Chapter 6: BASICS

In this chapter we pay homage to the basics. Ninety Nine percent of all database performance problems could be prevented before the first line of SQL code is written, by simply doing what we are supposed to do when we build a database. Each week, I see many problems related to basic flaws that arise due to lack of respect for Database Design theory. In any tuning book, a discussion of the value of the basics is always in order. Please let me be very clear of the importance of the basics with this quote:

> *At no time will you have a better opportunity to improve the performance of your database, than in the beginning, by paying attention to the basics of relational database design. No amount of tuning awesomeness can overcome an elemental flaw.*

Lots of Omitted Material

Statistics

Statistics in the context of a Relational Database means A DESCRIPTION OF YOUR DATA. Statistics are a form of METADATA which means data about data. For example, if we have a table that has one-hundred rows in it, we can count those rows and see that there are indeed one-hundred rows. If we then remember this fact by writing it down somewhere, we have a statistic about our table; that it has one-hundred rows. The actual number of rows in this table may change over time, but we will know that at least as of when we last looked, there were one-hundred rows.

This kind of metadata helps the database work better with our data. To understand this consider the simple question of which takes longer, looking at one-hundred rows, or looking at one-hundred thousand rows? My guess is it will take longer to look at one-hundred thousand rows than just one-hundred. But to compare these kinds of work efforts, we need some idea of the number of rows we might be looking at. Hence the need to count them ahead of time.

We can collect any kind of statistic we want and Oracle collects a lot of them. The purpose of these statistics, is to create a description of our data which will help the database work with our data more efficiently. But there has always been two problems with statistics in the Oracle world.

Two common problems in the world of Oracle metadata statistics.

- Statistics has always been and likely will always be an animal of change. Since statistics were first introduced in Oracle 7, the Oracle statistical model has been playing the catch up game. Each version of the database since statistics were introduced offered a statistical model with certain problems. So every new release of the database saw an upgrade to its statistical model which had a feature to address the previous deficiency. At the same time, each new version of the statistical model presented its own new problems. Though continuous improvement is a good thing, significant changes between releases in what statistics could do, made for a moving target when figuring out a statistical gathering strategy that worked without a lot of effort. But there is good news. 11gR2 though continuing the tradition of change, has released some game changing features that help mature its statistical model. So much so that it has become much easier to do statistics maintenance. As a DBA, we want to exploit this and we will discuss this later.
- The bigger problem with statistics though has always been people's understanding of why we need them. It is hard to develop a good strategy for statistics collection and monitoring, and use, when you do not really understand what statistics are doing for you, or the shortcomings of your statistical model that you need to actively overcome.

With this in mind we will first discuss some informational critical mass topics that help us understand Oracle statistics and what we should be looking for when we use them. Then we will head into the newer database versions and how the newer statistical model is mature enough in its offerings to make statistics gathering a much easier proposition with potentially a lot less headaches.

First we will discuss some fundamental aspects of Oracle's statistical model and its behaviors.

Fundamental concepts of statistics.

- NDV (Number of Distinct Values)
- Uniform Distribution of the Data
- Independence of Filter Predicates
- Default NDV and Unaccountable Expressions
- Dynamic Sampling

Then we will discuss where statistics can go wrong.

The most common ways statistics go wrong.

- Staleness
- Skew
- Dependence
- Defaulting
- Out-Of-Bounds
- Transiency
- Bloat

Lastly, to get some practicality from it, we will look at what 11gR2 and 12c offer regarding statistics collection.

Notable improvements in statistics management from 11gR2 and 12c.

- 11gR2 Simplicity
- Dealing with Common Statistics Problem Scenarios
- APPROXIMATE NDV and INCREMENTAL STATISTICS

Fundamentals of Statistics

An understanding of statistics starts with some basic ideas. These ideas influence many aspects of the Oracle statistical model. Let us review some ideas so that we have a common ground to work from.

NDV (Number of Distinct Values)

NDV or Number of Distinct Values refers to the number of different values in a column. Though all statistics gathered by Oracle have a purpose, NDV is arguably the most important of them. It (and the concept it represents) are the driver behind cardinality estimates in query plans. Consider this table.

Notice there are seven (7) rows in this table. Please notice also that the COLOR column has three different values (RED,BLUE,GREEN). This means NDV for column CAR_TABLE.COLOR = three (3). As the example also demonstrates, NULLS are not counted in NDV.

```
CAR TABLE
CAR# COLOR
---- -----
   1 RED
   2 RED
   3 BLUE
   4
```

```
   5 BLUE
   6 GREEN
   7 GREEN
```

In basic optimization, Oracle uses NDV to calculate cardinalities. Consider this query.

```
select * from CAR_TABLE where COLOR = 'RED';
```

How many rows should this query return? The answer of course is two. We can see this if we look at all the rows in the table. More importantly though, this can be computed without looking at all the rows in the table, by doing some simple math with NDV. The math looks like this:

```
(NUMBER OF ROWS WITH NON-NULL VALUES) / NDV =
ESTIMATED CARDINALITY =
(7 rows  -  1 row with null) / 3 =
6/3 = 2
```

Some simple math with NDV tells us that we should expect this query to return two rows. As we can see there are other statistics involved. But the basic concept is actually quite simple. And although there are a few variations on this theme used by the optimizer (like DENSITY and CLUSTERING FACTOR), this is the concept upon which the highly sophisticated statistics model of Oracle works.

Uniform Distribution of the Data

As a concept NDV is simple to grasp. But it relies on some assumptions. One of these is the idea of UNIFORM DISTRIBUTION. Recall from the CAR_TABLE that there were six rows with non-null values. If we examine these rows in more detail we can see that for the three distinct values in the column COLOR, each refers to exactly two rows. Since the number of rows referenced by each distinct value in the column COLOR is the same, the cardinality of column COLOR is considered UNIFORM. This is convenient. In fact it is highly unlikely that in a real scenario, all distinct values of a column will point back to exactly the same number of rows. BUT, Oracle assumes that in fact this is true. Oracle assumes that unless it somehow knows otherwise, all distinct values in a column refer to the same number of rows because the distribution of data of all columns is assumed to be uniform. Of course this assumption can lead to problems.

Consider this variation of the CAR_TABLE. Please notice, there are still only seven rows, and there is still one row that contains a null. And NDV for the column COLOR is still three because it has the same three distinct values. However the distribution of rows with respect to the COLOR column has changed such that it is no longer uniform. Each distinct value refers to a different number of rows with considerable variation. We see for the color RED there are four rows that use this color, whereas for BLUE there is only one row and for GREEN there is only one row. In this situation what would be the number of rows we expect to be returned for the same query we saw before?

```
select * from CAR_TABLE where COLOR = 'RED';
```

The answer is two. Why? Because the math has not changed. Even though we know by looking at all rows in the table that there are four rows which use the color RED, Oracle is not looking at all rows in the table when it evaluates this query (as long as we ignore the use of dynamic sampling anyway). Since Oracle is not looking at the table it will be using the NDV to calculate expected number of rows and the math is still:

```
(NUMBER OF ROWS WITH NON-NULL VALUES) / NDV =
ESTIMATED CARDINALITY =
(7 rows  -  1 row with null) / 3 =
6/3 = 2

CAR TABLE
CAR# COLOR
---- -----
   1 RED
   2 RED
   3 RED
   4
   5 RED
   6 BLUE
   7 GREEN
```

The query will of course return four rows and not the two we estimated using NDV. However this error in estimation brings out a good point. Statistics are an approximation of the truth. The Oracle statistical model uses this approximation of the truth only to calculate an estimated cardinality for situations like we see above such that the estimate is GOOD ENOUGH to produce a good query plan. Oracle does not have to estimate exact row counts in order to generate a good query plan which is a good thing since as we can clearly see, in a real world it almost never will estimate exact cardinalities using NDV and UNIFORM DISTRIBUTION. Fortunately for us, most of our data will come close enough to meeting the assumption of uniform distribution so as not to interfere with cardinality calculations.

