Everything You Wanted To Know About FIRST_ROWS_n But Were Afraid To Ask

Randolf Geist

Who am I

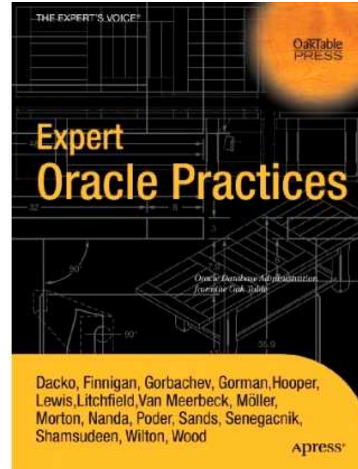
- Independent Consultant
- Located in Germany
- Oracle ACE, OCP 8i, 9i, 10g
- Member of the OakTable Network

- My Blog: Oracle related stuff

2

Who am I

Co-author of the forthcoming OakTable book "Expert Oracle Practices"



3

Highlights

- Introduction to the Cost-Based Optimizer's (CBO) OPTIMIZER_MODEs
- Evolution of the CBO – FIRST_ROWS, FIRST_ROWS_n
- Top N / Pagination queries, advanced techniques
- A deeper look at FIRST_ROWS_n, examples, quirks, oddities
- Questions and Answers

Everything You Wanted To Know About FIRST_ROWS_n

Introduction to the Cost-Based Optimizer's (CBO) OPTIMIZER_MODEs

5

Cost-Based Optimizer (CBO) - OPTIMIZER_MODEs

- CHOOSE between Rule Based Optimizer (RBO) and CBO: Default in pre-10g
- ALL_ROWS: Default in 10g and later
- FIRST_ROWS: Deprecated since 9i
- Oracle 9i introduced FIRST_ROWS_1, FIRST_ROWS_10, FIRST_ROWS_100 and FIRST_ROWS_1000 along with the FIRST_ROWS(n) hint

Cost-Based Optimizer (CBO) - OPTIMIZER_MODEs

When to use which mode?

- Often heard:

  "In OLTP environments use FIRST_ROWS / FIRST_ROWS_n, because typically you retrieve only a few rows with the queries"

Cost-Based Optimizer (CBO) - OPTIMIZER_MODEs

Philosophy ALL_ROWS:

The optimizer assumes that all rows from the generated result set will be processed / fetched by the client

This is true in most cases, therefore ALL_ROWS is a good general approach and default from 10g on

8

Cost-Based Optimizer (CBO) - OPTIMIZER_MODEs

Philosophy ALL_ROWS

- What about the "OLTP environment" statement?
- Assuming that the optimizer estimates the correct cardinality, queries identifying only a few rows still retrieve ***all*** of these few rows, so ALL_ROWS is applicable here as well, no need to use FIRST_ROWS/FIRST_ROWS_n

Cost-Based Optimizer (CBO) - OPTIMIZER_MODEs

So when do we need FIRST_ROWS(_n)?

Philosophy FIRST_ROWS(_n):

- The optimizer assumes that only a ***few*** first rows from a ***larger*** result set will be processed / fetched by the client

Cost-Based Optimizer (CBO) - OPTIMIZER_MODEs

In most cases these are so called Top N / Pagination queries:

- Show the top N rows from an ordered result set. Popular examples: Search results in Google / Amazon
- Retrieve the result set in pages - Pagination queries
- This is only reasonable if a deterministic order has been defined for the result set

11

Cost-Based Optimizer (CBO) - OPTIMIZER_MODEs

- If you don’t retrieve only the first rows from a larger result set but the FIRST_ROWS(_n) modes speed up the query processing, you very likely only were lucky due to the side effects of the FIRST_ROWS(_n) modes
- You should be able to get the same performance with ALL_ROWS provided that the optimizer estimates are in the right ballpark

12

Later we will see that the optimizer will still get it right when the query returns only a few rows, all of them are fetched/processed, the estimates are in the right ballpark and the FIRST_ROWS(n) mode gets used with n >= number of rows returned by the query

Evolution of the CBO – FIRST_ROWS, FIRST_ROWS_n

13

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

- FIRST_ROWS introduced with the CBO in Version 7
- FIRST_ROWS uses a mixture of heuristics and cost based optimization
- Some built-in high-level constraints are:
  - Use index to avoid SORT operation
  - Try to avoid Hash / Sort Merge Join, use NESTED LOOP join

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

Example Setup (1)

```
create table first_k_rows_test
(
   sort_order   varchar2(10) null,
   filter_col   number       null,
   filler char(200)
);

insert into first_k_rows_test
  select  dbms_random.string('x', 10) as sort_order,
          case when level <= 10 then 1 else 0 end as filter_col,
          'x' as filler
          from dual
          connect by level <= 100000;

commit;
```

15

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

Example Setup (2)

```
create index first_k_rows_test_idx1 on first_k_rows_test(filter_col);

create index first_k_rows_test_idx2 on first_k_rows_test(sort_order);

exec dbms_stats.gather_table_stats(null, 'first_k_rows_test',
method_opt=>'for all columns size 1 for columns filter_col size 254');
```

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

ALL_ROWS Baseline

```
select  /*+ all_rows */
        *
from
        first_k_rows_test a,
        first_k_rows_test b
where
        a.filter_col = 1
and     a.sort_order = b.sort_order
order by
        a.sort_order;
```

17

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

ALL_ROWS Baseline Plan

```
--------------------------------------------------------------------------------------
| Id  | Operation                       | Name                    | Rows  | Cost (%CPU)|
--------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                |                         |    10 |    23   (5)|
|   1 |  SORT ORDER BY                  |                         |    10 |    23   (5)|
|   2 |   NESTED LOOPS                  |                         |    10 |    22   (0)|
|   3 |    TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST       |    10 |     2   (0)|
|*  4 |     INDEX RANGE SCAN            | FIRST_K_ROWS_TEST_IDX1 |    10 |     1   (0)|
|   5 |    TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST       |     1 |     2   (0)|
|*  6 |     INDEX RANGE SCAN            | FIRST_K_ROWS_TEST_IDX2 |     1 |     1   (0)|
--------------------------------------------------------------------------------------

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

   4 - access("A"."FILTER_COL"=1)
   6 - access("A"."SORT_ORDER"="B"."SORT_ORDER")
```

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

FIRST_ROWS query

```
select  /*+ first_rows */
        *
from
        first_k_rows_test a,
        first_k_rows_test b
where
        a.filter_col = 1
and     a.sort_order = b.sort_order
order by
        a.sort_order;
```

19

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

FIRST_ROWS plan

```
-------------------------------------------------------------------------------------
| Id  | Operation                    | Name                    | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                         |     1 |   100K   (1)|
|   1 |  NESTED LOOPS                |                         |     1 |   100K   (1)|
|*  2 |   TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST       |     1 |   100K   (1)|
|   3 |    INDEX FULL SCAN           | FIRST_K_ROWS_TEST_IDX2  |  100K|   432    (2)|
|   4 |   TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST       |     1 |     2    (0)|
|*  5 |    INDEX RANGE SCAN          | FIRST_K_ROWS_TEST_IDX2  |     1 |     1    (0)|
-------------------------------------------------------------------------------------

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

   2 - filter("A"."FILTER_COL"=1)
   5 - access("A"."SORT_ORDER"="B"."SORT_ORDER")
```

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

- These high-level constraints can lead to inefficient execution plans:
  - Favoring an inefficient index over an highly selective index
  - Using inefficient NESTED LOOP joins where other join methods can be more efficient
- `_sort_elimination_cost_ratio` parameter influences FIRST_ROWS calculation

21

In previous releases (< 9.2.0.8) the FIRST_ROWS mode by default was avoiding the sort at all costs - in recent releases this has changed, but you can still get rather inefficient plans albeit the change - the gory details follow in the last part

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

- FIRST_ROWS_n introduced in Oracle 9i, FIRST_ROWS deprecated
- FIRST_ROWS_n: Fully cost based, no built-in heuristics (?)
- However: Still some internal tweaks that favor index access under certain circumstances, so it doesn't choose always the plan with the lowest cost

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

FIRST_ROWS_n approach:

- The optimizer knows about the requested number of rows (the “n” in FIRST_ROWS_n)
- Either explicitly set via OPTIMIZER_MODEs:
  - FIRST_ROWS_[1 | 10 | 100 | 1000]
  - or via FIRST_ROWS(n) hint: e.g. FIRST_ROWS(50)

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

FIRST_ROWS_n approach:

- Or implicitly via ROWNUM predicates:
  SELECT
  FROM (query [ORDER BY order])
  WHERE ROWNUM <= n
- "_optimizer_rownum_pred_based_fkr" parameter applies, default "true" in recent releases

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

FIRST_ROWS_n approach:

- Knowing the “n” is essential, because the optimizer now works out how many rows the result set is estimated to produce
- This is basically done performing an initial (and partial) ALL_ROWS optimization

25

Cost-Based Optimizer (CBO) - FIRST_ROWS(_n) Evolution

FIRST_ROWS_n approach:

- With the knowledge of “n” and the total number of rows estimated, the optimizer can calculate a “proration factor”, which is basically

  n / total_number_of_rows

- This “proration factor” is then used in the following optimization steps when calculating costs

Everything You Wanted To Know About FIRST_ROWS_n

Top N / Pagination queries, advanced techniques

27

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Modified example setup

```
truncate table first_k_rows_test;

insert into first_k_rows_test
  select  dbms_random.string('x', 10),
          case when level <= 5000 then 1 else 0 end,
          'x' as filler
          from dual
          connect by level <= 100000;

exec dbms_stats.gather_table_stats(null, 'first_k_rows_test',
method_opt=>'for all columns size 1 for columns filter_col size 254');
```

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Simple Top N query

```
select
        *
from    (
        select
                t.*
        from
                first_k_rows_test t
        order by
                sort_order
        )
where
        rownum <= 10;
```

29

We expect to get only 10 rows and we have an index in place on SORT_ORDER. Let's examine the execution plan

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan - first attempt

```
-------------------------------------------------------------------------
| Id  | Operation                | Name               | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |                    |    10 |  5516   (1)|
|*  1 |  COUNT STOPKEY           |                    |       |            |
|   2 |   VIEW                   |                    |   100K|  5516   (1)|
|*  3 |    SORT ORDER BY STOPKEY|                    |   100K|  5516   (1)|
|   4 |     TABLE ACCESS FULL    | FIRST_K_ROWS_TEST |   100K|   833   (2)|
-------------------------------------------------------------------------

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

   1 - filter(ROWNUM<=10)
   3 - filter(ROWNUM<=10)
```

Index doesn't get used - why?

