Indexing New Features: Oracle 11g Release 1 and Release 2

Richard Foote

Richard Foote

- Working in IT for 23+ years (scary stuff)
- Working with Oracle for 13+ years (almost as scary)
- Previously employed by Oracle Corporation for 5 ½ years (scary as hell)
- Currently employed by the Australian Federal Police as a Senior DBA
- Responsible for many large scale, mission critical, “life-dependant” classified Oracle systems
- Based in sunny Canberra, Australia
- Oracle OakTable Member since 2002 and Oracle ACE Director since 2008
- Interests includes all sports and music (especially David Bowie, Pink Floyd and Radiohead)
- **richard.foote@bigpond.com**
- Richard Foote’s Oracle Blog: **http://richardfoote.wordpress.com/**

So What Am I Going To Talk About ...

- Index Creation and Rebuild Locking Improvements
- Index Statistics and the Oracle 11g CBO
- Invisible Indexes
- Segment On Demand Indexes
- Zero Storage Unusable Indexes
- New Index Related Hints
- Analyze Validate Structure Improvements
- Bitmap-Join Indexes On IOT
- Virtual Columns Without Function-Based Indexes

Index Rebuild (and Creation) Locks

- Oracle's had the INDEX REBUILD ONLINE option for a long while
- However, only available with Enterprise Edition
- Prevents Parallel Execution
- Still requires two table locks on the base table at the start and at the end of the indexing process
- These table locks can still causes locking issues (before 11g)

Index Rebuild Locks (Pre 11g)

Session 1

```
SQL> CREATE TABLE bowie_stuff AS SELECT rownum id, 'David Bowie' name
        FROM dual CONNECT BY LEVEL <= 10000;
Table created.

SQL> CREATE INDEX bowie_stuff_i ON bowie_stuff(id);
Index created.
```

Session 2

```
SQL> INSERT INTO bowie_stuff VALUES (10001, 'Pink Floyd');
1 row created.       (No commit or rollback)
```

Session 1

```
SQL> ALTER INDEX bowie_stuff_i REBUILD ONLINE;

Session hangs !!  (Due to transaction in Session 2)
```

Index Rebuild Locks (Pre 11g)

Session 3

```
SQL> INSERT INTO bowie_stuff VALUES (10002, 'Radiohead');

Session hangs !!   (Due to lock required by index rebuild in Session 1)
```

Session 2

```
SQL> COMMIT;

Commit complete.
```

Releases lock in session 3 and index rebuild is free to proceed but it will eventually get stuck again by uncommitted session 3 as it requires another lock to complete the rebuild process...

In session 2, perform another insert before session 3 commits ...

```
SQL> INSERT INTO bowie_stuff VALUES (10003, 'Iggy Pop');

Session hangs !! (Due to the second lock required by index rebuild in Session 1)
```

Index Rebuild Locks (Pre 11g)

Session 3

```
SQL> COMMIT;

Commit complete.
```

Session 1

```
Index altered.
```

Session 2

```
1 row created.
```

Index Rebuild Locks (11g)

Session 1

```
SQL> CREATE TABLE bowie_stuff AS SELECT rownum id, 'David Bowie' name
        FROM dual CONNECT BY LEVEL <= 10000;
Table created.

SQL> CREATE INDEX bowie_stuff_i ON bowie_stuff(id);
Index created.
```

Session 2

```
SQL> INSERT INTO bowie_stuff VALUES (10001, 'Pink Floyd');
1 row created.       (No commit or rollback)
```

Session 1

```
SQL> ALTER INDEX bowie_stuff_i REBUILD ONLINE;

Session hangs !!  (Due to transaction in Session 2)
```

Note: No difference at this point ...

Index Rebuild Locks (11g)

Session 3

```
SQL> INSERT INTO bowie_stuff VALUES (10002, 'Radiohead');

1 row created.
```

Big Difference !! The index table lock no longer locks out other transactions

Session 2

```
SQL> COMMIT;

Commit complete.
```

Releases lock in session 1 and index rebuild is free to proceed but will eventually get stuck by uncommitted session 3 as it requires another lock to complete the rebuild ...

In session 2, perform another insert before session 3 commits ...

```
SQL> INSERT INTO bowie_stuff VALUES (10003, 'Iggy Pop');

1 row created.
```

Again, not a problem as the second index rebuild lock impacts no other transactions

Index Rebuild (and Creation) Locks

- With 11g, far safer to create or rebuild an index during busy Production periods
- The index DDL might be locked out for periods of time and take a while to complete …
- BUT it won't lock out other transactions
- Of course, most index rebuilds are a waste of time but that's another story

Index Statistics and the 11g CBO

Pre 11g example (Note: There are only 10 distinct combinations of data or 10% selectivity)

```
SQL> create table radiohead (id number, code varchar2(5), name varchar2(20));

Table created.

SQL> begin
  2  for i in 1..10000 loop
  3    insert into radiohead values(1, 'AAA', 'Description A');
  4    insert into radiohead values(2, 'BBB', 'Description B');
  5    insert into radiohead values(3, 'CCC', 'Description C');
  6    insert into radiohead values(4, 'DDD', 'Description D');
  7    insert into radiohead values(5, 'EEE', 'Description E');
  8    insert into radiohead values(6, 'FFF', 'Description F');
  9    insert into radiohead values(7, 'GGG', 'Description G');
 10    insert into radiohead values(8, 'HHH', 'Description H');
 11    insert into radiohead values(9, 'III', 'Description I');
 12    insert into radiohead values(10, 'JJJ', 'Description J');
 13  end loop;
 14  commit;
 15  end;
 16  /

PL/SQL procedure successfully completed.

SQL> create index radiohead_idx on radiohead(id, code);
Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'RADIOHEAD', estimate_percent=>null, cascade=>true,
method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

Index Statistics and the 11g CBO

SQL> select * from radiohead where id = 2 and code = 'BBB';

10000 rows selected.

Execution Plan

```
----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |  1000 | 21000 |    42  (10)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| RADIOHEAD |  1000 | 21000 |    42  (10)| 00:00:01 |
----------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
        365  consistent gets
          0  physical reads
          0  redo size
      50684  bytes sent via SQL*Net to client
        407  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      10000  rows processed
```

Note: Oracle assumes the selectivity will be 10 x 10 = 100 distinct values (1%), not 10% because there are 10 distinct values of both the ID and CODE columns...

Index Statistics and the 11g CBO

11g Rel 1 example (Note: There are only 10 distinct combinations of data or 10% selectivity)

```
SQL> create table radiohead (id number, code varchar2(5), name varchar2(20));

Table created.

SQL> begin
  2  for i in 1..10000 loop
  3    insert into radiohead values(1, 'AAA', 'Description A');
  4    insert into radiohead values(2, 'BBB', 'Description B');
  5    insert into radiohead values(3, 'CCC', 'Description C');
  6    insert into radiohead values(4, 'DDD', 'Description D');
  7    insert into radiohead values(5, 'EEE', 'Description E');
  8    insert into radiohead values(6, 'FFF', 'Description F');
  9    insert into radiohead values(7, 'GGG', 'Description G');
 10    insert into radiohead values(8, 'HHH', 'Description H');
 11    insert into radiohead values(9, 'III', 'Description I');
 12    insert into radiohead values(10, 'JJJ', 'Description J');
 13  end loop;
 14  commit;
 15  end;
 16  /

PL/SQL procedure successfully completed.

SQL> create index radiohead_idx on radiohead(id, code);
Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'RADIOHEAD', estimate_percent=>null, cascade=>true,
method_opt=>'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

Index Statistics and the 11g CBO

```
SQL> select * from radiohead where id = 2 and code = 'BBB';

10000 rows selected.

Execution Plan
----------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           | 10000 |   205K|   100   (1)| 00:00:02 |
|*  1 |  TABLE ACCESS FULL| RADIOHEAD | 10000 |   205K|   100   (1)| 00:00:02 |
----------------------------------------------------------------------------

Statistics
----------------------------------------------------------
        181  recursive calls
          0  db block gets
        391  consistent gets
          0  physical reads
          0  redo size
      50684  bytes sent via SQL*Net to client
        407  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
      10000  rows processed
```

Note: In 11g, Oracle gets the cardinality estimate exactly correct !!

Index Statistics and the 11g CBO

```
SQL> select index_name, distinct_keys from dba_indexes
       where index_name = 'RADIOHEAD_IDX';

INDEX_NAME     DISTINCT_KEYS
-------------- -------------
RADIOHEAD_IDX             10
```

Note: In 11g, Oracle can use the index DISTINCT_KEYS statistic to determine the selectivity of a predicate

There are only 10 distinct index key values, therefore one combination of values will return 10% of the data ...

Index Statistics and the 11g CBO

SQL> **DROP INDEX** radiohead_idx;

Index dropped.

SQL> select * from radiohead where id = 2 and code = 'BBB';

10000 rows selected.

```
Execution Plan
----------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |  1000 | 21000 |   100   (1)| 00:00:02 |
|*  1 |  TABLE ACCESS FULL| RADIOHEAD |  1000 | 21000 |   100   (1)| 00:00:02 |
----------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
        365  consistent gets
          0  physical reads
          0  redo size
      50684  bytes sent via SQL*Net to client
        407  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      10000  rows processed
```

After dropping the index, the CBO now gets cardinality estimates incorrect ...