Independence of Filter Predicates

Another assumption made by Oracle is INDEPENDENCE OF FILTER PREDICATES. What this means is that each test against the data that filters out rows from a result set is assumed to do so without regard to any other filter predicates that may be filtering rows out of the result set. This is to say that each column in a table is assumed to be not related to (aka. independent of) any other column in the same table. This is a reasonable assumption since if at some point during our database design process we referred to a third normal form version of our data model, one of the rules of normalization is that each column of a table must be independent of all the other non-key columns in the same table.

Expanding upon our CAR_TABLE we can see what this means to cardinality estimates of a query. Just as with the assumption of uniform distribution, sometimes the assumption of independence fails. It may be that our database design was not fully third normal form, or more commonly that even though a set of columns are independent, this independence is not true for subsets of rows in a table.

This version of the CAR_TABLE offers a slightly more complex example. We see a car table with three columns instead of just one. These columns are COLOR, SIZE , DOORS. We can see also the number of distinct values in each column and thus the NDV for each column. Please notice though that there are a couple of anomalies in this data.

For the color GREEN, all three attributes are the same unlike the other two colors. For the subset of rows identified by the color GREEN the three columns are not independent of each other. Looking at the colors RED and BLUE we see that there is no clear correlation between (COLOR and SIZE) or (COLOR and DOORS). But as we see, this is not true for the color GREEN.

Additionally we see another anomaly of the data between the columns SIZE and DOORS. The SIZE of a car seems to correlate pretty well with the number of doors the car has. For example, for this dataset, all COMPACT cars have two doors, whereas all MIDSIZE cars have four doors and all LUXURY cars have four doors. From this we can say that the columns SIZE and DOORS are not independent. Keep in mind that when estimating cardinalities, the optimizer is assuming that all columns are independent and the data here clearly violates this rule in at least two situations.

```
CAR TABLE                          DISCTINCT VALUES IN COLUMNS
CAR# COLOR SIZE    DOORS           COLOR SIZE    DOORS
---- ----- ------- -----           ----- ------- -----
   1 RED   COMPACT     2          RED   COMPACT     2
   2 RED   MIDSIZE     4          BLUD  MIDSIZE     4
   3 RED   LUXURY      4          GREEN LUXURY
   4 RED   COMPACT     2
   5 RED   MIDSIZE     4
   6 BLUE  LUXURY      4              3       3     2   COLUMN NDV
   7 BLUE  COMPACT     2
   8 BLUE  MIDSIZE     4
   9 BLUE  LUXURY      4
  10 BLUE  COMPACT     2
  11 GREEN LUXURY      4
  12 GREEN LUXURY      4
  13 GREEN LUXURY      4
  14 GREEN LUXURY      4
  15 GREEN LUXURY      4
```

So what does this mean for more involved queries that use more than one column in filter predicates?

```
select *
from CAR_TABLE
where COLOR = 'RED'
and SIZE = 'MIDSIZE';
```

We see here two filter predicates. Each filter predicate will remove rows from the result set. Since each column is assumed to be independent of other columns, each filter predicate can also be assumed to be independent of other filter predicates. That means in order to compute the combined effect of both filter predicates, we can use the same simple NDV math we used before and then combine results. It works like this.

```
where COLOR = 'RED'

(NUMBER OF ROWS WITH NON-NULL VALUES) / NDV =
ESTIMATED CARDINALITY =
(15 rows  -  0 row with null) / 3 =
15/3 = 5 = 33%
```

COLOR = RED will return 5 rows out of 15 or 5/15 of the data or 33% of the rows

```
and SIZE = 'MIDSIZE';

(NUMBER OF ROWS WITH NON-NULL VALUES) / NDV =
ESTIMATED CARDINALITY =
(15 rows  -  0 row with null) / 3 =
15/3 = 5 = 33%
```

SIZE = MIDSIZE will return 5 rows out of 15 or 5/15 of the data or 33% of the rows.

So how many rows will the combination of the two filter predicates return? Since the two filter predicates are assumed to be independent, we can use simple math to combine the two. We just multiply the two results together. Using NDV we computed that COLOR = RED should return about 33% of the data by itself and that SIZE = MIDSIZE should return about 33% of the data by itself. Together 33% * 33% = 11%. If we multiply this by 15 which is the number of rows in the table with non-null values for the columns we are using, we get 15*33%*33%=1.67. Since we cannot return a fraction of a row we round up to 2. Thus we expect this query to return about two rows.

```
ceil( 1/NDV(COLOR) * 1/NDV(SIZE) * (NUMBER OF ROWS WITH NON-NULL VALUES) ) =
CARDINALITY =
ceil( 1/3 * 1/3 * 15 ) =
ceil( 33% * 33% * 15 ) =
ceil( 11% * 15 ) =
ceil( 1.67 ) = 2
```

The point is that we calculated a combined percentage of returned rows for both predicates in this query, by multiplying the 1/NDV of each column used as dictated by each filter predicate. Since the NDV of COLOR is 3, each value from this column will return one row out of every three rows in the table or 1/3 of the rows or 33% of the rows in the table. Since the NDV of SIZE is 3, each value from this column will return one row out of every three rows in the table or 1/3 of the rows or 33% of the rows in the table. Since these two filter predicates are assumed to be independent of each other, the combined expected calculation from using both is 33% * 33% or 1/3 * 1/3 or 1/9 or one row out of every nine rows or 11% of the rows in the table. Since there are 15 rows in the table where neither COLOR nor SIZE has a null vales, 11%*15 = 1.67 rows or about 2 rows.

For any query, this math continues. Consider this query with three filter predicates.

```
select *
from CAR_TABLE
where COLOR = 'GREEN'
and SIZE = 'LUXURY'
and DOORS = 4;

NDV(COLOR) = 3 → 1/3 → 33%
NDV(SIZE)  = 3 → 1/3 → 33%
NDV(DOORS) = 2 → 1/2 → 50%
1/3 * 1/3 * 1/2 = 1/18 = 5.5%
5.5% * 15 = .83 rows.  Rounded up = 1 row.
```

We expect this query to return about one row. Of course if we examine the table we know that this query will in fact return five rows not one. This is because the subset of rows we want does not conform to the assumption of independence of filter predicates. This example demonstrates two things.

Math of NDV and importance of INDEPENDENCE.

- The process of combining filter predicates is pretty simple, just do the basic multiplication math with 1/NDV for AND predicates or addition math for OR predicates.
- When the data in a table does not meet the assumption of INDEPENDENCE OF FILTER PREDICATES, cardinality estimates that result from this simple math will be wrong.

Fortunately for us, most of our data will come close enough to meeting the assumption of independence so as not to interfere with cardinality calculations.