30

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Example Setup

```
create table first_k_rows_test
(
   sort_order   varchar2(10) null,
   filter_col   number       null,
   filler char(200)
);
```

B*tree indexes don't cover expressions that consist entirely of NULLs!

31

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

B*tree index can only be used for ordering if the index covers all values (NULLs!)

- Either define column as NOT NULL or have predicates in place that imply NOT NULL
- Or make the index cover NULL values e.g. contains a NOT NULL column or use function-based index:

```
create index first_k_rows_test_idx2 on first_k_rows_test(
sort_order,
' ');
```

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan - second attempt

```
---------------------------------------------------------------------------------------
| Id  | Operation                      | Name                    | Rows  | Cost (%CPU)|
---------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                         |    10 |    12   (0)|
|*  1 |  COUNT STOPKEY                 |                         |       |            |
|   2 |   VIEW                         |                         |    10 |    12   (0)|
|   3 |    TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST       |  100K|    12   (0)|
|   4 |     INDEX FULL SCAN            | FIRST_K_ROWS_TEST_IDX2 |    10 |     2   (0)|
---------------------------------------------------------------------------------------

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

   1 - filter(ROWNUM<=10)
```

33

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Modified example setup

```
drop index first_k_rows_test_idx1;

drop index first_k_rows_test_idx2;

create unique index first_k_rows_test_idx on first_k_rows_test
(
  filter_col
, sort_order
);
```

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Top N query including filter

```
select
        *
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        order by
                sort_order
        )
where
        rownum <= 10;
```

35

We expect to get only 10 rows and we have the ideal index in place supporting both filter and sort. Let's examine the execution plan

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan - first attempt

```
--------------------------------------------------------------------------
| Id  | Operation                 | Name              | Rows  | Cost (%CPU)|
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |                   |    10 |  1056   (2)|
|*  1 |  COUNT STOPKEY            |                   |       |            |
|   2 |   VIEW                    |                   |  4699 |  1056   (2)|
|*  3 |    SORT ORDER BY STOPKEY|                   |  4699 |  1056   (2)|
|   4 |     COUNT                 |                   |       |            |
|*  5 |      TABLE ACCESS FULL    | FIRST_K_ROWS_TEST |  4699 |   834   (2)|
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter(ROWNUM<=10)
   3 - filter(ROWNUM<=10)
   5 - filter("FILTER_COL"=1)
```

Index again doesn't get used - why?

36

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Example Setup

```
create table first_k_rows_test
(
   sort_order   varchar2(10) null,
   filter_col   number       null,
   filler char(200)
);
```

Sorting character strings:
Client NLS settings are relevant!

37

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

NLS sorting issues

- Set NLS_SORT to BINARY in session to support sort by conventional index

```
ALTER SESSION SET NLS_SORT = BINARY;
```

- Create function-based index using NLSSORT function

```
create index first_k_rows_test_idx on first_k_rows (
  filter_col
, NLSSORT(sort_order, 'NLS_SORT=GERMAN')
);
```

38

If you happen to have different NLS settings per client you theoretically need several NLSSORT based indexes - to support each different NLS based sort order

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan - second attempt

```
-------------------------------------------------------------------------------------
| Id  | Operation                        | Name                   | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                 |                        |    10 |    13   (0)|
|*  1 |  COUNT STOPKEY                   |                        |       |            |
|   2 |   VIEW                           |                        |    11 |    13   (0)|
|   3 |    COUNT                         |                        |       |            |
|   4 |     TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST        |    11 |    13   (0)|
|*  5 |      INDEX RANGE SCAN            | FIRST_K_ROWS_TEST_IDX |       |     2   (0)|
-------------------------------------------------------------------------------------


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

   1 - filter(ROWNUM<=10)
   5 - access("FILTER_COL"=1)
```

39

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Pagination query using ROWNUM

```
select
        *
from    (
        select
                rownum as rn
              , v1.*
        from    (
                select
                        t.*
                from
                        first_k_rows_test t
                where
                        filter_col = 1
                order by
                        sort_order
                ) v1
        where
                rownum <= 20
        ) v2
where rn >= 11
order by rn;
```

40

Extending the Top N query to a "Pagination" query - show the ordered result set in pages. The application requests a specific page.

It is obvious that using ROWNUM this is a bit cumbersome to write, since due to the nature how ROWNUM gets evaluated we need to write two levels of inline views.

Note that although you expect the result to be already correctly ordered if the index gets used, you still need to specify the final sort order to have the order of the result set guaranteed.

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Pagination query using ROWNUM

```
select
        *
from    (
        select
                rownum as rn
              , v1.*
        from    (
                select
                        t.*
                from
                        first_k_rows_test t
                where
                        filter_col = 1
                order by
                        sort_order
                ) v1
        where
                rownum <= 20
        ) v2
where rn >= 11
order by rn;
```

This corresponds to the Top N query, but it needs now to identify as many rows as indicated by the page requested

41

This part is the already shown Top N query

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Pagination query using ROWNUM

```
select
        *
from    (
        select
                rownum as rn
              , v1.*
        from    (
                select
                        t.*
                from
                        first_k_rows_test t
                where
                        filter_col = 1
                order by
                        sort_order
                ) v1
        where
                rownum <= 20
        ) v2
where rn >= 11
order by rn;
```

Now we discard all the rows except for those left for the page to display

42

We filter all rows that don't belong to the requested page.

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan

```
-------------------------------------------------------------------------------------
| Id  | Operation                        | Name                   | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                 |                        |    20 |    23   (5)|
|   1 |  SORT ORDER BY                   |                        |    20 |    23   (5)|
|*  2 |   VIEW                           |                        |    20 |    22   (0)|
|*  3 |    COUNT STOPKEY                 |                        |       |            |
|   4 |     VIEW                         |                        |    20 |    22   (0)|
|   5 |      TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST      |    20 |    22   (0)|
|*  6 |       INDEX RANGE SCAN           | FIRST_K_ROWS_TEST_IDX |       |     2   (0)|
-------------------------------------------------------------------------------------

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

   2 - filter("RN">=11)
   3 - filter(ROWNUM<=20)
   6 - access("FILTER_COL"=1)
```

43

Note that the optimizer doesn't recognize the "pagination" - it estimates a final cardinality of 20.

Optionally you could wrap the pagination query into a third view requesting on the first "pagesize" rows via the ROWNUM predicate - this will make sure that the optimizer recognizes the page size requested.

The same will be achieved by using the FIRST_ROWS(n) hint where n equals the pagesize.

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Same query, use FIRST_ROWS(n) hint

```
select  /*+ first_rows(10) */
        *
from    (
        select
                rownum as rn
              , v1.*
        from    (
                select
                        t.*
                from
                        first_k_rows_test t
                where
                        filter_col = 1
                order by
                        sort_order
                ) v1
        where
                rownum <= 20
        ) v2
where rn >= 11
order by rn;
```

44

Here is the same query using the FIRST_ROWS(10) hint - note the subtle difference in the execution plan

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan

```
-------------------------------------------------------------------------------------
| Id  | Operation                         | Name                   | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |                        |    10 |    13   (8)|
|   1 |  SORT ORDER BY                    |                        |    10 |    13   (8)|
|*  2 |   VIEW                            |                        |    10 |    12   (0)|
|*  3 |    COUNT STOPKEY                  |                        |       |            |
|   4 |     VIEW                          |                        |    10 |    12   (0)|
|   5 |      TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST      |    10 |    12   (0)|
|*  6 |       INDEX RANGE SCAN            | FIRST_K_ROWS_TEST_IDX |       |     2   (0)|
-------------------------------------------------------------------------------------

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

   2 - filter("RN">=11)
   3 - filter(ROWNUM<=20)
   6 - access("FILTER_COL"=1)
```

45

The estimated cardinality has dropped to 10

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Pagination query using analytic functions

```
select  /*+ first_rows(10) */
        *
from    (
        select
                row_number() over (order by sort_order) as rn
              , t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        order by
                sort_order
        )
where
        rn between 11 and 20
-- and     rownum <= 10 -- Instead of FIRST_ROWS(10) hint
order by
        rn;
```

46

Analytic functions allow to write the pagination query in a more expressive way. However they require either the FIRST_ROWS(n) hint or adding another ROWNUM predicate to activate the optimizer mode - there is no automatic detection similar to ROWNUM for the analytic function.

