When Object Statistics aren't Enough

Jonathan Lewis

jonathanlewis.wordpress.com

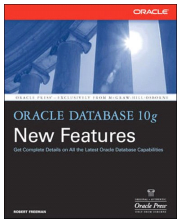
www.jlcomp.demon.co.uk

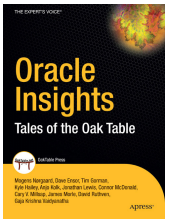
My History

Independent Consultant

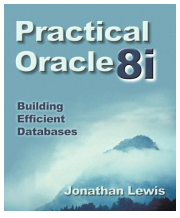
36+ years in IT
31+ using Oracle (5.1a on MSDOS 3.3)

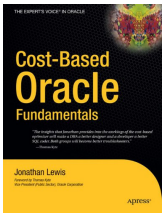
Strategy, Design, Review,
Briefings, Educational,
Trouble-shooting

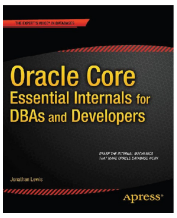




Founder Member of Oak Table Network
Best Presentation HROUG 2016
UKOUG Lifetime Award (IPA) 2013
ODTUG 2012 Best Presenter (d/b)
UKOUG Inspiring Presenter 2012
UKOUG Inspiring Presenter 2011
Select Editor's choice 2007
Oracle author of the year 2006
Oracle *ACE Director*
O1 visa for USA







- Problems using statistics
- Problems gathering statistics
- Faking statistics

Problems ***using*** Statistics

Out of Range (1a)

```
create table t1
as
select
        rownum                          id,
        mod(rownum-1,1000)              n1,
        lpad(rownum,10,'0')             v1,
        lpad('x',100,'x')               padding
from
        {large_row_source}
where
        rownum <= 1e5
;
```

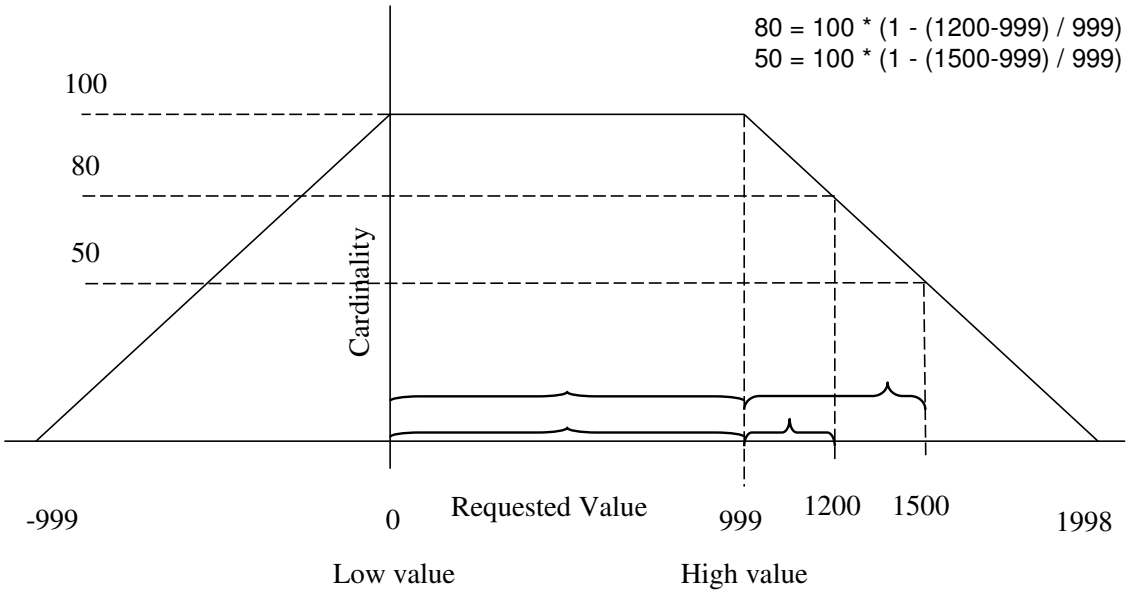

Column ***n1*** holds 1,000 distinct values ranging from 0 to 999 with 100 rows per value. We're going to look at equality predicates that go out of range. *"column = {constant}"*.

Out of Range (1b)

```
select  count(*) from    t1 where   n1 = 499;  -- in range

select  count(*) from    t1 where   n1 = 1200; -- out of range

select  count(*) from    t1 where   n1 = 1500; -- way out

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     4 |   228   (4)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |     4 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |   100 |   400 |   228   (4)| 00:00:01 |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL| T1   |    80 |   320 |   228   (4)| 00:00:01 |
-------------------------------------------------------------------------------
|*  2 |   TABLE ACCESS FULL| T1   |    50 |   200 |   228   (4)| 00:00:01 |
-------------------------------------------------------------------------------
```


Within range there are ***100*** rows per distinct value of ***n1***, and the estimate is correct. As you go outside the range the optimizer reduces the estimate using ***linear decay***.