Index Statistics and the 11g CBO

Let's create another table ...

```
SQL> create table ok_computer (id number, code varchar2(5), description varchar2(20));
Table created.

SQL> begin
  2  insert into ok_computer values(1, 'AAA', 'Description A');
  3  insert into ok_computer values(2, 'BBB', 'Description B');
  4  insert into ok_computer values(3, 'CCC', 'Description C');
  5  insert into ok_computer values(4, 'DDD', 'Description D');
  6  insert into ok_computer values(5, 'EEE', 'Description E');
  7  insert into ok_computer values(6, 'FFF', 'Description F');
  8  insert into ok_computer values(7, 'GGG', 'Description G');
  9  insert into ok_computer values(8, 'HHH', 'Description H');
 10  insert into ok_computer values(9, 'III', 'Description I');
 11  insert into ok_computer values(10, 'JJJ', 'Description J');
 12  commit;
 13  end;
 14  /
PL/SQL procedure successfully completed.

SQL> alter table ok_computer add primary key(id, code);
Table altered.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'OK_COMPUTER', estimate_percent=>null,
cascade=>true, method_opt=>'FOR ALL COLUMNS SIZE 1');
PL/SQL procedure successfully completed.
```

Index Statistics and the 11g CBO

SQL> **CREATE INDEX** radiohead_idx on radiohead(id, code);
Index created.

SQL> select * from radiohead r, ok_computer o where r.id = 5 and r.code = 'EEE';
100000 rows selected.

```
Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation            | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |   100K|  4101K|   989   (2)| 00:00:12 |
|   1 |  MERGE JOIN CARTESIAN|             |   100K|  4101K|   989   (2)| 00:00:12 |
|   2 |   TABLE ACCESS FULL  | OK_COMPUTER |    10 |   210 |     2   (0)| 00:00:01 |
|   3 |   BUFFER SORT        |             | 10000 |   205K|   987   (2)| 00:00:12 |
|*  4 |    TABLE ACCESS FULL | RADIOHEAD   | 10000 |   205K|    99   (2)| 00:00:02 |
----------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
        376  consistent gets
          0  physical reads
          0  redo size
    1203277  bytes sent via SQL*Net to client
        605  bytes received via SQL*Net from client
         21  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
     100000  rows processed
```

Note: although index not used in execution plan, it has used the index statistics to get the cardinality correct and uses a plan with 376 consistent gets.

Index Statistics and the 11g CBO

SQL> **DROP INDEX radiohead_idx;**
Index dropped.

SQL> select * from radiohead r, ok_computer o where r.id = 5 and r.code = 'EEE';
100000 rows selected.

```
Execution Plan
----------------------------------------------------------------------------------
| Id  | Operation            | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             | 10000 |   410K|   373   (1)| 00:00:05 |
|   1 |  MERGE JOIN CARTESIAN|             | 10000 |   410K|   373   (1)| 00:00:05 |
|*  2 |   TABLE ACCESS FULL  | RADIOHEAD   |  1000 | 21000 |   100   (1)| 00:00:02 |
|   3 |   BUFFER SORT        |             |    10 |   210 |   272   (0)| 00:00:04 |
|   4 |    TABLE ACCESS FULL | OK_COMPUTER |    10 |   210 |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
        165  recursive calls
          0  db block gets
        404  consistent gets
          0  physical reads
          0  redo size
    2603137  bytes sent via SQL*Net to client
        605  bytes received via SQL*Net from client
         21  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
     100000  rows processed
```

Note: Drop the index, Oracle now gets the cardinality wrong as it can't use the index statistics and chooses a more expensive execution plan ...

Index Statistics and the 11g CBO

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'RADIOHEAD', estimate_percent=>null, cascade=>true, **method_opt=>'FOR COLUMNS (id, code)** SIZE 1');
PL/SQL procedure successfully completed.

SQL> select * from radiohead r, ok_computer o where r.id = 5 and r.code = 'EEE';
100000 rows selected.

```
Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation            | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |   100K|  4101K|   989   (2)| 00:00:12 |
|   1 |  MERGE JOIN CARTESIAN|             |   100K|  4101K|   989   (2)| 00:00:12 |
|   2 |   TABLE ACCESS FULL  | OK_COMPUTER |    10 |   210 |     2   (0)| 00:00:01 |
|   3 |   BUFFER SORT        |             | 10000 |   205K|   987   (2)| 00:00:12 |
|*  4 |    TABLE ACCESS FULL | RADIOHEAD   | 10000 |   205K|    99   (2)| 00:00:02 |
----------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          8  recursive calls
          0  db block gets
        378  consistent gets
          0  physical reads
          0  redo size
    1203277  bytes sent via SQL*Net to client
        605  bytes received via SQL*Net from client
         21  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
     100000  rows processed
```

Note: 11g extended statistics can also be used to collect accurate statistics across column combinations and restore the efficient execution plan ...

Index Monitoring Trap

This new capability of using the DISTINCT_KEYS index statistic introduces a new trap when monitoring indexes.

```
SQL> alter index radiohead_idx monitoring usage;

Index altered.

SQL> select * from v$object_usage where index_name = 'RADIOHEAD_IDX';

INDEX_NAME     TABLE_NAME    MON USE START_MONITORING     END_MONITORING
-------------- ------------ --- --- -------------------- --------------------
RADIOHEAD_IDX RADIOHEAD     YES NO  09/16/2008 11:08:05
```

Let's see what happens when we begin to monitor the index ...

Index Monitoring – Index Cardinality Trap

```
SQL> select * from radiohead where id = 2 and code = 'BBB';

10000 rows selected.

Execution Plan
----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           | 10000 |   205K|   100   (1)| 00:00:02 |
|*  1 |  TABLE ACCESS FULL| RADIOHEAD | 10000 |   205K|   100   (1)| 00:00:02 |
----------------------------------------------------------------------------

Statistics
----------------------------------------------------------
        181  recursive calls
          0  db block gets
        391  consistent gets
          0  physical reads
          0  redo size
      50684  bytes sent via SQL*Net to client
        407  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
      10000  rows processed
```

Note: In 11g, Oracle gets the cardinality estimate exactly correct !!

Index Monitoring – Index Cardinality Trap

```
SQL> select * from v$object_usage where index_name = 'RADIOHEAD_IDX';

INDEX_NAME     TABLE_NAME    MON USE START_MONITORING     END_MONITORING
-------------- ------------ --- --- -------------------- --------------------
RADIOHEAD_IDX RADIOHEAD     YES NO  09/16/2008 11:08:05
```

Index Monitoring (in 11g Rel 1 and 2) does not picked up the index was “used” ...

As the index was not used within the execution plan, index monitoring doesn’t consider the index as being “used”.

Index Monitoring – Index Cardinality Trap

SQL> **DROP INDEX** radiohead_idx;

Index dropped.

SQL> select * from radiohead where id = 2 and code = 'BBB';

10000 rows selected.

```
Execution Plan
----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |  1000 | 21000 |   100   (1)| 00:00:02 |
|*  1 |  TABLE ACCESS FULL| RADIOHEAD |  1000 | 21000 |   100   (1)| 00:00:02 |
----------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          1  recursive calls
          0  db block gets
        365  consistent gets
          0  physical reads
          0  redo size
      50684  bytes sent via SQL*Net to client
        407  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
      10000  rows processed
```

After dropping the index, the CBO now gets the cardinality estimates and potentially the whole plan incorrect ...

Invisible Indexes

Invisible Indexes

```
SQL> SELECT * FROM invisible_bowie
       WHERE date_field BETWEEN '25-DEC-2006' AND '26-DEC-2006';

100 rows selected.

Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes |Cost(%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |   300 |  3600 |  304   (0)| 00:00:04|
|   1 |  TABLE ACCESS BY INDEX ROWID| INVISIBLE_BOWIE    |   300 |  3600 |  304   (0)| 00:00:04|
|*  2 |   INDEX RANGE SCAN           | INVISIBLE_BOWIE_I |   300 |       |    3   (0)| 00:00:01|
----------------------------------------------------------------------------------------
```

Note: By default, indexes are “visible” to the CBO and can potentially be used

Invisible Indexes

If an index:

- becomes problematic, or
- you think index is not being used and might be safe to drop

you can make it "**invisible**" ...

```
SQL> ALTER INDEX invisible_bowie_i INVISIBLE;

Index altered.

SQL> SELECT index_name, visibility FROM user_indexes
       WHERE index_name = 'INVISIBLE_BOWIE_I';

INDEX_NAME                     VISIBILITY
------------------------------ ---------------
INVISIBLE_BOWIE_I              INVISIBLE
```

Invisible Indexes

```
SQL> SELECT * FROM invisible_bowie
       WHERE date_field BETWEEN '25-DEC-2006' AND '26-DEC-2006';

100 rows selected.

Execution Plan
----------------------------------------------------------------------------
| Id  | Operation          | Name           | Rows | Bytes | Cost (%CPU)| Time   |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                |  300 |  3600 |   699   (2)|00:00:09|
|*  1 |  TABLE ACCESS FULL| INVISIBLE_BOWIE|  300 |  3600 |   699   (2)|00:00:09|
----------------------------------------------------------------------------
```

Note: The index is now invisible and is not considered by the CBO ...

