Basic Selectivity

Jonathan Lewis

jonathanlewis.wordpress.com

www.jlcomp.demon.co.uk

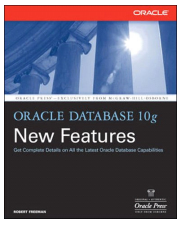
Who am I ?

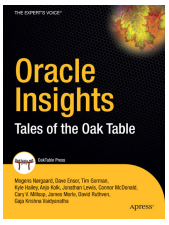
Independent Consultant

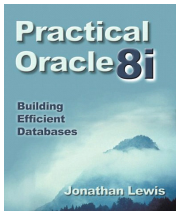
31+ years in IT
26+ using Oracle

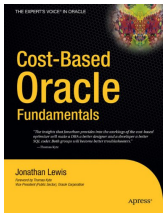
Strategy, Design, Review, Briefings, Educational, Trouble-shooting

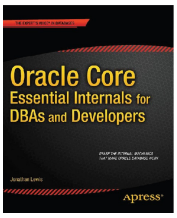
Oracle author of the year 2006
Select Editor's choice 2007
UKOUG Inspiring Presenter 2011
ODTUG 2012 Best Presenter (d/b)
UKOUG Inspiring Presenter 2012
UKOUG Lifetime Award (IPA) 2013
Member of the Oak Table Network
Oracle *ACE Director*
O1 visa for USA











Topics

- Single column predicates
- Effects of functions
- Character columns
- Effects of nulls
- Out of range predicates
- Multi-column predicates
- Strategies for solving problems

Why should you want to know?

Bad selectivity estimates lead to bad cardinality estimates

Bad cardinality estimates contribute to bad execution plans

If you understand why the bad plan appeared you can choose the least worst way to address the problem.

Selectivity

The ***fraction*** of a data set identified by a predicate.
The value will be between 0 and 1

Internally the optimizer uses ***selectivity*** to do its arithmetic, externally it displays the ***cardinality***.

Cardinality: the ***number*** of rows of a data set identified by a predicate or set of predicates.

Cardinality = selectivity * {number of rows in data set}

Warning (a)

```
create table t1 as
select
        rownum                          id,
        mod(rownum-1,200)               mod_200,
        mod(rownum-1,10000)             mod_10000,
        lpad(rownum,50)                 padding
from
        {very_large_rowsource}
where
        rownum <= 1e6
;
begin
        dbms_stats.gather_table_stats(
                ownname          => user,
                tabname          =>'T1',
                method_opt       => 'for all columns size 1'
        );
end;
/
```

Warning (b)

```
create index t1_i1 on t1(mod_200,mod_10000);
select * from t1 where mod_200 = 100 and mod_10000 = 100;
```

```
Execution Plan (10.2.0.5)          Name    Rows  Bytes  Cost (%CPU)  Time     .
| 0 | SELECT STATEMENT             |       |    1 |    63 |  103   (0)| 00:00:01 |
| 1 |  TABLE ACCESS BY INDEX ROWID| T1    |    1 |    63 |  103   (0)| 00:00:01 |
|*2 |   INDEX RANGE SCAN          | T1_I1 |  100 |       |    3   (0)| 00:00:01 |

Predicate Information (identified by operation id):
   2 - access("MOD_200"=100 AND "MOD_10000"=100)
```

```
Execution Plan (11.2.0.4)          Name    Rows  Bytes  Cost (%CPU)  Time     .
| 0 | SELECT STATEMENT             |       |  100 |  6300 |  103   (0)| 00:00:01 |
| 1 |  TABLE ACCESS BY INDEX ROWID| T1    |  100 |  6300 |  103   (0)| 00:00:01 |
|*2 |   INDEX RANGE SCAN          | T1_I1 |  100 |       |    3   (0)| 00:00:01 |

Predicate Information (identified by operation id):
   2 - access("MOD_200"=0 AND "MOD_10000"=0)
```


Stats for Selectivity

- user_tab_columns / **user_tab_cols** / user_tab_col_statistics
  - num_nulls
  - num_distinct, density
  - low_value, high_value
- User_indexes / user_ind_statistics
  - distinct_keys, *num_rows*
- User_tables / user_tab_statistics
  - *num_rows*
- User_tab_histograms
  - endpoint_number
  - endpoint_value
  - *(Endpoint_actual_value)*
  - *(endpoint_repeat_count -- 12c)*

