FALLACIES OF THE COST BASED OPTIMIZER

Wolfgang Breitling, Centrex Consulting Corporation

The essential task of the optimizer is to identify an execution plan among many possible plans that is least costly. A query evaluation plan determines the execution sequence of relational operators such as selections, joins and projections. The basis on which costs of different plans are compared with each other is the estimation of sizes (cardinalities) of – temporary or intermediate – relations after an operation. These estimates are derived from statistics on the tables, columns and indexes of the database system. Inaccurate base statistics or incorrect derived estimates from - even accurate - statistics may cause the optimizer to choose a very poor plan. "Although the initial error might be negligible for the first subplan – such as the first selection or join, the subsequent errors can grow very rapidly (i.e. exponentially). Good estimates for the cost of database operations are thus critical."[1]

This paper identifies three basic assumptions made by the cost based optimizer[1] in the estimation of cardinalities of the results of relational operations on the base and intermediate row sources and ultimately the query result set. These assumptions, if violated, can render the cardinality estimates to be off by orders of magnitude and with them the basis for choosing the plan with the lowest cost. Unfortunately, in reality these assumptions tend to be frequently violated. We will examine examples of queries that breach these assumptions and show how that affects the execution plan evaluation and selection. We will also show what remedies and workarounds, if any, are possible to right the estimation wrongs.

THE COST BASED OPTIMIZER

The following is taken from[2] to give definitions for the terms "cardinality" and "selectivity" which will be used throughout this paper.

Selectivity

The first type of measure is the selectivity, which represents a fraction of rows from a row set. The row set can be a base table, a view, or the result of a join or a GROUP BY operator. The selectivity is tied to a query predicate, such as last_ name = 'Smith', or a combination of predicates, such as last_name = 'Smith' AND job_type = 'Clerk'. A predicate acts as a filter that filters certain number of rows from a row set. Therefore, the selectivity of a predicate indicates how many rows from a row set will pass the predicate test. The selectivity lies in the value range 0.0 to 1.0. A selectivity of 0.0 means that no rows will be selected from a row set, and a selectivity of 1.0 means that all rows will be selected. When statistics are available, the estimator estimates selectivity based on statistics. For example, for an equality predicate (last_name = 'Smith') the selectivity is set to the reciprocal of the number of distinct values of last_name, because the query selects rows that all contain one out of N distinct values.

Cardinality

Cardinality represents the number of rows in a row set. Here, the row set can be a base table, a view, or the result from a join or GROUP BY operator. The base cardinality is the number of rows in a base table.

The effective cardinality is the number of rows that will be selected from a base table. The effective cardinality is dependent on the predicates specified on different columns of a base table. This is because each predicate acts as a successive filter on the rows of a base table. The effective cardinality is computed as the product of basecardinality and combined selectivity of all predicates specified on a table. When there is no predicate on a table, its effective cardinality equals its base cardinality. The join cardinality is the number of rows produced when two row sets are joined together. A join is a Cartesian product of two row sets with the join predicate applied as a filter to the result. Therefore, the join cardinality is the product of the cardinalities of two row sets, multiplied by the selectivity of the join predicate.

The optimizer calculates the cost of an access plan by accumulating the costs of basic building blocks:

---

[1] not just Oracle's but those of other commercial database vendors as well

Base Access Costs

- Base table access
  There are many base table access plans – full table scan, index unique, index range scan, etc. – each with its own cost formula, but ultimately the base table access cost is dependent on the estimated number of rows that need to be fetched: the estimated cardinality. That is true even for a full table scan or an index unique scan, irrespective of the fact that their costs are not directly related to the estimated cardinality, but their choice is predicated by the estimated cardinality.
  Some of the cost formulas are:
  - Table scan nblks / k[◈]
  - Unique scan blevel+1
  - Fast full scan leaf_blocks / k
  - Index-only blevel + FF * leaf_blocks
  - Range scan blevel + FF * leaf_blocks + FF * clustering_factor

FF (filter factor) is another term for selectivity. It is a measure of a predicate's power to reduce the result set. Remember that selectivity, and thus filter factor, was defined as the fraction of (expected) rows from a row set (Page 1):

$$\text{Selectivity} = FF = \text{card}_{est} / \text{card}_{base} \Leftrightarrow \text{card}_{est} = FF * \text{card}_{base}$$

where card_{base} is the base cardinality, which for base tables is the number of rows, i.e. NUM_ROWS. For an equality predicate for example, the selectivity (= FF) is set to the reciprocal of the number of distinct values of that column: 1/NDV. For a prime key, NDV is equal to NUM_ROWS of the table and the estimated cardinality therefore becomes:

$$\text{card}_{est} = 1/\text{NDV} * \text{NUM\_ROWS} = 1/\text{NUM\_ROWS} * \text{NUM\_ROWS} = 1$$

exactly what is to be expected. For the selectivity formulas for other predicates see [3] or [4].

Join Costs

The join explanations and cost formulas are again taken from[2]

- ***nested loop join*** *for every row in the outer row set, the inner row set is accessed to find all the matching rows to join. Therefore, in a nested loop join, the inner row set is accessed as many times as the number of rows in the outer row set.:*
  cost = outer access cost + (inner access cost * outer cardinality)
- ***sort merge join*** *the two row sets being joined are sorted by the join keys if they are not already in key order.*
  cost = outer access cost + inner access cost + sort costs (if sort is used)
- ***hash join*** *the inner row set is hashed into memory, and a hash table is built using the join key. Each row from the outer row set is then hashed, and the hash table is probed to join all matching rows. If the inner row set is very large, then only a portion of it is hashed into memory. This portion is called a hash partition. Each row from the outer row set is hashed to probe matching rows in the hash partition. The next portion of the inner row set is then hashed into memory, followed by a probe from the outer row set. This process is repeated until all partitions of the inner row set are exhausted.*
  cost = (outer access cost * # of hash partitions) + inner access cost

The three join methods each have their own cost formula but all are dependent on the base access costs of the outer and inner table and their effective cardinalities.