Invisible Indexes

SQL> ALTER SESSION SET **OPTIMIZER_USE_INVISIBLE_INDEXES** = true;

Session altered.

SQL> SELECT * FROM invisible_bowie WHERE date_field > sysdate - 1;

```
100 rows selected.

Execution Plan
----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name             | Rows  | Bytes | Cost(%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                  |   100 |  1200 |   103   (0)| 00:00:02|
|   1 |  TABLE ACCESS BY INDEX ROWID| INVISIBLE_BOWIE  |   100 |  1200 |   103   (0)| 00:00:02|
|*  2 |   INDEX RANGE SCAN           | INVISIBLE_BOWIE_I|   100 |       |     3   (0)| 00:00:01|
----------------------------------------------------------------------------------------------
```

Note: You can alter a session to make invisible indexes visible to the session ...

Invisible Indexes

SQL> ALTER INDEX invisible_bowie_i **VISIBLE**;

Index altered.

SQL> ALTER SESSION SET OPTIMIZER_USE_INVISIBLE_INDEXES = false;

Session altered.

SQL> SELECT * FROM invisible_bowie WHERE date_field > sysdate - 1;

```
100 rows selected.

Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation                    | Name             | Rows  | Bytes |Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                  |   100 |  1200 |   103   (0)| 00:00:02|
|   1 |  TABLE ACCESS BY INDEX ROWID| INVISIBLE_BOWIE  |   100 |  1200 |   103   (0)| 00:00:02|
|*  2 |   INDEX RANGE SCAN           | INVISIBLE_BOWIE_I|   100 |       |     3   (0)| 00:00:01|
----------------------------------------------------------------------------------------
```

Note: An index can instantly be converted visible again as the index is always being maintained by Oracle, even when invisible ...

Invisible Indexes – Foreign Key Indexes

First, create a “Parent” table with 3 parent values ...

```
SQL> create table daddy (id number constraint daddy_pk primary key, name varchar2(20));

Table created.

SQL> insert into daddy values (1, 'BOWIE');

1 row created.

SQL> insert into daddy values (2, 'ZIGGY');

1 row created.

SQL> insert into daddy values (3, 'THIN WHITE DUKE');

1 row created.

SQL> commit;

Commit complete.
```

Invisible Indexes – Foreign Key Indexes

Next create a “Child” table that references the table with an **indexed** FK constraint

```
SQL> create table kiddie (id number, name varchar2(20), fk number,
                  constraint kiddie_fk foreign key(fk) references daddy(id));

Table created.

SQL> insert into kiddie select rownum, 'MAJOR TOM', 1 from dual connect by level <= 1000000;

1000000 rows created.

SQL> commit;

Commit complete.

SQL> create index kiddie_fk_i on kiddie(fk);

Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'DADDY', estimate_percent=>null, cascade=>true,
method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'KIDDIE', estimate_percent=>null, cascade=>true,
method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

Invisible Indexes – Foreign Key Indexes

```
SQL> delete daddy where id = 2;

1 row deleted.

Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | DELETE STATEMENT   |             |     1 |     3 |     0   (0)| 00:00:01 |
|   1 |  DELETE            | DADDY       |       |       |            |          |
|*  2 |   INDEX UNIQUE SCAN| SYS_C009714 |     1 |     3 |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------------------

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

At **8 db block gets**, Oracle has used index on FK to determine whether any child rows exist ...

Invisible Indexes – Foreign Key Indexes

In Oracle 11g **Release 1**

```
SQL> rollback;

Rollback complete.

SQL> alter index kiddie_fk_i invisible;

Index altered.
```

If now make the index Invisible, does this change the manner in which the delete operation is performed ?

Invisible Indexes – Foreign Key Indexes

```
SQL> delete daddy where id = 2;

1 row deleted.

Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | DELETE STATEMENT   |             |     1 |     3 |     0   (0)| 00:00:01 |
|   1 |  DELETE            | DADDY       |       |       |            |          |
|*  2 |   INDEX UNIQUE SCAN| SYS_C009714 |     1 |     3 |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------------------

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

No !! The index, although invisible is still being used ...

Invisible Indexes– Foreign Key Indexes

```
SQL> drop index kiddie_fk_i;
Index dropped.

SQL> delete daddy where id = 3;
1 row deleted.

Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | DELETE STATEMENT   |             |     1 |     3 |     0   (0)| 00:00:01 |
|   1 |  DELETE            | DADDY       |       |       |            |          |
|*  2 |   INDEX UNIQUE SCAN| SYS_C009714 |     1 |     3 |     0   (0)| 00:00:01 |
----------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          7  recursive calls
          7  db block gets
       3172  consistent gets
          0  physical reads
        632  redo size
        674  bytes sent via SQL*Net to client
        554  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Note: dropped index can't be used to lookup child table and FTS on "KIDDIE" table is performed

Index Monitoring – Foreign Key Trap

```
SQL> select * from v$object_usage where index_name = 'KIDDIE_FK_I';

INDEX_NAME   TABLE_NAME MON USE START_MONITORING    END_MONITORING
------------ ---------- --- --- ------------------- -------------------
KIDDIE_FK_I  KIDDIE     YES NO  09/10/2008 12:10:52


SQL> drop index kiddie_fk_i;

Index dropped.
```

Note: Index Monitoring (in 11g Rel 1 and 2) also does not picked up the fact the index on the FK column of the KIDDIE table was actually used to ensure no record has the deleted value (2) ...

Deleting such a so-called “unused” index can be disastrous !!

Invisible Indexes: Index Statistics

Remember how Oracle11g can use index DISTINCT_KEY statistic

SQL> select * from radiohead where id = 2 and code = 'BBB';

10000 rows selected.

```
Execution Plan
----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           | 10000 |   205K|   100   (1)| 00:00:02 |
|*  1 |  TABLE ACCESS FULL| RADIOHEAD | 10000 |   205K|   100   (1)| 00:00:02 |
----------------------------------------------------------------------------

Statistics
----------------------------------------------------------
        181  recursive calls
          0  db block gets
        391  consistent gets
          0  physical reads
          0  redo size
      50684  bytes sent via SQL*Net to client
        407  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
      10000  rows processed
```

Invisible Indexes: Index Statistics

SQL> alter index radiohead_idx invisible;
Index altered.

SQL> select * from radiohead where id = 2 and code = 'BBB';
10000 rows selected.

```
Execution Plan
----------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           | 10000 |   205K|   100   (1)| 00:00:02 |
|*  1 |  TABLE ACCESS FULL| RADIOHEAD | 10000 |   205K|   100   (1)| 00:00:02 |
----------------------------------------------------------------------------

Statistics
----------------------------------------------------------
        181  recursive calls
          0  db block gets
        391  consistent gets
          0  physical reads
          0  redo size
      50684  bytes sent via SQL*Net to client
        407  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          6  sorts (memory)
          0  sorts (disk)
      10000  rows processed
```

In 11g Rel 1, Oracle stills sees index statistics of invisible indexes ...

“Visible” Invisible Indexes

- In Oracle11g Release 1, Invisible Indexes are still visible:
  - When used to check and police FK consistency
  - When use by the CBO to lookup index statistics
- Both these issues have been fixed in Oracle11g Release 2

Invisible Indexes: PK and UK Constraints

Unique Indexes and indexes used to police Primary Key and Unique Key constraints can also be made “Invisible”.

```
SQL> create table bowie (id number constraint bowie_pk primary key using index(create unique index
bowie_pk_i on bowie(id)), name varchar2(20));

Table created.

SQL> insert into bowie select rownum, 'DAVID BOWIE' from dual connect by level <= 10000;

10000 rows created.

SQL> commit;

Commit complete.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'BOWIE',
estimate_percent=>null, cascade=>true, method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

Invisible Indexes: PK and UK Constraints

```
SQL> select * from bowie where id = 42;

        ID NAME
---------- ---------------
        42 DAVID BOWIE

-------------------------------------------------------------------------------------
| Id  | Operation                    | Name       | Rows  | Bytes | Cost (%CPU)|Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |            |     1 |    15 |     2   (0)|00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| BOWIE      |     1 |    15 |     2   (0)|00:00:01 |
|*  2 |   INDEX UNIQUE SCAN          | BOWIE_PK_I |     1 |       |     1   (0)|00:00:01 |
-------------------------------------------------------------------------------------

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

Invisible Indexes: PK and UK Constraints

SQL> alter index bowie_pk_i invisible;
Index altered.

SQL> select * from bowie where id = 42;

```
        ID NAME
---------- ---------------
        42 DAVID BOWIE

-------------------------------------------------------------------------------
| Id  | Operation          | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |       |     1 |    15 |     8   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| BOWIE |     1 |    15 |     8   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

Invisible Indexes: PK and UK Constraints

However, if you attempt to insert a duplicate row that violates the constraint ...

