Join Selectivity

Jonathan Lewis

jonathanlewis.wordpress.com

www.jlcomp.demon.co.uk

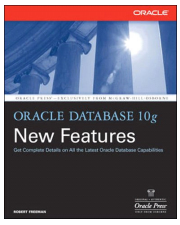
Who am I ?

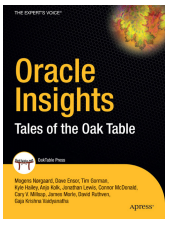
Independent Consultant

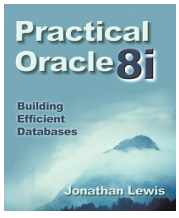
31+ years in IT
26+ using Oracle

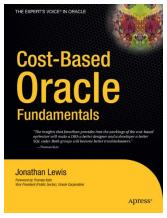
Strategy, Design, Review, Briefings, Educational, Trouble-shooting

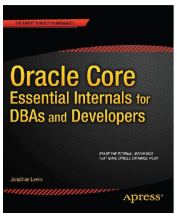
Oracle author of the year 2006
Select Editor's choice 2007
UKOUG Inspiring Presenter 2011
ODTUG 2012 Best Presenter (d/b)
UKOUG Inspiring Presenter 2012
UKOUG Lifetime Award (IPA) 2013
Member of the Oak Table Network
Oracle *ACE Director*
O1 visa for USA











Topics

- Comparing columns
- Distinct values
- Basic mechanism
- Sanity Checks

Warning (a)

```
select
        t1.small_vc,
        t2.small_vc,
        t3.small_vc
from
        t1, t2, t3
where
        t1.n1 between 40 and 50
and     t1.id1 = t2.id1
and     t1.ind_pad = t2.ind_pad
and     t1.id2 = t2.id2
and     t3.id = t1.id1
;
```

Warning (b)

```
| Id  | Operation            | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT     |      |   484 | 64856 |   278 |
|*  1 |  HASH JOIN           |      |   484 | 64856 |   278 |
|*  2 |   HASH JOIN          |      |   484 | 57596 |    18 |
|   3 |    TABLE ACCESS FULL| T2   |    20 |  1160 |     5 |
|*  4 |    TABLE ACCESS FULL| T1   |   484 | 29524 |    12 |
|   5 |   TABLE ACCESS FULL | T3   |  5000 | 75000 |   259 |
```

```
Predicate Information (identified by operation id):
   1 - access("T3"."ID"="T1"."ID1")
   2 - access("T1"."ID1"="T2"."ID1"
              AND "T1"."IND_PAD"="T2"."IND_PAD"
              AND "T1"."ID2"="T2"."ID2")
   4 - filter("T1"."N1">=40 AND "T1"."N1"<=50)
```

There's an index on t2 that isn't being used - so we'll ***drop*** it.

Warning (c)

```
| Id  | Operation                    | Name   | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT             |        |    52 |  6968 |    70 |
|   1 |  NESTED LOOPS                |        |    52 |  6968 |    70 |
|   2 |   NESTED LOOPS               |        |    52 |  6968 |    70 |
|*  3 |    HASH JOIN                 |        |    52 |  6188 |    18 |
|   4 |     TABLE ACCESS FULL        | T2     |    20 |  1160 |     5 |
|*  5 |     TABLE ACCESS FULL        | T1     |   484 | 29524 |    12 |
|*  6 |    INDEX UNIQUE SCAN         | T3_PK  |     1 |       |       |
|   7 |   TABLE ACCESS BY INDEX ROWID| T3     |     1 |    15 |     1 |
```

```
Predicate Information (identified by operation id):
   3 - access("T1"."ID1"="T2"."ID1"
              AND "T1"."IND_PAD"="T2"."IND_PAD"
              AND "T1"."ID2"="T2"."ID2")
   5 - filter("T1"."N1">=40 AND "T1"."N1"<=50)
   6 - access("T3"."ID"="T1"."ID1")
```

Dropping an "unused" index changed the execution plan

Comparing columns (a)

```
create table t1 as
select
        rownum                                  id,
        mod(rownum-1,200)                       mod_200,
        trunc(dbms_random.value(0,300))         rand_300
from
        {very_large_rowsource}
where   rownum <= 1e6
;

create table t2 as
select
        rownum                                          id,
        mod(rownum-1,200)                               mod_200,
        250 + trunc(dbms_random.value(0,300))           rand_300
from
        {very_large_rowsource}
where   rownum <= 1e6
;
```