Other Costs

- Auxiliary steps such as a sort for a GROUP BY or ORDER BY. Again, the cost of such steps is generally related to the size, or cardinality, of the data they operate on.

[◈] the value of k depends on the Oracle version and the value of the init.ora parameter db_file_multiblock_read_count. Contrary to what many believe, or event write or teach, k is not equal to db_file_multiblock_read_count !

As we have seen, the cost of an access plan is a function of the estimated cardinalities of its components. This is why it is so important that the cardinality estimates are accurate. If, for example, the optimizer underestimates the cardinality of a base table access, it may incorrectly find that the lowest cost plan is one that uses this table as the outer table of a NL join. Conversely. if it overestimates the cardinality, it may, again incorrectly, calculate the cost for a plan that uses the table as the outer table of a NL join as too high and discard it in favour of a different, inferior, plan.

This dependence of the cost on the estimated cardinalities is also the reason why costs of two different explain plans can not be compared. Something must have been different between the two environments in which the sql statements were parsed or else the two plans would have come out the same with the same costs (there is no random generator/selector in the CBO). This difference will have affected the estimated cardinalities which in turn affected the access plan choice. It is also the explanation for the apparent paradox of a higher cost figure for a faster access plan. The estimated cardinalities of the faster plan are better estimates of the real cardinalities and therefore the access plan chosen is better under the circumstances. The plan with the lower cost would have been good for data with lower cardinalities but goes awry as the real cardinalities differ. We will see examples of that.

THE ASSUMPTIONS

While the exact workings of the Oracle cost based optimizer are a proprietary secret of Oracle Corporation, the basics of query optimization in general, and cost calculation, selectivity determination, and cardinality estimation in particular, have been extensively studied in research labs and universities since E.F. Codd proposed the relational model for databases[5]. Published papers relevant to the topics covered in this paper are listed in the references. Given that selectivity of a predicate can also be defined as "the probability that a randomly selected row satisfies the predicate", the formulas for manipulating and combining predicates, and thus selectivities, borrow heavily from probability theory. The entire problem of query optimization in a relational database is, of course, steeped in relational theory. In order for solutions, published in the research papers, journals, and conference proceedings, to be valid and provable within the theoretical framework, certain assumptions had to be made.

As the solutions were implemented in commercial optimizers, the formulas used, and costs calculated based on these formulas, are only valid as long as the query and the referenced tables adhere to those assumptions. However, commercial relational database systems and their optimizers operate in the real world rather than in an ideal theoretical environment and the assumptions stipulated by the theory are often violated. This paper looks at 3 core assumptions – I call them fallacies because they are so often wrong – and shows how, in succession, the optimizer's selectivity, cardinality and cost estimates go astray when they are violated. It will demonstrate how miscalculated cardinalities can have a devastating effect on the access path choice and, where possible, identify solutions.

FALLACY I – THE UNIFORM DISTRIBUTION ASSUMPTION

There are three different forms of the uniform distribution assumption. One assumes that the column values are evenly distributed across all physical blocks of the table. The second assumes that the column values are evenly distributed across all rows of the table. Both can independently be true or false. The third assumes that the attribute values are evenly distributed across the spectrum of values, i.e. between the lowest and the highest value.

UNIFORM DISTRIBUTION OF COLUMN VALUES OVER ALL BLOCKS

Unlike the other assumptions, which affect the cost and plan estimates indirectly through the "selectivity ➔ cardinality ➔ cost" chain, adherence or violation of this assumption does not affect the cardinality estimate but goes purely and directly towards the cost of an index access.

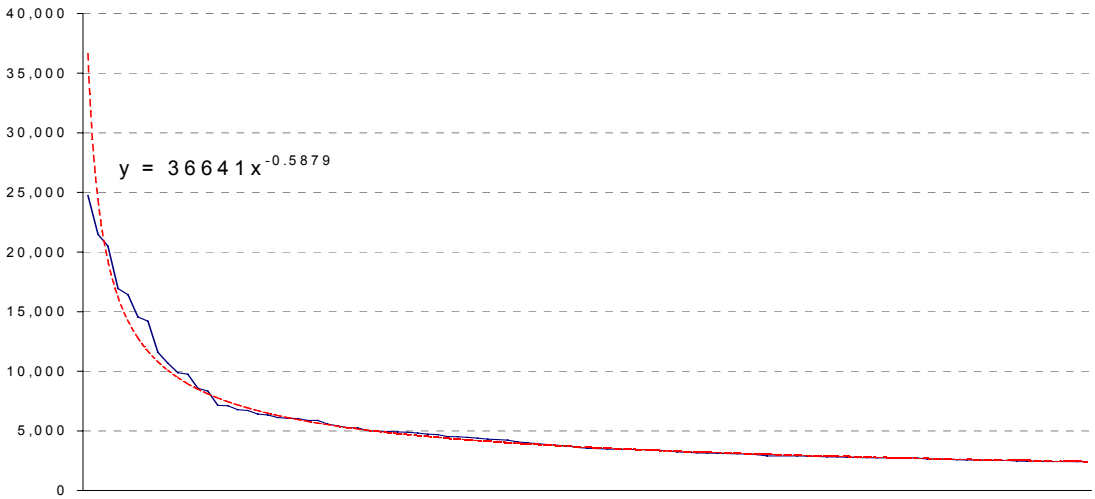
It should also be noted that, again in contrast to the other assumptions, the assumption of uniform distribution represents the worst case scenario – the more clustered the column values are the "cheaper" the access cost. If all n rows with a particular column value happen to be located in only 1 block, the I/O cost of access is just 1 rather than potentially n if they are uniformly distributed.

