
Performance specific new features in 11g

By
Riyaj Shamsudeen





your database maestros

www.pythian.com



Support Centers in :
North America | South America | Europe
Middle East | Asia | Australia

Managed services for:
Oracle | SQL Server | MySQL

Who am I?

- 16 years using Oracle products
- Over 15 years as Oracle DBA
- Certified DBA versions 7.0,7.3,8,8i &9i
- Specializes in performance tuning, Internals and E-business suite
- Currently working for The Pythian Group www.pythian.com
- OakTable member
- Email: [rshamsud at gmail.com](mailto:rshamsud@gmail.com)
- Blog : <http://orainternals.wordpress.com>



Disclaimer

These slides and materials represent the work and opinions of the author and do not constitute official positions of my current or past employer or any other organization. This material has been peer reviewed, but author assume no responsibility whatsoever for the test cases.

If you corrupt your databases by running my scripts, you are solely responsible for that.

This material should not should not be reproduced or used without the authors' written permission.

Agenda

- **(True) Online Index rebuild**
- **Invisible indices**
- **Virtual Columns**
- **LOB Performance improvements**
- **Compound triggers**
- **CBO – Extended statistics**
- **Fine Grained dependency**

Online index rebuild in 10g

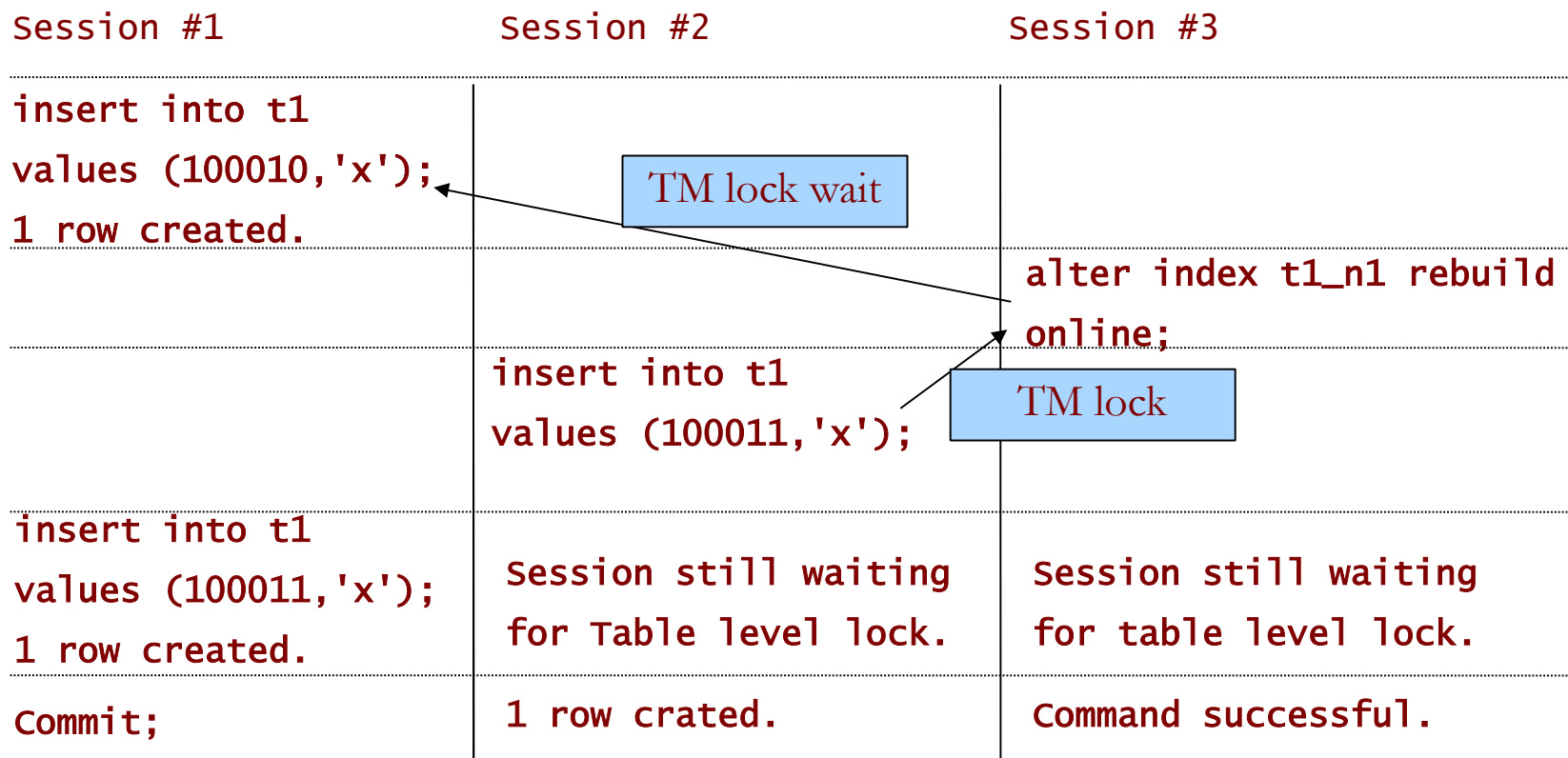
- In 10g, Online index rebuild still affects DML operations.
- Application can be locked out completely as completion of online index rebuild will acquire table level locks.
- It would be unwise for application to completely locked out to rebuild and index.