Note that in this case we can safely use the final ROWNUM predicate on the same level as the ORDER BY since the filter predicate on RN already identifies the correct rows

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan slightly different

```
-----------------------------------------------------------------------------------
| Id  | Operation                       | Name                   | Rows  | Cost (%CPU)|
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                |                        |    10 |    13   (8)|
|   1 |  SORT ORDER BY                  |                        |    10 |    13   (8)|
|*  2 |   VIEW                          |                        |    10 |    12   (0)|
|*  3 |    WINDOW NOSORT STOPKEY        |                        |    10 |    12   (0)|
|   4 |     TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST      |    10 |    12   (0)|
|*  5 |      INDEX RANGE SCAN           | FIRST_K_ROWS_TEST_IDX |        |     2   (0)|
-----------------------------------------------------------------------------------

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

   2 - filter("RN">=11 AND "RN"<=20)
   3 - filter(ROW_NUMBER() OVER ( ORDER BY "T"."SORT_ORDER")<=20)
   5 - access("FILTER_COL"=1)
```

47

It's interesting to note that the filter predicates suggest that the optimizer should be able to work out the FIRST_ROWS(n) scenario automatically. Also note the discrepancy between the filter predicate (ROW_NUMBER()... <= 20) and the estimates ROWS.

Another point to consider is that the ORDER BY is mandatory to get the FIRST_ROWS(n) mode to work - although the ROW_NUMBER() analytic function implies the sort operation already

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Effect of FIRST_ROWS(n) hint

```
select  /*+ first_rows(10) */
        *
from    (
        select
                row_number() over (order by sort_order) as rn
              , t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        order by
                sort_order
        )
where
        rn between 3991 and 4000
-- and     rownum <= 10 -- Instead of FIRST_ROWS(10) hint
order by
        rn;
```

48

The Analytic Function approach has a disadvantage - using the FIRST_ROWS(n) hint (or the final ROWNUM) confuses the optimizer - see the estimates and actual execution on the next slide

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Optimizer is "clueless" - actual execution

```
-----------------------------------------------------------------------------------------------------
| Id  | Operation                        | Name                   |  Cost | E-Rows | A-Rows | Buffers |
-----------------------------------------------------------------------------------------------------
|   1 |  SORT ORDER BY                   |                        |    13 |     10 |     10 |    3984 |
|*  2 |   VIEW                           |                        |    12 |     10 |     10 |    3984 |
|*  3 |    WINDOW NOSORT STOPKEY         |                        |    12 |     10 |   4000 |    3984 |
|   4 |     TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST      |    12 |     10 |   4001 |    3984 |
|*  5 |      INDEX RANGE SCAN            | FIRST_K_ROWS_TEST_IDX |     2 |        |   4001 |      15 |
-----------------------------------------------------------------------------------------------------

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

   2 - filter(("RN">=3991 AND "RN"<=4000))
   3 - filter(ROW_NUMBER() OVER ( ORDER BY "T"."SORT_ORDER")<=4000)
   5 - access("FILTER_COL"=1)
```

49

The estimated cost clearly doesn't reflect was is happening at runtime

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Things look better with ROWNUM

```
select
        *
from    (
        select
                rownum as rn
              , v1.*
        from    (
                select
                        t.*
                from
                        first_k_rows_test t
                where
                        filter_col = 1
                order by
                        sort_order
                ) v1
        where
                rownum <= 4000
        ) v2
where rn >= 3991
order by rn;
```

50

Requesting further pages with the ROWNUM approach makes the optimizer aware of the additional work required.

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan

```
-----------------------------------------------------------------------------------------
| Id  | Operation                       | Name                    | Rows   | Cost (%CPU)|
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                |                         |  4000 |  4226   (1)|
|   1 |  SORT ORDER BY                  |                         |  4000 |  4226   (1)|
|*  2 |   VIEW                          |                         |  4000 |  4017   (1)|
|*  3 |    COUNT STOPKEY                |                         |        |            |
|   4 |     VIEW                        |                         |  4000 |  4017   (1)|
|   5 |      TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST       |  4000 |  4017   (1)|
|*  6 |       INDEX RANGE SCAN          | FIRST_K_ROWS_TEST_IDX   |        |    14   (0)|
-----------------------------------------------------------------------------------------

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

   2 - filter("RN">=3991)
   3 - filter(ROWNUM<=4000)
   6 - access("FILTER_COL"=1)
```

51

Here the cardinality estimate is better, but as already shown the optimizer doesn't get the pagination right and therefore believes it has to sort 4000 rows.

This example also makes another point very obvious: The standard Top N / Pagination queries are working well when requesting only the first pages, but the work required increases with the page number requested.

Is there a way to minimize the work and ideally have the same amount of work regardless of the page requested?

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Advanced Pagination query - see next page

52

The purpose of this re-write is to avoid the costly access to the table for all the rows that will be discarded since they are prior to the requested page. This is achieved here using an inline view that only selects the ROWID from the original pagination query. This can be performed using the index only - a join is then performed to the original table using the ROWID.

Note that Oracle performs a similar operation "under the covers" when using sorted hash clusters - it gets the ROWIDs, sorts them and then gets the maining data from the cluster. This avoids large sort operations, but has the potential overhead of getting single rows by ROWIDs.

This way the performance ought to be almost the same independent from the number of pages requested.

The additional ROWNUM predicate limiting the inner set to 10 rows is necessary here, otherwise we see again that the optimizer doesn't recognize the "pagination" and estimates 4000 rows to join with the original table - eventually leading to a different join mechanism used with increasing number of rows.

```
select
        i.rn
      , t.*
from    (
        select
                *
        from
                (
                select
                        rownum as rn
                      , v1.*
                from    (
                        select
                                t.rowid as row_id
                        from
                                first_k_rows_test t
                        where
                                filter_col = 1
                        order by
                                sort_order
                        ) v1
                where
                        rownum <= 4000
                )
        where
                rn >= 3991
        and     rownum <= 10
        ) i
        , first_k_rows_test t
where
        t.rowid = i.row_id
order by
        rn;
```

53

The purpose of this re-write is to avoid the costly access to the table for all the rows that will be discarded since they are prior to the requested page. This is achieved here using an inline view that only selects the ROWID from the original pagination query. This can be performed using the index only - a join is then performed to the original table using the ROWID.

Note that Oracle performs a similar operation "under the covers" when using sorted hash clusters - it gets the ROWIDs, sorts them and then gets the maining data from the cluster. This avoids large sort operations, but has the potential overhead of getting single rows by ROWIDs.

This way the performance ought to be almost the same independent from the number of pages requested.

The additional ROWNUM predicate limiting the inner set to 10 rows is necessary here, otherwise we see again that the optimizer doesn't recognize the "pagination" and estimates 4000 rows to join with the original table - eventually leading to a different join mechanism used with increasing number of rows.

```
select
        i.rn
      , t.*
from    (
        select
                *
        from
                (
                select
                        rownum as rn
                      , v1.*
                from    (
                        select
                                t.rowid as row_id
                        from
                                first_k_rows_test t
                        where
                                filter_col = 1
                        order by
                                sort_order
                        ) v1
                where
                        rownum <= 4000
                )
        where
                rn >= 3991
        and     rownum <= 10
        ) i
        , first_k_rows_test t
where
        t.rowid = i.row_id
order by
        rn;
```

Get the first 4000 rows, ROWID only, therefore index only operation

54

This is achieved here using an inline view that only selects the ROWID from the original pagination query. This can be performed using the index only

Discard all rows (ROWIDs) of the previous pages except for the page requested

```
select
        i.rn
      , t.*
from    (
        select
                *
        from
                (
                select
                        rownum as rn
                      , v1.*
                from    (
                        select
                                t.rowid as row_id
                        from
                                first_k_rows_test t
                        where
                                filter_col = 1
                        order by
                                sort_order
                        ) v1
                where
                        rownum <= 4000
                )
        where
                rn >= 3991
        and     rownum <= 10
        ) i
        , first_k_rows_test t
where
        t.rowid = i.row_id
order by
        rn;
```

55

All rows (effectively ROWIDs only) not required are discarded in the next step

The additional ROWNUM predicate limiting the inner set to 10 rows is necessary here, otherwise we see again that the optimizer doesn't recognize the "pagination" and estimates 4000 rows to join with the original table - eventually leading to a different join mechanism used with increasing number of rows.

Finally pick only the rows from the table that belong to the page requested - minimizing the necessary visits to the table itself

```
select
        i.rn
      , t.*
from    (
        select
                *
        from
                (
                select
                        rownum as rn
                      , v1.*
                from    (
                        select
                                t.rowid as row_id
                        from
                                first_k_rows_test t
                        where
                                filter_col = 1
                        order by
                                sort_order
                        ) v1
                where
                        rownum <= 4000
                )
        where
                rn >= 3991
        and     rownum <= 10
        ) i
        , first_k_rows_test t
where
        t.rowid = i.row_id
order by
        rn;
```

56

Finally a join is then performed to the original table using the ROWID.

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan

```
---------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                   | Starts | E-Rows | A-Rows | Buffers |
---------------------------------------------------------------------------------------------------
|   1 |  SORT ORDER BY                |                        |      1 |     10 |     10 |      25 |
|   2 |   NESTED LOOPS                |                        |      1 |     10 |     10 |      25 |
|   3 |    VIEW                       |                        |      1 |     10 |     10 |      15 |
|*  4 |     COUNT STOPKEY             |                        |      1 |        |     10 |      15 |
|*  5 |      VIEW                     |                        |      1 |   4000 |     10 |      15 |
|*  6 |       COUNT STOPKEY           |                        |      1 |        |   4000 |      15 |
|   7 |        VIEW                   |                        |      1 |   4718 |   4000 |      15 |
|*  8 |         INDEX RANGE SCAN      | FIRST_K_ROWS_TEST_IDX  |      1 |   4718 |   4000 |      15 |
|   9 |    TABLE ACCESS BY USER ROWID| FIRST_K_ROWS_TEST      |     10 |      1 |     10 |      10 |
---------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter(ROWNUM<=10)
   5 - filter("RN">=3991)
   6 - filter(ROWNUM<=4000)
   8 - access("FILTER_COL"=1)
```

57

Compare this to the simple pagination query - the number of buffer gets is much less.

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Advanced Pagination query using subquery
see next page

58

This re-write is based on the assumption that the order expression is unique. In this case we can use a subquery to identify the first row of our page.