Sample data

```
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
;
```


Single Column Selectivity (SCS)

```
select * from t1 where rand_300  = 150;
select * from t1 where rand_300 != 150;
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |  3333 |   182K|  1117   (6)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |  3333 |   182K|  1117   (6)| 00:00:06 |

Predicate Information (identified by operation id):
    1 - filter("RAND_300"=150)
```

In the absence of histograms (and with no nulls)

for column = {constant}
selectivity = user_tab_cols.density = 1/user_tab_cols.num_distinct

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|*  1 |  TABLE ACCESS FULL| T1   |   996K|    53M|  1137   (8)| 00:00:06 |

   1 - filter("RAND_300"<>150)
```

column **!=** {constant}, selectivity = (1 - user_tab_cols.density)

SCS - functions

```
select * from t1 where sign(mod_10000) = 1;
select * from t1 where trunc(date_1000) != date'2015-Dec-01';
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|*  1 |  TABLE ACCESS FULL| T1   | 10000 |   546K|  1131   (7)| 00:00:06 |

   1 - filter(SIGN("MOD_10000")=1)
```

function(column) = {constant}, selectivity = 1%

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|*  1 |  TABLE ACCESS FULL| T1   | 50000 |  2734K|  1279  (18)| 00:00:07 |

   1 - filter(TRUNC(INTERNAL_FUNCTION("DATE_1000"))<>TO_DATE('
              2015-12-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

function(column) != {constant}, selectivity = 5%

Virtual Columns

```
alter table t1
add (trunc_date generated always as (trunc(date_1000))virtual)
;

begin
   dbms_stats.gather_table_stats(
        user,'t1',method_opt=>'for columns trunc_date size 1'
   );
end;

select * from t1 where trunc(date_1000) != date'2015-12-01';
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |   999K|    61M|  1287  (18)| 00:00:07 |
|*  1 |  TABLE ACCESS FULL| T1   |   999K|    61M|  1287  (18)| 00:00:07 |

Predicate Information (identified by operation id):
    1 - filter("T1"."TRUNC_DATE"<>TO_DATE(' 2015-12-01 00:00:00',
              'syyyy-mm-dd hh24:mi:ss'))
```

SCS - ranges (a)

```
explain plan set statement_id = 'GTLT' for
select * from t1 where mod_10000 > 1200 and mod_10000 < 1800;

explain plan set statement_id = 'GELT' for
select * from t1 where mod_10000 >= 1200 and mod_10000 < 1800;

explain plan set statement_id = 'GELE' for
select * from t1 where mod_10000 >= 1200 and mod_10000 <= 1800;


select
        statement_id, cardinality
from    plan_table
where   operation = 'TABLE ACCESS'
order by
        plan_id
;
```


SCS - ranges (b)

Informal argument

```
   mod_10000                  100 rows per value
   from 1200 to 1800          ca. 600 values
        Expected rows: 600 * 100 = 60,000
```

Selectivity (col between X and Y) is based on: (Y - X) / (high_value - low_value)

STATEMENT_ID	CARDINALITY	
GTLT	60006	>, < (1800-1200)/(9999-0)
GELT	60106	>=, < + 1/num_distinct
GELE	60206	>=, <= + 1/num_distinct

Our intuition is nearly correct - but it's not quite how Oracle reaches the answer.

The optimizer assumes the data is ***continuous***, and ***evenly spread*** between the ***low_value*** and the ***high_value***. As a rough approximation you don't often have to worry much about whether the range is open or closed.

SCS - ranges (c)

```
select * from t1 where mod_10000 > 9100;
select * from t1 where mod_10000 < 100;
```

selectivity (col > X) (high_value - X) / (high_value - low_value) … (+ 1/num_distinct)
selectivity (col < X) (X - low_value) / (high_value - low_value) … (+ 1/num_distinct)

```
select * from t1
where mod_10000 > (select 9100 from dual);

select * from t1
where mod_10000 between (select 9100 from dual)
                and     (select 9101 from dual);
```

These examples get special treatment in 12c (& 11.2.0.4)

selectivity (col > {unknown}) 5% -- no change for closed interval
selectivity (col < {unknown}) 5% -- ditto
selectivity (col between {unknown1} and {unknown2}) 0.25% -- ditto

But with indexes and unknown values the selectivities are 0.9% and 0.45% !!

SCS - index anomaly

```
create index t1_m10000 on t1(mod_10000);
select * from t1 where mod_10000  > (select 9100 from dual)
select * from t1 where mod_10000  between (select … ) and (select … )
```