Online index rebuild

- In 11g introduces true online index rebuild.
 - Index rebuild waits for transaction(s) to complete, instead of transactions queueing behind index creation.
 - At last, a true online rebuild as application does not wait.
 - But, all pending transactions on that table must complete before rebuild can be successful.
-

Test case (In 10g):

```
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);
```



Locking behaviour in 10g

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

SESS	ID1	ID2	LMODE	REQUEST	TY
Holder: 170	69663	0	3	0	TM
Waiter: 542	69663	0	2	4	TM
Waiter: 1262	69663	0	0	3	TM

- Essentially, application is standstill, even though online index rebuild was tried.

Test case (In 11g):

```
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);
```

Session #1

Session #2

Session #3

```
insert into t1  
values (100010,'x');  
1 row created.
```

TX lock wait

```
alter index t1_n1 rebuild  
online;
```

```
insert into t1  
values (100011,'x');  
1 row created.
```

TX lock wait

```
insert into t1  
values (100011,'x');  
commit;
```

```
Commit;
```

Session still waiting
for Row level lock.

```
Command successful.
```

(True) Online index rebuild

- Rebuild actions acquires row level locks, not higher level locks as in prior releases.

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

- Even two simultaneous online rebuild operations allowed. Still application connections do not wait for rebuild.

(True) Online index rebuild

- Create index `..online` does not acquire locks affecting application DML either.

For example, these two commands can be concurrently executed without affecting DML on `t1` table.

```
alter index t1_n1 rebuild online;
```

```
create index t1_n2 on t1(n2) online
```

- Works great for bitmap indices too.

(True) Online index rebuild

- But in 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?

(True) Online index rebuild

- Internally, a new type of lock type introduced too.

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 acquired to control concurrency.

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

(True) Online index rebuild

- An index organized table is used to keep track of changes to the base table which are merged at the end.

```
SQL> desc cbqt.sys_journal_72143
```

Name	Null?	Type
-----	-----	-----
C0	NOT NULL	NUMBER
OPCODE		CHAR(1)
PARTNO		NUMBER
RID	NOT NULL	ROWID

- Unfortunately, rebuild session waits even if another column, not part of that index is updated.

```
Session #1: SQL> update t1 set n2=n2 where n2=100001;
```

```
1 row updated.
```

```
Session #2: Alter index t1_n1 rebuild online;
```

```
--waits even though n1 is not in t1_n1 index.
```

Agenda

- **(True) Online Index rebuild**
- **Invisible indices**
- **Virtual Columns**
- **LOB Performance improvements**
- **Compound triggers**
- **CBO – Extended statistics**
- **Fine Grained dependency**

Question

- How many of you think that adding an index will not affect performance?

It can affect performance of queries in a production environment for many reasons:

1. CBO's poor chose of plan
 2. Inefficient indexing strategy etc
-

Invisible indices

- 11g introduces invisible indices.
- These indices are not visible to the optimizer and so optimizer can't choose that index.
- No, they aren't in their after life, just not visible to the optimizer.
- Parameter `optimizer_use_invisible_indexes` can be used to alter this behaviour. Default value is false.

