254');
end;
/
```

Let's review a SQL querying with n1=10 predicate. There are 100 rows in that table with the value n1=10 and Optimizer row estimate is also 100. This is good as estimate is matching with actual row count.

explain plan for select count(*) from t_vc where n1=10;

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	3	9 (0)	00:00:01
1	SORT AGGREGATE		1	3		
* 2	TABLE ACCESS FULL	T_VC	100	300	9 (0)	00:00:01

Now, let's add another predicate n2=10 to the SQL above. As per our data conditions n1=n2 for all rows and so, n1=10 and n2=10 should still return exactly 100 rows.

explain plan for select count(*) from t_vc where n1=10 and n2=10;

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	6	9 (0)	00:00:01
1	SORT AGGREGATE		1	6		
* 2	TABLE ACCESS FULL	T_VC	1	6	9 (0)	00:00:01

But optimizer row estimate is 1.

Why? Cost based optimizer assumes that predicates are independent. Let's look at arithmetic behind this.

selectivity of n1 is 1/100 (n1 has 100 distinct values)
selectivity of n2 is 1/100 (n2 has 100 distinct values)

selectivity of (n1=10 and n2 =10) is (1/100) * (1/100).

So, cardinality estimates for n1=10 and n2=10
 = num_rows * (1/100) * (1/100)
 = 10000 * (1/100) * (1/100)
 = 1

In version 11g, Oracle introduces extended statistics feature to combat this potential issue.

Version 11g extended stats:

First, let's create extended stats using dbms_stats package.

```
SELECT dbms_stats.create_extended_stats(
        ownname=>user, tabname => 'T_VC',
        extension => '(n1, n2)' ) AS n1_n2_correlation
FROM dual;

N1_N2_Correlation
-----
SYS_STUBZH0IHA7K$KEBJVX05LOHAS
```

Now, let's collect statistics on this table.

```
begin
  dbms_stats.gather_Table_stats( user, 'T_VC',
    estimate_percent => null,
    method_opt => 'for all columns size 254');
end;
```

Verifying cardinality estimates for this SQL.

explain plan for select count(*) from t_vc where n1=10 and n2=10;

```
-----
| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |
-----+-----+-----+-----+-----+-----+-----+
| 0  | SELECT STATEMENT |      |      |      | 1200 (9)| 00:00:01 |
|* 1 | TABLE ACCESS FULL | T_VC | 100  | 100  | 1200 (9)| 00:00:01 |
-----
```

Optimizer estimates for row cardinality is accurate at 100 after adding extended stats. So, optimizer realized that there is a strong correlation between these two columns.

Internals of Extended stats:

Adding extended stats adds a virtual column and an undocumented clause is used: 'BY USER for statistics'.

```
alter table T_VC add (SYS_STUBZH0IHA7K$KEBJVX05LOHAS
  as ( sys_op_combined_hash(n1, n2)) virtual BY USER for statistics);
```

Collecting histograms on all columns collects histograms on this column too. Using histogram buckets on this new virtual column, CBO is able to calculate correct selectivity.

```
SINGLE TABLE ACCESS PATH
Single Table Cardinality Estimation for T_VC[T_VC]

ColGroup (#1, VC) SYS_STUBZH0IHA7K$KEBJVX05LOHAS
  Col#: 1 2      <b> CorStregth: 100.00</b>
  ColGroup Usage:: PredCnt: 2  Matches Full: #0  Partial: Sel: 0.0100
  Table: T_VC Alias: T_VC
  Card: Original: 10001.000000  Rounded: 100  Computed: 100.00  Non Adjusted:
100.00
```

Function `sys_op_combined_hash` is the function. This function returns a hash value based upon combined values of arguments passed.³

6. Fine grained dependency

Prior to version 11g, any DDL change will invalidate all dependent objects. This affects performance since invalid objects need to be recompiled and in high concurrency environments causes 'library cache pin' and 'library cache lock' waits since parse locks need to be broken. For example, let us consider the following three objects.

```
create table t(a number);

REM Explicit column selection
create view v as select a from t;

REM Use of '*' instead of column names
create or replace procedure p1
is
    a1 number;
begin
    select * into a1 from t;
end;
/
REM Use of explicit column names.
create or replace procedure p2
is
    a1 number;
begin
    select a into a1 from t;
end;
/
```

In the test case above, only procedure p1 is badly written. Since '*' is used in the select statement, adding a column to table t will break this procedure p1 since variables are not defined to receive any additional columns. But other objects v and p2 need not be invalidated.

In Oracle version 10g, adding a column will mark all three objects v, p1 and p2. But, in Oracle version 11g, adding a column marks only p1 invalid and other objects are not affected.

```
select owner, object_name, status from dba_objects where object_name in
('T','V','P1','P2','T2')
```

OWNER	OBJECT_NAME	STATUS
SYS	P1	VALID
SYS	T	VALID
SYS	V	VALID
SYS	P2	VALID

```
REM adding a column
Alter table t add column ( b number);

REM only p1 is invalid.
```

³ Author does not have enough information about `sys_op_combined_hash` to show more detail.