```
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |           | 50000 |  2734K|  1072   (2)|
|   1 |  TABLE ACCESS BY INDEX ROWID| T1        | 50000 |  2734K|  1070   (2)|
|*  2 |   INDEX RANGE SCAN          | T1_M10000 |  9000 |       |    21   (0)|
|   3 |    FAST DUAL                |           |     1 |       |     2   (0)|
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
    2 - access("MOD_10000"> (SELECT 9100 FROM "SYS"."DUAL" "DUAL"))
```

```
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |           |  2500 |   136K|  1067   (1)|
|   1 |  TABLE ACCESS BY INDEX ROWID| T1        |  2500 |   136K|  1063   (1)|
|*  2 |   INDEX RANGE SCAN          | T1_M10000 |  4500 |       |    12   (0)|
|   3 |    FAST DUAL                |           |     1 |       |     2   (0)|
|   4 |    FAST DUAL                |           |     1 |       |     2   (0)|
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
   2 - access("MOD_10000">= (SELECT 99 FROM "SYS"."DUAL" "DUAL") AND
              "MOD_10000"<= (SELECT 100 FROM "SYS"."DUAL" "DUAL"))
```

SCS - character (a)

```
select * from t1 where alpha_06 like 'mm%';
select * from t1 where alpha_06 >= 'mm' and alpha_06 < 'mn';
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |   157 |  8792 |  1143   (8)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |   157 |  8792 |  1143   (8)| 00:00:06 |

Predicate Information (identified by operation id):
   1 - filter("ALPHA_06" LIKE 'mm%')
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |   157 |  8792 |  1138   (8)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |   157 |  8792 |  1138   (8)| 00:00:06 |

Predicate Information (identified by operation id):
   1 - filter("ALPHA_06"<'mn' AND "ALPHA_06">='mm')
```

The selectivity of "like 'X%'" seems to derived from ">= X and < {first value that is too large}"

SCS - character (b)

```
select * from t1 where alpha_06 like '%mm%';
select * from t1 where alpha_06 not like '%mm%';
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      | 50000 |  2734K|  1143   (8)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   | 50000 |  2734K|  1143   (8)| 00:00:06 |

Predicate Information (identified by operation id):
   1 - filter("ALPHA_06" LIKE '%mm%' AND "ALPHA_06" IS NOT NULL)
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |   950K|    50M|  1151   (9)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |   950K|    50M|  1151   (9)| 00:00:06 |

Predicate Information (identified by operation id):
   1 - filter("ALPHA_06" NOT LIKE '%mm%' AND "ALPHA_06" IS NOT NULL)
```

Like '%x%' uses a 5% "guess"
Not like '%x%' uses a 95% "guess"

-- in 11.1 and earlier this was also 5%
The "is not null" appeared in 11gR2

SCS - char arithmetic (a)

How do you do range arithmetic with characters anyway ?
The clue comes from histograms

```
begin
   dbms_stats.gather_table_stats(
        user,'t1',
        method_opt=>'for columns alpha_20 size 10'
   );
end;


select
        endpoint_number           E_no,
        endpoint_value            E_val,
        to_char(endpoint_value,'XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX') hex_value
from
        user_tab_histograms
where
        table_name = 'T1'
and     column_name = 'ALPHA_20'
/
```


SCS - char arithmetic (b)

E_NO	E_VAL	HEX_VALUE
0	***505,627,989,088,765***,000,000,000,000,000,000,000	616162736A6675391170BAA4200000
	Equivalent	a a b s j f ***U 9 . P*** . . . .
	Actual stored	a a b s j f b d I m s e j j s
1	516,318,405,360,374,***000,000,000,000,000,000,000***	637076796265886F49F6BC2DC00000
2	531,731,684,843,213,000,000,000,000,000,000,000	666865696E765F3CD43C8206200000
3	542,360,142,838,457,000,000,000,000,000,000,000	68746B69697662EF6E7A7621A00000
4	557,633,828,476,402,000,000,000,000,000,000,000	6B65786D6B6671C56A0C8793400000
5	568,382,236,986,073,000,000,000,000,000,000,000	6D77686A6A6666D4B7128BF5A00000
6	583,818,343,141,460,000,000,000,000,000,000,000	707077776B7253799D7A4F0C800000
7	599,292,152,917,341,000,000,000,000,000,000,000	736B626764638C4F000F8400200000
8	609,879,809,669,898,000,000,000,000,000,000,000	7575656B72785C523CD3A992400000
9	625,254,034,284,170,000,000,000,000,000,000,000	786B676A686E77431D16F0C2400000
10	635,944,374,441,818,000,000,000,000,000,000,000	7A7A7A7A706F5CD711C230A4400000

Strings can be viewed as numbers - but have to be "normalised"
Take the first 15 ***bytes*** of the string
(padded with zeros (varchar2) or 0x20 (char) if necessary)
Convert to decimal
Restrict to 15 significant digits
Precision is poor - roughly first 6 characters for a ***single-byte*** character set

SCS - Nulls

As a first approximation, Oracle simply applies the selectivity to the value (user_tables.num_rows - user_tab_cols.num_nulls)