```
SQL> insert into bowie values (1, 'ZIGGY');
insert into bowie values (1, 'ZIGGY')
*
ERROR at line 1:
ORA-00001: unique constraint (BOWIE.BOWIE_PK) violated
```

In both Oracle11g Release 1 & 2, the index is still “visible” in that it’s used to police the constraint.

Invisible Indexes: PK and UK Constraints

Same scenario if you use a Non-Unique Index to police a PK or UK constraint

```
SQL> create table bowie (id number constraint bowie_pk primary key using index(create index bowie_pk_i
on bowie(id)), name varchar2(20));
Table created.

SQL> insert into bowie select rownum, 'DAVID BOWIE' from dual connect by level <= 10000;
10000 rows created.

SQL> commit;
Commit complete.

SQL> alter index bowie_pk_i invisible;
Index altered.

SQL> insert into bowie values (1, 'ZIGGY');
insert into bowie values (1, 'ZIGGY')
*
ERROR at line 1:
ORA-00001: unique constraint (BOWIE.BOWIE_PK) violated
```

Invisible Indexes: Unique Indexes

Same scenario if you just create a Unique Index ...

```
SQL> create table bowie as select rownum id, 'DAVID_BOWIE' name from dual connect by level <=10000;

Table created.

SQL> create unique index bowie_id_i on bowie(id);

Index created.

SQL> alter index bowie_id_i invisible;

Index altered.

SQL> insert into bowie values (1, 'ZIGGY');
insert into bowie values (1, 'ZIGGY')
*
ERROR at line 1:
ORA-00001: unique constraint (BOWIE.BOWIE_ID_I) violated
```

Invisible Indexes: 11g Rel. 1 Bug

Nasty bug with Oracle11g Release 1 when it comes to collecting statistics on an Invisible index:

```
SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'BOWIE', estimate_percent=>null,
cascade=>true, method_opt=> 'FOR ALL COLUMNS SIZE 1');
BEGIN dbms_stats.gather_table_stats(ownname=>null, tabname=>'BOWIE', estimate_percent=>null,
cascade=>true, method_opt=> 'FOR ALL COLUMNS SIZE 1'); END;

*
ERROR at line 1:
ORA-00904: : invalid identifier
ORA-06512: at "SYS.DBMS_STATS", line 17806
ORA-06512: at "SYS.DBMS_STATS", line 17827
ORA-06512: at line 1
```

Fixed In Oracle11g Release 2

Bitmap-Join Indexes

- Useful index structure for Data Warehouses
- Can create a bitmap index on a table based on the column of another table
- Can potentially make joins unnecessary and associated SQL queries more efficient
- But has a big restriction prior to Oracle11g ...

Bitmap-Join Indexes

```
SQL> CREATE TABLE big_dwh_table (id NUMBER PRIMARY KEY, album_id NUMBER, artist_id NUMBER, country_id NUMBER, format_id
NUMBER, release_date DATE, total_sales NUMBER);
Table created.

SQL> CREATE SEQUENCE dwh_seq;
Sequence created.

SQL> create or replace procedure pop_big_dwh_table as
  2  v_id         number;
  3  v_artist_id   number;
  4  begin
  5    for v_album_id in 1..10000 loop
  6       v_artist_id:= ceil(dbms_random.value(0,100));
  7       for v_country_id in 1..100 loop
  8         select dwh_seq.nextval into v_id from dual;
  9         insert into big_dwh_table values (v_id, v_album_id, v_artist_id, v_country_id, ceil(dbms_random.value(0,4)), trunc(sysdate-
mod(v_id,ceil(dbms_random.value(0,1000)))), ceil(dbms_random.value(0,500000)));
 10       end loop;
 11    end loop;
 12 commit;
 13 end;
 14 /
Procedure created.

SQL> exec pop_big_dwh_table
PL/SQL procedure successfully completed.

SQL> create bitmap index big_dwh_table_album_id_i on big_dwh_table(album_id);
Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=> 'BOWIE', tabname=> 'BIG_DWH_TABLE', estimate_percent=> null, cascade=> true,
method_opt=> 'FOR ALL COLUMNS SIZE 1');
PL/SQL procedure successfully completed.
```

Bitmap-Join Indexes

```
SQL> CREATE TABLE albums (album_id number, album_details varchar2(30));

Table created.

SQL> INSERT INTO albums SELECT rownum, substr(object_name,1,30) FROM dba_objects WHERE rownum <= 10000;

10000 rows created.

SQL> commit;

Commit complete.

SQL> alter table albums add primary key(album_id);

Table altered.

SQL> create index albums_details_i on albums(album_details);

Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=> 'BOWIE', tabname=> 'ALBUMS', estimate_percent=> null,
cascade=> true, method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

Bitmap-Join Indexes

```
SQL> SELECT b.id, b.album_id, b.format_id FROM big_dwh_table b, albums a WHERE b.album_id = a.album_id and a.album_details = 'TAB$';

100 rows selected.

-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                       | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                            |   125 |  4250 |    25   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |                            |       |       |            |          |
|   2 |   NESTED LOOPS                |                            |   125 |  4250 |    25   (0)| 00:00:01 |
|   3 |    TABLE ACCESS BY INDEX ROWID| ALBUMS                     |     1 |    22 |     2   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | ALBUMS_DETAILS_I           |     1 |       |     1   (0)| 00:00:01 |
|   5 |    BITMAP CONVERSION TO ROWIDS|                            |       |       |            |          |
|*  6 |     BITMAP INDEX SINGLE VALUE | BIG_DWH_TABLE_ALBUM_ID_I   |       |       |            |          |
|   7 |   TABLE ACCESS BY INDEX ROWID | BIG_DWH_TABLE              |   100 |  1200 |    25   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         10  consistent gets
          0  physical reads
          0  redo size
       1648  bytes sent via SQL*Net to client
        396  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        100  rows processed
```

Bitmap-Join Indexes

Let's now create a Bitmap-Join Index ...

```
SQL> drop index albums_details_i;

Index dropped.

SQL> CREATE BITMAP INDEX big_dwh_album_details_i ON big_dwh_table(a.album_details)
    FROM big_dwh_table b, albums a
    WHERE b.album_id = a.album_id;

Index created.
```

Bitmap-Join Indexes

```
SQL> SELECT b.id, b.album_id, b.format_id FROM big_dwh_table b, albums a WHERE b.album_id = a.album_id and a.album_details = 'TAB$';

100 rows selected.

-----------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                          |   125 |  1500 |    26   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID  | BIG_DWH_TABLE            |   125 |  1500 |    26   (0)| 00:00:01 |
|   2 |   BITMAP CONVERSION TO ROWIDS|                          |       |       |            |          |
|*  3 |    BITMAP INDEX SINGLE VALUE  | BIG_DWH_ALBUM_DETAILS_I |       |       |            |          |
-----------------------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
       1648  bytes sent via SQL*Net to client
        396  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        100  rows processed
```

There is no join step in the execution plan, the ALBUMS table is not even referenced and consistent gets has dropped down from 10 to 6

Bitmap-Join Indexes

```
SQL> drop table albums;
Table dropped.

SQL> CREATE TABLE albums (album_id number primary key, album_details varchar2(30)) organization index;
Table created.

SQL> INSERT INTO albums SELECT rownum, substr(object_name,1,30) FROM dba_objects WHERE rownum <= 10000;
10000 rows created.

SQL> commit;
Commit complete.

SQL> exec dbms_stats.gather_table_stats(ownname=> 'BOWIE', tabname=> 'ALBUMS', estimate_percent=> null, cascade=> true,
method_opt=> 'FOR ALL COLUMNS SIZE 1');
PL/SQL procedure successfully completed.

SQL> CREATE BITMAP INDEX big_dwh_album_details_i ON big_dwh_table(a.album_details)
  2      FROM big_dwh_table b, albums a
  3      WHERE b.album_id = a.album_id;
CREATE BITMAP INDEX big_dwh_album_details_i ON big_dwh_table(a.album_details)
                                              *
ERROR at line 1:
ORA-25966: join index cannot be based on an index organized table
```

However, can't create a Bitmap-Join Index if either table is an Index Organized Table

11g Bitmap-Join Indexes on IOTs

However, since Oracle11g Rel 1, Bitmap-Join Indexes on IOTs are fully supported ...

```
SQL> CREATE BITMAP INDEX big_dwh_album_details_i ON big_dwh_table(a.album_details)
   FROM big_dwh_table b, albums a
   WHERE b.album_id = a.album_id;
Index created.

SQL> SELECT b.id, b.album_id, b.format_id FROM big_dwh_table b, albums a WHERE b.album_id = a.album_id and a.album_details = 'TAB$';
100 rows selected.

-------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                          |   125 |  1500 |    26   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID | BIG_DWH_TABLE            |   125 |  1500 |    26   (0)| 00:00:01 |
|   2 |   BITMAP CONVERSION TO ROWIDS|                          |       |       |            |          |
|*  3 |    BITMAP INDEX SINGLE VALUE | BIG_DWH_ALBUM_DETAILS_I |       |       |            |          |
-------------------------------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          6  consistent gets
          0  physical reads
          0  redo size
       1648  bytes sent via SQL*Net to client
        396  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
        100  rows processed
```