Comparing columns (b)

```
select * from t1 where mod_200 = rand_300;

| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |  3333 |   178K|  1117   (6)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |  3333 |   178K|  1117   (6)| 00:00:06 |

Predicate Information (identified by operation id):
   1 - filter("MOD_200"="RAND_300")
```

The optimizer treats this as one of:

```
        mod_200  = {unknown}
        rand_300 = {unknown}
```

and uses the column with the larger number of distinct values / lower selectivity.

3333 = 1M / 300

There is no allowance for "badly overlapping" ranges - t2 gets the same cardinality

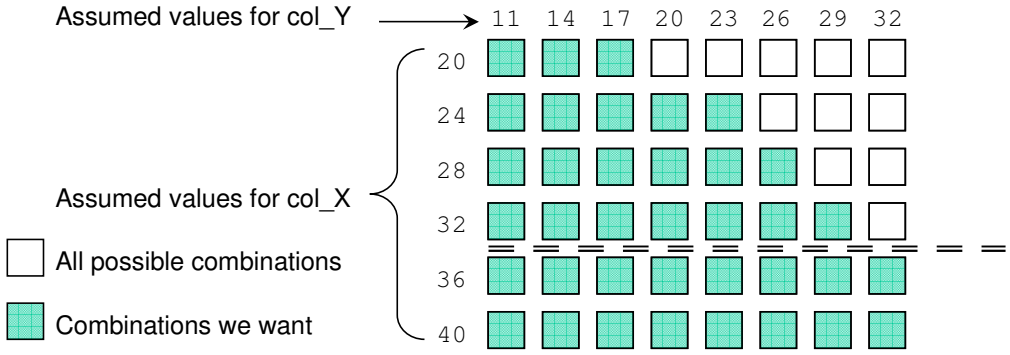
Comparing columns (c)

```
select * from t1 where col_X > col_Y;
```

Col_X: low = 20, high = 40, NDV = 6
Col_Y: low = 11, high = 32, NDV = 8

Simple visual approximation

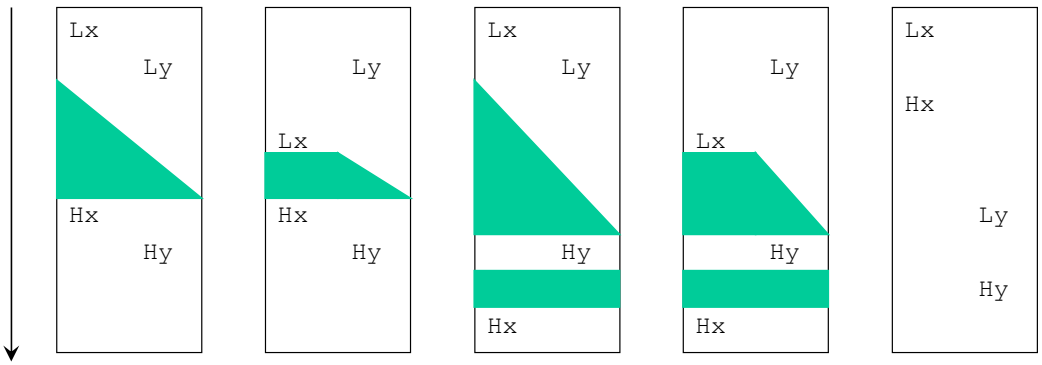
The model assumes uniform distribution of data



Comparing columns (c)

Four patterns (plus 1 degenerate)