Out of Range (1c)



Out of Range (2a)

```
create table t1 as
select
        rownum                                  id,
        sysdate +
                trunc((rownum - 1)/3) / 86400 -
                trunc((1e5    - 1)/3) / 86400
        as                                      place_time,
        lpad(rownum,10,'0')                     v1,
        lpad('x',100,'x')                       padding
from
        {large_row_source}
where
        rownum <= 1e5
;
```

Out of Range (2b)

```
select  /*+ 2 seconds later */ count(*)
from    t1
where   place_time > sysdate - 5/1440    -- 5 mins / ca. 900 rows
;

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     8 |   244   (7)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |     8 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |   888 |  7104 |   244   (7)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("PLACE_TIME">SYSDATE@!-.00347222222222222222222222222222222
              22222222)
```


I'm interested only in the cardinality estimate, not the access method. The hint explains the timing of the query. 5 minutes of data = 5 * 60 * 3 = 900 rows.

Out of Range (2c)

```
select  /*+ 5 minutes later */ count(*)
from    t1
where   place_time > sysdate - 5/1440    -- by now we're O-o-R
;

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     8 |   244   (7)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |     8 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |     3 |    24 |   244   (7)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("PLACE_TIME">SYSDATE@!-.00347222222222222222222222222222222
              22222222)
```


The query has been re-optimised and the input used is now slightly greater than the ***high_value*** for the column. The cardinality estimate is that of *"column = {constant}"*.

Guessing subqueries

```
create table t1 as select * from all_objects where rownum <= 20000;
create index t1_i1 on t1(owner, object_id);
execute dbms_stats.gather_table_stats(user,'t1',cascade=>true)

select  * from  t1
where   object_id > (select max(object_id) from t1 where owner = 'SYS');
```

```
| Id | Operation                      | Name   | Rows | Bytes | Cost  |
|  0 | SELECT STATEMENT               |        | 1000 |   119K|    57 |
|* 1 |  TABLE ACCESS FULL             | T1     | 1000 |   119K|    55 |
|  2 |   SORT AGGREGATE               |        |    1 |    10 |       |
|  3 |    FIRST ROW                   |        |    1 |    10 |     2 |
|* 4 |     INDEX RANGE SCAN (MIN/MAX)| T1_I1  |    1 |    10 |     2 |

Predicate Information (identified by operation id):
   1 - filter("OBJECT_ID">
            (SELECT MAX("OBJECT_ID") FROM "T1" "T1" WHERE "OWNER"='SYS'))
   4 - access("OWNER"='SYS')
```


Oracle has no idea about the value coming out of the subquery, so uses a fixed ***5%*** selectivity. Always look for 1% and 5% - sometimes the only option is to hint.

Function(column) (a)

```
create  table t1 as
select  cast(rownum as number(8,0)) id1
from    all_objects where rownum <= 10000

select  *
from    t1
where   mod(id1,2) = 0
;
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT   |      |   100 |   400 |     4 |
|*  1 |  TABLE ACCESS FULL| T1   |   100 |   400 |     4 |

Predicate Information (identified by operation id):
   1 - filter(MOD("ID1",2)=0)
```

Selectivity of:

function(column) = constant	1%
function(column) > constant	5%
function(column) between …	5% of 5% (= 0.25%)

Function(column) (b)

```
alter table t1 add
   m1 invisible generated always as (mod(id1,2)) virtual
;

execute dbms_stats.gather_table_stats(user,'t1');

select  *
from    t1
where   mod(id1,2) = 0
;
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT   |      |  5000 | 35000 |     4 |
|*  1 |  TABLE ACCESS FULL| T1   |  5000 | 35000 |     4 |

Predicate Information (identified by operation id):
   1 - filter("T1"."M1"=0)
```


Correlated Columns (a)

selectivity of (predicate1 **and** predicate2) =
selectivity of (predicate1) * selectivity of (predicate2)

```
create table t1 as
select
        rownum                          id,
        mod(rownum,100)                 n1,         -- sel = 1/100
        mod(rownum,100)                 n2,         -- sel = 1/100
        lpad(rownum,10,'0')             v1,
        lpad('x',100,'x')               padding
from
       large_row_source
where
        rownum <= 1e4
;
```

Correlated Columns (b)

```
select  count(*)
from    t1
where   n1 = 50
and     n2 = 50
;

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |     6 |    24   (5)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |     6 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |     1 |     6 |    24   (5)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("N1"=50 AND "N2"=50)
```


Default selectivity = (1/100 * 1/100) = 1/1e4.
Cardinality = num_rows * selectivity = 1e4 * 1/1e4 = 1.

Correlated Columns (c)

```
create index t1_i1 on t1(n1, n2);            -- index creation

select  count(v1)                            -- avoid index-only query
from    t1
where   n1 = 50
and     n2 = 50
;

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    17 |    24   (5)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |    17 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |   100 |  1700 |    24   (5)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("N1"=50 AND "N2"=50)
```


The optimizer can use the ***distinct_keys*** from ***user_indexes*** to check the actual number of combinations - even if the index is not used.