Creation On Demand Segments

- Prior to 11g Release 2, when a segment is created, it's allocated at least 1 initial extent
- However, many large applications create many segments that are not actually used (e.g. SAP)
- Many tables can have many associated indexes which are likewise never used
- This can result in a large amount of essentially wasted storage overall

Creation On Demand Segments

```
SQL> create table empty (a number, b number, c number, d number, e number);
Table created.

SQL> create index empty_a_i on empty(a);
Index created.

SQL> create index empty_b_i on empty(b);
Index created.

SQL> create index empty_c_i on empty(c);
Index created.

SQL> create index empty_d_i on empty(d);
Index created.

SQL> create index empty_e_i on empty(e);
Index created.

SQL> select segment_name, blocks, bytes, extents from dba_segments where segment_name like 'EMPTY%';

SEGMENT_NAME      BLOCKS      BYTES    EXTENTS
------------ ---------- ---------- ----------
EMPTY                128    1048576          1
EMPTY_A_I            128    1048576          1
EMPTY_B_I            128    1048576          1
EMPTY_C_I            128    1048576          1
EMPTY_D_I            128    1048576          1
EMPTY_E_I            128    1048576          1

6 rows selected.
```

Creation On Demand Segments

However, create the same segments in Oracle11g Release 2 and no storage is allocated at all ...

```
SQL> create table empty (a number, b number, c number, d number, e number);
Table created.

SQL> create index empty_a_i on empty(a);
Index created.

SQL> create index empty_b_i on empty(b);
Index created.

SQL> create index empty_c_i on empty(c);
Index created.

SQL> create index empty_d_i on empty(d);
Index created.

SQL> create index empty_e_i on empty(e);
Index created.

SQL> select segment_name, blocks, bytes, extents from dba_segments where segment_name like
'EMPTY%';
no rows selected
```

Creation On Demand Segments

If we now create another table with lots of rows ...

```
SQL> create table bowie as select rownum id, 'BOWIE' name from dual connect by level <= 1000000;
Table created.

SQL> create index bowie_id_i on bowie(id);
Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'BOWIE', cascade=>true,
estimate_percent=>null, method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'EMPTY', cascade=>true,
estimate_percent=>null, method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

Creation On Demand Segments

SQL> select * from bowie, empty where bowie.id=empty.a and bowie.id = 42;
no rows selected

```
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name       | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |            |     1 |    76 |     2   (0)| 00:00:01 |
|   1 |  MERGE JOIN CARTESIAN         |            |     1 |    76 |     2   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID | EMPTY      |     1 |    65 |     1   (0)| 00:00:01 |
|*  3 |    INDEX RANGE SCAN           | EMPTY_A_I  |     1 |       |     1   (0)| 00:00:01 |
|   4 |   BUFFER SORT                 |            |     1 |    11 |     1   (0)| 00:00:01 |
|   5 |    TABLE ACCESS BY INDEX ROWID| BOWIE      |     1 |    11 |     1   (0)| 00:00:01 |
|*  6 |     INDEX RANGE SCAN          | BOWIE_ID_I |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

Creation On Demand Segments

If we insert the first row in a pre-Oracle11g Rel 2 table:

```
SQL> insert into empty2 (a, b) values (1,1);

1 row created.

-------------------------------------------------------------------------------
| Id  | Operation                 | Name   | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | INSERT STATEMENT          |        |     1 |     1   (0)| 00:00:01 |
|   1 |  LOAD TABLE CONVENTIONAL  | EMPTY2 |       |            |          |
-------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
          3  recursive calls
         10  db block gets
          6  consistent gets
          0  physical reads
       1056  redo size
        389  bytes sent via SQL*Net to client
        322  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Overheads are minimal ...

Creation On Demand Segments

However, when we insert the first row in an Oracle11g Rel 2 table

```
SQL> insert into empty (a, b) values (1,1);

1 row created.

-------------------------------------------------------------------------------
| Id  | Operation                 | Name  | Rows  | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | INSERT STATEMENT          |       |     1 |     1   (0)| 00:00:01 |
|   1 |  LOAD TABLE CONVENTIONAL  | EMPTY |       |            |          |
-------------------------------------------------------------------------------

Statistics
----------------------------------------------------------
        759  recursive calls
        129  db block gets
        133  consistent gets
          3  physical reads
      21232  redo size
        389  bytes sent via SQL*Net to client
        321  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Creation On Demand Segments

The first row creates all dependent segments, even if indexes are not populated

```
SQL> select segment_name, blocks, bytes, extents from dba_segments where segment_name like 'EMPTY%';

SEGMENT_NAME      BLOCKS      BYTES    EXTENTS
------------ ---------- ---------- ----------
EMPTY                128    1048576          1
EMPTY_A_I            128    1048576          1
EMPTY_B_I            128    1048576          1
EMPTY_C_I            128    1048576          1
EMPTY_D_I            128    1048576          1
EMPTY_E_I            128    1048576          1

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'EMPTY', cascade=>true,
estimate_percent=>null, method_opt=> 'FOR ALL COLUMNS SIZE 1');
PL/SQL procedure successfully completed.

SQL> select index_name, blevel, leaf_blocks, status from dba_indexes where index_name like 'EMPTY%';

INDEX_NAME      BLEVEL LEAF_BLOCKS STATUS
---------- ---------- ----------- --------
EMPTY_A_I            0           1 VALID
EMPTY_B_I            0           1 VALID
EMPTY_C_I            0           0 VALID
EMPTY_D_I            0           0 VALID
EMPTY_E_I            0           0 VALID
```

Quotas With Creation On Demand Segments

Prior to 11g R2, could not create a segment in a tablespace without sufficient quotas ...

```
SQL> create user muse identified by muse default tablespace user_data temporary tablespace
temp;

User created.

SQL> grant create session, create table to muse;

Grant succeeded.

SQL> connect muse/muse;
Connected.


SQL> create table fred (id number primary key using index (create index fred_pk on fred(id)
tablespace user_data), name varchar2(20));
create table fred (id number primary key using index (create index fred_pk on fred(id) tablespace
user_data), name varchar2(20))
*
ERROR at line 1:
ORA-01950: no privileges on tablespace 'USER_DATA'
```

Quotas With Creation On Demand Segments

However, in 11g R2, you can with quotas only enforced when data is actually inserted ...

```
SQL> create user muse identified by muse default tablespace user_data temporary tablespace
temp;

User created.

SQL> grant create session, create table to muse;

Grant succeeded.

SQL> connect muse/muse
Connected.


SQL> create table fred (id number primary key using index (create index fred_pk on fred(id)
tablespace user_data), name varchar2(20));

Table created.

SQL> insert into fred values (1, 'BOWIE');
insert into fred values (1, 'BOWIE')
            *
ERROR at line 1:
ORA-01950: no privileges on tablespace 'USER_DATA'
```

Zero Sized Unusable Indexes

Prior to Oracle11g Release 2

SQL> create table bowie as select rownum id, 'BOWIE' name from dual connect by level <= 1000000;
Table created.

SQL> create index bowie_id_i on bowie(id);
Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'BOWIE', cascade=> true, estimate_percent=> null, method_opt=> 'FOR ALL COLUMNS SIZE 1');
PL/SQL procedure successfully completed.

SQL> alter index bowie_id_i **unusable**;
Index altered.

SQL> select index_name, blevel, leaf_blocks, num_rows, status, dropped from dba_indexes where index_name = 'BOWIE_ID_I';

```
INDEX_NAME      BLEVEL LEAF_BLOCKS   NUM_ROWS STATUS   DRO
---------- ---------- ----------- ---------- -------- ---
BOWIE_ID_I          2        2226    1000000 UNUSABLE NO
```

SQL> select segment_name, bytes, blocks, extents from dba_segments where segment_name = 'BOWIE_ID_I';

```
SEGMENT_NAME       BYTES     BLOCKS    EXTENTS
------------ ---------- ---------- ----------
BOWIE_ID_I     18874368       2304         18
```

Zero Sized Unusable Indexes

Oracle11g Release 2

```
SQL> create table bowie as select rownum id, 'BOWIE' name from dual connect by level <= 1000000;
Table created.

SQL> create index bowie_id_i on bowie(id);
Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=>null, tabname=>'BOWIE', cascade=> true,
estimate_percent=> null, method_opt=> 'FOR ALL COLUMNS SIZE 1');
PL/SQL procedure successfully completed.

SQL> alter index bowie_id_i unusable;
Index altered.

SQL> select index_name, blevel, leaf_blocks, num_rows, status, dropped from dba_indexes where index_name
= 'BOWIE_ID_I';

INDEX_NAME      BLEVEL LEAF_BLOCKS   NUM_ROWS STATUS   DRO
---------- ---------- ------------ ---------- -------- ---
BOWIE_ID_I           2         2226    1000000 UNUSABLE NO

SQL> select segment_name, bytes, blocks, extents from dba_segments where segment_name = 'BOWIE_ID_I';

no rows selected
```