Default NDV and Unaccountable Expressions

So far we have seen how the optimizer deals with statistics when it has them. But when the optimizer has no statistics or cannot use those it has, what does it do? The short answer is: the optimizer plays JAZZ or fakes it. Start on the right note, end on the right note, and do whatever you want in the middle.

Without any statistics, the optimizer still needs a value for NDV in order to do its calculations. So… it uses 100 for NDV. Why 100? Who knows. I recall a Doctorate of Applied Math many years ago who developed formulae related to percentages of rows given uniform distributions of data in which this gentleman determined that the 2% RULE (as we call it) was really 1% which in a way equates to NDV=100. Maybe Oracle geniuses did something similar. In any event, we can see the NDV=100 with a very simple example.

In this example we create a little table with no rows and no statistics and then we lie to the optimizer about how many rows the table has. We will tell the optimizer that this table has some specific number of rows by manually setting the NUM_ROWS statistic for the table and then ask the optimizer to construct a query plan for our sample query. From this we will learn that default NDV is 100 (for a simple equality predicate anyway) which yields 1/100 rows or 1 out of every 100 rows or 1% of the rows in the table.

```
create table kevtemp1
(a integer not null,b integer not null,c integer not null);

exec dbms_stats.set_table_stats(user,'KEVTEMP1',NUM_ROWS=>100)

explain plan for
select /*+ dynamic_sampling(a 0) */ * from kevtemp1 where a = 1;

------------------------------------------------------------
| Id  | Operation                 | Name     | Rows  | Bytes |
------------------------------------------------------------
|   0 | SELECT STATEMENT          |          |     1 |    13 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |     1 |    13 |
------------------------------------------------------------

exec dbms_stats.set_table_stats(user,'KEVTEMP1',NUM_ROWS=>1000)

explain plan for
select /*+ dynamic_sampling(a 0) */ * from kevtemp1 where a = 1;

------------------------------------------------------------
| Id  | Operation                 | Name     | Rows  | Bytes |
------------------------------------------------------------
|   0 | SELECT STATEMENT          |          |    10 |   130 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |    10 |   130 |
------------------------------------------------------------

exec dbms_stats.set_table_stats(user,'KEVTEMP1',NUM_ROWS=>10000

explain plan for
select /*+ dynamic_sampling(a 0) */ * from kevtemp1 where a = 1;

------------------------------------------------------------
| Id  | Operation                 | Name     | Rows  | Bytes |
------------------------------------------------------------
|   0 | SELECT STATEMENT          |          |   100 |  1300 |
```

```
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |   100 |  1300 |
-------------------------------------------------------------
```

These query plans have been abbreviated to fit on the page better. Note also that the query turns off dynamic sampling for the table it is querying so that we can see the default NDV in action.

If it is true that when the optimizer cannot determine a suitable NDV by any other method it will use NDV=100, then if we tell the optimizer that the table it is using has 100 rows, the sample query we see here should produce a query plan with a cardinality estimate of 1 row. And in fact that is what we see in the first query plan. Further as we scale up the number of rows, the cardinality estimate scales up the same. So when rows in the table is made to look like 1000, NDV=100 results in a cardinality estimate of 10 rows. When rows in the table is made to look like 10,000, NDV=100 results in a cardinality estimate of 100 rows. Remember that NDV is just for the one column we are checking in this one predicate. If there are more predicates in the query the optimizer will also combine NDV percentages to get a final cardinality estimate.

Here we set NUM_ROWS to one million. Then we run a query with two predicates for which there are no column statistics. If the simple math to be used is as we described above, then this query will want 1% of 1% of the data or 1/100 * 1/100 or 1/10,000 rows. One million * 1/10,000 = 100. And that is what we see.

```
exec dbms_stats.set_table_stats(user,'KEVTEMP1',NUM_ROWS=>1000000)

explain plan for
select /*+ dynamic_sampling(a 0) */ * from kevtemp1
where a = 1
and b = 1;

-------------------------------------------------------------
| Id  | Operation                 | Name     | Rows  | Bytes |
-------------------------------------------------------------
|   0 | SELECT STATEMENT          |          |   100 |  3900 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |   100 |  3900 |
-------------------------------------------------------------

NDV(A) = 100 = 1/100 = 1%
NDV(B) = 100 = 1/100 = 1%
NUM_ROWS = 1000000
1/100 * 1/100 * 1000000 = 100
```

This example though is pretty extreme. So extreme that I would never expect to see it on an 11gR2 or higher database as was used by me to generate these plans. Normally on 11gR2 and higher, in a situation where a table has no statistics, dynamic sampling will kick in, in order get the optimizer some real statistics. But as was noted, dynamic sampling was turned off for these tests.

However, there is a common situation where NO USABLE STATISTICS can happen. It is our old friend, known as COLUMN MODIFICATION. Consider these queries and their query plans. Notice how they each involve the use of some expression. The important thing is that the optimizer has not collected statistics on any of these expressions, only on the underlying columns. The statistics it has on the underlying columns are of no use to it in evaluating these expressions so the optimizer once again uses its default NDV=100. We will use a row count of 1000 for our next three plans to demonstrate.

```
exec dbms_stats.set_table_stats(user,'KEVTEMP1',NUM_ROWS=>1000)

explain plan for
select /*+ dynamic_sampling(a 0) */ * from kevtemp1 where a+b = 1;
-------------------------------------------------------------
| Id  | Operation                 | Name     | Rows  | Bytes |
-------------------------------------------------------------
|   0 | SELECT STATEMENT          |          |    10 |   390 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |    10 |   390 |
-------------------------------------------------------------

explain plan for
select /*+ dynamic_sampling(a 0) */ * from kevtemp1 where a||b = 'a';
-------------------------------------------------------------
| Id  | Operation                 | Name     | Rows  | Bytes |
```

```
-------------------------------------------------------------
|   0 | SELECT STATEMENT          |          |    10 |   390 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |    10 |   390 |
-------------------------------------------------------------

explain plan for
select /*+ dynamic_sampling(a 0) */ * from kevtemp1 where substr(a,2,3) = '123';
 -------------------------------------------------------------
| Id  | Operation                 | Name     | Rows  | Bytes |
-------------------------------------------------------------
|   0 | SELECT STATEMENT          |          |    10 |   390 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |    10 |   390 |
-------------------------------------------------------------
```

So we see that when in doubt, Oracle will use default statistical percentages. We have only presented a simple case here, all equality predicates. The fact is that for differing situations, Oracle will use different percentages (usually 1% or 5%). In either case however, the default is likely not correct. Fortunately there is an alternative in today's Oracle called Dynamic Sampling.

Dynamic Sampling

Dynamic Sampling is the attempt by the optimizer to acquire statistics by reading a sample of rows from a table on the fly. The purpose of dynamic sampling is to try and estimate cardinalities of table accesses and joins. When the optimizer has no statistics for a table, or cannot use those that it does have, or for any number of other reasons decides it just feels like sampling your data, it will make an attempt to acquire statistics by reading a sample of rows.

Consider the example we just discussed of a table with no statistics on it. At query plan generation time, the optimizer will recognize this condition of missing statistics and BEFORE it generates a query plan, will invoke dynamic sampling. The dynamic sampling process will read a sample of rows from the table and through various means, compute a cardinality estimate for each table it samples. It will then give these cardinality estimates back to the optimizer. The optimizer will use these cardinality estimates in generating a query plan. Dynamic Sampling is a great feature when used correctly. However dynamic sampling can be overused, incorrectly used, and is a seemingly unpredictable feature. Let us consider some insights about it.