Correlated Columns (d)

```
drop index t1_i1;              -- replace index with column group

execute dbms_stats.gather_table_stats( -
    ownname    => null, -
    tabname    => 't1', -
    method_opt=> 'for columns (n1,n2) size 1'   -
)

select count(v1) from t1 where n1 = 50 and n2 = 50;

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    17 |    24   (5)| 00:00:01 |
|   1 |  SORT AGGREGATE    |      |     1 |    17 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |   100 |  1700 |    24   (5)| 00:00:01 |
-------------------------------------------------------------------------------
```


If you don't want an index a ***column group*** (best use of ***extended stats***) will do. There is a limit: https://jonathanlewis.wordpress.com/2016/12/07/extended-stats-4/

Correlated Columns (e)

Effects of *"out of range"* predicates

```
method_opt=> 'for columns (n1, n2) size 1'
select count(v1) from t1 where n1 = 50 and n2 = 101;

|* 2 |  TABLE ACCESS FULL| T1   |     1 |    1 |    24   (5)|


create index t1_i1 on t1(n1, n2);
select count(v1) from t1 where n1 = 50 and n2 = 101;

|  2 |  TABLE ACCESS BY INDEX ROWID BATCHED| T1   |     1 |   17 |     2   (0)|
```


If you use "out of range" predicates the optimizer will not use the column group stats or ***distinct_keys***. It goes back to linear decay and multiplication of selectivities.

Correlated Columns (f)

Effects of histograms

```
method_opt=> 'for columns (n1, n2) size 1 for columns n1 size 255'

select count(v1) from t1 where n1 = 50 and n2 = 50;

|* 2 |  TABLE ACCESS FULL| T1   |     1 |     1 |    24   (5)|
```

```
create index t1_i1 on t1(n1, n2);
method_opt=> 'for columns n1 size 255'

select count(v1) from t1 where n1 = 50 and n2 = 50;

|  2 |  TABLE ACCESS BY INDEX ROWID BATCHED| T1   |     1 |    17 |     2   (0)|
```


Histograms will stop the optimizer using the index ***distinct_keys*** and will block the use of column groups if the column group doesn't have its own histogram.

Correlated Columns (g)

Effects of "is null"

```
select  count(*) from t1
where   col1 = 1 and col2 = 1 and col3 = '04-Nov-2015';

select  count(*) from t1
where   col1 = 1 and col2 = 1 and col3 is null;
```

I have a column group defined on *(col1, col2, col3)*
Its frequency histogram records frequencies for ***both*** the combinations shown above.
The optimizer does not use the column group frequency for the *"is null"* version.

```
ColGroup (#1, VC) SYS_STU2IZIKAO#T0YCS1GYYTTOGYE
  Col#: 1 2 3    CorStregth: 1.60
ColGroup Usage:: PredCnt: 3  Matches Full: #1  Partial:  Sel: 0.3000

ColGroup (#1, VC) SYS_STU2IZIKAO#T0YCS1GYYTTOGYE
  Col#: 1 2 3    CorStregth: 1.60
ColGroup Usage:: PredCnt: 2  Matches Full:  Partial:
```


https://jonathanlewis.wordpress.com/2015/11/05/column-groups/
For matching purposes *"col3 is null"* doesn't appear to be a predicate !

The present is not the past (a)

- An orders table is partitioned by order date
- An order goes through 6 states.
  - new, picked, sent, invoiced, paid, done
- After a couple of weeks almost all are 'Done'
- Today almost everything is 'New' or 'Picked'

Typical queries:

- All orders not "Done" more than 4 weeks old.
- All orders from the last 48 hours that are still "New".
- All orders "Sent" in the last 48 hours.

The present is not the past (b)

```
select  *
from    (
        select trunc(date_placed) placed,   status
        from   orders
        where  date_placed > trunc(sysdate - 15)
        )       piv
        pivot   (
                        count(status) ct
                 for    status in (
                                'New'  as New,
                                'Pick' as Pick,
                                'Sent' as Sent,
                                'Inv'  as Inv,
                                'Paid' as Paid,
                                'Done' as Done
                        )
        )
order by placed
;
```

The present is not the past (c)

PLACED	NEW_CT	PICK_CT	SENT_CT	INV_CT	PAID_CT	DONE_CT
23-AUG-18	**0**	**0**	**0**	**0**	**0**	**4114**
24-AUG-18	0	0	0	0	0	4114
25-AUG-18	0	0	0	0	252	3862
26-AUG-18	0	0	0	0	2523	1592
27-AUG-18	0	0	0	0	2499	1615
28-AUG-18	0	0	0	2518	1070	526
29-AUG-18	0	0	0	3762	353	0
30-AUG-18	**0**	**0**	**885**	**2964**	**265**	**0**
31-AUG-18	0	0	3805	309	0	0
01-SEP-18	0	0	3811	303	0	0
02-SEP-18	0	3053	998	64	0	0
03-SEP-18	0	3807	307	0	0	0
04-SEP-18	1405	2513	196	0	0	0
05-SEP-18	3807	308	0	0	0	0
06-SEP-18	3783	331	0	0	0	0
07-SEP-18	**213**	**16**	**0**	**0**	**0**	**0**

The present is not the past (d)

```
select  partition_name, sample_size, histogram, num_buckets
from    user_part_col_statistics
where   table_name = 'ORDERS' and column_name = 'STATUS'
order by
        partition_name        -- weekly, interval partitioned
;
```

PARTITION_NAME	Sample	HISTOGRAM	NUM_BUCKETS
SYS_P527	5,554	FREQUENCY	1
SYS_P528	5,479	FREQUENCY	1
SYS_P529	5,513	FREQUENCY	1
SYS_P530	5,509	FREQUENCY	1
SYS_P531	5,391	FREQUENCY	1
SYS_P532	5,569	FREQUENCY	1
SYS_P533	28,800	FREQUENCY	2
SYS_P534	28,800	FREQUENCY	4
SYS_P535	20,801	FREQUENCY	4

These stats were from 18.3 with incremental stats enabled, with one partition gathered at a time. (Why is the sample 5,500 for a frequency histogram ?)