Oracle automatically drops the storage associated with such unusable indexes

Zero Sized Unusable Indexes

Create and populate a simple partitioned table

```
SQL> CREATE TABLE big_album_sales(id number, album_id number, country_id number,
        release_date date, total_sales number)  PARTITION BY RANGE (release_date)
(PARTITION ALBUMS_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007', 'DD-MON-YYYY')),
 PARTITION ALBUMS_2007 VALUES LESS THAN (TO_DATE('01-JAN-2008', 'DD-MON-YYYY')),
 PARTITION ALBUMS_2008 VALUES LESS THAN (TO_DATE('01-JAN-2009', 'DD-MON-YYYY')),
 PARTITION ALBUMS_2009 VALUES LESS THAN (MAXVALUE));

Table created.

SQL> INSERT INTO big_album_sales SELECT rownum, mod(rownum,5000)+1, mod(rownum,100)+1, sysdate-
mod(rownum,2000), ceil(dbms_random.value(1,500000)) FROM dual CONNECT BY LEVEL <= 1000000;

1000000 rows created.

SQL> commit;

Commit complete.
```

Zero Sized Unusable Indexes

Create a non-partition index, a global index and a local index on the partitioned table

```
SQL> CREATE INDEX big_album_tot_sales_i ON big_album_sales(total_sales);

Index created.

SQL> CREATE INDEX big_album_country_id_i  ON big_album_sales(country_id)
  2  GLOBAL PARTITION BY RANGE (country_id)
  3  (PARTITION TS1 VALUES LESS THAN (26),
  4   PARTITION TS2 VALUES LESS THAN (51),
  5   PARTITION TS3 VALUES LESS THAN (76),
  6   PARTITION TS4 VALUES LESS THAN (MAXVALUE));

Index created.

SQL> CREATE INDEX big_album_album_id_i ON big_album_sales(album_id) LOCAL;

Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=> 'BOWIE', tabname=> 'BIG_ALBUM_SALES',
estimate_percent=> null, method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.
```

Zero Sized Unusable Indexes

```
SQL> ALTER TABLE big_album_sales SPLIT PARTITION ALBUMS_2009
  2  AT (TO_DATE('01-JAN-2010', 'DD-MON-YYYY'))
  3  INTO (PARTITION ALBUMS_2009, PARTITION ALBUMS_2010);

Table altered.

SQL> select index_name, status from dba_indexes where table_name = 'BIG_ALBUM_SALES';

INDEX_NAME                     STATUS
------------------------------ --------
BIG_ALBUM_TOT_SALES_I          UNUSABLE
BIG_ALBUM_COUNTRY_ID_I         N/A
BIG_ALBUM_ALBUM_ID_I           N/A

SQL> select index_name, partition_name, status, leaf_blocks from dba_ind_partitions where index_name like
'BIG_ALBUM_%';

INDEX_NAME                     PARTITION_NAME                 STATUS   LEAF_BLOCKS
------------------------------ ------------------------------ -------- -----------
BIG_ALBUM_ALBUM_ID_I           ALBUMS_2006                    USABLE           807
BIG_ALBUM_ALBUM_ID_I           ALBUMS_2007                    USABLE           381
BIG_ALBUM_ALBUM_ID_I           ALBUMS_2008                    USABLE           383
BIG_ALBUM_ALBUM_ID_I           ALBUMS_2009                    UNUSABLE
BIG_ALBUM_ALBUM_ID_I           ALBUMS_2010                    UNUSABLE
BIG_ALBUM_COUNTRY_ID_I         TS1                            UNUSABLE         629
BIG_ALBUM_COUNTRY_ID_I         TS2                            UNUSABLE         629
BIG_ALBUM_COUNTRY_ID_I         TS3                            UNUSABLE         629
BIG_ALBUM_COUNTRY_ID_I         TS4                            UNUSABLE         629
```

Zero Sized Unusable Indexes

All unusable index partitions from both the global and local index no longer have allocated storage.

However, the unusable non-partitioned index segment has not been dropped ...

```
SQL> select segment_name, partition_name, bytes, blocks from dba_segments where segment_name like
'BIG_ALBUM_%' and segment_type like 'INDEX%';

SEGMENT_NAME           PARTITION_NAME                   BYTES     BLOCKS
---------------------- ------------------------------ ---------- ----------
BIG_ALBUM_ALBUM_ID_I   ALBUMS_2006                     7340032        896
BIG_ALBUM_ALBUM_ID_I   ALBUMS_2007                     3145728        384
BIG_ALBUM_ALBUM_ID_I   ALBUMS_2008                     4194304        512
BIG_ALBUM_TOT_SALES_I                                 23068672       2816
```

Zero Sized Unusable Indexes

Can use zero sized unusable indexes to your advantage to index only useful portions of a table. Most data here is processed:

```
SQL> create table bowie_stuff (id number, processed varchar2(10));
Table created.

SQL> insert into bowie_stuff select rownum, 'YES' from dual connect by level <= 1000000;
1000000 rows created.

SQL> commit;
Commit complete.

SQL> update bowie_stuff set processed = 'NO' where id in (999990, 999992, 999994, 999996,
999998);
5 rows updated.

SQL> commit;
Commit complete.

SQL> create index bowie_stuff_i on bowie_stuff(processed) pctfree 0;
Index created.
```

Zero Sized Unusable Indexes

SQL> select index_name, leaf_blocks from dba_indexes where index_name = 'BOWIE_STUFF_I';

```
INDEX_NAME                     LEAF_BLOCKS
------------------------------ -----------
BOWIE_STUFF_I                         1877
```

SQL> select segment_name, blocks from dba_segments where segment_name = 'BOWIE_STUFF_I';

```
SEGMENT_NAME                     BLOCKS
-------------------- ----------
BOWIE_STUFF_I                      1920
```

SQL> exec dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'BOWIE_STUFF', estimate_percent=>null, cascade=> true, method_opt=> 'FOR ALL COLUMNS SIZE 1');

PL/SQL procedure successfully completed.

SQL> exec dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>'BOWIE_STUFF', estimate_percent=>null, method_opt=> 'FOR COLUMNS PROCESSED SIZE 5');

PL/SQL procedure successfully completed.

Zero Sized Unusable Indexes

SQL> select * from bowie_stuff where processed = 'NO';

```
Execution Plan
----------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     5 |    40 |     4   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| BOWIE_STUFF   |     5 |    40 |     4   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | BOWIE_STUFF_I |     5 |       |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------------------

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

Zero Sized Unusable Indexes

However in 11g R2, if we now recreate the index as a partitioned index with only the “useful” portion of the index usable ...

```
SQL> drop index bowie_stuff_i;

Index dropped.

SQL> create index bowie_stuff_i on bowie_stuff(processed)
  2  global partition by range (processed)
  3  (partition not_processed_part values less than ('YES'),
  4   partition processed_part values less than (MAXVALUE))
  5  unusable;

Index created.

SQL> alter index bowie_stuff_i rebuild partition not_processed_part;

Index altered.
```

Zero Sized Unusable Indexes

We now only use a fraction of the storage for the index and the “useful” portion of the indexed data is just a single leaf block in size ...

```
SQL> select index_name, partition_name, leaf_blocks from dba_ind_partitions where
index_name = 'BOWIE_STUFF_I';

INDEX_NAME           PARTITION_NAME        LEAF_BLOCKS
-------------------- -------------------- -----------
BOWIE_STUFF_I        PROCESSED_PART                 0
BOWIE_STUFF_I        NOT_PROCESSED_PART             1

SQL> select segment_name, partition_name, blocks from dba_segments where segment_name
= 'BOWIE_STUFF_I';

SEGMENT_NAME         PARTITION_NAME            BLOCKS
-------------------- -------------------- ----------
BOWIE_STUFF_I        NOT_PROCESSED_PART             8
```

Zero Sized Unusable Indexes

```
SQL> select * from bowie_stuff where processed = 'NO';

Execution Plan
----------------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |     5 |    45 |     1   (0)| 00:00:01 |       |       |
|   1 |  PARTITION RANGE SINGLE      |               |     5 |    45 |     1   (0)| 00:00:01 |     1 |     1 |
|   2 |   TABLE ACCESS BY INDEX ROWID| BOWIE_STUFF   |     5 |    45 |     1   (0)| 00:00:01 |       |       |
|*  3 |    INDEX RANGE SCAN          | BOWIE_STUFF_I |     5 |       |     1   (0)| 00:00:01 |     1 |     1 |
----------------------------------------------------------------------------------------------------------------

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

Note: The query itself is also more efficient with consistent gets reduced from 6 down to 4 ...

IGNORE_ROW_ON_DUPKEY_INDEX

Create table with a PK and attempt to insert duplicate PK rows ...

SQL> create table radiohead (id number constraint radiohead_pk_i primary key using index (create unique index radiohead_pk_i on radiohead(id)), name varchar2(20));

Table created.

SQL> select index_name, uniqueness, table_name from dba_indexes where index_name = 'RADIOHEAD_PK_I';

```
INDEX_NAME                     UNIQUENES TABLE_NAME
------------------------------ --------- ------------------------------
RADIOHEAD_PK_I                 UNIQUE    RADIOHEAD
```

SQL> insert into radiohead select rownum, 'OK COMPUTER' from dual connect by level **<= 10**;

10 rows created.