Col_X	Low value Lx,	High value Hx
Col_Y	Low value Ly,	High value Hy



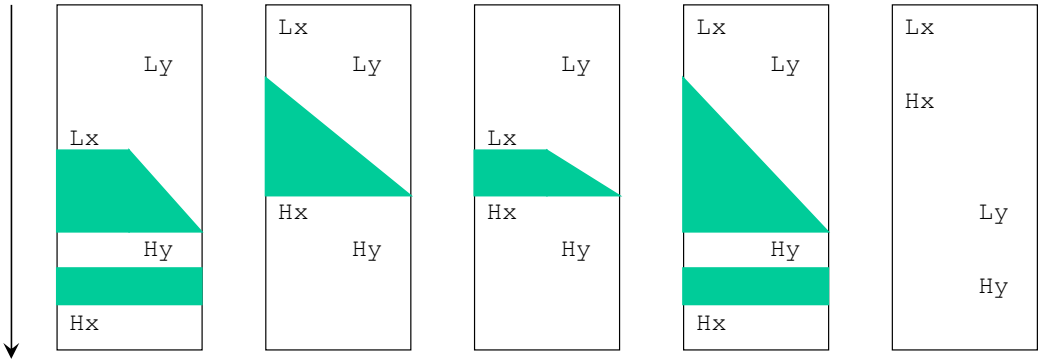
$$\sum_{i = Lx \ldots Hx} ((greatest(least(i,Hy),Ly) - Ly))$$

For "similar" columns the selectivity is 0.5

Comparing columns (c)

Four patterns (plus 1 degenerate)

Col_X Low value Lx, High value Hx
Col_Y Low value Ly, High value Hy



$$\sum_{i = Lx \ldots Hx} ((greatest(least(i,Hy),Ly) - Ly))$$

For "similar" columns the selectivity is 0.5

Distinct (a)

```
select   distinct rand_300
from     t1
where    date_1000 = trunc(sysdate) + 10
;

| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |   289 |  3468 |  1288   (17)| 00:00:01 |
|   1 |  HASH UNIQUE       |      |   289 |  3468 |  1288   (17)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| T1   |  1000 | 12000 |  1287   (17)| 00:00:01 |

Predicate Information (identified by operation id):
    2 - filter("DATE_1000"=TRUNC(SYSDATE@!)+10)
```

The result is not ***num_distinct*** from user_tab_columns

The same algorithm is used for *"select rand_300, count(*) group by"*

Distinct (b)

t1 has 1,000,000 rows (number of rows) ("nr").
mod_300 has 300 distinct values (number of distinct) ("nd")
date_1000 = {constant} returns a sample of 1,000 rows ("s")

So how many different values are you likely to get for:

```
select distinct mod_300 from t1 where date_1000 = {constant};
```

Selection without replacement

```
distinct = 300 * (1 - power(1 - 1000/1000000, 1000000/1000))
         = 289.3156

distinct = nd * (1 - power(1 - s/nr, nr/nd))
```

Required Reading, Alberto Dell'Era: http://www.adellera.it/investigations

Distinct (c)

```
create or replace function distinct_formula(
        i_num_distinct in      number,
        i_row_count    in      number,
        i_sample_size  in      number
) return number is
begin
        return  i_num_distinct *
                (1 -
                        power(
                                1 - i_sample_size/i_row_count,
                                i_row_count/i_num_distinct
                        )
                )
   ;
end;
/
```

Distinct (d)

```
create table t1 as
select
        trunc(dbms_random.value(0,1000))      n_1000,
        trunc(dbms_random.value(0,750))       n_750
        trunc(dbms_random.value(0,600))       n_600
        trunc(dbms_random.value(0,400))       n_400,
        trunc(dbms_random.value(0,90))        n_90,
        trunc(dbms_random.value(0,72))        n_72,
        trunc(dbms_random.value(0,40))        n_40,
        trunc(dbms_random.value(0,3))         n_3
from
        {very large rowsource}
where
        rownum <= 1e6
;                                  -- clone to create tables t2 and t3

Arbitrary scaling factor:     sqrt(2)
Limiting value:               number of rows in table
```


Distinct (e)

```
select
        n_40, n_72, n_90,  count(*)
from
        t1
group by
        n_40, n_72, n_90
;
```

Id	Operation	Name	Rows	Bytes	TempSpc	Cost
0	SELECT STATEMENT		129K	1139K		2020
1	HASH GROUP BY		***129K***	1139K	19M	2020
2	TABLE ACCESS FULL	T1	1000K	8789K		614

Distinct `= 40 * 72 * 90 / (sqrt(2) * sqrt(2)) = 129600`

N columns => divide by sqrt(2) ^ (N-1)