The present is not the past (e)

user_part_histograms

Partition_name	Bucket	EP_actual_value
SYS_P532	5569	Done
SYS_P533	28548	Done
	28800	Paid
SYS_P534	3733	Done
	13589	Inv
	20299	Paid
	28800	Sent
SYS_P535	64	Inv
	9272	New
	19300	Pick
	20801	Sent

user_tab_histograms

Bucket	Actual Value	
208744		Done
218664	(9,920)	Inv
227872	(9,208)	New
234834	(6,962)	Paid
244862	(10,028)	Pick
254864	(10,002)	Sent

The present is not the past (f)

```
select
        trunc(date_placed), count(*)
from
        orders
where
        status = 'New'
group by
       trunc(date_placed)
;
```

Actual

TRUNC(DAT	COUNT(*)
03-SEP-18	0
04-SEP-18	1405
05-SEP-18	3807
06-SEP-18	3783
07-SEP-18	213
Sum	9208

Predicted

Id	Operation	Name	Rows	Bytes	TempSpc	Pstart	Pstop
0	SELECT STATEMENT		36123	458K			
1	HASH GROUP BY		36123	458K	864K		
2	PARTITION RANGE ALL		36129	458K		1	1048575
* 3	TABLE ACCESS FULL	ORDERS	36129	458K		1	1048575

```
Predicate Information (identified by operation id):
   3 - filter("STATUS"='New')
```

The present is not the past (g)

```
| Id | Operation                    | Name    | Rows | Bytes | Pstart| Pstop |
|  0 | SELECT STATEMENT             |         |  900 | 11700 |       |       |
|  1 |  HASH GROUP BY               |         |  900 | 11700 |       |       |
|  2 |   PARTITION RANGE ITERATOR|         |  900 | 11700 |    35 |1048575|
|* 3 |    TABLE ACCESS FULL         | ORDERS  |  900 | 11700 |    35 |1048575|

Predicate Information (identified by operation id):
    3 - filter("DATE_PLACED">=TO_DATE(' 2018-09-01 00:00:00',
              'syyyy-mm-dd hh24:mi:ss') AND "STATUS"='New')

| Id  | Operation                    | Name    | Rows  | Bytes | Pstart| Pstop |
|   0 | SELECT STATEMENT             |         |  5566 | 72358 |       |       |
|   1 |  HASH GROUP BY               |         |  5566 | 72358 |       |       |
|   2 |   PARTITION RANGE ITERATOR|         |  5566 | 72358 |    36 |1048575|
|*  3 |    TABLE ACCESS FULL         | ORDERS  |  5566 | 72358 |    36 |1048575|

Predicate Information (identified by operation id):
    3 - filter("STATUS"='New' AND "DATE_PLACED">=TO_DATE(' 2018-09-04 00:00:00',
              'syyyy-mm-dd hh24:mi:ss'))
```


Lost Skew (a)

```
create table facts as
select
        rownum                                  id,
        trunc(3 * abs(dbms_random.normal))      id_status,
        lpad(rownum,10,'0')                     v1,
        lpad('x',100,'x')                       padding
from
        {large row source}
where
        rownum <= 1e5
;

create table statuses as
select
        id, chr(65 + id)    status_code,
        rpad('x',100,'x')   description
from    (select distinct(id_status) id from facts)
;
```

Lost Skew (b)

```
select id_status, count(*)
from   facts
group by id_status
order by id_status
;
```

ID_STATUS	COUNT(*)
0	26050
1	23595
2	18995
3	13415
4	8382
5	4960
6	2643
7	1202
8	490
9	194
10	***55***
11	17
12	2

```
select id, status_code
from   statuses
order by id
;
```

ID	S
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	***K***
11	L
12	M

Note that there are ***13 values*** available for status - we'll be using that number later - and the data is clearly very skewed. So we gather a frequency histogram on it.

Lost Skew (c)

```
select  sum(fct.id)
from    facts   fct
where   fct.id_status = 10;
```

```
-------------------------------------------------------------------------------
| Id | Operation                           | Name    | Rows | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                    |         |    1 |     8 |    12   (0)|
|  1 |  SORT AGGREGATE                     |         |    1 |     8 |            |
|  2 |   TABLE ACCESS BY INDEX ROWID BATCHED| FACTS   |   55 |   440 |    12   (0)|
|  3 |    BITMAP CONVERSION TO ROWIDS      |         |      |       |            |
|* 4 |     BITMAP INDEX SINGLE VALUE       | FCT_B1  |      |       |            |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("FCT"."ID_STATUS"=10)
```


When we query by a column on the ***fact*** table the optimizer uses the histogram to get an accurate estimate of rows returned.

Lost Skew (d)

```
select  sum(fct.id)
from    facts fct, statuses sta
where   fct.id_status = sta.id
and     sta.status_code = 'K';
```

```
-------------------------------------------------------------------------------
| Id  | Operation           | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |          |     1 |    13 |   233   (4)| 00:00:01 |
|   1 |  SORT AGGREGATE     |          |     1 |    13 |            |          |
|*  2 |   HASH JOIN         |          |  7692 | 99996 |   233   (4)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL| STATUSES |     1 |     5 |     2   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| FACTS    |  100K |  781K |   229   (3)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("FCT"."ID_STATUS"="STA"."ID")
   3 - filter("STA"."STATUS_CODE"='K')
```


When we query by a column on the ***statuses*** table the optimizer uses the number of distinct values as its basis for estimation: 7692 = 100,000 / 13.