Invisible indices

```
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))
```

Invisible indices

```
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))
```

Invisible indices

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

10053 shows that CBO considers that index same as unusable index.

Agenda

- (True) Online Index rebuild
- Invisible indices
- Virtual Columns
- LOB Performance improvements
- Compound triggers
- CBO – Extended statistics
- Fine Grained dependency

Question ?

- How do you tune query of this form ?

```
select * from emp
where upper(employee_name) = :b1
```

- Usual way is to create function based index.

```
Create index fb_i1 on emp
        upper (employee_number);
```

Virtual columns

- 11g introduces virtual columns:

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

- But column `emp_name_upper` values are not stored in the database and “calculated” every time, column is accessed.
-

Virtual columns

- 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.

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

```
COLUMN_EXPRESSION
```

```
-----
```

```
"CBQT"."F_UPPER"("EMP_NAME")
```

Virtual columns – 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
```

Function must be 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;
```

```
/
```

Just a CPU consumer.

Thanks to Jonathan Lewis.

Virtual columns – Test case

- 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.

```
SQL> desc emp
```

Name	Null?	Type
-----	-----	
EMP_ID		NUMBER
EMP_NAME		VARCHAR2(32)
EMP_NAME_UPPER		VARCHAR2
(4000)		

Default return type of a function with varchar2 is varchar2(4000).

Virtual columns – Test case

- Predicates specifying virtual column will call the function to calculate column value, for each row.

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

```
EMP_ID EMP_NAME EMP_NAME_UPPER
-----
20 ICOL$ ICOL$
```

...

6 rows selected.

Elapsed: 00:00:52.06

- tkprof shows function calls 11 times.

```
SELECT COUNT(*) FROM KILL_CPU CONNECT BY N > PRIOR N START WITH N = 1
```

call	count	cpu	elapsed	disk	query	current	rows
...							
Fetch	11	59.39	59.81	0	44	0	11
total	23	59.40	59.81	0	44	0	11

Virtual columns – Test case

- 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.

```
1* 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

Id	Operation	Name	Rows
0	SELECT STATEMENT		5
1	TABLE ACCESS BY INDEX ROWID	EMP	5
* 2	INDEX RANGE SCAN	EMP_F1	5

Virtual columns – Test case

- If the function call is costlier, it is almost important to use index based access to avoid costly function calls.

```
SQL> select * from emp where emp_name_upper like '%';
```

```
Elapsed: 00:00:57.59
```

```
-----
| Id | Operation          | Name | Rows | Bytes | Cost (%CPU)| Time     |
-----
|  0 | SELECT STATEMENT   |      |  10 |  430 |      3 (0) | 00:00:01 |
SQL> select TABL|E+ACC|ESS(emp)|*EMP from emp where emp_name_upper like '%';
Elapsed: 00:00:00.01
-----
```

```
-----
| Id | Operation          | Name | Rows | Bytes | Cost (%CPU)| Time     |
-----
|  0 | SELECT STATEMENT   |      |  10 |  430 |      2 (0) | 00:00:01 |
|  1 | TABLE ACCESS BY INDEX ROWID| EMP  |  10 |  430 |      2 (0) | 00:00:01 |
|*  2 | INDEX FULL SCAN    | EMP_F1 |  10 |      |      1 (0) | 00:00:01 |
-----
```

Virtual columns – Test case

- Associating statistics with this function with high I/O cost increases cost estimation. Still, access is with full table function:

```
associate statistics with functions f_upper default cost (3000,1000,0);
select * from emp where emp_name_upper like '%';
Elapsed: 00:00:57.59
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		10	430	10003 (0)	00:02:01
* 1	TABLE ACCESS FULL	EMP	10	430	10003 (0)	00:02:01

Virtual columns – event 10053

SINGLE TABLE ACCESS PATH

Single Table Cardinality Estimation for EMP[EMP]

Default costs for function F_UPPER CPU: 3000, I/O: 1000

***** Virtual column Adjustment *****

Column name EMP_NAME_UPPER

cost_cpu 3100.00

cost_io 1000.00

***** End virtual column Adjustment *****

Column (#3):

NewDensity:0.100000, OldDensity:0.050000 BktCnt:10, PopBktCnt:0, PopValCnt:0, NDV:10

Table: EMP Alias: EMP

Card: Original: 10.000000 Rounded: 10 Computed: 10.00 Non Adjusted: 10.00

Access Path: TableScan

Cost: 10003.00 Resp: 10003.00 Degree: 0

Cost_io: 10003.00 Cost_cpu: 68307

Resp_io: 10003.00 Resp_cpu: 68307

Best:: AccessPath: TableScan

~~Cost: 10003.00 Degree: 1 Resp: 10003.00 Card: 10.00 Bytes: 0~~



Index access is not even considered!