SQL> commit;

Commit complete.

SQL> insert into radiohead select rownum, 'OK COMPUTER' from dual connect by level **<= 12**;
insert into radiohead select rownum, 'OK COMPUTER' from dual connect by level <= 12
*
ERROR at line 1:
ORA-00001: unique constraint (BOWIE.RADIOHEAD_PK_I) violated

IGNORE_ROW_ON_DUPKEY_INDEX

With the new Oracle11g Rel. 1 hint, duplicate violation rows are automatically ignored.

```
SQL> insert /*+ ignore_row_on_dupkey_index(radiohead,radiohead_pk_i) */ into radiohead select rownum, 'OK COMPUTER'
from dual connect by level <= 12;
2 rows created.

SQL> insert /*+ ignore_row_on_dupkey_index(radiohead(id)) */ into radiohead select rownum, 'OK COMPUTER' from dual
connect by level <= 13;
1 row created.

SQL> commit;
Commit complete.

SQL> select * from radiohead;

        ID NAME
---------- --------------------
         1 OK COMPUTER
         2 OK COMPUTER
         3 OK COMPUTER
         4 OK COMPUTER
         5 OK COMPUTER
         6 OK COMPUTER
         7 OK COMPUTER
         8 OK COMPUTER
         9 OK COMPUTER
        10 OK COMPUTER
        11 OK COMPUTER
        12 OK COMPUTER
        13 OK COMPUTER
13 rows selected.
```

IGNORE_ROW_ON_DUPKEY_INDEX

The index must be Unique for the hint to be valid ...

```
SQL> create table radiohead (id number constraint radiohead_pk_i primary key using index (create index radiohead_pk_i on radiohead(id)), name varchar2(20));

Table created.

SQL> insert into radiohead select rownum, 'OK COMPUTER' from dual connect by level <= 10;

10 rows created.

SQL> commit;

Commit complete.

SQL> select index_name, uniqueness, table_name from dba_indexes where index_name='RADIOHEAD_PK_I';

INDEX_NAME                     UNIQUENES TABLE_NAME
------------------------------ --------- ------------------------------
RADIOHEAD_PK_I                 NONUNIQUE RADIOHEAD

SQL> insert /*+ ignore_row_on_dupkey_index(radiohead,radiohead_pk_i) */ into radiohead select rownum, 'OK COMPUTER' from dual connect by level <= 12;
insert /*+ ignore_row_on_dupkey_index(radiohead,radiohead_pk_i) */ into radiohead select rownum, 'OK COMPUTER' from dual connect by level <= 12
                                                                  *
ERROR at line 1:
ORA-38913: Index specified in the index hint is invalid
```

IGNORE_ROW_ON_DUPKEY_INDEX

The UPDATE statement is not allowed with this hint ...

```
SQL> update /*+ ignore_row_on_dupkey_index(radiohead,radiohead_pk_i) */ radiohead set id = 13 where id = 3;
update /*+ ignore_row_on_dupkey_index(radiohead,radiohead_pk_i) */ radiohead set id = 13 where id = 3
*
ERROR at line 1:
ORA-38917: IGNORE_ROW_ON_DUPKEY_INDEX hint disallowed for this operation
```

ANALYZE VALIDATE STRUCTURE FAST

Oracle11g Release 1 introduces a new “FAST”, more efficient VALIDATE STRUCTURE command option

Will identified a corruption exists but no specific details about the corruption

```
SQL> analyze table general_audit_log validate structure cascade;

Table analyzed.

Elapsed: 00:01:34.12
SQL> analyze table general_audit_log validate structure cascade fast;

Table analyzed.

Elapsed: 00:01:45.75
SQL> analyze table general_audit_log validate structure cascade;

Table analyzed.

Elapsed: 00:01:34.12
SQL> analyze table general_audit_log validate structure cascade fast;

Table analyzed.

Elapsed: 00:01:53.73
```

However, initial investigation with FAST option has sometimes yielded “disappointing” results ...

ANALYZE VALIDATE STRUCTURE FAST

If we trace a session using just the VALIDATE STRUCTURE CASCADE option:

```
analyze table log_entries validate structure cascade

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.06          0          0          0           0
Execute      1     37.56     127.03      44679   14300957          2           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2     37.57     127.10      44679   14300957          2           0
```

We notice that it causes a massive number of query based I/Os

ANALYZE VALIDATE STRUCTURE FAST

If we trace a session using the new VALIDATE STRUCTURE CASCADE **FAST** option:

```
analyze table log_entries validate structure cascade fast

select /*+ full(LOG_ENTRIES) */ ORA_HASH(DATE_TIME_ACTIONED || rowid)
from LOG_ENTRIES MINUS select /*+ index_ffs(LOG_ENT_DATE_ACT_I) */
  ORA_HASH(DATE_TIME_ACTIONED || rowid) from LOG_ENTRIES

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      5.93      15.49      13022      13190          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      5.94      15.50      13022      13190          0           0

select /*+ full(LOG_ENTRIES) */ ORA_HASH(CASE_ID || DATE_TIME_ADDED ||  rowid)
from LOG_ENTRIES MINUS select /*+  index_ffs(LOG_ENT_CASE_ID_DTETME_ADDED_I) */ ORA_HASH(CASE_ID ||
  DATE_TIME_ADDED || rowid) from LOG_ENTRIES

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      6.22       7.64       1345       3148          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      6.23       7.65       1345       3148          0           0
```

We notice it utilises the ORA_HASH function to compare differences between a full table scan and a fast full index scan

ANALYZE VALIDATE STRUCTURE FAST

If we look at the overall total resources:

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        2      0.00       0.01          2          5          0           0
Execute      2      0.03       0.00      55212      77243          2           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.03       0.02      55214      77248          2           0

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse       42      0.07       0.07          0          0          0           0
Execute     87      0.05       0.07         11          5         60           4
Fetch      140     80.33     111.80      55203      77243          0         240
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total      269     80.45     111.95      55214      77248         60         244
```

We notice it's using significantly less I/Os but much more CPU

Overall, this can sometimes result in the overall elapsed times actually being greater with FAST than without ...

Function-Based Indexes

```
SQL> CREATE TABLE func_tab AS SELECT ROWNUM id, 'DAVID BOWIE '||
ceil(dbms_random.value(0,10000000)) name FROM DUAL CONNECT BY LEVEL <= 100000;
Table created.

SQL> INSERT INTO func_tab VALUES (100001, 'Ziggy');
1 row created.

SQL> INSERT INTO func_tab VALUES (100002, 'ZIGGY');
1 row created.

SQL> INSERT INTO func_tab VALUES (100003, 'ZiGgY');
1 row created.

SQL> commit;
Commit complete.

SQL> CREATE INDEX func_tab_name_i ON func_tab(name);
Index created.

SQL> exec dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=>
'FUNC_TAB',cascade=> true, estimate_percent=>null, method_opt=> 'FOR ALL COLUMNS
SIZE 1');
PL/SQL procedure successfully completed.
```

Function-Based Indexes

The use of the UPPER function negates the use of the index.

```
SQL> SELECT * FROM func_tab WHERE UPPER(name) = 'ZIGGY';

        ID NAME
---------- -------
    100001 Ziggy
    100002 ZIGGY
    100003 ZiGgY

Execution Plan
----------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          |  1000 | 24000 |    89   (5)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| FUNC_TAB |  1000 | 24000 |    89   (5)| 00:00:01 |
----------------------------------------------------------------------------

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

Note also that the cardinality estimate of 1000 rows is way off ...

Function-Based Indexes

```
SQL> CREATE INDEX func_tab_upper_name_i ON func_tab(UPPER(name))
       COMPUTE STATISTICS;
Index created.

SQL> SELECT * FROM func_tab WHERE UPPER(name) = 'ZIGGY';

     ID NAME
---------- --------------------------------------------------
   100001 Ziggy
   100002 ZIGGY
   100003 ZiGgY

Execution Plan
----------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                  | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                       |  1000 | 24000 | 84      (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| FUNC_TAB              |  1000 | 24000 | 84      (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | FUNC_TAB_UPPER_NAME_I |   400 |       |  3      (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------
```

The cardinality estimate is still way off at 1000 rows when only 3 rows are actually returned ...

Function-based index has fortunately been used but such wrong cardinality estimates can potentially result in disastrous execution plans.