```
select * from t1
where date_1000 between date'2013-12-01' and date'2014-02-28';
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|*  1 |  TABLE ACCESS FULL| T1   | 91089 |  4981K|  1132   (7)| 00:00:06 |
```

We have 1,000 consecutive dates, so high value - low value = 999
28th Feb 14 - 1st Dec 13 = 89; add 1/num_distinct twice for BETWEEN
Selectivity = 89/999+ 2/999 = 0.091091091…

After updating of 2% of the rows to null ***(and gathering stats)***:

```
update t1 set date_1000 = null where mod(id,50) = 0
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|*  1 |  TABLE ACCESS FULL| T1   | 89267 |  4881K|  1132   (7)| 00:00:06 |
```

91089 * 0.98 = 89267.22

SCS - Out of Range (a)

```
select * from t1 where mod_200  = 100;
select * from t1 where mod_200  = 250;       -- out of range
select * from t1 where mod_200  = 350;       -- well out of range
select * from t1 where mod_200 >= 350;       -- not just equality
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |  5000 |   268K|  1113   (6)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   | 5000 |   268K|  1113   (6)| 00:00:06 |

Predicate Information (identified by operation id):
    1 - filter("MOD_200"=100)
```

200 distinct values, selectivity = 1/200, num_rows = 1M
Cardinality = 1M/200 = 5,000

SCS - Out of Range (b)

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|*  1 |  TABLE ACCESS FULL| T1   |  3719 |   199K|  1113   (6)| 00:00:06 |
```

1 – filter("MOD_200"=250)

Value outside known range - estimate uses "linear decay"

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|*  1 |  TABLE ACCESS FULL| T1   |  1206 | 66330 |  1113   (6)| 00:00:06 |
```

1 – filter("MOD_200"=350)

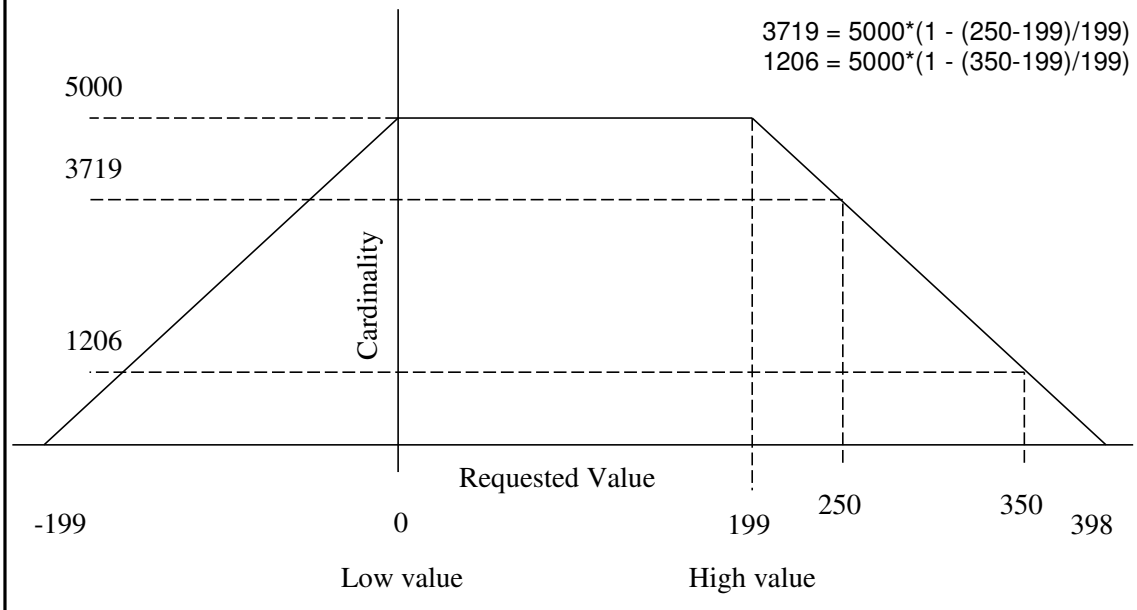
Value further outside known range - increases "linear decay" effect

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|*  1 |  TABLE ACCESS FULL| T1   |  1206 | 66330 |  1113   (6)| 00:00:06 |
```

