

Q328 Stats with Intelligence

Arup Nanda
Starwood Hotels

Effect of Stats on Two Columns

- Optimizer Statistics on tables and indexes are vital for the optimizer to compute **optimal** execution plans
- If there are stats on two different columns used in the query, how does the optimizer decide?
- It takes the selectivity of each column, and multiplies that to get the selectivity for the query.

Example

- Two columns
 - Month of Birth: selectivity = $1/12$
 - Zodiac Sign: selectivity = $1/12$
- What will be the selectivity of a query
 - Where zodiac sign = 'Pisces'
 - And month of birth = 'January'
- Problem:
 - According to the optimizer it will be $1/12 \times 1/12 = 1/144$
 - In reality, it will be 0, size the combination is not possible
- What will be the selectivity of a query
 - Where zodiac sign = 'Capricorn'
 - And month of birth = 'January'

Multi-column Intelligence

- If the Optimizer knew about these combinations, it would have been able to choose the proper path
- How would you let the optimizer learn about these?
- In Oracle 10g, we saw a good approach – SQL Profiles
 - which allowed data to be considered for execution plans
 - but was not a complete approach
 - it still lacked a dynamism – applicability in all circumstances
- In 11g, there is an ability to provide this information to the optimizer
 - **Multi-column stats**

An Example

- Table BOOKINGS
- Index on (HOTEL_ID, RATE_CODE)
- What will be plan for the following?

```
select min(book_txn)
from bookings
where hotel_id = 10
and rate_code = 23
```

HOTEL_ID	RATE_CODE	COUNT (1)
10	11	444578
10	12	50308
20	22	100635
20	23	404479

The Plan

Here is the plan

expl1.sql

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	769 (3)	00:00:10
1	SORT AGGREGATE		1	10		
* 2	TABLE ACCESS FULL	BOOKINGS	199K	1951K	769 (3)	00:00:10

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT

2 - filter("RATE_CODE"=23 AND "HOTEL_ID"=10))

- It didn't choose index scan
- The estimated number of rows are 199K, or about 20%; so full table scan was favored over index scan

Solution

- Create Extended Stats in the related columns –
HOTEL_ID and RATE_CODE
var ret varchar2(2000)
begin
 :ret := dbms_stats.create_extended_stats(
 'ARUP', 'BOOKINGS', '(HOTEL_ID, RATE_CODE)'
);
end;
/
print ret
- The variable “ret” shows the name of the extended statistics xstats.sql

Then Collect Stats Normally

```
begin
  dbms_stats.gather_table_stats (
    ownname      => 'ARUP',
    tabname       => 'BOOKINGS',
    estimate_percent=> 100,
    method_opt    => 'FOR ALL COLUMNS SIZE SKEWONLY',
    cascade       => true
  );
end;
/
```

stats.sql

The Plan Now

- After extended stats, the plan looks like this:

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1	10	325 (1)	00:00:04
1	SORT AGGREGATE		1	10		
2	TABLE ACCESS BY INDEX ROWID	BOOKINGS	23997	234K	325 (1)	00:00:04
* 3	INDEX RANGE SCAN	IN_BOOKINGS_01	23997		59 (0)	00:00:01

- Note:
 - No of Rows is now more accurate
 - As a result, the index scan was chosen

expl1.sql

Extended Stats

- Extended stats store the correlation of data among the columns
 - The correlation helps optimizer decide on an execution path that takes into account the data
 - Execution plans are more accurate
- Under the covers,
 - extended stats create an invisible virtual column
 - Stats on the columns collects stats on this virtual column as well

10053 Trace Shows the Virtual Column

```
Single Table Cardinality Estimation for
BOOKINGS[BOOKINGS]
  Column (#2):
    NewDensity:0.247422, OldDensity:0.000000
  BktCnt:1000000, PopBktCnt:1000000, PopValCnt:2, NDV:2
  Column (#3):
    NewDensity:0.025295, OldDensity:0.000000
  BktCnt:1000000, PopBktCnt:1000000, PopValCnt:4, NDV:4
  Column (#5):
    NewDensity:0.025295, OldDensity:0.000000
  BktCnt:1000000, PopBktCnt:1000000, PopValCnt:4, NDV:4
  ColGroup (#1, VC) SYS_STU4JHE7J4YQ3ZLDXSW5L108KX
    Col#: 2 3    CorStrength: 2.00
  ColGroup Usage:: PredCnt: 2  Matches Full:    Using
density: 0.025295 of col #5 as selectivity of unpopular
value pred
```