Lost Skew (e)

```
create bitmap index fct_b2 on facts (sta.status_code)
from   facts  fct,  statuses sta
where  sta.id = fct.id_status
;
```

```
-------------------------------------------------------------------------------
| Id | Operation                           | Name   | Rows | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                    |        |    1 |     8 |   441   (1)|
|  1 |  SORT AGGREGATE                     |        |    1 |     8 |            |
|  2 |   TABLE ACCESS BY INDEX ROWID BATCHED| FACTS  | 7692 | 61536 |   441   (1)|
|  3 |    BITMAP CONVERSION TO ROWIDS      |        |      |       |            |
|* 4 |     BITMAP INDEX SINGLE VALUE       | FCT_B2 |      |       |            |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("FCT"."SYS_NC00005$"='K')
```


A ***bitmap join index*** doesn't fix the cardinality estimate even if we ***set_column_stats()*** to fake a histogram onto the virtual column supporting the index definition.

Lost Skew (f)

```
select  sum(fct.id)
from    facts   fct
where   fct.id_status in (
                select  /*+ precompute_subquery */ id
                from    statuses where status_code = 'K'
        );

-------------------------------------------------------------------------------
| Id | Operation                           | Name    | Rows | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                    |         |    1 |     8 |    12   (0)|
|  1 |  SORT AGGREGATE                     |         |    1 |     8 |            |
|  2 |   TABLE ACCESS BY INDEX ROWID BATCHED| FACTS   |   55 |   440 |    12   (0)|
|  3 |    BITMAP CONVERSION TO ROWIDS      |         |      |       |            |
|* 4 |     BITMAP INDEX SINGLE VALUE       | FCT_B1  |      |       |            |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("FCT"."ID_STATUS"=10)
```


There is a workaround - used inside Oracle OLAP - which is to use an IN subquery with the undocumented ***/*+ precompute_subquery */*** hint.

Distributed woes

In a distributed query the optimizer

1. doesn't get histogram information from remote database(s).
2. ignores remote function based indexes
   … including reverse key and "descending" indexes
3. ignores remote virtual columns and extended stats
   … including the indexes on such columns
4. Ignores remote ***distinct_keys*** in selectivity calculations
   … considers only 20 remote indexes anyway
   … (possibly the last 20 created - highest ***object_id***)

https://jonathanlewis.wordpress.com/2013/08/19/distributed-queries-3/

https://jonathanlewis.wordpress.com/2013/11/11/reverse-key/

https://jonathanlewis.wordpress.com/2018/05/04/fbi-limitation/

https://jonathanlewis.wordpress.com/2018/05/08/20-indexes/

The limit of 20 indexes is documented - the suggestion about which 20 is speculative backed up with a little experimentation. Watch out for plan changes on drop/receate

Simple distributed join (a)

```
create table t_{local | remote} as
select
        rownum                          id,         -- Primary Key
        mod(rownum,100)                 n1,
        mod(rownum,100)                 n2,
        mod(rownum,100)                 n3,
        mod(rownum,100)                 n4,
        lpad(rownum,10,'0')             v1,
        lpad('x',100,'x')               padding
from
        {large row source}
where
        rownum <= 1e4;


create index t_remote_i1 on t_remote(n3 , n4 );
```


Columns ***n3*** and ***n4*** are correlated so that there are 100 distinct combinations, and the optimizer should spot this in the index ***distinct_keys***. (Only created on t_remote)

Simple distributed join (b)

```
select
        /*+
                leading (t_local t_remote)
                use_nl_with_index(t_remote)
        */
        t_local.*,
        t_remote.*
from
        t_local,
        t_remote
--      t_remote@&m_remote      t_remote
where
        t_local.id between 1 and 10          -- 10 rows
and     t_remote.n3 = t_local.n1
and     t_remote.n4 = t_local.n2
;
```


We construct a query that should do a nested loop join from ***t_local*** to ***t_remote***, using an indexed nested loop. Optionally the table t_remote could be local or remote.

Simple distributed join (c)

Local join

```
-------------------------------------------------------------------------------
| Id | Operation                              | Name        | Rows | Cost (%CPU)|
-------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                       |             | 1000 |  1015   (1)|
|  1 |  NESTED LOOPS                          |             | 1000 |  1015   (1)|
|  2 |   NESTED LOOPS                         |             | 1000 |  1015   (1)|
|  3 |    TABLE ACCESS BY INDEX ROWID BATCHED| T_LOCAL     |   10 |     3   (0)|
|* 4 |     INDEX RANGE SCAN                   | T_LOCAL_PK  |   10 |     2   (0)|
|* 5 |    INDEX RANGE SCAN                    | T_REMOTE_I1 |  100 |     1   (0)|
|  6 |   TABLE ACCESS BY INDEX ROWID          | T_REMOTE    |  100 |   101   (0)|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("T_LOCAL"."ID">=1 AND "T_LOCAL"."ID"<=10)
   5 - access("T_REMOTE"."N3"="T_LOCAL"."N1" AND "T_REMOTE"."N4"="T_LOCAL"."N2")
```


When ***t_remote*** is a local table the rowsource estimate for access to it is 100 rows per operation - which is what we expect thanks to the correlation information.