Distinct (f)

```
select  distinct n_400, n_3
from    t1
where   n_1000 = 0
;

| Id  | Operation           | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT    |      |   779 |  8569 |   601 |
|   1 |  HASH UNIQUE        |      |   779 |  8569 |   601 |
|*  2 |   TABLE ACCESS FULL| T1   |  1000 | 11000 |   600 |

Predicate Information (identified by operation id):
   2 - filter("N_1000"=0)
```

Adjusted num_distinct:

```
400 * (1 - power(1 - 1000/1000000, 1000000/400)) = 367.203
  3 * (1 - power(1 - 1000/1000000, 1000000/  3)) =   3.000
```

Distinct = 3 * 367.203 / sqrt(2) = 779

Joins (a)

Join Cardinality: *(joining t1 to t2)* *-- MOS 68992.1 (rewritten)*

```
join selectivity *
filtered cardinality(t1) * filtered cardinality(t2)
```

Join Selectivity: *(for t1.c1 = t2.c2)*

```
((num_rows(t1) - num_nulls(t1.c1)) / num_rows(t1)) *
((num_rows(t2) - num_nulls(t2.c2)) / num_rows(t2)) /
greater(num_distinct(t1.c1), num_distinct(t2.c2))
```

1) Use ***non-join*** predicates to calculate cardinality of ***Cartesian*** join.
2) Use ***non-join*** predicates to adjust "num_distinct" of join predicates
3) Use "single-table" algorithms to calculate the adjusted selectivity of the join predicates on the resulting Cartesian join.

Joins (b)

```
create table t1 as
select
        rownum                                          id,
        mod(rownum-1,200)                               mod_200,
        trunc(dbms_random.value(0,300))                 rand_300,
        mod(rownum-1,10000)                             mod_10000,
        trunc(sysdate) +
                trunc(dbms_random.value(0,1000))        date_1000,
        dbms_random.string('l',6)                       alpha_06,
        dbms_random.string('l',20)                      alpha_20
from
        {large_data_source}
where
        rownum <= 1e6
;
```

Original single-table example with table t2 as a clone of t1

Joins (c)

```
select  *
from
        t1, t2
where
        t1.date_1000 = trunc(sysdate) + 10
and     t2.mod_200   = t1.rand_300
;

| Id  | Operation           | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT    |      |  3448K|   368M|  2466 |
|*  1 |  HASH JOIN          |      |  3448K|   368M|  2466 |
|*  2 |   TABLE ACCESS FULL| T1   |  1000 | 56000 |  1269 |
|   3 |   TABLE ACCESS FULL| T2   |  1000K|    53M|  1127 |

Predicate Information (identified by operation id):
   1 - access("T2"."MOD_200"="T1"."RAND_300")
   2 - filter("T1"."DATE_1000"=TRUNC(SYSDATE@!)+10)
```

Joins (d)

Cartesian Join:

```
T1:        1,000 rows          (date_1000 = constant)
T2:        1,000,000 rows      (no filter predicate)
    Cartesian Join:     1,000,000,000 rows
```

Num_distinct:

```
t2.mod_200        no filtering, so unchanged:       200
t1.rand_300       after filtering to 1,000 rows:    289.3156
    use greater(num_distinct): 289.3156
```

Result:

```
join selectivity = 1/289.3156 (no nulls to consider)
join cardinality = 1e9/289.3156 = 3,456,433
```


Sanity Checks (a)

```
select
        t1.*, t2.*
from
        t1, t2
where
        t1.n_400 = 0
and     t2.n_72  = t1.n_90
and     t2.n_750 = t1.n_600
and     t2.n_400 = 1
;
```

1,000,000 rows in every table
No nulls in any columns

Cartesian Join:

```
T1:             2500 rows       (n_400 = 0)
T2:             2500 rows       (n_400 = 1)
Cartesian Join: 6,250,000 rows
```

```
select
   distinct_formula( 72, 1000000, 2500) dist_72,     --  72
   distinct_formula( 90, 1000000, 2500) dist_90,     --  90
   distinct_formula(750, 1000000, 2500) dist_750,    -- 723
   distinct_formula(600, 1000000, 2500) dist_600     -- 590
from   dual;
```

Sanity Checks (b)