Cary Milsap has covered this topic extensively in his paper[6].

UNIFORM DISTRIBUTION OF COLUMN VALUES OVER ALL ROWS

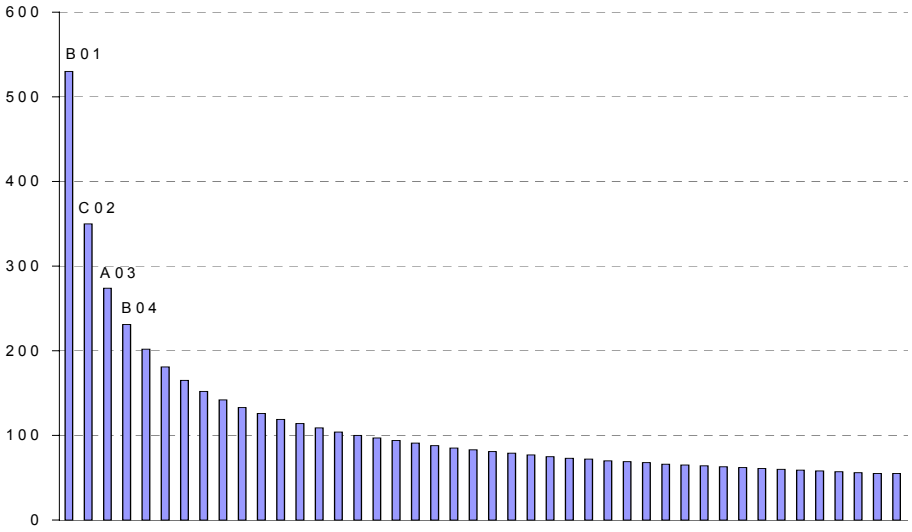
Several selectivity estimation methods have been reported in the literature. The earliest, simplest, and most commonly used is based on the uniform distribution assumption[7]. It assumes that attribute[2] values occur equally frequently in the table, i.e. each 1/NDV times and sets the selectivity to this value. However, this assumption rarely holds in practical situations. It has been observed that frequency distributions of attribute values often follow a power, or "Zipf"[8] distribution.

Below, for example, is the frequency distribution of the department_id in a production general ledger table, including the trendline which clearly shows the power distribution:



The following example shows the effect of the non-uniform distribution on the selectivity and cardinality estimate and, ultimately, the execution plan. The frequencies of the company column values have been crafted to follow a power distribution like that above.

```
COM   COUNT(0)
--- ----------
B01         530
C02         350
A03         274
B04         231
C05         202
A06         181
B07         165
C08         152
A09         142
B10         133
 …
C00          28
```




```
Select
emplid, jobcode, salary
from ps_job5 b
```

```
Explain Plan
 card operation
   50 SELECT STATEMENT
```

[2] attribute and column may be used interchangeably in this paper. Attribute and tuple are terms used in relational theory while column and table are preferred in relational database practice.

```
where b.company ='B01'                50   TABLE ACCESS BY INDEX ROWID PS_JOB5
                                      50     INDEX RANGE SCAN PSBJOB5
```

```
Execution Plan from tkprof
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.47	0.47[3]	21	359	5	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	37	0.48	0.47	420	567	0	530
total	39	0.95	0.94	441	926	5	530

```
Rows     Row Source Operation
    530 SELECT STATEMENT   GOAL: CHOOSE
    530   TABLE ACCESS BY INDEX ROWID PS_JOB5
    531     INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'PSBJOB5' (NON-UNIQUE)
```

In this example, the optimizer underestimated the cardinality by one order of magnitude. Had we used the predicate "`where b.company ='C00'`", it would have overestimated the cardinality by a factor of 2. When compounded over several query blocks of an access plan, or combined with other assumptions the effects of inaccurate cardinality estimates due to skew can be profound.

Furthermore, it is reasonable to assume that the use of the column values in a predicate follows a distribution pattern similar to that of the column values in the database. If a value occurs in more rows in the database, it is likely also more often the subject of interest., therefore, averaged over all SQL, the optimizer is more likely to underestimate cardinalities than to overestimate them.

REMEDY

Of course, the remedy for this fallacy is well known and documented: the use of histograms. Collecting a histogram on the company column gives the optimizer better, if not exact, selectivity and thus cardinality estimates

```
Explain Plan
```

```
     card operation
      534 SELECT STATEMENT
      534   TABLE ACCESS FULL PS_JOB5
```

```
Execution Plan
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.17	0.15	25	424	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	37	0.24	0.22	912	943	15	530
total	39	0.41	0.37	937	1367	15	530

```
Rows     Row Source Operation
    530 SELECT STATEMENT   GOAL: CHOOSE
    530   TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'PS_JOB5'
```

However, if the query uses a bind variable, we are back to where we started:

```
Select emplid, jobcode, salary
from ps_job5 b
where b.company = :b1
```

```
Explain Plan
```

```
    card operation
      61 SELECT STATEMENT
      61   TABLE ACCESS FULL PS_JOB5
```

Let me digress for a moment. Everywhere you hear and read that histograms are useless, being ignored by the CBO, if a query uses bind variables. But look at the explain plan above. Notice that the cardinality is now 61. The explanation for this is that the optimizer is not using the reciprocal of NDV for the column selectivity, but the value of

[3] The parse time is so high because the 10053 event trace had been enabled.

the density statistic of the column. Without histograms, density is equal to 1/NDV, but with histograms density is calculated differently. The column statistics after the histogram collection are:

table	column	NDV	density	bkts
PS_JOB5	COMPANY	200	6.0644E-03	67

10,000 * 6.0644^{e-3} = 60.644 rounded up to 61.