Simple distributed join (d)

Distributed Join

```
-------------------------------------------------------------------------------
| Id | Operation                              | Name       | Rows | Cost (%CPU)|
-------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                       |            |   10 |    23   (0)|
|  1 |  NESTED LOOPS                          |            |   10 |    23   (0)|
|  2 |   TABLE ACCESS BY INDEX ROWID BATCHED| T_LOCAL    |   10 |     3   (0)|
|* 3 |    INDEX RANGE SCAN                    | T_LOCAL_PK |   10 |     2   (0)|
|  4 |   REMOTE                               | T_REMOTE   |    1 |     2   (0)|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("T_LOCAL"."ID">=1 AND "T_LOCAL"."ID"<=10)

Remote SQL Information (identified by operation id):
----------------------------------------------------
   4 - SELECT /*+ USE_NL_WITH_INDEX ("T_REMOTE")*/ "ID","N1","N2","N3","N4","V1",
       "PADDING","N1V","N2V" FROM "T_REMOTE" "T_REMOTE" WHERE "N3"=:1 AND "N4"=:2
       (accessing 'OR32@LOOPBACK' )
```


When ***t_remote*** is in the remote database the optimizer doesn't use ***distinct_keys*** for its estimate, and seems to use only the simple column selectivity. (Unchanged in 18.3)

Problems ***gathering*** Statistics

Histograms (a)

- 12c - new types (good)
- 12c - changed meaning of "size repeat" (bad)
- Small sample for hybrid/height => unstable

```
create table t1 as
select
        ceil(sqrt(1 + mod(rownum-1,10000)))     n1, ...
from
        large_row_source
where
        rownum <= 1e6                   -- 1M rows, 100 distinct values
;
```

Repeat 4 times:

```
dbms_stats.gather_table_stats(.. method_opt  => 'for columns n1 size 10');
```


12c is relatively safe for *Frequency* and *Top-N* histograms as the ***auto_sample_size*** becomes 100% at virtually no marginal cost; but *Hybrid* still uses a sample.

Histograms (b)

```
select  column_name, sample_size, histogram
from
        user_tab_cols
where
        column_name = 'N1'
and     table_name  = 'T1'
;

COLUMN_NAME                Sample HISTOGRAM
-------------------- ------------ ---------------
N1                          5,542 HYBRID
N1                          5,519 HYBRID
N1                          5,480 HYBRID
N1                          5,463 HYBRID
```


Repeating the sample 4 times in a row doesn't even result in the same number of rows being sampled. The starting target is 5,500 rows, adjusted to allow for ***num_nulls***.

Histograms (c)

```
select  endpoint_number, endpoint_value
from    user_tab_histograms
where   table_name  = 'T1'
and     column_name = 'N1'
order by endpoint_number;
```

EP_NUM	VAL	EP_NUM	VAL	EP_NUM	VAL	EP_NUM	VAL
2	1	1	1	1	1	2	1
649	33	626	33	637	***35***	633	33
1292	47	1253	47	1254	***48***	1282	48
1951	58	1890	58	1901	***59***	1894	58
2609	68	2517	***67***	2519	68	2514	67
3263	76	3187	76	3225	***77***	3130	***75***
3949	84	3804	***83***	3926	***85***	3786	83
4600	91	4472	***90***	4619	***92***	4417	90
5231	97	5097	***96***	5271	***98***	5037	96
5542	100	5519	100	5480	100	5463	100

Even for the values with large numbers of repetitions we don't necessarily see the same values re-appearing in the sequence of 4 samples.

Histogram bug 12c (a)

```
create table t1 as
select case
                when mod(rownum,2) = 0       -- 50% have special value
                        then '999960'
                        else lpad(rownum,6,'0')
        end     as bad_col ...
from    large_row_source
where   rownum < 1e6;

begin
        dbms_stats.gather_table_stats(
                ownname         => 'TEST_USER',
                tabname         => 'T1',
                method_opt      => 'for columns bad_col size 254'
        );
end;
/
```


Half the data holds the value '999960' and every other value is "rare", but there are some rows with a higher value - with '999999' as the highest value.

Histogram bug 12c (b)

```
select  histogram, sample_size, num_buckets
from    user_tab_columns
where   table_name = 'T1'
and     column_name = 'BAD_COL'
;

select
        endpoint_number         end_no,
        to_char(endpoint_value,'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx') end_val,
        endpoint_actual_value   end_av,
        endpoint_repeat_count   end_rpt
from
        user_tab_histograms
where
        table_name  = 'T1'
and     column_name = 'BAD_COL'
order by
        endpoint_number
;
```


The ***hybrid*** (and ***Top-N***) histogram ***must*** hold the highest and lowest values so if the sample doesn't happen to capture them the captured values get overwritten.