```
select owner, object_name, status from dba_objects where object_name in
('T','V','P1','P2','T2')
```

OWNER	OBJECT_NAME	STATUS
SYS	P1	INVALID
SYS	T	VALID
SYS	V	VALID
SYS	P2	VALID

7. SQL Result cache

Prior to 10g, every query must be re-executed even if there is no change in the tables. This re-execution is unnecessary for few static tables and only caching using at client tools resolve this issue. But, in 11g, query results can be cached in SGA. Just to avoid confusion, this is much different from buffer cache caching database blocks. Result cache in addition to that.

Results are retrieved immediately from the result cache if there is no change to the underlying base tables. Frequently executed queries will see performance improvements.

There are two ways to use results cache:

- Manual mode setting `result_cache_mode` to manual. Only queries with `result_cache` hint will use result cache.
- Force mode setting `result_cache_mode` parameter to force. All queries will use the result cache.

In test case below, result cache is not in use. Queries were executed one after another. Same amount of consistent gets used

```
SQL> set autotrace traceonly stat
```

```
SQL> select count(n1) , count(n2) from
t1 where n2=10;
```

```
statistics
```

```
-----
0 db block gets
1011 consistent gets
0 physical reads
0 redo size
481 bytes sent via SQL*Net to client
416 bytes received via SQL*Net from
client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

```
SQL> select count(n1) , count(n2) from
t1 where n2=10;
```

```
statistics
```

```
-----
0 db block gets
1011 consistent gets
0 physical reads
0 redo size
481 bytes sent via SQL*Net to client
416 bytes received via SQL*Net from
client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

In the test case below result cache is turned on using an hint `/*+ result_cache */`. Second execution of same SQL consumed 0 consistent gets, since results were retrieved from result case without actually performing any work. As Cary Millsapp would say, best way to tune something is not do it in the first place.

```
SQL> set autotrace traceonly stat
```

```
SQL> select /*+ result_cache */
count(n1) , count(n2) from
t1 where n2=10;
```

Statistics

```
-----
0 db block gets
1011 consistent gets
0 physical reads
0 redo size
481 bytes sent via SQL*Net to client
416 bytes received via SQL*Net from
client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

```
SQL> select /*+ result_cache */
count(n1) , count(n2) from
t1 where n2=10;
```

Statistics

```
-----
0 db block gets
0 consistent gets
0 physical reads
0 redo size
481 bytes sent via SQL*Net to client
416 bytes received via SQL*Net from
client
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

8. Function Result cache

Similar to SQL result cache, function results can be cached too. Following test case illustrates this.

REM creating a function with result_cache keyword indicating that result_cache to be used.

```
create or replace function f1 (v_n1 number)
return number result_cache
is
l_sum_n2 number;
begin
select sum(n2) into l_sum_n2 from t1 where n1=v_n1;
return l_sum_n2;
end;
/
```

Following test case shows that there was 0 consistent gets for the second execution of function, which returned results from the result cache.

```
SQL> set autotrace traceonly stat
```

```
SQL >select f1(10 ) from dual;
```

Statistics

```
-----
27 recursive calls
0 db block gets
72 consistent gets
0 physical reads
0 redo size
415 bytes sent via SQL*Net to client
416 bytes received via SQL*Net from t
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

```
SQL >select f1(10 ) from dual;
```

Statistics

```
-----
0 recursive calls
0 db block gets
0 consistent gets
0 physical reads
0 redo size
415 bytes sent via SQL*Net to client
416 bytes received via SQL*Net from
2 SQL*Net roundtrips to/from client
0 sorts (memory)
0 sorts (disk)
1 rows processed
```

Summary:

In summary, there are many new features in 11g and will be very useful for performance tuning.

About the author

Riyaj Shamsudeen has 15+ years of experience in Oracle and 14+ years as an Oracle DBA/Oracle Applications DBA. He is the principal DBA behind ora!internals (<http://www.orainternals.com> - performance/recovery/EBS11i consulting company). He specializes in RAC, performance tuning and database internals and frequently blogs about them in <http://orainternals.wordpress.com>. He has authored many articles such as internals of locks, internals of hot backups, redo internals etc. He also teaches in community colleges in Dallas such as North lake college and El Centro College. He is a proud member of OakTable network.

References

1. Oracle support site. Metalink.oracle.com. numerous documents
2. Innovate faster with Oracle database 11g- An Oracle white paper
3. My blog: <http://orainternals.wordpress.com>
4. Oracle database 11g: An Oracle white paper:
5. Internal's guru Steve Adam's: <http://www.ixora.com.au>
6. Jonathan Lewis: <http://www.jlcomp.daemon.co.uk>
7. Julian Dyke: <http://www.julian-dyke.com>
8. Costing PL/SQL functions: Joze Senegachnick – HOTSOS 2006
9. Pythian blog: Query result cache: by Alex fatkulin:
<http://www.pythian.com/authors/fatkulin>
10. Metalink note : 453567.1 on result cache
11. Oracle database 11g: secure files : An Oracle white paper.
12. Securefile performance:
<http://www.oracle.com/technology/products/database/securefiles/pdf/securefilesperformancepaper.pdf>

Appendix #1: Environment details

Linux/Solaris 11g
Oracle version 10gR2 and 9iR2
Locally managed tablespaces
No ASM
No ASSM