So histograms **can** make a difference even with queries that use bind variables. By using different numbers of buckets one can influence the value of density and therefore the cardinality estimate.
For this sample table and its company column the following bucket–density values and resulting cardinality estimates were observed:

buckets	density	card
10	1.0870E-02	109
25	8.5039E-03	86
50	7.4833E-03	75
75	6.0644E-03	61
90	5.5556E-03	56
100	5.0000E-03	50
150	3.3333E-03	50
199	2.5381E-03	50
200	5.0000E-05	50

It is obvious that density, and therefore selectivity and cardinality estimates, decrease as the number of buckets increases. Density is equal to 1/NDV at the midpoint (# buckets = NDV/2) and generally greater than 1/NDV before and smaller after, although there can be minor "aberrations" around the midpoint.
Note the big drop in density as the number of buckets increases from 199 to 200. This is because at 200 buckets – the number of distinct values of company – the histogram changes from a height based to a value based histogram and density is calculated differently again.

Note also how the reciprocal of NDV acts as a lower bound for the column selectivity and thus the estimated cardinality, i.e selectivity = max[density, 1/NDV].

UNIFORM DISTRIBUTION OF COLUMN VALUES OVER THE RANGE OF VALUES

Still another case of uniform distribution assumption occurs with range predicates. To quote again from [2]"*The optimizer assumes that employee_id values are distributed evenly in the range between the lowest value and highest value.*"

If that is not taken into consideration during development, this assumption may be violated "by design". For example, ACCOUNTING_PERIOD in the Peoplesoft Financials general ledger table has 15 distinct values with a range 0-999. Period 0 holds opening balances, periods 1-12 hold the ledger entries for the months, and periods 998 and 999 are used for special processing[4]. The frequencies for the periods 1-12 are roughly uniform and the selectivity of an equality predicate is not far off:

accounting_period = n [n ε {1 .. 12}]
⇒ selectivity = 1/NDV = 1/15 = 6.6667e-2

However, the selectivity of a range predicate is underestimated because of the artificially high maximum range:

accounting_period between 1 and 12
⇒ selectivity[⊗] = 12/(999-0) + 1/15 = 7.8679e-2

when in reality the selectivity should be close to 1

Paradoxically, according to the range selectivity formula, the selectivity of the range "accounting_period < 12" would be (see [9])

⇒ selectivity = (12-0)/(999-0) = 1.2012e-2

much smaller than the equality selectivity, even though the expected cardinality, and therefore the selectivity should be 11 times as big. However, similar to the selectivity of a bind variable on a column with a histogram, the optimizer

[4] This use of outliers is not uncommon in database design, especially by inexperienced designers.

[⊗] The calculation of the selectivity differs from the formula published in Note 68992.1 "Predicate Selectivity". It has been deduced from the selectivity value observed in the CBO trace.

uses 1/NDV as a lower bound for any range selectivity. Yet, even at 1/15, the range selectivity of "accounting_period < 12" is still severely underestimated.

One possible remedy would be to correct min or max outliers in the statistics. Changing the high value in the column statistics for accounting_period to 14 yields much more realistic cardinality estimates as shown below in the comparison of 10053 traces. The differences are highlighted.

```
select sum(posted_total_amt) from ps_ledger
where accounting_period between 1 and 12

Column: ACCOUNTING  Col#: 11     Table: PS_LEDGER   Alias: PS_LEDGER
    NDV: 15        NULLS: 0        DENS: 6.6667e-002 LO:  0  HI: 999
  TABLE: PS_LEDGER     ORIG CDN: 745198  CMPTD CDN: 58632

Column: ACCOUNTING  Col#: 11     Table: PS_LEDGER   Alias: PS_LEDGER
    NDV: 15        NULLS: 0        DENS: 6.6667e-002 LO:  0  HI: 14
  TABLE: PS_LEDGER     ORIG CDN: 745198  CMPTD CDN: 684873
```

The following selectivity calculation yields the observed values for both cases:

- with high_value = 999:
  $card_{est}$ = selectivity * $card_{base}$ = [max((12-1)/(999-0), 1/15) + min(12/(999-0), 2/15)] * 745,198
  = [1.2012e-2 + 6.6667e-2] * 745,198 = 7.8679e-2 * 745,198 = 58,631.19
  rounded up to 58,632

- with high_value = 14:
  $card_{est}$ = selectivity * $card_{base}$ = [max((12-1)/(14-0), 1/15) + min(12/(14-0), 2/15)] * 745,198
  = [1.3333e-1 + 7.8571e-1] * 745,198 = 9.1905e-1 * 745,198 = 684,872.45
  rounded up to 684,873

```
select sum(posted_total_amt) from ps_ledger
where accounting_period < 12

Column: ACCOUNTING  Col#: 11     Table: PS_LEDGER   Alias: PS_LEDGER
    NDV: 15        NULLS: 0        DENS: 6.6667e-002 LO:  0  HI: 999
  TABLE: PS_LEDGER     ORIG CDN: 745198  CMPTD CDN: 49680

Column: ACCOUNTING  Col#: 11     Table: PS_LEDGER   Alias: PS_LEDGER
    NDV: 15        NULLS: 0        DENS: 6.6667e-002 LO:  0  HI: 14
  TABLE: PS_LEDGER     ORIG CDN: 745198  CMPTD CDN: 638742
```

Here this selectivity calculation yields the observed values:

- with high_value = 999:
  $card_{est}$ = selectivity * $card_{base}$ = max[12/(999-0), 1/15] * 745,198 = max(1.2012e-2, 6.6667e-2)
  = 6.6667e-2 * 745,198 = 49679.87 rounded up to 49,680

- with high_value = 14:
  $card_{est}$ = selectivity * $card_{base}$ = max[12/(14-0), 1/15] * 745,198 = max(8.5714e-1, 6.6667e-2)
  = 8.5714e-1 * 745,198 = 638,741.14 rounded up to 638,742

It may be interesting to note that DB2 does not collect and store the minimum and maximum value of a column in the catalog, but the 2nd lowest and 2nd highest values in an attempt to eliminate outliers. Evidently, that would not have helped here since the 2nd highest value (998) would still have been an outlier.