Dynamic Sampling is NOT a substitute for normal statistics gathering for most tables. Dynamic Sampling does not gather statistics in the normal sense, nor does it update the data dictionary with what it finds (at least not yet). It is only interested in computing cardinality estimates from an evaluation of predicate selectivity, for the query being optimized.

Also, under most conditions, Dynamic Sampling samples a limited number of blocks from a table. This sampling can be rather small. Typically the number of blocks sampled is 32 or 64. Not surprisingly then, the value of dynamic sampling can be limited due to its limited sample size.

Dynamic Sampling is very dynamic. There are so many factors that can influence this feature that it is difficult to know how even Oracle keeps it all straight; so much so that it almost seems alive. The behavior of dynamic sampling will change depending upon a plethora of conditions like, what your query looks like, whether your query uses parallel mode or executes serially, how dynamic sampling initially gets invoked, what kind of statistics already exist on a table being considered for sampling, and even if SQL profiles are in use, and of course any bug-fix patches you have installed. Though it is certainly not random it can seem so. I for one would love very much to see the detailed decision tree for how it works (not the high level tree given in Oracle documentation).

Dynamic Sampling is so sophisticated and changes so often, I am unable to give it a full treatment of the subject. It would take hundreds of pages to do it justice; a book in its own right. But I would not even try that since everyone who attempts it gets its wrong anyway. Even the experts I respect greatly from Oracle Corp. make mistakes when they talk about it. And of course I would never claim to know it all. The good news for us is that Dynamic Sampling is one of those 99%/1% features. 99% of what we want to get out of it we can get by understanding about 1% of the details. So that is what I am going to concentrate on, the necessary 1% of the details needed to use dynamic sampling appropriately. And like most things I believe the way to understanding is in the WHY. There are at least three scenarios where dynamic sampling has the potential to make a very big and positive difference. The first is obvious, the other two being not so well known.

Three situations where dynamic sampling can have a positive impact.

- A table has no statistics at all
- The optimizer is guessing about predicate selectivity

- Parallel query is being invoked for one or more large tables

A table has no statistics at all

The scenario of a table with no statistics really brings forward the purpose of dynamic sampling and how it works. Consider this example.

```
create table kevtemp1 (adate date not null) nologging;
insert into kevtemp1 values (to_date('01-sep-2013','dd-mon-rrrr'));
insert into kevtemp1 select * from kevtemp1;
insert into kevtemp1 select * from kevtemp1;
commit;
```

This is a simple table with one date column and four rows all of which sport the same date of 01-SEP-2013.

```
NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
optimizer_dynamic_sampling           integer     2
optimizer_features_enable            string      11.2.0.2
```

Our system is an 11gR2 database with dynamic sampling set to 2. Levels can be from 0 to 10 (11 if you are on 12c (actually I believe this level is available as of 11.2.0.4)). Consult Chapter 5: HINTS for a fuller descriptions of these levels. Level 2 dynamic sampling simply means that the optimizer is allowed to collect statistics for tables that have no statistics. This is the default setting and is what most databases use and what most people expect when they think of dynamic sampling; though as I look around, most databases I support do not use this setting…hmm. Also, we will see 11g interprets a setting of level=2 differently for parallel queries.

So how would the database optimize this statement?

```
select * from kevtemp1 where adate = to_date('01-sep-2013','dd-mon-rrrr');
```

We know that his query should return four rows because we just loaded this table with four rows that have this date. But there are no statistics for the optimizer to work with because we have not collected any for this table. So how will the optimizer compute a correct cardinality for the table access this query intends to do?

```
00:18:19 SQL> @showcolstats kmuser kevtemp1

no rows selected

00:18:33 SQL> select NUM_ROWS from user_tables where table_name = 'KEVTEMP1';

  NUM_ROWS
----------

1 row selected.
```

The table clearly has neither column level statistics nor table level statistics.

```
------------------------------------------------------------
| Id  | Operation                 | Name     | Rows  | Bytes |
------------------------------------------------------------
|   0 | SELECT STATEMENT          |          |     4 |    36 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |     4 |    36 |
------------------------------------------------------------

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

   1 - storage("ADATE"=TO_DATE(' 2013-09-01 00:00:00', 'syyyy-mm-dd
              hh24:mi:ss'))
       filter("ADATE"=TO_DATE(' 2013-09-01 00:00:00', 'syyyy-mm-dd
              hh24:mi:ss'))
```

Yet the query plan clearly shows the optimizer expecting to find four rows. How does it know this? The answer is dynamic sampling.

The primary purpose of dynamic sampling is to try and estimate cardinalities of table accesses. To do this dynamic sampling will issue one or more recursive SQL statements to the database that will read a sample of rows from the table and count these rows. For this query, this is the recursive SQL that was executed. At first glance it looks a little scary but it is not really.

```
SELECT /* OPT_DYN_SAMP */
/*+
    ALL_ROWS
    IGNORE_WHERE_CLAUSE
    NO_PARALLEL(SAMPLESUB) opt_param('parallel_execution_enabled', 'false')
    NO_PARALLEL_INDEX(SAMPLESUB)
    NO_SQL_TUNE
*/
     Nvl(SUM(c1), :"SYS_B_0"),
     Nvl(SUM(c2), :"SYS_B_1")
FROM   (SELECT
              /*+
                   IGNORE_WHERE_CLAUSE
                   NO_PARALLEL("KEVTEMP1")
                   FULL("KEVTEMP1")
                   NO_PARALLEL_INDEX("KEVTEMP1")
              */
    :"SYS_B_2" AS C1,
    CASE
    WHEN "kevtemp1"."adate" = To_date(:"SYS_B_3", :"SYS_B_4") THEN :"SYS_B_5"
    ELSE :"SYS_B_6"
    END AS C2
        FROM   "KMUSER"."kevtemp1" "KEVTEMP1") SAMPLESUB
```

This recursive SQL query is issued on behalf of your query by the optimizer BEFORE the optimizer generates a query plan. The important pieces are the CASE expression aliased as C2, and the associated SUM expression. We can see that the WHERE clause of our query and the PREDICATE INFORMATION section of our query plan, and the CASE EXPRESSION in the recursive SQL statement all line up. The same filter predicate is being used in different ways.

```
where adate = to_date('01-sep-2013','dd-mon-rrrr')

1 - storage("ADATE"=TO_DATE(' 2013-09-01 00:00:00', 'syyyy-mm-dd
           hh24:mi:ss'))
    filter("ADATE"=TO_DATE(' 2013-09-01 00:00:00', 'syyyy-mm-dd
           hh24:mi:ss'))

CASE
   WHEN "kevtemp1"."adate" = To_date(:"SYS_B_3", :"SYS_B_4") THEN :"SYS_B_5"
   ELSE :"SYS_B_6"
END AS C2
```

If we plug in values for the variables in the CASE expression and NVL expression, it might make more sense.

```
     Nvl(SUM(c2), 0)

CASE
   WHEN "kevtemp1"."adate" = to_date('01-sep-2013','dd-mon-rrrr') THEN 1
   ELSE 0
END AS C2
```

It should be clear now that the filter predicates applied to the table were taken from the WHERE clause and converted to a CASE expression that returns 1 if a row meets the conditions and 0 if it does not. Summing up the result of the case expression gives the number of rows in the sample that meet the filter predicates.