Agenda

- **(True) Online Index rebuild**
- **Invisible indices**
- **Virtual Columns**
- **LOB Performance improvements**
- **Compound triggers**
- **CBO – Extended statistics**
- **Fine Grained dependency**

LOB enhancements

- **New type LOB introduced: securefile lobs.**
- **Older lob types are now known as basicfile lob.**
- **Internal implementation of securefile is very different from basicfile lobs.**

LOB enhancements

- This new type of lob supports few important features such as deduplicate and encrypt.
- An internal hash algorithm converts LOB value to a hash value.
- Before storing a LOB value, RDBMS code checks if that lob value stored in that table already.
- LOB index stores hash value and quick search on lob index identifies existence of LOB value.
- Of course, classic space vs time optimization.

LOB enhancements - deduplicate

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

Few keywords:
securefile, deduplicate

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

Every row LOB value
is unique

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

Size : 88MB

LOB enhancements - deduplicate

```
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);
```

All lob values
are same

```
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

Size is very
small

LOB enhancements - basicfile

```
create table t1 (n1 number, c1 clob)
      lob(c1) store as basicfile;
```

```
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: 00:00:12.28

```
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_LOB0000072356C00002\$\$	80

For basic files
no deduplication

LOB enhancements

- In prior slides, basic file uses 80MB and securefile uses 88MB for non-deduplicate case.
- That's a bug : 6698457 fixed in 11.2

LOB enhancements - compression

- **LOBs can be compressed and stored too.**
- **11g uses industry standard algorithms to compress lobs.**
- **If LOB values are already compressed or if doesn't provide any compression, defaults back to nocompress.**

LOB enhancements - compression

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

Text values, better
compression ratio.

```
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

Size is just .125KB

LOB writes

- LOBs are written using 4MB write-gather-cache (wgc) to improve LOB write performance.
- Parameter `_kdlw_enable_write_gathering` enables write gather cache and by default, it is true.
- Write gather cache is per transaction basis and one wgc is allocated per transaction.
- Write gather cache is flushed when it is 4MB full or when transaction ends. Parameter that controls this behaviour: `_kdlwp_flush_threshold`

LOB internals - deduplication

LOB logical ID
00000001000000169767

LOB Index block:

row#0[7436] flag: -----, lock: 0, len=300, data:(255):

00 00 00 01 00 00 00 16 97 67 00 00 00 03 00 00 0c 21 00 0f a0 01 00 01 01
00 2f a8 01 00

...

00 00 00 00 00

col 0; len 20; (20): 2b f8 d8 65 7e 68 ff 43 5f 83 7d 6b b9 e0 bc bb c4 11 23 79

col 1; len 20; (20): 2b f8 d8 65 7e 68 ff 43 5f 83 7d 6b b9 e0 bc bb c4 11 23 79

col 2; NULL

LOB block
0x01002fa8

LOB column value is hashed and
LOB
index seems to hold hash value.

LOB internals - deduplication

LOB data block:

bdba [0x01002fa8] → Block DBA

kdlich [0c54224c 56]

flg0 0x20 [ver=0 typ=data lock=n]

flg1 0x00

scn 0x0000.003e1236

lid 00000001000000169767 → Logical ID

rid 0x00000000.0000

kdlih [0c542264 24]

flg2 0x00 [ver=0 lid=short-rowid hash=n cmap=n pfill=n]

flg3 0x00

...

78
78
78 78

LOB enhancements - basicfile

```
create table t1 (n1 number, c1 clob)
  lob(c1) store as basicfile;
```

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.

```
alter system flush buffer_cache;
```

```
alter session set events '10046 trace name context forever, level 12';
```

```
insert into t1 select n1 , lpad(1, 8192, 'x') from
  (select level n1 from dual connect by level <=12000);
```

12000 rows created.

```
tkprof orcl11g_ora_1756.trc orcl11g_ora_1756.trc.out sort=execpu, fchcpu
```

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

In basicfile, all writes are performed by user session introducing a lag.