REMEDY

The best and easiest remedy for a violation of this assumption is, of course, again a histogram on the column, especially if the number of distinct values is less than 255 and a value-based histogram can be used.

See [10] and [11] for good information on histograms.

FALLACY II – THE PREDICATE INDEPENDENCE ASSUMPTION

When there is more than one predicate on a table, the selectivities of the individual predicates are combined according to the following rules:[9]

P1 AND P2 S(P1&P2) = S(P1) * S(P2)
P1 OR P2 S(P1|P2) = S(P1) + S(P2) -[S(P1) * S(P2)]

These rules are the exact replicas of the rules for combining probabilities and are subject to the same restrictions: they are only valid in this simple form if the events, in our case the predicates, are independent. As with the uniform distribution assumption, this assumption is also often breached in practical applications. This violation is, however, more difficult to detect.

Here is an example of the same query issued against two different tables:

```
select emplid, jobcode, salary
from ps_job1 b
where b.company = 'CCC'
  and b.paygroup = 'FGH';

250 rows selected.
```

```
select emplid, jobcode, salary
from ps_job2 b
where b.company = 'CCC'
  and b.paygroup = 'FGH';

2500 rows selected.
```

The explain plans are identical and the estimated cardinalities for table PS_JOB1, where the predicates are indeed independend, match the actual cardinalities, whereas estimated and actual cardinalities for table PS_JOB2 differ by one order of magnitude:

```
Explain Plan

 card operation
  251 SELECT STATEMENT
  251   TABLE ACCESS BY INDEX ROWID PS_JOB1
  251     INDEX RANGE SCAN PSBJOB1

Execution Plan

 card operation
  250 SELECT STATEMENT
  250   TABLE ACCESS BY INDEX ROWID PS_JOB1
  251     INDEX RANGE SCAN PSBJOB1
```

```
Explain Plan

 card operation
  251 SELECT STATEMENT
  251   TABLE ACCESS BY INDEX ROWID PS_JOB2
  251     INDEX RANGE SCAN PSBJOB2

Execution Plan

 card operation
 2500 SELECT STATEMENT
 2500   TABLE ACCESS BY INDEX ROWID PS_JOB2
 2501     INDEX RANGE SCAN PSBJOB2
```

and the corresponding tkprof details:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.06	0.07	0	0	0	0
Exec	1	0	0	0	0	0	0
Fetch	18	0.03	0.03	252	272	0	250
total	20	0.09	0.1	252	272	0	250

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.06	0.08	0	0	0	0
Exec	1	0	0	0	0	0	0
Fetch	168	0.34	0.36	2518	2692	0	2500
total	170	0.4	0.44	2518	2692	0	2500

The statistics for both tables are up-to-date and virtually identical. The only significant difference is highlighted:

Table Statistics

table	free	used	rows	blks	empty	chain	avg_row_len
PS_JOB1	8	65	50,000	4,547	3	0	317

table	free	used	rows	blks	empty	chain	avg_row_len
PS_JOB2	8	65	50,000	4,547	3	0	317

Column Statistics[5]

table	column	NDV	density	bkts
PS_JOB1	EMPLID	10,000	1.0000E-04	1
PS_JOB1	JOBCODE	198	5.0505E-03	1
PS_JOB1	COMPANY	10	1.0000E-01	1
PS_JOB1	PAYGROUP	20	5.0000E-02	1
PS_JOB1	SALARY	49,597	2.0163E-05	1

table	column	NDV	density	bkts
PS_JOB2	EMPLID	10,000	1.0000E-04	1
PS_JOB2	JOBCODE	199	5.0251E-03	1
PS_JOB2	COMPANY	10	1.0000E-01	1
PS_JOB2	PAYGROUP	20	5.0000E-02	1
PS_JOB2	SALARY	49,848	2.0061E-05	1

Index Statistics

table	index	column	NDV	CLUF	#LB	lvl	#LB/K	#DB/K
PS_JOB1	PSBJOB1		**200**	.0000E+00	400	2	2	250
		COMPANY	10	1.0000E-01				
		PAYGROUP	20	5.0000E-02				
	PS_JOB1	U	50,000	.0000E+00	740	2	1	1
		EMPLID	10,000	1.0000E-04				
		EMPL_RCD#	1	1.0000E+00				
		EFFDT	21,842	4.5783E-05				
		EFFSEQ	1,006	9.9404E-04				

table	index	column	NDV	CLUF	#LB	lvl	#LB/K	#DB/K
PS_JOB2	PSBJOB2		**20**	.0000E+00	449	2	22	2,500
		COMPANY	10	1.0000E-01				
		PAYGROUP	20	5.0000E-02				
	PS_JOB2	U	50,000	.0000E+00	740	2	1	1
		EMPLID	10,000	1.0000E-04				
		EMPL_RCD	1	1.0000E+00				
		EFFDT	20,037	4.9908E-05				
		EFFSEQ	1,006	9.9404E-04				

In this case the index statistics clearly show that in table PS_JOB2 the attributes company and paygroup are not independent. The PSBJOB2 index has only 20 distinct values rather than the expected 200 if the attributes were independent (like PSBJOB1). Or, looking at it differently, each paygroup can occur in only one company since there are the same number of company-paygroup combinations as there are paygroup values, which clearly violates the attribute independence. However, if the index included another column, these conclusions could not be drawn just from the index statistics.