The dynamic sampling query reads a sample of the table. Then it evaluates the filter predicates via a CASE EXPRESSION to determine how many rows from the sample satisfy these filter predicates. This provides a cardinality for the sample which can then be scaled up or down as the dynamic sampling feature sees fit, to provide a cardinality estimate for the table access. Dynamic sampling then provides this cardinality estimate to the optimizer so that the optimizer can use this newly obtained information to generate a query plan.

If we have good statistics on a table, and a simple query like this one, then certainly we do not want to be doing dynamic sampling. But for a table without any statistics, dynamic sampling is an excellent feature for optimizing a query. There are tables which legitimately may not have statistics. These include GLOBAL TEMPORARY tables, WORK tables, and STAGING tables. These are situations where rows are of a transient nature and for such tables, dynamic sampling is one possible solution to performance issues that might arise from a lack of statistics.

The optimizer is guessing about predicate selectivity

Of course, most of the Oracle world is not so simple. Our tables are not simple tables such as we just saw, and our queries are not simple one table one predicate queries. The complexity of SQL in fact, can create situations where the optimizer must guess about predicate selectivity even when statistics do exist on a table. There are two common scenarios for this:

Two commons situations where the optimizer must guess about statistics.

- Expressions
- Complex Predicates

Consider these queries. For each of these queries, the optimizer must make a guess as to the selectivity of the predicates involved.

```
select * from T where substr(astring,3,2) = 'CT';
select * from T where anum1 = 1 or anum2 = 2 or anum3 = 3;
```

Consider that for the first query, even if statistics have been collected on the columns of the table, statistics have not been collected for the expression SUBSTR (ASTRING, 3, 2). So Oracle has no clue how selective this predicate is. Dynamic Sampling can do the same thing with this predicate that we saw it do in our prior example.

Consider that for the second query, although we have statistics on each of the three columns, this query has a COMPLEX PREDICATE meaning its predicate is based on multiple columns of the table. This complex predicate brings with it the possibility of DEPENDENCE between columns affecting cardinality estimates. Once again Dynamic Sampling can do its magic and examine the data directly to understand the filtering capability of these combined predicates and thus eliminate any effect of dependence between columns.

Indeed no matter how complex the filter predicate is for a table, dynamic sampling can convert it to a CASE expression and do its magic to compute a cardinality estimate. For completeness, these two statements represent the next two levels of dynamic sampling sophistication. To get dynamic sampling to recognize the expression predicate, we need set OPTIMIZER_DYNAMIC_SAMPLING=3. To get dynamic sampling to recognize the complex predicate (multi-column predicate) we need to set OPTIMIZER_DYNAMIC_SAMPLING=4.

Parallel Query is being invoked for one or more large tables

This is new to 11gR2. In this release, if a query is running in parallel, the database assumes that the query is going to take a long time. After all, what is parallel query for if not to make really big queries get done in a shorter period of time? Since queries running in parallel are expected by the database to take a long time, the database figures dynamic sampling would be a good idea. If the database is using OPTIMIZER_DYNAMIC_SAMPLING=2 (the default) then instead of leaving things alone, the database will control the sampling process for parallel queries. But the kind of sampling it does is a bit different. Notably, the database will decide how big a sample to take and it will scale its sample size up with the size of the tables being sampled with possibly some much larger sample sizes. So really large tables will get much bigger samples. The basic theory is that the time spent doing this sampling is negligible compared to the expected long runtime of the query so having the advantage of the sampled data will help avoid mistakes that could turn the query into a never ending process. Ideally extra sampling that is done will produce a net savings in total runtime by generating much better plans or by at least avoiding devastatingly bad plans.

Personally I do not have much experience with this one. The databases I support mostly use OPTIMIZER_DYNAMIC_SAMPLING=4. I am not suggesting your databases should use this setting, only pointing out that a value of 4 disables this parallel query quirk of dynamic sampling and so I do not have much to say about Dynamic Sampling invoked due to Parallel Query, since I have do not deal with it on a daily basis. What I state here is pretty much a restatement of the Oracle documentation.

Dynamic Sampling and the future

Dynamic Sampling is not going away so we better get used to it. In fact, in Oracle 12c, Oracle again makes significant changes. The trends are clear. Oracle is very committed to the dynamic sampling mechanism. And Oracle continues to increase the autonomy of the database to decide for itself the best way to apply this feature.

Where statistics can go wrong

We now explore the most common situations that arise when statistics are not doing their job and how you might fix them. Although the Oracle optimizer is very good at its job, it still has limitations. Fortunately Oracle offers some new solutions for a few of these limitations in 11gR2 and 12c. Additionally, all but the last of the items we will review can be addressed by Dynamic Sampling, with varying degrees of success.

Staleness

Staleness means statistics that no longer provide a fair description of the objects upon which they were collected. Statistics are static. Once collected they do not change until we collect them again. The more our data changes, the more likely our statistics no longer provide a fair representation of the data. Staleness is a problem because the farther away our approximation of reality is to true reality of the data, the more divergent the cardinality estimates generated by the optimizer will become. Thus over time it becomes necessary to re-collect statistics. Ideally we do this before staleness reaches a point that it negatively affects cardinality estimates in query plans. There are a few ways I know of to determine if statistics are stale and warrant a possible re-collection event.

The new way is to use DBA_TAB_MODIFICATIONS introduced in 11gR2. This view records changes to tables. There are several types of changes it monitors including number of inserts, updates, and deletes. There is no guarantee that these numbers are exact, and there is a delay from when a change occurs and when it might show up recorded here, and the change modification feature must be turned on for this recording to happen, and the monitor does not appear to compare old and new values in an update so changing a value to itself still records as an update (something to think about in terms of generic code).

Using this information, some simple math of the total number of changes compared to number of rows in the table can yield a gross percentage change figure that can guide us in determining if statistics need re-collection. Oracle by default uses 10% as the figure for its automatic statistics gathering process. So one way to deal with staleness of statistics is to use the modifications view and act accordingly.

```
11:46:28 SQL> desc dba_tab_modifications
 Name                          Null?    Type
 ----------------------------- -------- ------------------
 TABLE_OWNER                            VARCHAR2(30)
 TABLE_NAME                             VARCHAR2(30)
 PARTITION_NAME                         VARCHAR2(30)
 SUBPARTITION_NAME                      VARCHAR2(30)
 INSERTS                                NUMBER
 UPDATES                                NUMBER
 DELETES                                NUMBER
 TIMESTAMP                              DATE
 TRUNCATED                              VARCHAR2(3 CHAR)
 DROP_SEGMENTS                          NUMBER
```

Something like this perhaps. Looks like some staleness potential here.

```
  1  select a.inserts,a.updates,a.deletes,b.NUM_ROWS,a.table_name
  2        ,round((a.inserts+a.updates+a.deletes)/b.NUM_ROWS*100) pct_change
  3  from dba_tab_modifications a
  4      ,dba_tables b
  5  where b.owner = a.table_owner
  6  and b.table_name = a.table_name
  7  and b.owner = 'EDW_SHR'
  8  and inserts > 1000
  9  and NUM_ROWS > 0
 10* and rownum < 5
12:14:33 SQL> /
                                                                          %
   INSERTS    UPDATES    DELETES   NUM_ROWS TABLE_NAME                Change
---------- ---------- ---------- ---------- ---------------------- ----------
```

```
      1802        345       2483      43520 BATCH_HISTORY                  11
      9747       2267       6746      40405 ENTITY_MAPPING                 46
      8830          0          0      12500 ENTITY_PROCESS_STATUS          71
      2916         69       2896       3424 TBL_MSTR                      172

4 rows selected.
```

A second less precise way to check statistics, is to look at them. We can compare NUM_ROWS to the true row count in the table or use the SHOWCOLSTATS <owner> <table> script you will find on the web set for this book. This method requires you to know your data enough to tell if something looks wrong. As such it is prone to inaccuracy but it is a valuable thing to do in order to get you familiar with statistics on objects you care about.