Histogram bug 12c (c)

```
COLUMN_NAME          HISTOGRAM              Sample    Buckets
-------------------- -------------- ------------ ----------
BAD_COL              HYBRID                5,440        254

    END_NO END_VAL                          END_AV          END_RPT
---------- -------------------------------- ------------ ----------
         1  303030303031001f0fe211e0800000 000001                1
        12  303034393930e637e4f5ed8de00000 004991                1
        23  3030393439390c1a6e316105200000 009499                1
...
      2727  393936393434fbc31b1f8296200000 996945                1
      2736  393939363734ebff9d408ce3000000 999675                1
      5440  393939393938f86f9b35437a800000 999999                1
```

This histogram says half the rows have values evenly spread between 999676 and 999999, which is true - but it has lost sight of the specific very popular value.

https://jonathanlewis.wordpress.com/2018/01/15/histogram-hassle/

Histogram Strategies

- Do not gather histograms
  - Unless you ***know*** they will be Frequency
  - (or Top-N if you've patched 12c).
  - *(special cases: dates stored as number, special values for null)*
- Get rid of all *"size repeat"* (in 12c+)
- set_global_prefs
  - `method_opt => 'for all columns size 1'`
- set_table_prefs
  - `method_opt => 'for all columns size 1' ||`
  - `'for columns XXX size N'`
- Write scripts for "unusual" histograms

Index Sampling

Auto_sample_size = 100% for tables.

- Really samples for indexes - target 1,100 blocks.
- Full scan for indexes < ca. 5,000 blocks
- Sample fast full scan for larger indexes
- Doesn't use *approximate_ndv* for distinct !

Side effects:

- doing a separate gather using 100% can be very slow
- auto_sample_size can give inaccurate results for ***distinct_keys***
- auto_sample_size may leave blocks needing cleanout

Update (10th Sept)

- There is a recent patch for 18.0.0.0 and 12.2.0.1 to address this:
- 27268249: ENHANCE INDEX NDK STATISTICS GATHERING
- Pre-requisite: 27674384 April 2018 update
  - Superseded by 28161333 by Jul 2018 update !

Index Strategies (a)

- Create a column group on the index definition
  - Gets you an accurate ***distinct_keys*** "free"
- Schedule serial fast full scans for critical indexes
  - `select /*+ index_ffs(alias index) */` etc.
  - Don't forget you need a non-null column in the index
  - (or a predicate that implies the values will be in the index)
  - Best to set `"_serial_direct_read" = never` for this.
- Write code using the supplied APIs
  - `dbms_stats.set_index_stats()`

Index clustering_factor

- Often too high.
  - Slight scattering looks like extreme scattering to Oracle
  - Concurrency benefits of ASSM exaggerate this
  - So optimizer uses wrong index (or no index)
- Directly addressable since 11g.
  - Set table preference *table_cached_blocks*

Index Strategoes (b)

- Adjust *'table_cached_blocks'*
  - set_global_prefs()
  - set_table_prefs()

```
execute dbms_stats.set_global_prefs('table_cached_blocks',16)
```

Default value: 1
Maximum value: 255
larger numbers increase the CPU needed to gather index stats.

Sensible strategic values:

Single instance	16
RAC	16 * Number of nodes
special case	dbms_stats. auto_table_cached_blocks (0)

"auto" = `least(ceil(least(1% table blocks, 0.1% buffer_cache)), 255)`

Table Sampling (a)

- Auto_sample_size
  - Uses 100%, but doesn't do *"count distinct()"*.
    - Samples for histograms (version dependent)
  - This is a very good thing (mostly)
  - Reasonable speed, very accurate
- Special cases (threats)
  - Chained rows (not migrated)
  - IoTs with overflows
  - Expensive "virtual" columns

Table Sampling (b)

https://jonathanlewis.wordpress.com/2014/03/02/auto-sample-size/

Each row in an IOT overflow is accessed by "rowid" as the scan takes place.
This can result in a huge volume of single block read requests

Similarly if a table has chained rows, Oracle follows each row as the scan takes place

https://community.oracle.com/thread/2620088

"Gathering stats takes hours after I created an index"

```
create bitmap index i_s_rmp_eval_csc_msg_actions on
    s_rmp_evaluation_csc_message (
        decode(instr(xml_message_text,' '),0,0,1)
    )
```

This creates a hidden virtual column in the table - and Oracle gathers stats on it.

Table strategy

- Set table preference to avoid "dangerous" columns
  - Except it doesn't currently help!
  - Not even if you set an explicit ***method_opt***
  - So write code using *set_column_stats()*

```
begin
    dbms_stats.set_table_prefs(
        ownname =>      user,
        tabname =>      't1',
        pname   =>      'METHOD_OPT',
        pvalue  =>      'for columns size 1 id v1'
);
end;
/
```


It would be nice if Oracle has a ***method_opt*** option of *"except columns …"* (and really meant it).

Limited gather ? (a)

This table has 3 real columns, one set of extended stats, one virtual, one FBI

```
COLUMN_NAME                     DATA_DEFAULT
ID
V1
V2
SYS_STUIBQVZ_50PU9_NIQ6_G6_2Y7  SYS_OP_COMBINED_HASH("V1","V2")
ID_12                           MOD("ID",12)
SYS_NC00006$                    MOD("ID",10)

begin
        dbms_stats.gather_table_stats(
                ownname     => user,
                tabname     => 't1',
                method_opt  => 'for columns size 1 id v1',
                cascade     => false
        );
end;
/
```


This column listing comes from ***user_tab_cols***, ordered by ***internal_column_id***.