This can be done using the existing index as an index-only operation and is therefore very efficient.

The outer query then simply retrieves the n rows of the page again using the same index using the row determined to be the first one as start.

```
select
        3991 + rownum - 1 as rn
      , t.*
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        and     sort_order >=
                (
                select
                        sort_order
                from    (
                        select
                                sort_order
                              , rownum rn
                        from    (
                                select
                                        sort_order
                                from
                                        first_k_rows_test t
                                where
                                        filter_col = 1
                                order by
                                        sort_order
                                )
                        where rownum <= 3991
                        )
                where rn = 3991
                )
        order by sort_order
        ) t
where rownum <= 10;
```

59

This re-write is based on the assumption that the order expression is unique. In this case we can use a subquery to identify the first row of our page.

This can be done using the existing index as an index-only operation and is therefore very efficient.

The outer query then simply retrieves the n rows of the page again using the same index using the row determined to be the first one as start.

```
select
        3991 + rownum - 1 as rn
      , t.*
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        and     sort_order >=
                (
                select
                        sort_order
                from    (
                        select
                                sort_order
                              , rownum rn
                        from    (
                                select
                                        sort_order
                                from
                                        first_k_rows_test t
                                where
                                        filter_col = 1
                                order by
                                        sort_order
                                )
                        where rownum <= 3991
                        )
                where rn = 3991
                )
        order by sort_order
        ) t
where rownum <= 10;
```

Get the SORT_ORDER for the first 3991 rows, index-only

60

The innermost query identifies the order expression of all the rows up to the first row of the requested page. This can be done as an index-only operation in our case.

```
select
        3991 + rownum - 1 as rn
      , t.*
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        and     sort_order >=
                (
                select
                        sort_order
                from    (
                        select
                                sort_order
                              , rownum rn
                        from    (
                                select
                                        sort_order
                                from
                                        first_k_rows_test t
                                where
                                        filter_col = 1
                                order by
                                        sort_order
                                )
                        where rownum <= 3991
                        )
                where rn = 3991
                )
        order by sort_order
        ) t
where rownum <= 10;
```

Discard all rows except for the first row of the requested page

61

All rows will be discarded except for the first row of the identified order expressions

```
select
        3991 + rownum - 1 as rn
      , t.*
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        and     sort_order >=
                (
                select
                        sort_order
                from    (
                        select
                                sort_order
                              , rownum rn
                        from    (
                                select
                                        sort_order
                                from
                                        first_k_rows_test t
                                where
                                        filter_col = 1
                                order by
                                        sort_order
                                )
                        where rownum <= 3991
                        )
                where rn = 3991
                )
        order by sort_order
        ) t
where rownum <= 10;
```

Get the actual rows from the table, using the same filter expression again, but starting with the first row identified in the subquery

62

The outer query then simply retrieves the n rows of the page again using the same index using the row determined to be the first one as start.

Limit the number of rows accessed to the page size - so we need to visit only as many table rows as the page size

```
select
        3991 + rownum - 1 as rn
      , t.*
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        and     sort_order >=
                (
                select
                        sort_order
                from    (
                        select
                                sort_order
                              , rownum rn
                        from    (
                                select
                                        sort_order
                                from
                                        first_k_rows_test t
                                where
                                        filter_col = 1
                                order by
                                        sort_order
                                )
                        where rownum <= 3991
                        )
                where rn = 3991
                )
        order by sort_order
        ) t
where rownum <= 10;
```

63

So again the number of rows that need to be obtained from the actual table are limited to the number of rows per page.

Cost-Based Optimizer (CBO) - Top N - Pagination Queries

Execution plan

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                       | Name                  | Starts | E-Rows | A-Rows | Buffers |
-------------------------------------------------------------------------------------------------
|*  1 |  COUNT STOPKEY                  |                       |      1 |        |     10 |      27 |
|   2 |   VIEW                          |                       |      1 |     10 |     10 |      27 |
|   3 |    TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST     |      1 |     10 |     10 |      27 |
|*  4 |     INDEX RANGE SCAN            | FIRST_K_ROWS_TEST_IDX |      1 |     42 |     10 |      17 |
|*  5 |      VIEW                       |                       |      1 |    200 |      1 |      15 |
|*  6 |       COUNT STOPKEY             |                       |      1 |        |   3991 |      15 |
|   7 |        VIEW                     |                       |      1 |    200 |   3991 |      15 |
|*  8 |         INDEX RANGE SCAN        | FIRST_K_ROWS_TEST_IDX |      1 |    200 |   3991 |      15 |
-------------------------------------------------------------------------------------------------

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

   1 - filter(ROWNUM<=10)
   4 - access("FILTER_COL"=1 AND "SORT_ORDER">= AND "SORT_ORDER" IS NOT NULL)
   5 - filter("RN"=3991)
   6 - filter(ROWNUM<=3991)
   8 - access("FILTER_COL"=1)
```

64

As you can see - although the query looks much more complicated - it is almost as efficient as the one joining by ROWID - the only difference is the second access to the index which adds only a very small overhead (two buffer gets in this example here to retrieve the 10 rows of the page).

One advantage of this query is that we don't need to join by ROWID but can simply use the existing index (twice).

Everything You Wanted To Know About FIRST_ROWS_n

A deeper look at FIRST_ROWS(_n), examples, quirks, oddities

65

A Deeper Look - FIRST_ROWS mechanics

Overview of FIRST_ROWS mode (1)

- Two different scenarios
  - Single table access
  - Joins
- `_sort_elimination_cost_ratio` defines the "breakpoint" between index usage to avoid sort and plans with a sort operation

A Deeper Look - FIRST_ROWS mechanics

Overview of FIRST_ROWS mode (2)

- `_sort_elimination_cost_ratio` default value = 0, see example next slides
- When setting it to a non-default, positive integer then the index to avoid the sort is used if the cost of using it is less than the cost of doing the sort multiplied by the value of the parameter

A Deeper Look - FIRST_ROWS mechanics

_sort_elimination_cost_ratio = 0

Example Setup (1)

```
create table first_k_rows_test
(
   sort_order   varchar2(10) not null,
   filter_col   number       null,
   filler char(200)
);

insert into first_k_rows_test
  select  dbms_random.string('x', 10) as sort_order,
          case when level <= 120 then 1 else 0 end as filter_col,
          'x' as filler
          from dual
          connect by level <= 100000;

commit;
```

68

A Deeper Look - FIRST_ROWS mechanics

Example Setup (2)

```
create index first_k_rows_test_idx2 on first_k_rows_test(sort_order);

exec dbms_stats.gather_table_stats(null, 'first_k_rows_test',
method_opt=>'for all columns size 1 for columns filter_col size 254');
```

69

A Deeper Look - FIRST_ROWS mechanics

ALL_ROWS Baseline

```
select  /*+ all_rows */
        *
from
        first_k_rows_test
where
        filter_col = 1
order by
        sort_order;
```

70

A Deeper Look - FIRST_ROWS mechanics

ALL_ROWS Baseline Plan

```
--------------------------------------------------------------------
| Id  | Operation          | Name              | Rows   | Cost (%CPU)|
--------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                   |   120 |    835   (2)|
|   1 |  SORT ORDER BY     |                   |   120 |    835   (2)|
|*  2 |   TABLE ACCESS FULL| FIRST_K_ROWS_TEST |   120 |    834   (2)|
--------------------------------------------------------------------

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

   2 - filter("FILTER_COL"=1)
```

71

A Deeper Look - FIRST_ROWS mechanics

FIRST_ROWS query

```
select  /*+ first_rows */
        *
from
        first_k_rows_test
where
        filter_col = 1
order by
        sort_order;
```

72

A Deeper Look - FIRST_ROWS mechanics

FIRST_ROWS Plan

```
-------------------------------------------------------------------------
| Id  | Operation             | Name                | Rows   | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                     |   120 |   835    (2)|
|   1 |  SORT ORDER BY        |                     |   120 |   835    (2)|
|*  2 |   TABLE ACCESS FULL| FIRST_K_ROWS_TEST |   120 |   834    (2)|
-------------------------------------------------------------------------

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

   2 - filter("FILTER_COL"=1)
```

73

A Deeper Look - FIRST_ROWS mechanics

Example Setup (3) - slight modification

```
truncate table first_k_rows_test;

insert into first_k_rows_test
  select  dbms_random.string('x', 10) as sort_order,
          case when level <= 121 then 1 else 0 end as filter_col,
          'x' as filler
          from dual
          connect by level <= 100000;

commit;

exec dbms_stats.gather_table_stats(null, 'first_k_rows_test',
method_opt=>'for all columns size 1 for columns filter_col size 254');
```

74

A Deeper Look - FIRST_ROWS mechanics

FIRST_ROWS Plan

```
-----------------------------------------------------------------------------------
| Id  | Operation                    | Name                    | Rows  | Cost (%CPU)|
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                         |   121 |   100K   (1)|
|*  1 |  TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST       |   121 |   100K   (1)|
|   2 |   INDEX FULL SCAN            | FIRST_K_ROWS_TEST_IDX2 |  100K|   426    (2)|
-----------------------------------------------------------------------------------

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

   1 - filter("FILTER_COL"=1)
```

120 * 835 = 100,200

121 * 835 = 101,035

_sort_elimination_cost_ratio = num_rows

75

It looks like that in recent releases the former behaviour of treating the default setting of _sort_elimination_cost_ratio = 0 as "infinite" cost for sort operations and therefore avoiding the sort at all costs has changed to setting it to the number of rows estimated to be returned by the query. However this seems only to apply to single-table queries.

A Deeper Look - FIRST_ROWS mechanics

Example Setup (4) - Joins