1 – filter("MOD_200">=350)

Outside known range, range-based predicate treated the same as equality

SCS - Out of Range (c)



Multiple predicates - and

```
select
        *
from    t1
where   mod_200 = 100           -- 1/200, 5000
and     rand_300 = 150          -- 1/300, 3333
;
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |    17 |   935 |  1117   (6)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |    17 |   935 |  1117   (6)| 00:00:06 |

Predicate Information (identified by operation id):
   1 - filter("RAND_300"=150 AND "MOD_200"=100)
```

The result set is 1 in 300 of the 1 in 200 -
Alternatively it is 1 in 200 of the 1 in 300.

Selectivity (p1) AND (p2) = selectivity(p1) * selectivity(p2)
1/200 * 1/300 = 1/60000 = 0.0000166…

Multiple predicates - or

```
select
   *
from    t1
where   mod_200 = 100           -- 1/200, 5000
or      rand_300 = 150          -- 1/300, 3333
;                               -- sum = 8,333
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |  8317 |  446K |  1127   (7)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |  8317 |  446K |  1127   (7)| 00:00:06 |

Predicate Information (identified by operation id):
   1 - filter("MOD_200"=100 OR "RAND_300"=150)
```

If we simply take the sum we have double counted some rows (the overlap)

Selectivity (p1) OR (p2) = selectivity(p1) + selectivity(p2) - selectivity(p1 and p2)
= selectivity(p1) + selectivity(p2) - selectivity(p1) * selecitivyt(p2)

1/200 + 1/300 - 1/60000 = 0.00831666…

Multiple predicate problem

Oracle assumes all columns are independent.

Assume I run a parcel delivery service
I have a 'parcel' table with ***date_col*** (date collected) and ***del_flag*** (delivered)

Query: *"List parcels collected in the last 24 hours but not yet delivered"*

Approx. 1 in 1,000 was collected in the last 24 hours
Approx. 1 in 1,000 has not yet been delivered

Mostly it's the same 1,000 parcels
Oracle thinks that 1 in 1,000,000 match the combined predicate.

Column Group stats (a)

```
select
        *
from    t1
where   mod_200    = 100                -- 1/200, 5000
and     mod_10000 = 100                 -- 1/10000, 100
;

| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |     1 |    57 |  1121   (6)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |     1 |    57 |  1121   (6)| 00:00:06 |

Predicate Information (identified by operation id):
   1  1 - filter("MOD_10000"=100 AND "MOD_200"=100)
```

But whenever mod_10000 = 10 we know that mod_200 will also be 10, so the cardinality estimate should be 100, not 1.

Column Group stats (b)

```
begin
    dbms_stats.gather_table_stats(
        ownname          => user,
        tabname          => 't1',
        method_opt       => 'for columns (mod_200, mod_10000) size 1'
    );
end;
/
```

```
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
|   0 | SELECT STATEMENT   |      |   100 |  5700 |  1121   (6)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| T1   |   100 |  5700 |  1121   (6)| 00:00:06 |

Predicate Information (identified by operation id):
    1 - filter("MOD_10000"=100 AND "MOD_200"=100)
```

Only works for equality - could be useful with a frequency histogram

Bypassed if either predicate goes "out of range"

Maximum of 20 "extended stats" per table

Solution Summary

If the optimizer has used a guess (typically 1%, 5%, or 0.25%) that is most inappropriate then setting *optimizer_dynamic_sampling* to level 3 may help. In some cases it may be best to use level 4. Generally it's better to use a cursor or table-level hint to force sampling.

`http://jonathanlewis.wordpress.com/?s=optimizer_dynamic_sampling`

11g lets you declare (and collect stats on) *"virtual columns"* – which may solve many of the selectivity problems due to predicates on *function(col)*. In harder cases you may find that *"extended stats"* (in particular, *"column groups"*) will help, but you are limited to 20 sets of extended stats per table. There may be side effects in (e.g. IOTs, replication, dbms_redefinition et. al.)

In earlier versions of Oracle, creating function-based indexes and multi-column indexes may address some of the selectivity problems. You could create these indexes as ***unusable*** to avoid maintenance costs - so long as nothing accidentally makes them usable (e.g. a truncate).