Limited gather ? (b)

SQL run by the gather command (from trace file, cosmetically enhanced):

```
select
        /*+
                full(t) no_parallel(t) no_parallel_index(t) dbms_stats
                cursor_sharing_exact use_weak_name_resl dynamic_sampling(0)
                no_monitoring xmlindex_sel_idx_tbl no_substrb_pad
        */
        to_char(count(ID)),
        substrb(dump(min(ID),16,0,64),1,240),
        substrb(dump(max(ID),16,0,64),1,240),
        to_char(count(V1)),
        substrb(dump(min(V1),16,0,64),1,240),
        substrb(dump(max(V1),16,0,64),1,240),
        to_char(count(V2)),
        to_char(count(SYS_STUIBQVZ_50PU9_NIQ6_G6_2Y7)),
        to_char(count(ID_12)),
        to_char(count(SYS_NC00006$))
from
        TEST_USER.T1 t  /* NDV,NIL,NIL,NDV,NIL,NIL,ACL,ACL,ACL,ACL*/
```


The two columns identified by the preferences get the full works (***substrb()***, and ***ndv*** in the hint show this). But the rest of them get evaluated for the count.

Reporting stats gathering - 12c

```
set long 1000000
set pagesize   0
set linesize 255
set trimspool on

select
        dbms_stats.report_stats_operations(
                since => sysdate - 1
        ) text_line
from dual;

select
        dbms_stats.report_single_stats_operation(
                opid         => 25809,
                detail_level => 'ALL'
        ) text_line
from dual;
```


For a sample of output, see:
https://jonathanlewis.wordpress.com/2018/09/10/stats-time-2/

Faking Stats

Story so far …

- There are problems you can't fix with stats.
- There are configuration options you can set
- Correlation handling is fragile
- low_value / high_value effects can be dire
- Histograms may be unstable
  - Frequency can be gathered safely in 12c
  - Bug fix will allow safe Top-N gather in 12c
- Gathering stats may do a lot of work.

tl;dr

- Create a simple general strategy
- Ongoing identification of special cases
- Program stats ***creation*** as part of the codebase
  - Globally disable histogram collection
    - *set_global_prefs('method_opt', 'for all columns size 1')*
  - Set a few table prefs … *method_opt, table_cached_blocks*
  - ***Program*** special cases for high (and low) values
  - ***Program*** for histograms … *especially hybrid (height)*
  - ***Program*** for partitioning
  - unlock; scripted changes; lock; (run autostats)

dbms_stats

The Carlsberg functions (probably)

- lock_table_stats()
- set_table_prefs()
- set_column_stats()

- ***lock_schema_stats()***
- ***unlock_table_stats()***
- ***set_global_prefs()***
- ***set_table_stats(), set_index_stats()***

Create Histogram (a)

```
declare
        c_array         dbms_stats.chararray;
        m_rec           dbms_stats.statrec;
        m_distcnt       number;
        m_density       number;
        m_nullcnt       number;
        m_avgclen       number;

begin
        m_distcnt       := 5;
        m_density       := 0.00001;
        m_nullcnt       := 0;
        m_avgclen       := 1;
```

http://jonathanlewis.wordpress.com/2009/05/28/frequency-histograms/ (frequency)
http://jonathanlewis.wordpress.com/2010/03/23/fake-histograms/ (height balanced)
http://jonathanlewis.wordpress.com/2015/09/04/histogram-tip/

The ***histogram_tip*** article comments makes some comments about the "repeat count" array that you need to populate if you want to create a 12c ***Hybrid*** histogram.

Create Histogram (b)

```
        c_array      := dbms_stats.chararray( 'C', 'P', 'R', 'S', 'X' );
        m_rec.bkvals := dbms_stats.numarray (5000,   3,   3,   3,  5000);
--      m_rec.rpcnts := dbms_stats.numarray (   0,   0,   0,   0,  0);
        m_rec.epc    := 5;

        dbms_stats.prepare_column_values(m_rec, c_array);
        dbms_stats.set_column_stats(
                ownname         => user,
                tabname         => 'T1',
                colname         => 'STATUS',
                distcnt         => m_distcnt,
                density         => m_density,
                nullcnt         => m_nullcnt,
                srec            => m_rec,
                avgclen         => m_avgclen
                );
end;
```


The ***repcnts*** are for the 12c ***Hybrid*** histogram, the array has to be zero'ed for frequency histograms. The notes in ***dbmsstat.sql*** about this array need some clarification.

Adjusting low/high

```
dbms_stats.get_column_stats(
   ownname      => NULL,  tabname      => 'T1',
   colname      => 'N1',  srec         => m_rec, ...
);

m_rec.novals := dbms_stats.numarray(1,1000);
m_rec.rpcnts := dbms_stats.numarray(0,   0);        -- 12c

m_rec.epc    := 2;                    -- endpoint count
m_rec.bkvals := null;                 -- not frequency histogram

dbms_stats.prepare_column_values(m_rec, m_rec.novals);

dbms_stats.set_column_stats(
   ownname      => NULL,  tabname      => 'T1',
   colname      => 'N1',  srec         => m_rec, ...
);
```


Adjusting the low/high values is simply a case of installing a "two value histogram". You can start by reading the current column stats, then setting one or both values.

Partitioned Tables

- "Bug" - collects stats too often
  - Table preference to avoid over-collection in 12.2
- "Global" stats cover all history
  - We often want stats to represent recent history
- Lock stats
  - Create stats only by program
- dbms_stats.copy_table_stats