```
truncate table first_k_rows_test;

insert into first_k_rows_test
  select  dbms_random.string('x', 10) as sort_order,
          case when level <= 10 then 1 else 0 end as filter_col,
          'x' as filler
          from dual
          connect by level <= 100000;

commit;

exec dbms_stats.gather_table_stats(null, 'first_k_rows_test',
method_opt=>'for all columns size 1 for columns filter_col size 254');
```

76

A Deeper Look - FIRST_ROWS mechanics

ALL_ROWS Baseline

```
select  /*+ all_rows */
        *
from
        first_k_rows_test a,
        first_k_rows_test b
where
        a.filter_col = 1
and     a.sort_order = b.sort_order
order by
        a.sort_order;
```

77

A Deeper Look - FIRST_ROWS mechanics

ALL_ROWS Baseline Plan

```
-----------------------------------------------------------------------------------
| Id  | Operation                     | Name                   | Rows  | Cost (%CPU)|
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                        |    10 |   855   (2)|
|   1 |  SORT ORDER BY                |                        |    10 |   855   (2)|
|   2 |   TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST      |     1 |     2   (0)|
|   3 |    NESTED LOOPS               |                        |    10 |   854   (2)|
|*  4 |     TABLE ACCESS FULL         | FIRST_K_ROWS_TEST      |    10 |   834   (2)|
|*  5 |     INDEX RANGE SCAN          | FIRST_K_ROWS_TEST_IDX2 |     1 |     1   (0)|
-----------------------------------------------------------------------------------

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

   4 - filter("A"."FILTER_COL"=1)
   5 - access("A"."SORT_ORDER"="B"."SORT_ORDER")
```

78

A Deeper Look - FIRST_ROWS mechanics

FIRST_ROWS query

```
select  /*+ first_rows */
        *
from
        first_k_rows_test a,
        first_k_rows_test b
where
        a.filter_col = 1
and     a.sort_order = b.sort_order
order by
        a.sort_order;
```

A Deeper Look - FIRST_ROWS mechanics

FIRST_ROWS Plan

```
--------------------------------------------------------------------------------
| Id  | Operation                      | Name                   | Rows  | Cost (%CPU)|
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                        |    10 |  100K   (1)|
|   1 |  TABLE ACCESS BY INDEX ROWID   | FIRST_K_ROWS_TEST      |     1 |     2   (0)|
|   2 |   NESTED LOOPS                 |                        |    10 |  100K   (1)|
|*  3 |    TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST      |    10 |  100K   (1)|
|   4 |     INDEX FULL SCAN            | FIRST_K_ROWS_TEST_IDX2 |  100K|   426   (2)|
|*  5 |    INDEX RANGE SCAN            | FIRST_K_ROWS_TEST_IDX2 |     1 |     1   (0)|
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("A"."FILTER_COL"=1)
   5 - access("A"."SORT_ORDER"="B"."SORT_ORDER")
```

So we are back at old behaviour with joins

_sort_elimination_cost_ratio = 0 means infinite

80

If joins are involved then I couldn't identify any similar "breakpoints" - it looks like the old behaviour of treating 0 as "infinite" is used in this scenario.

A Deeper Look - FIRST_ROWS_n mechanics

Overview of FIRST_ROWS_n mode

- Controlled via `_optimizer_fkr_index_cost_bias`
- Default value = 10
- Range of values: 1..1000
- ```
  index_cost <
  greatest(
  FTS_cost,
  reduced FTS_cost * _optimizer_fkr_index_cost_bias
  )
  ```
- But... ?

81

The FIRST_ROWS_n also has at least one internal parameter the affects the calculation. It seems however not to be limited to the "sort elimination" but obviously controls the "bias" towards index access.

In general above formula seems to apply, which means that an index will be used if its cost is less than the cost of the full table scan multiplied by the parameter.

Since the FIRST_ROWS_n mode "knows" the number of rows requested it can actually scale the full table scan cost accordingly - based on the assumption that you don't need to visit all rows from the table to obtain the first n rows.

Of course this assumption is only valid if a SORT is not involved - otherwise you need to obtain all rows identified to perform a sort operation on those rows, or you need to use an index to avoid the sort operation.

There seems to be a sanity check built into the calculation: If the "reduced" FTS cost multiplied by the parameter is less than the cost of the non-scaled FTS cost, then the non-scaled FTS cost is used as lower limit.
  ```

A Deeper Look - FIRST_ROWS_n mechanics

Test setup (1)

```
create table t2
pctfree 40
pctused 1
as
select
        mod(id, 10) + 1 as col1
      , id as col2
      , id as colx
      , rpad('x', 1000, 'x') as filler
from    (
        select
                level as id
        from
                dual
        connect by
                level <= 1000
);
```

82

Here are some examples that do not adhere to the formula provided - so there is obviously something influencing the calculation that I haven't identified yet

A Deeper Look - FIRST_ROWS_n mechanics

Test setup (1)

```
update t2 set col1 = 1 where col1 = 2 and rownum <= 1;

commit;

create index t2_idx_filter on t2 (col1);

create index t2_idx_sort on t2 (col1, col2);

exec dbms_stats.gather_table_stats(null, 't2',
   method_opt=>'for all columns size 1 for columns col1 size
   254')
```

83

The table has 1,000 rows and 100 rows from 1 to 10 in COL1. However I change the distribution slightly - value 1 will be there 101 times, whereas value 2 only 99 times.

A Deeper Look - FIRST_ROWS_n mechanics

ALL_ROWS result

```
select * from t2 where col1 = 1;

--------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)|
--------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   101 |    69   (0)|
|*  1 |  TABLE ACCESS FULL| T2   |   101 |    69   (0)|
--------------------------------------------------------

select * from t2 where col1 = 2;

--------------------------------------------------------
| Id  | Operation          | Name | Rows  | Cost (%CPU)|
--------------------------------------------------------
|   0 | SELECT STATEMENT   |      |    99 |    69   (0)|
|*  1 |  TABLE ACCESS FULL| T2   |    99 |    69   (0)|
--------------------------------------------------------
```

A Deeper Look - FIRST_ROWS_n mechanics

_optimizer_fkr_index_cost_bias = 1

```
select * from (
select * from t2 where col1 = 1
)
where rownum <= 67;
---------------------------------------------------------------------------
| Id  | Operation                     | Name            | Rows  | Cost (%CPU)|
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                 |    67 |    69   (0)|
|*  1 |  COUNT STOPKEY                |                 |       |            |
|   2 |   TABLE ACCESS BY INDEX ROWID| T2              |    67 |    69   (0)|
|*  3 |    INDEX RANGE SCAN           | T2_IDX_FILTER |       |     1   (0)|
---------------------------------------------------------------------------
select * from (
select * from t2 where col1 = 1
)
where rownum <= 68;
--------------------------------------------------------
| Id  | Operation           | Name | Rows  | Cost (%CPU)|
--------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    68 |    48   (0)|
|*  1 |  COUNT STOPKEY      |      |       |            |
|*  2 |   TABLE ACCESS FULL| T2   |    68 |    48   (0)|
--------------------------------------------------------
```

85

We can clearly see the breakpoint here when the cost exceeds the cost of the full table scan in ALL_ROWS mode

A Deeper Look - FIRST_ROWS_n mechanics

_optimizer_fkr_index_cost_bias = 1

```
select * from (
select * from t2 where col1 = 2
)
where rownum <= 98;
-------------------------------------------------------------------------
| Id  | Operation                     | Name           | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                |    98 |    100   (0)|
|*  1 |  COUNT STOPKEY                |                |       |            |
|   2 |   TABLE ACCESS BY INDEX ROWID| T2              |    98 |    100   (0)|
|*  3 |    INDEX RANGE SCAN           | T2_IDX_FILTER |       |      1   (0)|
-------------------------------------------------------------------------
```

86

But it doesn't apply to this case obviously...

A Deeper Look - FIRST_ROWS_n mechanics

_optimizer_fkr_index_cost_bias = 1

```
select * from (
select * from t2 where col1 = 2
order by col2
)
where rownum <= 67;

-------------------------------------------------------------------------
| Id  | Operation                      | Name         | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |              |    67 |    70   (0)|
|*  1 |  COUNT STOPKEY                 |              |       |            |
|   2 |   VIEW                         |              |    67 |    70   (0)|
|   3 |    TABLE ACCESS BY INDEX ROWID| T2           |    99 |    70   (0)|
|*  4 |     INDEX RANGE SCAN           | T2_IDX_SORT |    67 |     2   (0)|
-------------------------------------------------------------------------
```

87

Adding a sort operation gets us back to the expected behaviour

A Deeper Look - FIRST_ROWS_n mechanics

_optimizer_fkr_index_cost_bias = 1

```
select * from (
select * from t2 where col1 = 2
order by col2
)
where rownum <= 68;

-------------------------------------------------------------
| Id  | Operation                | Name | Rows   | Cost (%CPU)|
-------------------------------------------------------------
|   0 | SELECT STATEMENT         |      |    68 |     70   (2)|
|*  1 |  COUNT STOPKEY           |      |       |            |
|   2 |   VIEW                   |      |    99 |     70   (2)|
|*  3 |    SORT ORDER BY STOPKEY|      |    99 |     70   (2)|
|*  4 |     TABLE ACCESS FULL    | T2   |    99 |     69   (0)|
-------------------------------------------------------------
```

A Deeper Look - FIRST_ROWS_n mechanics

_optimizer_fkr_index_cost_bias = 1000 ????

```
select * from (
select * from t2 where col1 = 2
order by col2
)
where rownum <= 68;

-------------------------------------------------------------
| Id  | Operation              | Name | Rows  | Cost (%CPU)|
-------------------------------------------------------------
|   0 | SELECT STATEMENT       |      |    68 |    70   (2)|
|*  1 |  COUNT STOPKEY         |      |       |            |
|   2 |   VIEW                 |      |    99 |    70   (2)|
|*  3 |    SORT ORDER BY STOPKEY|     |    99 |    70   (2)|
|*  4 |     TABLE ACCESS FULL  | T2   |    99 |    69   (0)|
-------------------------------------------------------------
```

89

But the index usage can't be forced in this case via the parameter

A Deeper Look - FIRST_ROWS_n mechanics

Test setup (2)

```
create table first_k_rows_test
(
   sort_order   varchar2(10),
   filter_col   number,
   filler       char(200)
);

insert into first_k_rows_test
  select  dbms_random.string('x', 10),
          case
          when level <= 5000
          then 1
          else 0 end,
          'x' as filler
          from dual
          connect by level <= 100000;

commit;
```

90

Now the previous example suggests that introducing an ORDER BY made the formular work again - potentially due to the fact that there is no "reduced" FTS cost if a sort operation is involved - all rows of an unsorted source need to be processed in order to find the top N rows.