Join Selectivity: *(for t1.c1 = t2.c2)*

```
((num_rows(t1) - num_nulls(t1.c1)) / num_rows(t1)) *
((num_rows(t2) - num_nulls(t2.c2)) / num_rows(t2)) /
greater(num_distinct(t1.c1), num_distinct(t2.c2))
```

```
t2.n_72  = t1.n_90              90 (t1) > 72 (t2)
t2.n_750 = t1.n_600             723 (t2) > 590 (t1)
```

We have seen: selectivity (predicate(A) AND predicate(B)) =
selectivity (predicate(A)) * selectivity (predicate(B))

We might expect the join selectivity to be: `1/90 * 1/723 = 0.000015368`
Which gives a join cardinality of: `96 (selectivity x 6.25M)`

Sanity Checks (c)

```
| Id  | Operation           | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT    |      |  2554 |   139K|  1225 |
|*  1 |  HASH JOIN          |      |  2554 |   139K|  1225 |
|*  2 |   TABLE ACCESS FULL| T1   |  2500 | 70000 |   612 |
|*  3 |   TABLE ACCESS FULL| T2   |  2500 | 70000 |   612 |
```

Multi-column sanity check

The optimizer picks the original selectivities from one side only.
In this case (72 * 750) = (90 * 600) = 54,000,
Oracle would pick the larger ***then*** apply the adjustment algorithm

```
Join Selectivity =
        distinct_formula( 54000, 1000000, 2500) =
        2446.00097

Join Cardinality = 6,250,000 / 2,446 = 2,554
```

Sanity Checks (d)

Index sanity check

If there were a UNIQUE index on (n_72, n_750) and/or (n_90, n_600) the optimizer would use distinct_keys from the index in place of the multiplication.

This has to be a unique index - a unique constraint with a supporting non-unique index does not have the same effect. (I can think of no good reason why not.)

Column Groups

If you had extended statistics on (n_72, n_750) and/or (n_90, n_600) the optimizer would use num_distinct for the column group instead of multiplying.

Joins - argh!

```
select
        t1.*, t2.*, t3.*
from
        t1, t2, t3
where
        t2.n_90  = t1.n_90
and     t3.n_90  = t2.n_90
and     t3.n_600 = t2.n_600
and     t1.n_400 = 1
and     t2.n_400 = 2
and     t3.n_400 = 3
;
```

```
select
        t1.*, t2.*, t3.*
from
        t1, t2, t3
where
        t2.n_90  = t1.n_90
and     t3.n_90  = t1.n_90
and     t3.n_600 = t2.n_600
and     t1.n_400 = 1
and     t2.n_400 = 2
and     t3.n_400 = 3
;
```

Logically these two queries are the same query
So they should give the same plan with the same estimated cardinality.

Joins - argh!

With multi-column sanity check

```
| Id  | Operation           | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT    |      | 70949 |  5820K|  1839 |
|*  1 |  HASH JOIN          |      | 70949 |  5820K|  1839 |
|*  2 |   TABLE ACCESS FULL | T1   |  2500 | 70000 |   612 |
|*  3 |   HASH JOIN         |      |  2554 |   139K|  1225 |
|*  4 |    TABLE ACCESS FULL| T2   |  2500 | 70000 |   612 |
|*  5 |    TABLE ACCESS FULL| T3   |  2500 | 70000 |   612 |
```

No sanity check

```
| Id  | Operation           | Name | Rows  | Bytes | Cost  |
|   0 | SELECT STATEMENT    |      |  3264 |   267K|  1838 |
|*  1 |  HASH JOIN          |      |  3264 |   267K|  1838 |
|*  2 |   TABLE ACCESS FULL | T1   |  2500 | 70000 |   612 |
|*  3 |   HASH JOIN         |      | 10575 |   578K|  1225 |
|*  4 |    TABLE ACCESS FULL| T2   |  2500 | 70000 |   612 |
|*  5 |    TABLE ACCESS FULL| T3   |  2500 | 70000 |   612 |
```


Summary

- Column compare - EQ ignores out of range
- Single distinct - classic probability theory
- Multi-column distinct - odd sqrt(2) factor
- Joins are two-step
  - Cartesian based on filter predicates
  - Column comparison on Cartesian result
  - Various "sanity check" on multi-column joins
- Be careful about changing unique indexes
- Column group stats have a big impact