A third way to judge the staleness of statistics is to evaluate the quality of cardinality estimates they yield in query plans. Since the purpose of statistics is to generate good plans and since good plans rely heavily on the quality of cardinality estimates, we can use the closeness of estimated to actual values as a way to judge how good a job our statistics are doing and thus if staleness might be a problem. We do this by comparing estimates to actuals to see how wide a difference there is between what the optimizer expected and what actually happened. Consider this example obtained using the GATHER_PLAN_STATISTICS hint.

```
select /*+ gather_plan_statistics */ count(*) from kevtemp1 where c1 > 100;
select * FROM table(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));
-------------------------------------------------------------------
| Id  | Operation                | Name     | E-Rows | A-Rows |
-------------------------------------------------------------------
|   0 | SELECT STATEMENT         |          |     10 |    390 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |     10 |    390 |
-------------------------------------------------------------------
```

Or the alternative old school method.

```
explain plan for select count(*) from kevtemp1 where c1 > 100;
select * from table(dbms_xplan.display('PLAN_TABLE',NULL,'ADVANCED'));
-----------------------------------------------------
| Id  | Operation                | Name     | Rows |
-----------------------------------------------------
|   0 | SELECT STATEMENT         |          |   10 |
|*  1 |  TABLE ACCESS STORAGE FULL| KEVTEMP1 |   10 |
-----------------------------------------------------
12:44:02 SQL> select count(*) from kevtemp1 where c1 > 100;

  COUNT(*)
----------
       396

1 row selected.
```

With either check, we can see that by a wide margin, the estimated row count does not reflect the actual row count given the filters applied to the table. How much off they are before you get worried is of some debate, but surely a difference of an order of magnitude will result in errors in query plans. If estimates like this are commonly off then staleness may very well be the culprit. In any event additional investigation given cardinality estimates off by an order of magnitude, is in order.

To fix staleness in statistics, re-collect statistics. Or better yet, turn on Oracle's automatic statistics gathering process and let it do the work of monitoring and correcting staleness for you.

Skew

Skew means data that is not evenly distributed (at least that is going to be our definition of it). We saw this before so let us look at the example again.

```
CAR TABLE #1       CAR TABLE #2
CAR# COLOR         CAR# COLOR
   1 RED              1 RED
   2 RED              2 RED
```

```
   3 BLUE              3 RED
   4                   4
   5 BLUE              5 RED
   6 GREEN             6 BLUE
   7 GREEN             7 GREEN
```

Recall please, the CAR_TABLE from previous examples and these two different data sets we used with it. The first data set (left) shows data that is evenly distributed. Indeed on the left table each value in the COLOR column appears on exactly two rows. The second data set (right) shows SKEW. One or more values has a larger or smaller number of rows it is found on, compared to the expected average of two (2) calculated using NDV.

SKEW is a problem because Oracle assumes by default that values in a column are uniformly distributed across rows in a table and thus have no skew. The optimizer generates cardinality estimates using math that relies on this assumption of uniformity of distribution. It is common for some values in a column to be skewed while others are not. This is one of the problems with bind variable peeking (not discussed in this book) and often results in query plans that are good for some query executions but not for other executions of the same query.

SKEW is a statistically complicated topic. This is because skew has multiple meanings for different situations. For example, statistically we might say that UNIFORM DISTRIBUTION does not mean evenly distributed, but rather that the distribution we see in our data matches a distribution we can statistically equate to it. For example a NORMAL DISTRIBUTION is certainly not evenly distributed. But a data distribution that matches a NORMAL DISTRIBUTION would not be skewed with respect to the statistical concept of a NORMAL DISTRIBUTION even though none of the data points in the distribution would point to the same number of measurements. In this sense, skew refers to sameness or symmetry. I apologize to the statistical experts for my mangling of the topic but we are database people looking for practical uses of the concept.

In this book though, we will be using only a simple explanation of Uniform Distribution as meaning an even distribution such that a uniformly distributed column will show each distinct value in the column appearing on approximately the same number of rows across the table. We will also conveniently ignore the idea of clustered data in which rows with the same values appear close to each other and thus localize themselves to a small number of blocks. That is a totally different topic.

From a practical perspective, finding skew is not really worth attempting. First you need to define a measure of it that works for you and then you need a way to calculate it and then you need to deal with the fact that you have likely calculated a general description of skew for your data which does not help at all with specific instances of skew in the data values. So in the end, what typically happens is we find out we have skew only after it becomes a problem for some query. If you really want to look for skew, you could try something like this but let us be realistic, if we saw this does it mean there is a skew problem?

```
  1  select stddev(count(*)),avg(count(*)),max(count(*)),min(count(*))
  2  from edw_dm.batch_history
  3* group by APPLICATION_NM
13:50:29 SQL> /

STDDEV(COUNT(*)) AVG(COUNT(*)) MAX(COUNT(*)) MIN(COUNT(*))
---------------- ------------- ------------- -------------
      1417.93668    839.980392          5575             1

1 row selected.
```

I suppose the maximum number of rows pointed to by a single value is 7 times bigger than the average which is almost an order of magnitude. That sounds like skew. But the reality is I do not plan to go out of my way executing scripts like this on all my columns to look for problems that do not yet exist. Most data has some skew. When that skew results in runway queries we need to deal with it.

The fix for skew is HISTOGRAMS. Histograms are another big topic and we are going to avoid a deep discussion of them. Think of a histogram as a more detailed description of your data. What we want to keep in mind is that histograms are one of Oracle's defenses against skew so when we are faced with skew that is negatively affecting a query plan, it is appropriate to create histograms on the columns that have it. The reverse is also true. Unless skew is causing a problem, we should not create histograms where we do not need them. For the above table, we might apply a METHOD_OPT parameter like this in our DBMS_STATS.GATHER_TABLE_STATS call.

```
method_opt=>'for all columns size auto for columns application_nm size 254'
```

This would collect a histogram on the column with skew without changing default statistics for other columns. Or better yet, we might just use AUTO for everything and let Oracle do the driving.

Dependence

We talked about Dependence once before when we discussed Dynamic Sampling, but let us look at another example. Consider the correlation between DATE-OF-BIRTH (month) and Signs of the Zodiac. Suppose you are building a database for a dating service. One question people like to ask for fun is "what is your sign?".

The ZODIAC is a circle parted into twelve divisions of 30 degree increments on a 360 degree circle. It defines a celestial longitude centered upon the ecliptic which is itself the perceived path of the sun across the celestial sphere which itself is an imaginary sphere larger than the earth with the earth inside it; think of the Earth as a ball inside a larger ball. All objects appear in the sky as if they sit on the inside of this larger ball surrounding the earth. This concept provides a simple way to plot the position (without distance) of objects in the sky. It so happens that these 12 divisions of the ZODIAC are date based, aligned with the vernal equinox which is the first of two times of the year when the tilt of the axis of the Earth is effectively 0 being neither toward nor away from the Sun (also that moment when the Sun is exactly over the Earth's Equator more or less) which occurs on 20-March more or less.