LOB enhancements - securefile

```
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.

```
alter system flush buffer_cache;
```

```
alter session set events '10046 trace name context forever, level 12';
```

```
insert into t1 select n1 , lpad(1, 8192, 'x') from
      (select level n1 from dual connect by level <=12000);
```

12000 rows created.

```
tkprof orcl11g_ora_3220.trc orcl11g_ora_3220.trc.out sort=execpu, fchcpu
```

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

In securefile, there is no writes by user process. Writes are performed through WGC, and user sessions does not suffer from write performance.

LOB enhancements - securefile

basicfile:

```
tkprof orcl11g_ora_1756.trc orcl11g_ora_1756.trc.out sort=execpu, fchcpu
```

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

securefile:

```
tkprof orcl11g_ora_3220.trc orcl11g_ora_3220.trc.out sort=execpu, fchcpu
```

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

- For securefile deduplication inserts, for every LOB insert, existing LOB is read using 'direct path read' and checked for deduplication .
- Also, LOB writes are not handled by foreground process and foreground process does not need to wait for LOB writes to complete.

LOB enhancements - Parameters

- There are few other LOB parameters controlling LOB behaviour. Need to do more research before discussing it.

```
_kdli_cache_read_threshold
  minimum lob size for cache->nocache read (0 disables heuristic)
0
_kdli_cache_size
  maximum #entries in inode cache
8
_kdli_cache_verify
  verify cached inode via deserialization
FALSE
```

Agenda

- **(True) Online Index rebuild**
- **Invisible indices**
- **Virtual Columns**
- **LOB Performance improvements**
- **Compound triggers**
- **CBO – Extended statistics**
- **Fine Grained dependency**

Multiple triggers in 10g

```
create or replace trigger t1_b4_row
before insert or update or delete on t1
for each row
begin
    null;
end;
/
```

```
create or replace trigger t1_b4_stmt
before insert or update or delete on t1
begin
    null;
end;
/
```

```
create or replace trigger t1_b4_stmt
after insert or update or delete on t1
begin
    null;
end;
/
```

```
create or replace trigger t1_b4_stmt
after insert or update or delete on t1
for each row
begin
    null;
end;
/
```

Compound triggers

- **Compound triggers:**

- One program to fire at various timing points instead of many simple triggers to fire at a single timing point.

- **Since there is one program unit, instead of many program units performance also supposed to improve.**

- **We will create a small test case to test this.**

Compound triggers

```
create or replace trigger
  t2_compound
for insert or update or delete on
  t2
compound trigger
-- Declaration Section:
before EACH ROW IS
BEGIN
  null;
end before each row;

before statement is
begin
  null;
end before statement;
```

```
AFTER EACH ROW IS
BEGIN
  null;
end after each row;

after statement is
begin
  null;
end after statement;

end t2_compound;
```

Performance comparison

```
SQL> set serveroutput on size 100000
SQL> exec sys.runstats_pkg.rs_stop(10);
Run1 ran in 490 hsecs
Run2 ran in 2841 hsecs
run 1 ran in 17.25% of the time
```

Surprisingly compound triggers are costlier than simple triggers!

Name	Run1	Run2	Diff
....			
STAT...session pga memory	179,580	262,144	82,564
STAT...session pga memory max	179,580	262,144	82,564
STAT...session uga memory	65,464	261,856	196,392

Run1 latches total versus runs -- difference and pct

Run1	Run2	Diff	Pct
17,047	72,824	55,777	23.41%

This difference in performance could be a bug too..

PL/SQL procedure successfully completed.

Agenda

- **(True) Online Index rebuild**
- **Invisible indices**
- **Virtual Columns**
- **LOB Performance improvements**
- **Compound triggers**
- **CBO – Extended statistics**
- **Fine Grained dependency**

CBO – extended stats

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

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

- For 100% rows, $n1 = n2$.
- For 50% of rows, $n1 = n3$.
- For 20% of rows, $n1 = n4$.

CBO extended stats

There are 100 rows with n1=10

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

There are still 100 rows but CBO estimate is way off!

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

CBO extended stats

- CBO is using uniform distribution:

n1 has 100 distinct values, so selectivity of n1 is 1/100.

n2 has 100 distinct values, so selectivity of n2 is 1/100.

Selectivity of (n1=10 and n2 =10) is

$$(1/100) * (1/100).$$

So, row estimates for n1=10 and n2=10

$$= \text{num_rows} * (1/100) * (1/100)$$

$$= 10000 * (1/100) * (1/100) = 1$$

- CBO assumption: n1=10 and n2=10 are two independent predicates, an incorrect assumption.

CBO extended stats

- 11g introduces extended statistics to calculate and store correlation between columns.