REMEDY

There is no remedy for the incorrect selectivity and cardinality estimates due to predicates not being independent. Where possible you can use hints or stored outlines to guide the optimizer to a better plan. If that is not possible the only remedy in severe cases may be to adjust the statistics such that the CBO's estimates better reflect the reality of the query. The problem with that approach is that different statistics may be required for different predicate combinations. That could be solved if Oracle gave us an extension of the stored outline model: the option to use statistics from "stored statistics" which would be a stattab-statid combination.

[5] Statistics for columns that do not appear in the query have been omitted.

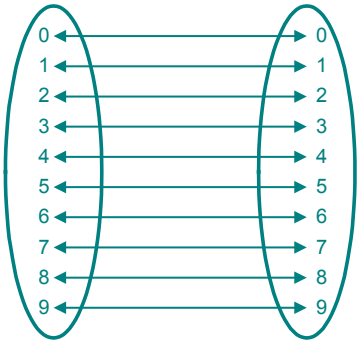
FALLACY III – THE JOIN UNIFORMITY ASSUMPTION

The join uniformity assumption states that a row from one table is equally likely to join with any row from the second table. It is essentially an extension of the uniform distribution assumption to two tables. This is the most difficult assumption to verify, and, by corollary, to detect when it is violated. Nonetheless, there are again many situations in which this assumption is violated. To understand how violation of this assumption affects the cardinality of a join we look at a simple example of joining to tables with 10 rows each.

Example 1

First an example with join uniformity intact – we join a table with 10 distinct values to itself.

```
SQL> select 'A-'||a.n1, 'B-'||b.n1
t1 a, t1 b
a.n1 = b.n1;

Explain Plan

    card operation
-------- -------------------------
      10 SELECT STATEMENT
      10   HASH JOIN
      10     TABLE ACCESS FULL T1
      10     TABLE ACCESS FULL T1

execution results

A-0   B-0
A-1   B-1
A-2   B-2
A-3   B-3
A-4   B-4
A-5   B-5
A-6   B-6
A-7   B-7
A-8   B-8
A-9   B-9

10 rows selected.
```




```
2  from
3  where
```

The join cardinality is determined by applying the join selectivity estimate to the cardinality of the cartesion join of the two tables:

Join cardinality = $card_A$ * $card_B$ * join selectivity[9]

The join selectivity is estimated as the smaller of the individual selectivities of the join predicate for each table, adjusted for nulls if necessary. Since attribute selectivity – under the uniform distribution assumption – is the reciprocal of the number of distinct attribute values, the join selectivity becomes:

join selectivity = $1/max(ndv_A, ndv_B)$[6]

This is how the estimated cardinality of 10 for the hash join above breaks down:

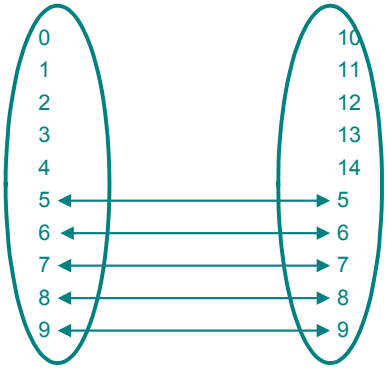
10 = $card_A$ * $card_B$ * /max(ndv_A, ndv_B) = 10 * 10 * 1/max(10,10) = 10 * 10 / 10

[6] See metalink note 68992.1. For simplicity reasons, and since it doesn't apply here, the adjustment for null values has been omitted.

Example 2

Now if we use a second table, also with 10 rows with distinct values, but only 5 of the values match any of the first table, we get the following:

```
SQL> select 'A-'||a.n1, 'B-'||b.n1
t1 a, t2 b
where a.n1 = b.n1;

Explain Plan
    card operation
-------- -------------------------
     10 SELECT STATEMENT
     10   HASH JOIN
     10     TABLE ACCESS FULL T1
     10     TABLE ACCESS FULL T2

execution results
A-5   B-5
A-6   B-6
A-7   B-7
A-8   B-8
A-9   B-9

5 rows selected.
```



The join cardinality estimate is calculated as above:

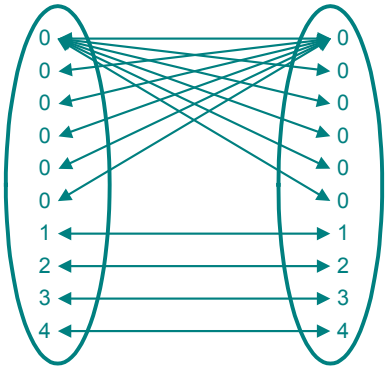
$10 = card_A * card_B * /max(ndv_A, ndv_B) = 10 * 10 * 1/max(10,10) = 10 * 10 / 10$

Obviously, it is wrong. It is overestimated by a factor of 2. This is a consequence of the violation of the join uniformity assumption.

Example 3

One more example, this time of an underestimation of the join cardinality. Here the breach of the join uniformity is related to a breach of the distribution uniformity. We again join a table with 10 rows to itself, but this time there are only 5 distinct attribute values: 6 of the rows have the same attribute, the remaining 4 are distinct.

```
SQL> select 'A-'||a.n1, 'B-'||b.n1
t2 a, t2 b
where a.n1 = b.n1;

Explain Plan
    card operation
-------- -----------------------
     20 SELECT STATEMENT
     20   HASH JOIN
     10     TABLE ACCESS FULL T2
     10     TABLE ACCESS FULL T2

execution results
A-0   B-0
A-0   B-0
A-0   B-0
…
A-0   B-0
A-1   B-1
A-2   B-2
A-3   B-3
A-4   B-4

40 rows selected.
```



The diagram shows the join arrows only for the first of the "0" attributes. There are 36 (6 * 6) combinations of "0" attributes between the two sources. The remaining four attributes add another 4 for the total of 40. The join cardinality estimate is calculated as above:

$20 = card_A * card_B * /max(ndv_A, ndv_B) = 10 * 10 * 1/max(5,5) = 10 * 10 / 5$

This time the join cardinality is underestimated by a factor of 2.