Here is however another example that uses a SORT ORDER BY and always uses the index, regardless of the parameter setting.

A Deeper Look - FIRST_ROWS_n mechanics

Test setup (2)

```
create index first_k_rows_test_idx on first_k_rows_test
   (filter_col, sort_order);

exec dbms_stats.gather_table_stats(null, 'first_k_rows_test',
   method_opt=>'for all columns size 1 for columns filter_col
   size 254', estimate_percent=>null);
```

91

A Deeper Look - FIRST_ROWS_n mechanics

ALL_ROWS result

```
select
        t.*
from
        first_k_rows_test t
where
        filter_col = 1
order by
        sort_order;

---------------------------------------------------------------------
| Id  | Operation          | Name               | Rows  | Cost (%CPU)|
---------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                    |  5000 |  1071   (2)|
|   1 |  SORT ORDER BY     |                    |  5000 |  1071   (2)|
|*  2 |   TABLE ACCESS FULL| FIRST_K_ROWS_TEST  |  5000 |   834   (2)|
---------------------------------------------------------------------
```

92

A Deeper Look - FIRST_ROWS_n mechanics

_optimizer_fkr_index_cost_bias = 1 ????

```
select
        *
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        order by
                sort_order
        )
where
        rownum <= 4999;
-----------------------------------------------------------------------------------
| Id  | Operation                     | Name                   | Rows  | Cost (%CPU)|
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                        |  4999 |  5024   (1)|
|*  1 |  COUNT STOPKEY                |                        |       |            |
|   2 |   VIEW                        |                        |  5000 |  5024   (1)|
|   3 |    TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST      |  5000 |  5024   (1)|
|*  4 |     INDEX RANGE SCAN          | FIRST_K_ROWS_TEST_IDX  |       |    18   (0)|
-----------------------------------------------------------------------------------
```

93

In this example however the index usage can not be prevented regardless of the setting of the parameter.

Further tests indicate that by reading the 10053 optimizer trace debug file it looks like that even with SORT ORDER BYs involved the calculation is based on the "reduced/scaled down" FTS cost in general, which seems to be odd - there is no "reduced" FTS cost if the source is unsorted.

A Deeper Look - FIRST_ROWS_n

Requesting more rows than estimated in ALL_ROWS mode

```
select
        t.*
from
        first_k_rows_test t
where
        filter_col = 1;
------------------------------------------------------------------
| Id  | Operation          | Name              | Rows  | Cost (%CPU)|
------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                   |  5003 |   834   (2)|
|*  1 |  TABLE ACCESS FULL| FIRST_K_ROWS_TEST |  5003 |   834   (2)|
------------------------------------------------------------------
```


Requesting more rows than estimated in ALL_ROWS mode effectively disables the FIRST_ROWS(n) mode - the statement is optimized using ALL_ROWS mode

A Deeper Look - FIRST_ROWS_n

```
select
        *
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        order by
                sort_order
        )
where
        rownum <= 4999;
```

```
-------------------------------------------------------------------------------------
| Id  | Operation                      | Name                   | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                        |  4999 |  5029   (1)|
|*  1 |  COUNT STOPKEY                 |                        |       |            |
|   2 |   VIEW                         |                        |  5003 |  5029   (1)|
|   3 |    TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST      |  5003 |  5029   (1)|
|*  4 |     INDEX RANGE SCAN           | FIRST_K_ROWS_TEST_IDX  |       |    21   (0)|
-------------------------------------------------------------------------------------
```

95

Requesting less than 5003 rows will activate FIRST_ROWS(n) optimization

A Deeper Look - FIRST_ROWS_n

```
select
        *
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        order by
                sort_order
        )
where
        rownum <= 5010;
---------------------------------------------------------------------------
| Id  | Operation               | Name              | Rows  | Cost (%CPU)|
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |                   |  5003 |  1071   (2)|
|*  1 |  COUNT STOPKEY          |                   |       |            |
|   2 |   VIEW                  |                   |  5003 |  1071   (2)|
|*  3 |    SORT ORDER BY STOPKEY|                   |  5003 |  1071   (2)|
|*  4 |     TABLE ACCESS FULL   | FIRST_K_ROWS_TEST |  5003 |   834   (2)|
---------------------------------------------------------------------------
```

96

Requesting more rows than estimated in ALL_ROWS mode effectively disables the FIRST_ROWS(n) mode - the statement is optimized using ALL_ROWS mode

A Deeper Look - FIRST_ROWS_n

```
alter session set tracefile_identifier = 'first_rows_n_greater_than_all_rows';

alter session set events '10053 trace name context forever, level 1';

explain plan for
select
        *
from    (
        select
                t.*
        from
                first_k_rows_test t
        where
                filter_col = 1
        order by
                sort_order
        )
where
        rownum <= 5010;

alter session set events '10053 trace name context off';
```

97

This can also be seen from the 10053 optimizer trace debug file where no FIRST_ROWS(n) specific output can be found after the initial evaluation of the estimated number of rows in ALL_ROWS mode.

A Deeper Look - FIRST_ROWS_n And DML

DML and FIRST_ROWS_n don't mix

```
update
          first_k_rows_test t
set
          filter_col = 1
where
          filter_col = 0
and       rownum <= 1
returning
          sort_order into :retval;
```

Special case: Get just the first row(s) for update

98

Oracle optimizes DML always using the ALL_ROWS mode regardless of any hints / ROWNUM predicates.

In general this is a good idea since DML must process all rows anyway - however there are special cases where you really would like to get just a single row - an example scenario is getting just one code from a pool codes.

A Deeper Look - FIRST_ROWS_n And DML

Execution plan

```
-----------------------------------------------------------------------------------
| Id  | Operation            | Name              | Starts | E-Rows | A-Rows | Buffers |
-----------------------------------------------------------------------------------
|   1 |  UPDATE              | FIRST_K_ROWS_TEST |      1 |        |      1 |     312 |
|*  2 |   COUNT STOPKEY      |                   |      1 |        |      1 |     307 |
|*  3 |    TABLE ACCESS FULL| FIRST_K_ROWS_TEST |      1 |  90415 |      1 |     307 |
-----------------------------------------------------------------------------------

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

   2 - filter(ROWNUM<=1)
   3 - filter("FILTER_COL"=0)
```

99

The FULL TABLE SCAN here is definitely not the most efficient access to update just a single row. A range scan on the existing index on FILTER_COL would be a much better idea.

A Deeper Look - FIRST_ROWS_n And DML

Hints required to get it right

```
update     /*+ index_rs(t) */
           first_k_rows_test t
set
           filter_col = 1
where
           filter_col = 0
and        rownum <= 1
returning
           sort_order into :retval;
```

100

Oracle optimizes DML always using the ALL_ROWS mode regardless of any hints / ROWNUM predicates.

In general this is a good idea since DML must process all rows anyway - however there are special cases where you really would like to get just a single row - an example scenario is getting just one code from a pool codes.

A Deeper Look - FIRST_ROWS_n And DML

Execution plan

```
-------------------------------------------------------------------------------------
| Id  | Operation           | Name                  | Starts | E-Rows | A-Rows | Buffers |
-------------------------------------------------------------------------------------
|   1 |  UPDATE             | FIRST_K_ROWS_TEST     |      1 |        |      1 |       7 |
|*  2 |   COUNT STOPKEY     |                       |      1 |        |      1 |       2 |
|*  3 |    INDEX RANGE SCAN| FIRST_K_ROWS_TEST_IDX |      1 |  90415 |      1 |       2 |
-------------------------------------------------------------------------------------

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

   2 - filter(ROWNUM<=1)
   3 - filter("FILTER_COL"=0)
```

101

Notice the much less buffer gets

A Deeper Look - Bitmap Indexes

- FIRST_ROWS(n) doesn't consider Bitmap Indexes for avoiding ORDER BYs
- FIRST_ROWS does

A Deeper Look - Bitmap Indexes

Modified test setup

```
drop index first_k_rows_test_idx2;

create bitmap index first_k_rows_test_bitmap on
first_k_rows_test (sort_order);

alter table first_k_rows_test
modify sort_order null;
```

103

The test setup gets just slightly modified - the conventional index is replaced with a bitmap index - and additionally since bitmap indexes also index NULL expressions the column is modified to be nullable.