The term ZODIAC itself means CIRCLE OF ANIMALS. Having a date epoch and a consistent size, the ZODIAC is in one sense an alternative set of months each of which was given one of the following twelve names contrived from clusters of stars that looked like something to someone at some time (think 4th century BC, Babylonians): Aries, Taurus, Gemini, Cancer, Leo, Virgo, Libra, Scorpio, Sagittarius, Capricorn, Aquarius and Pisces: collectively known as SIGNS OF THE ZODIAC.

If we look at a comparison of the Gregorian Calendar (what most of us use for dates) to the SIGNS, we see there is a clear correlation. Here is I believe the mapping for 2013. Note please that each of the SIGNS corresponds to a specific number of days during the year. Looking at it from the reverse side, every single day of the year has one sign. Thus for example my Birth Day is on 22-OCT which means I am a Libra (the pleaser) and love is in the stars.

```
ZODIAC SIGN GREGORIAN DATE RANGE
----------- -------------------------
Aries       March 21 to April 19
Taurus      April 20 to May 20
Gemini      May 21 to June 20
Cancer      June 21 to July 22
Leo         July 23 to August 22
Virgo       August 23 to September 22
Libra       September 23 to October 22
Scorpio     October 23 to November 21
Sagittarius November 22 to December 21
Capricorn   December 22 to January 19
Aquarius    January 20 to February 18
Pisces      February 19 to March 20
```

That is some neat history but what does this have to do with dependence and Oracle database performance? Clearly Zodiac signs and Gregorian dates have a strong correlation. In the context of statistics and cardinality calculations, what does Oracle do with a query like the following?:

```
SELECT *
FROM HOOKUP_DATA
WHERE BIRTH_DATE = TO_DATE('22-OCT-1962','DD-MON-RRRR')
AND SIGN = 'LIBRA';
```

For this query, given what we just learned about the ZODIAC, we know that the predicate AND SIGN = 'LIBRA' offers nothing to a cardinality computation for this query because if we know the date, then we know the sign. With or without the predicate AND SIGN = 'LIBRA', this query will return the same rows. SIGN is dependent on BIRTH_DATE. But the database does not know this dependence exists. Oracle has only statistics on the two individual columns to work with. Oracle believes the two columns to be independent and will calculate cardinality for this query accordingly, which given the clear dependence between these two columns, will be wrong. There are two ways to get around this dependence issue. The first is dynamic sampling which we discussed previously.

The second way is to use EXTENDED STATISTICS. By using extended statistics we can create a COLUMN GROUP and then collect statistics on that column group. This would be done as follows.

```
DBMS_STATS.create_extended_stats(ownname   => 'SCOTT',
                                 tabname   => 'HOOKUP_DATA',
                                 extension => '(BIRTH_DATE,SIGN)');
```

This command defines to the database a group of columns that should be treated as a single object. After executing this command, whenever statistics are gathered for this table, the database will also collect statistics on this group of columns as if it were a single column. Then whenever the database sees these two columns used together in a conjunctive expression (AND) as we see above, the optimizer will consult the group statistics rather than compute statistics using stats of the individual columns. This means that any dependence between columns in the group will be accounted for by using group statistics.

As with everything there are limitations to extended statistics. One such limitation is disjunctive form (OR). This query would not use the column group statistics.

```
SELECT *
FROM SOMEDATA
WHERE BIRTH_DATE = TO_DATE('22-OCT-1962','DD-MON-RRRR')
OR SIGN = 'LIBRA';
```

Defaulting

Defaulting refers to any situation where Oracle cannot use column based statistics and must rely on DEFAULT percentages to do cardinality computations. The most common scenarios where this happens is in the use of functions on columns and other expressions. For example, these two queries both show a situation where simple column statistics cannot be used. The reason is that the thing upon which we need to calculate cardinality is not something we have collected statistics for.

```
select *
from person
where to_char(birth_date,'MON') = 'OCT';

select *
from emp
where salary+commission > 75000;
```

The first query is looking for anyone born in October. The second query is looking for employees that make more than the average household income in America. Both queries have done something to the columns. These queries are evaluating expression, not individual columns and we do not have statistics for these expressions, only the columns. So Oracle is forced to use default percentages in cardinality computations. But as before there are two ways to deal with this problem. Once again dynamic sampling can have a go at it.

The second way is to exploit EXTENDED STATISTICS like we did with dependence. These extended statistics would provide the statistical data we need.

```
DBMS_STATS.create_extended_stats(ownname   => 'SCOTT',
                                 tabname   => 'SOMETABLE',
                                 extension => '(to_char(birth_date,'MON'))');

DBMS_STATS.create_extended_stats(ownname   => 'SCOTT',
                                 tabname   => 'SOMETABLE',
                                 extension => '(salary+commission)');
```

Depending upon database version, one limitation is that it is not possible to created groups of expressions using extended statistics or to combine expressions with columns. In other words we cannot determine if there is dependence between expressions or expressions and columns. Thus for this query, the following extended statistics would be illegal syntax.

```
select *
from somedata
where SIGN = 'LIBRA'
and salary+commission > 75000;

DBMS_STATS.create_extended_stats(ownname   => 'SCOTT',
                                 tabname   => 'SOMETABLE',
                                 extension => '(SIGN,salary+commission)');
```

Out-Of-Bounds

Out-of-Bounds is a condition in which the value found in a predicate is outside the LOW_VALUE/HIGH_VALUE range that had been seen by statistics collection for some column or column group or expression upon which we collected statistics. Consider this sampling of statistics (abbreviated).

```
COLUMN_NAME     AVG_COL_LEN    DENSITY GLO USE LOW_VALUE        HIGH_VALUE
--------------- ----------- ---------- --- --- ---------------- -----------
ABS_HR                    3 .000020877 YES NO  "-1440"          "12516"
...
ABS_DATE_PK_ID            6 .000165071 YES NO  "19961031"       "20130731"
...
```

This is a short extract of column statistics from a table. I have removed most rows and columns from this data dump for clarity. We see in this statistics dump two columns called LOW_VALUE and HIGH_VALUE. When statistics were collected, these were the two extreme values the database saw for each column and they were recorded. These will be used when computing cardinalities if necessary.

Check out column ABS_DATE_PK_ID. Notice the absolute brilliance seen here of converting a DATE to a NUMBER which in turn introduces more complication as witnessed by the need to keep some semblance of ordering by using a specific format mask YYYYMMDD to generate the numbers. Nothing like taking a date, turning into a number but then trying to make the number still behave like a DATE. I need a hanky to wipe the tears from my eyes.

Yes this is a FACT table in a dimensional data model. Yes these guys read Kimball yet still did it wrong by gaffing their surrogate key by making it have intelligence. Yes, times have changed and a time dimension without a surrogate key is a valid design choice given the benefits of date based partitioning which to Kimball's defense did not exist at the time he first proposed his rules on surrogate keys. Additionally I seem to recall that Kimball in his latest literature has reconsidered the question of surrogates keys used for DATE based dimensions. But I am getting off topic.