Example 4

Using the same table as in the first example, but loading 5 sets of data:

```
insert into t1(n1,n2,n3)
select mod(rownum,10),mod(rownum,5),mod(rownum,25)
from dba_objects where rownum <= 50;

insert into t2(n1,n2,n3)
select mod(rownum,10),mod(rownum,5),mod(rownum,25)
from dba_objects where rownum <= 50;
```

column	NDV	nulls	density	lo	hi	bkts
N1	10	0	1.0000E-01	0	9	1
N2	5	0	2.0000E-01	0	4	1
N3	25	0	4.0000E-02	0	24	1

```
select 'A.'||A.n1||'-B.'||B.n1
from t1 a, t2 b
where a.n1 = b.n1;

Explain Plan
```

card	operation
250	SELECT STATEMENT
250	HASH JOIN
50	TABLE ACCESS FULL T1
50	TABLE ACCESS FULL T2

```
Execution Plan
```

Rows	Execution Plan
0	SELECT STATEMENT GOAL: CHOOSE
250	HASH JOIN
50	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'T1'
50	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'T2'

No surprise so far. But how many sql statements do you have that join two tables with nothing but the join predicate(s). Consider adding a predicate:

```
select 'A.'||A.n1||'-B.'||B.n1
from t1 a, t2 b
where a.n1 = b.n1
and a.n2 = 5;

Explain Plan
```

card	operation
50	SELECT STATEMENT
50	HASH JOIN
10	TABLE ACCESS FULL T1
50	TABLE ACCESS FULL T2

Rows	Execution Plan
0	SELECT STATEMENT GOAL: CHOOSE
50	HASH JOIN
10	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'T1'
50	TABLE ACCESS GOAL: ANALYZED (FULL) OF 'T2'

The estimated cardinalities in explain plan still match the actual cardinalities of the execution plan.

Finally, consider the following sql:

```
select 'A.'||A.n1||'-B.'||B.n1
from t1 a, t2 b
where a.n1 = b.n1
and a.n1 = 5;
```

Since n1 has 10 distinct values in t1, the predicate "n1 = 5" reduces the estimated cardinality of t1 to 5 (50/10). Because of the join equality predicate the same is true for t2 and thus the overall cardinality of the resultset is 25 – all values are evenly distributed. And the execution plan shows exactly that.

```
Rows     Execution Plan
      0  SELECT STATEMENT   GOAL: CHOOSE
     25   HASH JOIN
      5    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'T1'
      5    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'T2'
```

In the explain plan, however, the optimizer underestimates the join cardinality by a factor of 5:

```
Explain Plan

   card operation
      5 SELECT STATEMENT
      5   HASH JOIN
      5     TABLE ACCESS FULL T1
      5     TABLE ACCESS FULL T2
```

Below is the extract from the 10053 event trace showing the relevant statistics and calculations. There is no explanation, unfortunately, of how the optimizer arrived at the join selectivity of 0.2. In this case it should be 1 since the selectivity has already been taken care of by the predicate and the applied transitive closure.

```
Table stats    Table: T2   Alias:  B
  TOTAL ::  CDN: 50  NBLKS:  1  TABLE_SCAN_CST: 1  AVG_ROW_LEN:  8

Table stats    Table: T1   Alias:  A
  TOTAL ::  CDN: 50  NBLKS:  1  TABLE_SCAN_CST: 1  AVG_ROW_LEN:  8

SINGLE TABLE ACCESS PATH
Column:         N1  Col#: 1      Table: T1   Alias:  A
    NDV: 10        NULLS: 0         DENS: 1.0000e-001 LO:  0  HI: 9
  TABLE: T1     ORIG CDN: 50  CMPTD CDN: 5
SINGLE TABLE ACCESS PATH
Column:         N1  Col#: 1      Table: T2   Alias:  B
    NDV: 10        NULLS: 0         DENS: 1.0000e-001 LO:  0  HI: 9
  TABLE: T2     ORIG CDN: 50  CMPTD CDN: 5
...
Join cardinality:  5 = outer (5) * inner (5) * sel (2.0000e-001)  [flag=0]
```

This example, coupled with the observation that many attributes do not have a uniform distribution and that predicates are no always independent, could explain many of the, sometimes severe, underestimations of join cardinalities experienced.

Consider for example the following join of the ps_job5 table introduced earlier (page 4) joined to table ps_pay_check5 with a similar distribution of company and paygroup columns. It is a simplified version of a real query, not just a made-up example.

```
select A.COMPANY, A.PAYGROUP, SUM(B.SALARY)
from PS_PAY_CHECK5 A, PS_JOB5 B
where A.COMPANY = 'B01'
  and A.PAYCHECK_STATUS = 'B'
  and B.COMPANY=A.COMPANY
  and B.PAYGROUP=A.PAYGROUP
group by A.COMPANY, A.PAYGROUP
order by A.COMPANY, A.PAYGROUP
```

Table Statistics

table	free	used	rows	blks	empty	chain	avg_row_len
PS_JOB5	10	40	10,000	911			317

table	free	used	rows	blks	empty	chain	avg_row_len
PS_PAY_CHECK5	10	40	40,000	3,638	0		303