A Deeper Look - Bitmap Indexes

FIRST_ROWS query

```
select  /*+ first_rows */
        *
from
        first_k_rows_test t
order by
        sort_order;

-------------------------------------------------------------------------------------
| Id  | Operation                      | Name                        | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                             |   100K|  5271   (1)|
|   1 |  TABLE ACCESS BY INDEX ROWID   | FIRST_K_ROWS_TEST           |   100K|  5271   (1)|
|   2 |   BITMAP CONVERSION TO ROWIDS  |                             |       |            |
|   3 |    BITMAP INDEX FULL SCAN      | FIRST_K_ROWS_TEST_BITMAP    |       |            |
-------------------------------------------------------------------------------------
```

104

A Deeper Look - Bitmap Indexes

FIRST_ROWS(n) query

```
select * from (
select  *
from
        first_k_rows_test t
order by
        sort_order
)
where rownum <= 10;

--------------------------------------------------------------------------
| Id  | Operation              | Name              | Rows  | Cost (%CPU)|
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                   |    10 |  5516   (1)|
|*  1 |  COUNT STOPKEY         |                   |       |            |
|   2 |   VIEW                 |                   |  100K|  5516   (1)|
|*  3 |    SORT ORDER BY STOPKEY|                  |  100K|  5516   (1)|
|   4 |     TABLE ACCESS FULL  | FIRST_K_ROWS_TEST |  100K|   833   (2)|
--------------------------------------------------------------------------
```

105

A Deeper Look - Bitmap Indexes

FIRST_ROWS(n) query - hinted

```
select * from (
select  /*+ index(t, first_k_rows_test_bitmap) */
        *
from
        first_k_rows_test t
order by
        sort_order
)
where rownum <= 10;
```

```
--------------------------------------------------------------------------------
| Id  | Operation                     | Name                     | Rows  | Cost (%CPU)|
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                          |    10 |     7   (0)|
|*  1 |  COUNT STOPKEY                |                          |       |            |
|   2 |   VIEW                        |                          |    10 |     7   (0)|
|   3 |    TABLE ACCESS BY INDEX ROWID | FIRST_K_ROWS_TEST        |    10 |     7   (0)|
|   4 |     BITMAP CONVERSION TO ROWIDS|                          |       |            |
|   5 |      BITMAP INDEX FULL SCAN   | FIRST_K_ROWS_TEST_BITMAP |       |            |
--------------------------------------------------------------------------------
```

106

A Deeper Look - Bitmap Indexes

FIRST_ROWS(n) query - hinted

Runtime Statistics

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                     | Starts | E-Rows | A-Rows | Buffers |
-----------------------------------------------------------------------------------------------------------
|*  1 |  COUNT STOPKEY                |                          |      1 |        |     10 |      12 |
|   2 |   VIEW                        |                          |      1 |     10 |     10 |      12 |
|   3 |    TABLE ACCESS BY INDEX ROWID | FIRST_K_ROWS_TEST        |      1 |     10 |     10 |      12 |
|   4 |     BITMAP CONVERSION TO ROWIDS|                          |      1 |        |     10 |       2 |
|   5 |      BITMAP INDEX FULL SCAN    | FIRST_K_ROWS_TEST_BITMAP |      1 |        |     10 |       2 |
-----------------------------------------------------------------------------------------------------------
```

107

A Deeper Look - Bitmap Indexes

FIRST_ROWS(n) query - filter added

```
select * from (
select
        *
from
        first_k_rows_test t
where
        sort_order >= '0'
order by
        sort_order
)
where rownum <= 10;
----------------------------------------------------------------------------------------
| Id  | Operation                     | Name                     | Rows   | Cost (%CPU)|
----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                          |    10  |     7   (0)|
|*  1 |  COUNT STOPKEY                |                          |        |            |
|   2 |   VIEW                        |                          |    10  |     7   (0)|
|   3 |    TABLE ACCESS BY INDEX ROWID | FIRST_K_ROWS_TEST        |    10  |     7   (0)|
|   4 |     BITMAP CONVERSION TO ROWIDS|                          |        |            |
|*  5 |      BITMAP INDEX RANGE SCAN   | FIRST_K_ROWS_TEST_BITMAP |        |            |
----------------------------------------------------------------------------------------
```

A Deeper Look - Potential parsing overhead

- In more recent releases (in particular in 11g and later) there is a measurable parsing overhead when using FIRST_ROWS(n) mode
- The following block simply hard parses a query 1,000 times

A Deeper Look - Potential parsing overhead

```
declare
  c integer;
begin
  c := dbms_sql.open_cursor;
  for i in 1..1000 loop
    dbms_sql.parse(c, '
select   /* PARSE_' || to_char(i, 'TM') || ' */
         *
from
         first_k_rows_test a
where a.filter_col > (
select
         avg(b.filter_col)
from
         first_k_rows_test b
where
         b.sort_order = a.sort_order
)
    ', dbms_sql.NATIVE);
  end loop;
  dbms_sql.close_cursor(c);
end;
/
```


It is a rather simply query with a single subquery

A Deeper Look - Potential parsing overhead

- In ALL_ROWS mode this takes on my test system approx. **5.5** seconds (11.1.0.7)
- Let's run the same in FIRST_ROWS(1) mode

111

A Deeper Look - Potential parsing overhead

```
declare
  c integer;
begin
  c := dbms_sql.open_cursor;
  for i in 1..1000 loop
    dbms_sql.parse(c, '
select    /*+ FIRST_ROWS(1) */ /* PARSE_' || to_char(i, 'TM') || ' */
          *
from
          first_k_rows_test a
where a.filter_col > (
select
          avg(b.filter_col)
from
          first_k_rows_test b
where
          b.sort_order = a.sort_order
)
    ', dbms_sql.NATIVE);
  end loop;
  dbms_sql.close_cursor(c);
end;
/
```

112

A Deeper Look - Potential parsing overhead

- In FIRST_ROWS(1) mode this takes on my test system approx. **10** seconds (11.1.0.7)
- With well behaving applications not relevant, but there is some overhead that might lead to problems with applications that do a lot of hard parsing

A Deeper Look - Things Can Go Wrong (1)

Test setup - function-based index

```
create table test_sort(name,object_name)
as
select 'select',object_name from
dba_objects;

create index i_test_sort_1 on
test_sort(substr(name,0,3),object_name);

exec dbms_stats.gather_table_stats(null,'TEST_SORT');
```

114

Oracle 10.2.0.4 has problems when using function-based indexes and first_rows(n) mode. This might be bug 3940803 which was filed for 9.2.0.5, but there are many bugs in this context, so it could also be a regression of some of these bugs.

A Deeper Look - Things Can Go Wrong (1)

Simple Top-N query using expression

```
select
        *
from    (
        select
                *
        from
                test_sort
        where
                substr(name,0,3) = 'sel'
        order by
                object_name
        )
where
        rownum<=10;
```

115

A Deeper Look - Things Can Go Wrong (1)

Execution plan 10.2.0.4

```
-----------------------------------------------------------------
| Id  | Operation              | Name      | Rows  | Cost (%CPU)|
-----------------------------------------------------------------
|   0 | SELECT STATEMENT       |           |    10 |   529   (4)|
|*  1 |  COUNT STOPKEY         |           |       |            |
|   2 |   VIEW                 |           | 50744 |   529   (4)|
|*  3 |    SORT ORDER BY STOPKEY|          | 50744 |   529   (4)|
|*  4 |     TABLE ACCESS FULL  | TEST_SORT | 50744 |    75   (6)|
-----------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(ROWNUM<=10)
   3 - filter(ROWNUM<=10)
   4 - filter(SUBSTR("NAME",0,3)='sel')
```

116

The function-based index doesn't get used in 10.2.0.4

A Deeper Look - Things Can Go Wrong (1)

Execution plan 11.1.0.7

```
---------------------------------------------------------------------------
| Id  | Operation                      | Name            | Rows  | Cost (%CPU)|
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                 |    10 |      8   (0)|
|*  1 |  COUNT STOPKEY                 |                 |       |             |
|   2 |   VIEW                         |                 |    10 |      8   (0)|
|   3 |    TABLE ACCESS BY INDEX ROWID| TEST_SORT       |    10 |      8   (0)|
|*  4 |     INDEX RANGE SCAN           | I_TEST_SORT_1 |       |      3   (0)|
---------------------------------------------------------------------------

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

   1 - filter(ROWNUM<=10)
   4 - access(SUBSTR("NAME",0,3)='sel')
```

117

It gets used in 11.1.0.7 however

A Deeper Look - Things Can Go Wrong (1)

Workaround in 10.2.0.4

```
select
        *
from    (
        select
                *
        from
                test_sort
        where
                substr(name,0,3) = 'sel'
        order by
                substr(name,0,3), object_name
        )
where
        rownum<=10;
```

118

Adding the expression to the ORDER BY enables the index usage in 10.2.0.4

A Deeper Look - Things Can Go Wrong (1)

Workaround execution plan 10.2.0.4

```
--------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Cost (%CPU)|
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |    10 |     8   (0)|
|*  1 |  COUNT STOPKEY               |               |       |            |
|   2 |   VIEW                       |               |    10 |     8   (0)|
|   3 |    TABLE ACCESS BY INDEX ROWID| TEST_SORT    | 50744 |     8   (0)|
|*  4 |     INDEX RANGE SCAN         | I_TEST_SORT_1 |    10 |     3   (0)|
--------------------------------------------------------------------------

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

   1 - filter(ROWNUM<11)
   4 - access(SUBSTR("NAME",0,3)='sel')
```

119

A Deeper Look - Things Can Go Wrong (2)

Test setup - Join with group by and filter

```
create table first_k_rows_test1
as
select
        id
      , mod(id, 100)   as id_join
      , mod(id, 100)   as filter_col
      , rpad('x', 100) as filler
from
        (
        select
                level as id
        from
                dual
        connect by
                level <= 100000
        );
```

http://forums.oracle.com/forums/thread.jspa?threadID=934895

120

This example for an issue with GROUP BY is based on a test case posted on the OTN forums, however the original setup was much more complicated and therefore I've tried to simplify as much as possible.

We have a simple table with 100,000 rows

A Deeper Look - Things Can Go Wrong (2)

Test setup - Join with group by and filter