Extended Stats

- This hidden virtual column shows up in column statistics

```
select column_name, DENSITY, num_distinct
from user_tab_col_statistics
where table_name = 'BOOKINGS'
```

COLUMN_NAME	DENSITY	NUM_DISTINCT
BOOKING_ID	.000001	1000000
HOTEL_ID	.0000005	2
RATE_CODE	.0000005	4
BOOK_TXN	.002047465	2200
SYS_STU4JHE7J4YQ3ZLDXSW5L108KX	.0000005	4

Checking for Extended Stats

- To check the presence of extended stats, check the view `dba_stat_extensions`.

```
select extension_name, extension
from dba_stat_extensions
where table_name='BOOKINGS';
```

Output:

EXTENSION_NAME	EXTENSION
SYS_STU4JHE7J4YQ3ZLDXSW5L108KX	("HOTEL_ID", "RATE_CODE")

check.sql

Deleting Extended Stats

- If you want, you can drop the extended stats, you can use the `dbms_stats` package, specifically the procedure `drop_extended_stats`

```
begin
  dbms_stats.drop_extended_stats (
    ownname => 'ARUP',
    tabname => 'BOOKINGS',
    extension => ' ("HOTEL_ID", "RATE_CODE") '
  );
end;
```

drop.sql

Another way

- You can collect the extended stats using the normal `dbms_stats` as well:

```
begin
  dbms_stats.gather_table_stats (
    ownname => 'ARUP',
    tabname => 'BOOKINGS',
    estimate_percent => 100,
    method_opt =>
      'FOR ALL COLUMNS SIZE SKEWONLY FOR COLUMNS
      (HOTEL_ID,RATE_CODE)',
    cascade      => true
  );
end;
/
```

startx.sql

The Case on Case Sensitivity

- A table of CUSTOMERS with 1 million rows
- LAST_NAME field has the values
 - McDonald – 20%
 - MCDONALD – 10%
 - McDONALD – 10%
 - mcdonald – 10%
- They make up 50% of the rows, with the variation of the same name.
- When you issue a query like this:

```
select * from customers where upper(last_name) =
'MCDONALD'
```


Normal Plan

- The plan looks like this:

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		10000	498K	2140 (2)	00:00:26
* 1	TABLE ACCESS FULL	CUSTOMERS	10000	498K	2140 (2)	00:00:26

No of rows
wrongly
estimated

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT

1 - filter(UPPER("LAST_NAME")='MCDONALD')

expl2.sql

Extended Stats

- You collect the stats for the UPPER() function

begin

```
dbms_stats.gather_table_stats (  
    ownname      => 'ARUP',  
    tabname       => 'CUSTOMERS',  
    method_opt    => 'for all columns size  
skewonly for columns (upper(last_name))'  
);  
end;
```

statsx_cust.sql

With Extended Stats

- The plan is now:

No of rows
correctly
estimated

Id	Operation	Name	Rows	Bytes	Cost	(%CPU)	Time
0	SELECT STATEMENT		500K	33M	2140	(2)	00:00:26
* 1	TABLE ACCESS FULL	CUSTOMERS	500K	33M	2140	(2)	00:00:26

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT

1 - filter("CUSTOMERS"."SYS_STUJ6BPFDTTE396EPTURAB2DBI5"='MCDONALD')

Extended stats
name come here

expl2.sql

Alternatives

- Remember, the extended stats create a virtual column – hidden from you
- You can have the same functionality as extended stats by defining virtual columns
- Advantage
 - You can have a column name of your choice
 - You can index it, if needed
 - You can partition it
 - You can create Foreign Key constraints on it

Restrictions

- Has to be 11.0 or higher
- Not for SYS owned tables
- Not on IOT, clustered tables, GTT or external tables
- Can't be on a virtual column
- An Expression
 - can't contain a subquery
 - must have ≥ 1 columns
- A Column Groups
 - should no of columns ≤ 32 and ≥ 2
 - can't contain expressions
 - can't have the same column repeated

Summary

- Normally the optimizer does not know the correlation between the columns
 - e.g. no one born in January can have a sun sign of Pisces
 - Therefore, perform an index scan if that combination is passed as predicate
- Extended statistics enable the optimizer to know that relationship
- More information to the optimizer
- ... results in better plans

Thank You!

Please fill out Evaluation Forms

Q328

Stats with Intelligence