Virtual Columns

- When a Function-Based Index is created, Oracle creates a “Virtual” column for the table
- Prior to 11g, these Virtual columns were “Hidden”
- These columns are used to store column statistics to be subsequently used by the CBO
- The table, not the index, needs to have statistics collected to populate Virtual columns

Virtual Columns

```
SQL> SELECT table_name, column_name, num_distinct, density
       FROM dba_tab_columns WHERE table_name = 'FUNC_TAB';

TABLE_NAME   COLUMN_NAME  NUM_DISTINCT    DENSITY
------------ ------------ ------------ ----------
FUNC_TAB     ID                 100003 9.9997E-06
FUNC_TAB     NAME                99495 .000010051

SQL> SELECT table_name, column_name, num_distinct, density, virtual_column,
       hidden_column FROM dba_tab_cols WHERE table_name = 'FUNC_TAB';

TABLE_NAME   COLUMN_NAME  NUM_DISTINCT    DENSITY VIR HID
------------ ------------ ------------ ---------- --- ---
FUNC_TAB     SYS_NC00003$                        YES YES
FUNC_TAB     NAME                99495 .000010051 NO  NO
FUNC_TAB     ID                 100003 9.9997E-06 NO  NO

SQL> SELECT column_name, num_distinct, density, avg_col_len
       FROM dba_tab_col_statistics  WHERE table_name = 'FUNC_TAB';

COLUMN_NAME  NUM_DISTINCT    DENSITY AVG_COL_LEN
------------ ------------ ---------- -----------
ID                 100003 9.9997E-06           5
NAME                99495 .000010051          18
```

Virtual Columns

SQL> exec dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=> 'FUNC_TAB', cascade=> true, estimate_percent=>null, method_opt=> **'FOR ALL HIDDEN COLUMNS SIZE 1**');

PL/SQL procedure successfully completed.

SQL> SELECT table_name, column_name, num_distinct, density, virtual_column, hidden_column FROM dba_tab_cols WHERE table_name = 'FUNC_TAB';

```
TABLE_NAME   COLUMN_NAME   NUM_DISTINCT    DENSITY VIR HID
------------ ------------ ------------ ---------- --- ---
FUNC_TAB     SYS_NC00003$        99493 .000010051 YES YES
FUNC_TAB     NAME                99495 .000010051 NO  NO
FUNC_TAB     ID                 100003 9.9997E-06 NO  NO
```

Note: the virtual statistics are now populated ...

Virtual Columns

```
SQL> SELECT * FROM func_tab WHERE UPPER(name) = 'ZIGGY';

 ID NAME
---------- --------------------------------------------------
    100001 Ziggy
    100002 ZIGGY
    100003 ZiGgY

Execution Plan
----------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name                 | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                      |     1 |    44 |     5   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| FUNC_TAB             |     1 |    44 |     5   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | FUNC_TAB_UPPER_NAME_I |     1 |       |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------------
```

Note: statistics are now much more accurate, remembering that all rows are unique, except the “ZIGGY” rows ...

11g Virtual Columns

- Prior to 11g, virtual columns were created through Function-Based Indexes
- Virtual columns useful due to statistics but had to create index even if it wasn't required
- Virtual columns were "hidden" and had to be manually derived in queries
- 11g now allows Virtual Columns to be defined and visible in a table definition

11g Virtual Columns

SQL> drop index func_tab_upper_name_i;
Index dropped.

SQL> ALTER TABLE func_tab **ADD (upper_name AS (UPPER(name)))**;
Table altered.

SQL> desc func_tab

Name	Null?	Type
ID		NUMBER
NAME		VARCHAR2(52)
UPPER_NAME		**VARCHAR2(52)**

SQL> SELECT * FROM user_indexes WHERE table_name = 'FUNC_TAB';
no rows selected

SQL> exec dbms_stats.gather_table_stats(ownname=>'BOWIE', tabname=> 'FUNC_TAB', cascade=> true, estimate_percent=>null, method_opt=> 'FOR ALL COLUMNS SIZE 1');
PL/SQL procedure successfully completed.

Can define a visible, virtual column to a table ...

11g Virtual Columns

```
SQL> SELECT table_name, column_name, num_distinct, density
       FROM dba_tab_columns WHERE table_name = 'FUNC_TAB';

TABLE_NAME COLUMN_NAME  NUM_DISTINCT    DENSITY
---------- ------------ ------------ ----------
FUNC_TAB   ID                100003 9.9997E-06
FUNC_TAB   NAME               99495 .000010051
FUNC_TAB   UPPER_NAME         99493 .000010051

SQL> SELECT table_name, column_name, num_distinct, density, virtual_column,
       hidden_column FROM dba_tab_cols WHERE table_name = 'FUNC_TAB';

TABLE_NAME COLUMN_NAME  NUM_DISTINCT    DENSITY VIR HID
---------- ------------ ------------ ---------- --- ---
FUNC_TAB   UPPER_NAME         99493 .000010051 YES NO
FUNC_TAB   NAME               99495 .000010051 NO  NO
FUNC_TAB   ID                100003 9.9997E-06 NO  NO

SQL> SELECT column_name, num_distinct, density, avg_col_len FROM dba_tab_col_statistics
       WHERE table_name = 'FUNC_TAB';

COLUMN_NAME NUM_DISTINCT    DENSITY AVG_COL_LEN
----------- ------------ ---------- -----------
UPPER_NAME         99493 .000010051          18
NAME               99495 .000010051          18
ID                100003 9.9997E-06           5
```

11g Virtual Columns

Partial block dump of table

```
block_row_dump:
tab 0, row 0, @0x1f69
tl: 23 fb: --H-FL-- lb: 0x0  cc: 2
col  0: [ 3]  c2 06 58
col  1: [15]  44 41 56 49 44 20 42 4f 57 49 45 20 35 38 37
tab 0, row 1, @0x1f52
tl: 23 fb: --H-FL-- lb: 0x0  cc: 2
col  0: [ 3]  c2 06 59
col  1: [15]  44 41 56 49 44 20 42 4f 57 49 45 20 35 38 38
tab 0, row 2, @0x1f3b
tl: 23 fb: --H-FL-- lb: 0x0  cc: 2
col  0: [ 3]  c2 06 5a
col  1: [15]  44 41 56 49 44 20 42 4f 57 49 45 20 35 38 39
```

Note each row only has the 2 column values physically stored in the table

11g Virtual Columns

```
SQL> SELECT * FROM func_tab WHERE UPPER(name) = 'ZIGGY';

        ID NAME       UPPER_NAME
---------- ---------- ----------
    100001 Ziggy      ZIGGY
    100002 ZIGGY      ZIGGY
    100003 ZiGgY      ZIGGY

Execution Plan
----------------------------------------------------------------------------
| Id  | Operation         | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |          |     1 |    44 |    89   (5)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| FUNC_TAB |     1 |    44 |    89   (5)| 00:00:01 |
----------------------------------------------------------------------------

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

11g Virtual Columns

```
SQL> create index upper_name_idx on func_tab(upper(name));
create index upper_name_idx on func_tab(upper(name))
                               *
ERROR at line 1:
ORA-54018: A virtual column exists for this expression

SQL> create index upper_name_idx on func_tab(upper_name);

Index created.

SQL> SELECT index_name, index_type FROM user_indexes WHERE index_name = 'UPPER_NAME_IDX';

INDEX_NAME                     INDEX_TYPE
------------------------------ ---------------------------
UPPER_NAME_IDX                 FUNCTION-BASED NORMAL

SQL> SELECT index_name, column_expression FROM user_ind_expressions WHERE index_name =
'UPPER_NAME_IDX';

INDEX_NAME                     COLUMN_EXPRESSION
------------------------------ --------------------
UPPER_NAME_IDX                 UPPER("NAME")
```

11g Virtual Columns

```
SQL> select * from func_tab where upper_name = 'ZIGGY';

        ID NAME       UPPER_NAME
---------- ---------- ----------
    100001 Ziggy      ZIGGY
    100002 ZIGGY      ZIGGY
    100003 ZiGgY      ZIGGY

Execution Plan
----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                |     1 |    45 |     5   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| FUNC_TAB       |     1 |    45 |     5   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | UPPER_NAME_IDX |     1 |       |     3   (0)| 00:00:01 |
----------------------------------------------------------------------------------------------

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

11g Virtual Columns

Virtual columns can be used where non-virtual columns are used, including as the partitioning key in a partitioned table:

```
SQL> CREATE TABLE album_sales(id number, album_id number, country_id number,
  2  release_date date, total_sales number, total_value as (total_sales*10.95))  PARTITION BY RANGE
(total_value)
  3  (PARTITION ALBUMS_POOR       VALUES LESS THAN (100000),
  4   PARTITION ALBUMS_500000     VALUES LESS THAN (500001),
  5   PARTITION ALBUMS_1000000    VALUES LESS THAN (1000001),
  6   PARTITION ALBUMS_2000000    VALUES LESS THAN (2000001),
  7   PARTITION ALBUMS_3000000    VALUES LESS THAN (3000001),
  8   PARTITION ALBUMS_4000000    VALUES LESS THAN (4000001),
  9   PARTITION ALBUMS_5000000    VALUES LESS THAN (5000001),
 10   PARTITION ALBUMS_6000000    VALUES LESS THAN (6000001),
 11   PARTITION ALBUMS_7000000    VALUES LESS THAN (7000001),
 12   PARTITION ALBUMS_8000000    VALUES LESS THAN (8000001),
 13   PARTITION ALBUMS_9000000    VALUES LESS THAN (9000001),
 14   PARTITION ALBUMS_BESTSELLERS VALUES LESS THAN (MAXVALUE));

Table created.
```

http://richardfoote.wordpress.com/

Thank you ☺