```
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;  
/
```

CBO extended stats

- After adding extended stats, now CBO row estimate calculations are closer to reality.

```
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 | | 100 | 1200 | 9 (0)| 00:00:01 |  
|* 1 | TABLE ACCESS FULL| T_VC | 100 | 1200 | 9 (0)| 00:00:01 |  
-----
```

CBO extended stats-internals

- Adding extended stats adds a virtual column.

```
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 also.
- With this histograms information, CBO is able to calculate correlation strength (lines from 10053 trace):

SINGLE TABLE ACCESS PATH

Single Table Cardinality Estimation for T_VC[T_VC]

ColGroup (#1, VC) SYS_STUBZH0IHA7K\$KEBJVX05LOHAS

Col#: 1 2 CorStregth: 100.00

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

Agenda

- **(True) Online Index rebuild**
- **Invisible indices**
- **Virtual Columns**
- **LOB Performance improvements**
- **Compound triggers**
- **CBO – Extended statistics**
- **Fine Grained dependency**

Fine Grained dependency

- Dependency is tracked fine grained. For example, adding a column to a table does not validate dependent objects, if it is not necessary to do so.

```
create table t(a number);
create view v as
    select a from t;
create or replace procedure p1
is
    a1 number;
begin
    select * into a1 from t;
end;
/
```

```
create or replace procedure p2
is
    a1 number;
begin
    select a into a1 from t;
end;
/
```

Fine Grained dependency

- Let's Check status of these objects, add a column and check status again.

```
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

```
Alter table t add column ( b number);
```

```
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

Fine Grained dependency

- Column `d_attrs` in `dependency$` seems to be used to track this dependency. For procedure P1, 4th and 5th positions are 01.

```
select do.name, dp.name, d.d_attrs
from dependency$ d, obj$ do, obj$ dp
where d.d_obj#= do.obj# and d.p_obj#= dp.obj# and
p_obj#=(select object_id from dba_objects where object_name='T')
```

NAME	NAME	D_ATTRS
P1	T	0001010006 ←
V	T	0001000002
P2	T	0001000002
T2	T	

Fine Grained dependency

- Let's recreate the procedure with column names specified and check value for d_attrs column.

```
create or replace procedure p1
is
  a1 number;
begin
  select a1 into a1 from t;
end;
/
```

NAME	NAME	D_ATTRS
P1	T	0001000002 ←
V	T	0001000002
P2	T	0001000002
T2	T	

SQL result cache

```
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>
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 result cache

```
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>
```

```
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
```

More to come..

- RAC performance improved for specific class of workload.
 - Optimizer statistics gathering and accuracy improved.
 - Native compiler for PL/SQL and Java improves performance.
 - Oracle streams performance improvement.
 - Result caching
 - Real Application Testing
 - Automated partition creation.
-

Questions?

References

- Oracle support site. [Metalink.oracle.com](http://metalink.oracle.com). numerous documents
- Innovate faster with Oracle database 11g- An Oracle white paper
- My blog: <http://orainternals.wordpress.com>
- Oracle database 11g: An Oracle white paper:
- Internal's guru Steve Adam's website
www.ixora.com.au
- Jonathan Lewis' website
www.jlcomp.daemon.co.uk
- Julian Dyke's website
www.julian-dyke.com
- Costing PL/SQL functions: Jozse Senegocnick – HOTSOS 2006
- Pythian blog: Query result cache: by Alex fatkulin: www.pythian.com/authors/fatkulin
- Metalink note : 453567.1 on result cache
- Oracle database 11g: secure files : An Oracle white paper.