```
create table first_k_rows_test2
as
select
        id_join
      , upper(dbms_random.string('a',15)) as group_col
from
        (
        select  distinct
                id_join
        from
                first_k_rows_test1
        );

create index first_k_rows_test2_idx on first_k_rows_test2 (group_col);

exec dbms_stats.gather_table_stats(null, 'first_k_rows_test1')

exec dbms_stats.gather_table_stats(null, 'first_k_rows_test2')
```

121

The second table is only a lookup table - the crucial point here is that there is an index on the grouping column.

A Deeper Look - Things Can Go Wrong (2)

The query

```
select
        group_col
from
        (
        select
                t2.group_col
              , count(*) as cnt
        from
                first_k_rows_test1 t1
              , first_k_rows_test2 t2
        where
                t1.id_join = t2.id_join
        and     t2.group_col != 'BLA'
        and     t1.filter_col between 1 and 2
        group by
                t2.group_col
        order by
                cnt desc
        )
where
        rownum <= 1;
```

122

Having the index and a filter predicate on the grouping column encourages the optimizer to evaluate an execution plan that avoids the grouping operation.

A Deeper Look - Things Can Go Wrong (2)

Execution plan 10.2.0.4 + 11.1.0.7

```
-------------------------------------------------------------------------------------------
| Id  | Operation                      | Name                     | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT               |                          |     1 |    35   (3)|
|*  1 |  COUNT STOPKEY                 |                          |       |            |
|   2 |   VIEW                         |                          |     2 |    35   (3)|
|*  3 |    SORT ORDER BY STOPKEY       |                          |     2 |    35   (3)|
|   4 |     SORT GROUP BY NOSORT       |                          |     2 |    35   (3)|
|   5 |      NESTED LOOPS              |                          |     2 |    35   (3)|
|   6 |       TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST2     |    99 |     2   (0)|
|*  7 |        INDEX FULL SCAN         | FIRST_K_ROWS_TEST2_IDX |     1 |     1   (0)|
|*  8 |       TABLE ACCESS FULL        | FIRST_K_ROWS_TEST1     |     2 |    33   (4)|
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(ROWNUM<=1)
   3 - filter(ROWNUM<=1)
   7 - filter("T2"."GROUP_COL"<>'BLA')
   8 - filter("T1"."FILTER_COL"<=2 AND "T1"."FILTER_COL">=1 AND
     "T1"."ID_JOIN"="T2"."ID_JOIN")
```

123

The problem seems to be that the ROWNUM predicate is pushed far down and therefore the plan is based on the assumption that only a single row needs to read from the small table - however the nested loop will have to run through all iterations to perform the group by operation.

A Deeper Look - Things Can Go Wrong (2)

Runtime statistics

```
-----------------------------------------------------------------------------------------------------------
| Id  | Operation                        | Name                    | Starts | E-Rows | A-Rows | Buffers |
-----------------------------------------------------------------------------------------------------------
|*  1 |  COUNT STOPKEY                   |                         |      1 |        |      1 |    164K|
|   2 |   VIEW                           |                         |      1 |      2 |      1 |    164K|
|*  3 |    SORT ORDER BY STOPKEY         |                         |      1 |      2 |      1 |    164K|
|   4 |     SORT GROUP BY NOSORT         |                         |      1 |      2 |      2 |    164K|
|   5 |      NESTED LOOPS                |                         |      1 |      2 |   2000 |    164K|
|   6 |       TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST2      |      1 |     99 |    100 |      2 |
|*  7 |        INDEX FULL SCAN           | FIRST_K_ROWS_TEST2_IDX  |      1 |      1 |    100 |      1 |
|*  8 |       TABLE ACCESS FULL          | FIRST_K_ROWS_TEST1      |    100 |      2 |   2000 |    164K|
-----------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(ROWNUM<=1)
   3 - filter(ROWNUM<=1)
   7 - filter("T2"."GROUP_COL"<>'BLA')
   8 - filter(("T1"."FILTER_COL"<=2 AND "T1"."FILTER_COL">=1 AND
    "T1"."ID_JOIN"="T2"."ID_JOIN"))
```

124

It is obviously not a good idea to run a full table scan 100 times in a nested loop

A Deeper Look - Things Can Go Wrong (2)

Potential workarounds:

- Use ALL_ROWS hint
- Use pre-10.2.0.3 optimizer features
- Use "_first_k_rows_dynamic_proration" parameter

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

- Obviously functionality changes from release to release, since 10.2.0.3 enabled by default (true)

125

The issue is fixed in 11.2.0.1. Switching to pre-10.2.0.3 optimizer settings simply sets the "_first_k_rows_dynamic_proration" parameter to false.

The "_first_k_rows_dynamic_proration" seems to enable more sophisticated code paths regarding the FIRST_ROWS(n) mode - as always new features not always result in better execution plans.

A Deeper Look - Things Can Go Wrong (2)

Workaround execution plan

```
-------------------------------------------------------------------------------------------
| Id  | Operation               | Name                | Starts | E-Rows | A-Rows | Buffers |
-------------------------------------------------------------------------------------------
|*  1 |  COUNT STOPKEY          |                     |      1 |        |      1 |    1644 |
|   2 |   VIEW                  |                     |      1 |     99 |      1 |    1644 |
|*  3 |    SORT ORDER BY STOPKEY|                     |      1 |     99 |      1 |    1644 |
|   4 |     HASH GROUP BY       |                     |      1 |     99 |      2 |    1644 |
|*  5 |      HASH JOIN          |                     |      1 |   2931 |   2000 |    1644 |
|*  6 |       TABLE ACCESS FULL | FIRST_K_ROWS_TEST2  |      1 |     99 |    100 |       3 |
|*  7 |       TABLE ACCESS FULL | FIRST_K_ROWS_TEST1  |      1 |   2990 |   2000 |    1641 |
-------------------------------------------------------------------------------------------

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

   1 - filter(ROWNUM<=1)
   3 - filter(ROWNUM<=1)
   5 - access("T1"."ID_JOIN"="T2"."ID_JOIN")
   6 - filter("T2"."GROUP_COL"<>'BLA')
   7 - filter(("T1"."FILTER_COL"<=2 AND "T1"."FILTER_COL">=1))
```


Notice the much less buffer gets

A Deeper Look - Things Can Go Wrong (2)

- But using the "_first_k_rows_dynamic_proration" not always helps. Slightly changing the query - for example removing the filter predicate - will result in the bad execution plan, even with the parameter set to "false"
- Let's see how the plan looks like when forcing the nested loop operation and the parameter set to false

127

A Deeper Look - Things Can Go Wrong (2)

Execution plan with workaround

```
---------------------------------------------------------------------------
| Id  | Operation                 | Name               | Rows  | Cost (%CPU)|
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |                    |     1 |   458   (3)|
|*  1 |  COUNT STOPKEY            |                    |       |            |
|   2 |   VIEW                    |                    |    99 |   458   (3)|
|*  3 |    SORT ORDER BY STOPKEY|                    |    99 |   458   (3)|
|   4 |     HASH GROUP BY         |                    |    99 |   458   (3)|
|*  5 |      HASH JOIN            |                    |  2931 |   456   (2)|
|*  6 |       TABLE ACCESS FULL | FIRST_K_ROWS_TEST2 |    99 |     2   (0)|
|*  7 |       TABLE ACCESS FULL | FIRST_K_ROWS_TEST1 |  2990 |   454   (2)|
---------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(ROWNUM<=1)
   3 - filter(ROWNUM<=1)
   5 - access("T1"."ID_JOIN"="T2"."ID_JOIN")
   6 - filter("T2"."GROUP_COL"<>'BLA')
   7 - filter("T1"."FILTER_COL"<=2 AND "T1"."FILTER_COL">=1)
```

128

A Deeper Look - Things Can Go Wrong (2)

Execution plan with forced NL

```
-------------------------------------------------------------------------------------
| Id  | Operation                         | Name                     | Rows  | Cost (%CPU)|
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                  |                          |     1 |   463   (3)|
|*  1 |  COUNT STOPKEY                    |                          |       |            |
|   2 |   VIEW                            |                          |    23 |   463   (3)|
|*  3 |    SORT ORDER BY STOPKEY          |                          |    23 |   463   (3)|
|   4 |     SORT GROUP BY NOSORT          |                          |    23 |   463   (3)|
|   5 |      NESTED LOOPS                 |                          |    23 |   463   (3)|
|   6 |       TABLE ACCESS BY INDEX ROWID| FIRST_K_ROWS_TEST2       |    99 |     2   (0)|
|*  7 |        INDEX FULL SCAN            | FIRST_K_ROWS_TEST2_IDX   |     2 |     1   (0)|
|*  8 |       TABLE ACCESS FULL           | FIRST_K_ROWS_TEST1       |    15 |   231   (3)|
-------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(ROWNUM<=1)
   3 - filter(ROWNUM<=1)
   7 - filter("T2"."GROUP_COL"<>'BLA')
   8 - filter("T1"."FILTER_COL"<=2 AND "T1"."FILTER_COL">=1 AND
    "T1"."ID_JOIN"="T2"."ID_JOIN")
```


So the plan just got discarded because it is slightly more expensive - a small change to the query will result in that execution plan potentially becoming cheaper.

Everything You Wanted To Know About FIRST_ROWS_n

Summary (1)

- Important to understand when to use FIRST_ROWS(_n) optimization
- FIRST_ROWS is deprecated since 9i and might produce suboptimal execution plans, in particular when joins are involved

Everything You Wanted To Know About FIRST_ROWS_n

Summary (2)

- Advanced techniques for Pagination queries when frequently accessing later pages
- If there are issues with executions plans, one can try to set the "_first_k_rows_dynamic_proration" parameter to false in 10.2.0.4 and later

Everything You Wanted To Know About FIRST_ROWS_n

Questions

&

Answers

132

Everything You Wanted To Know About FIRST_ROWS_n

Thank you!