Hmm… how many days are there between 20130731 and 19961031? Let us see…

```
14:01:03 SQL> select 20130731-19961031 from dual;

20130731-19961031
-----------------
           169700
```

Really? No! Does this look stupid? Not yet? Well how about this? How many rows will come out of this query? How many rows will Oracle estimate will come out of this query? Why is Oracle's estimate so far from reality? Do you think this does not happen in your databases? Do you believe there are other variations on this theme that are more complex and interesting and do a great job of messing up cardinality estimates?

```
select * from time_dimension where time_PK between 19961031 and 20130731;
```

Once again I get off topic. This is a different problem which we discussed earlier. I just could not help myself. When getting material to discuss OUT-OF-BOUNDS, I saw this data show up as the root of one of today's problems. Let us get back to OUT-OF-BOUNDS.

What can we say about the following query?

```
select *
from somedata
where ABS_DATE_PK_ID = 20130831;
```

This query is looking for this year's AUGUST month end rows. How many rows are in the table for AUGUST MONTH END? Being that the current month is SEPTEMBER, I can tell you that there is exactly one month's worth of rows in the table for this month end date (about 30 thousand). But how many rows does the Optimizer think are in the table with this date? The answer is one (1) but only because the optimizer won't let itself think zero.

Referring back to the column statistics shown earlier, please notice that the HIGH_VALUE for ABS_DATE_PK_ID is 20130731. The query is looking for 20130831. The value this query wants is greater than the currently known HIGH_VALUE. This is called OUT-OF-BOUNDS (the same being true for a value lower than the LOW_VALUE of column statistics). So Oracle thinks this table has no rows for the requested date. What else can it think?

In fact the optimizer tries to be smarter. It will attempt to scale or prorate the out-of-bounds value based on its distance from the HIGH_VALUE. Yet this only takes us back to the problem that this DATE was converted to a NUMBER. The optimizer will be doing its scaling using NUMBER math instead of DATE math, and so the scaling no longer works right. Making things more difficult, this problem changes when histograms are created, and which database patch sets have been applied to your database.

This OUT-OF-BOUNDS problem is very common for date based tables where rows are sensitive to time. Month end reporting typically suffers badly when this occurs. There are two solutions to this problem. First is Dynamic Sampling which we already talked about.

Second is better statistics collection. The basic rule about statistics has not changed. They need to be representative of your data. If your SQL WORKLOAD looks at recently loaded data, then after you load that data you need to recollect statistics on it in order to obtain a new HIGH_VALUE (and possibly LOW VALUE). In most cases this means after month end data loads we need to recollect stats. We may not like it. I know my guys hate it when I tell them this. But that is what we have to do.

Transiency

Transiency refers to data that is temporary. Rows enter a table, they stay there for a short while, then they are removed.

Good examples of transiency.

- GLOBAL TEMPORARY TABLES
- WORK TABLES
- STAGING TABLES

These tables present a special problem because of their nature. For one of several different reasons the rows in these tables have a limited life span and this means that these tables will soon contain a different set of rows which leads us to address the problem of statistics and determine if the statistics for the previous set of rows are good enough for the new set of rows or if we need something different. There are three solutions to dealing with statistics for transient rows. First is Dynamic Sampling (hmm… a pattern is emerging) which we already talked about.

Second is simple statistics collection. We collect new statistics every time we load new rows. Thus we know our stats are representative all the time.

Third is statistics lockdown. In this option we select a statistics collection that seems to do a good job for us and lock it down so that it cannot be changed. Then we do not collect statistics again.

All three choices can work. Though I suggest you look at dynamic sampling. Dynamic Sampling was originally intended to solve this specific problem.

Bloat

Bloat refers to the special condition of tables with lots of blocks but very few rows. These tables have lots of empty space BELOW the high water mark. Consider a table with 11 rows taking about 10K of data space, where the table has allocated to it 500 blocks. On a typical database with a 16K or 8K blocks size, these 11 rows will occupy one or two blocks. Having a large number of blocks allocated is a waste of space.

OK so space wastage is sad but it is also normal. A typical Oracle database has hundreds of ways of wasting space. And developers seem to find a way to use all of them. So what makes BLOAT such a big deal? Bloat affects dynamic sampling. We have seen above that dynamic sampling is one possible solution to all of the common statistics related problems we face (though admittedly with varying degrees of success). And we know that Oracle is committed to dynamic sampling in the future regardless of how many times its name changes because they keep expanding its capabilities and autonomy. So our future will be filled with dynamic sampling in one form or another.

Unfortunately in my experience, dynamic sampling is not very particular about the blocks it samples. As long as a block is below the high water mark dynamic sampling can sample it. So as the percentage of empty blocks below the high water mark increases, the quality of dynamically sampled statistics decreases. Thus it is to our advantage to check the number of blocks below the high water mark for a table, and if the ratio is more than some threshold (say twice?), then we should consider rebuilding these tables in order to push empty blocks back above the high water mark thus improving dynamic sampling results for these tables. The following query has helped me with the problem.

```
select blocks,NUM_ROWS,avg_row_len,owner,table_name
,ceil((blocks*8192)/decode(avg_row_len,0,null,avg_row_len)/1.2/decode(NUM_ROWS,0,null,
NUM_ROWS)) size_multiplier
from dba_tables
where owner in ('SCOTT','SCOTT1','SCOTT2')
```

```
and
nvl(ceil((blocks*8192)/decode(avg_row_len,0,null,avg_row_len)/1.2/decode(NUM_ROWS,0,nu
ll,NUM_ROWS)),3) > 2
order by table_name,owner
/
BLOCKS     NUM_ROWS   AVG_ROW_LEN OWNER           TABLE_NAME      Multiplier
---------- ---------- ----------- --------------- --------------- ----------
     16000       5000         128 SCOTT           ABSNC_FREQ_DIM         171
…
        16          7         118 SCOTT           FCT_SNPSHT_DT_D        133
…
         0          0           0 SCOTT1          RPTG_BAND_DIM
         4         63         130 SCOTT2          RPTG_BAND_DIM            4
```

This query shows how many more times space is allocated compared to the data the table contains. For example, table ABSNC_FREQ_DIM has 16 thousand blocks allocated to it, but based on the number of rows and average row length, it requires on only a fraction of that amount. Thus it has 171 times as many blocks as it needs to actually store the data it contains. This is a table with BLOAT and dynamic sampling might have significant issues getting a good sampling from the table since 99% of the space available for sampling is empty. We also see other tables in this list where the multiplier is high but the number of blocks is low. For these tables, dynamic sampling if it occurs on these tables, will simply sample the entire table so these are not a problem. I would recommend that table ABSNC_FREQ_DIM be rebuilt. Something like this would do it.

```
ALTER TABLE ABSNC_FREQ_DIM MOVE;
ALTER INDEX ABSNC_FREQ_DIM REBUILD;
EXEC DBMS_STATS.GATHER_TABLE_STATS('SCOTT','ABSNC_FREQ_DIM')
```

The first statement will rebuild the table in the same table space it currently resides in. This rebuild operation will repack the data so that most empty blocks will reside above the high water mark. This table move will make indexes on the table unusable so the second statement is needed to address that. The third statement recollects statistics so we can see the effect of the change. After doing a rebuild on the table noted above, number of blocks went from 16000 to 82. All we did was repack the data so that dynamic sampling would not have a hard time getting a good sampling.

More Omitted Material