Column Statistics

table	column	NDV	density	bkts
PS_JOB5	COMPANY	200	5.0000E-03	1
PS_JOB5	PAYGROUP	300	3.3333E-03	1

table	column	NDV	density	bkts
PS_PAY_CHECK5	COMPANY	200	5.0000E-03	1
PS_PAY_CHECK5	PAYGROUP	300	3.3333E-03	1
PS_PAY_CHECK5	PAYCHECK_STATUS	5	2.0000E-01	1

Index Statistics

table	index	column	NDV	CLUF	#LB	lvl	#LB/K	#DB/K
PS_JOB5	PSBJOB5		8,636	1.5403E-01	55	1	1	1
		COMPANY	196	5.1020E-03	0			
		PAYGROUP	301	3.3223E-03	0			
	PS_JOB5	U	10,000	7.7016E-02	104	1	1	1
		EMPLID	10,000	1.0000E-04				
		EMPL_RCD#	1	1.0000E+00				
		EFFDT	9,224	1.0841E-04				
		EFFSEQ	998	1.0020E-03				

table	index	column	NDV	CLUF	#LB	lvl	#LB/K	#DB/K
PS_PAY_CHECK5	PSAPAY_CHECK5		10,000	.0000E+00	288	2	1	4
		EMPLID	12,771	7.8302E-05	0			
		EMPL_RCD#	1	1.0000E+00	0			
	PS_PAY_CHECK5	U	40,000	1.6501E-02	389	2	1	1
		COMPANY	200	5.0000E-03				
		PAYGROUP	300	3.3333E-03				
		PAY_END_DT	26	3.8462E-02				
		OFF_CYCLE	3	3.3333E-01				
		PAGE#	9	1.1111E-01				
		LINE#	39	2.5641E-02				

Explain Plan

card	operation
1	SELECT STATEMENT
1	SORT GROUP BY
1	NESTED LOOPS
50	TABLE ACCESS BY INDEX ROWID PS_JOB5
50	INDEX RANGE SCAN PSBJOB5
40	TABLE ACCESS BY INDEX ROWID PS_PAY_CHECK5[7]
40	INDEX RANGE SCAN PS_PAY_CHECK5

[7] The estimated cardinality of 40 is the result of transitive closure:
$card_{est} = sel_{PAYCHECK\_STATUS} * sel_{COMPANY} * card_{base} = 1/5 * 1/200 * 40{,}000 = 40$

Execution Plan

```
Rows     Execution Plan
      0  SELECT STATEMENT   GOAL: CHOOSE
    183   SORT (GROUP BY)
   1206    NESTED LOOPS
    531     TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'PS_JOB5'
    531      INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'PSBJOB5' (NON-UNIQUE)
   1206     TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'PS_PAY_CHECK5'
   6458      INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'PS_PAY_CHECK5' (UNIQUE)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.02	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	14	0.29	0.31	2376	13963	0	183
total	16	0.31	0.33	2376	13963	0	183

Instead of the estimated 50 rows, 530 rows of table ps_job5 qualify for predicate "`COMPANY = 'B01'`" and the inner table ps_pay_check5 is therefore scanned 530 times rather than the estimated 50 times. But the really big difference is in the cardinality of the join result – 1206 rather than the estimated 1. In this example this underestimation of the join cardinality only affects the cost estimate for the subsequent "group by" sort and has no effect on the overall access plan. In queries that contain multiple joins, however, errors in the cardinality estimates of intermediate join results propagate[12] and can – and do – lead to disastrous access plans.

Explain Plan with histograms

Next we examine what difference histograms make. After collecting histograms for all predicate columns on both tables using the default 75 buckets, these are the relevant column statistics:

table	column	NDV	density	bkts
PS_PAY_CHECK5	COMPANY	200	6.2294E-03	67
PS_PAY_CHECK5	PAYGROUP	300	3.4257E-03	75
PS_PAY_CHECK5	PAYCHECK_STATUS	5	1.2500E-05	4

table	column	NDV	density	bkts
PS_JOB5	COMPANY	200	6.6044E-03	67
PS_JOB5	PAYGROUP	300	3.4257E-03	75

```
   card operation
      7 SELECT STATEMENT
      7   SORT GROUP BY
      7     HASH JOIN
    427       TABLE ACCESS FULL PS_PAY_CHECK5
    534       TABLE ACCESS FULL PS_JOB5

Rows     Execution Plan
      0  SELECT STATEMENT   GOAL: CHOOSE
    183   SORT (GROUP BY)
   1206    HASH JOIN
    414     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'PS_PAY_CHECK5'
    530     TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'PS_JOB5'
```

The histograms improved the cardinality estimates for the base tables to near accurate, leading to a different access plan, but did little to correct the severe underestimation of the join cardinality.

"proof" that the optimizer completed transitive closure on the company predicate:
A.COMPANY = 'B01' and B.COMPANY = A.COMPANY ⇒ B.COMPANY = 'B01'

REMEDY

There is again no easy remedy for the incorrect selectivity and cardinality estimates resulting from a violation of the join uniformity assumption. The same workarounds as for non-independent predicates apply: hints, outlines, and setting rather than gathering statistics.

GLOSSARY

NDV	number of distinct values. A.k.a. NUM_DISTINCT
Selectivity	represents a fraction of rows from a row set.
FF (FILTER FACTOR)	Another term for selectivity. Used especially for the combined selectivity of multiple predicates.
Cardinality	the number of rows in a row set.
Base cardinality	the number of rows in a base table.
Effective cardinality	the estimated number of rows that are selected from a base table. The effective cardinality depends on the predicates specified on different columns of a base table.
Join cardinality	the number of rows produced when two row sets are joined together. The join cardinality is the product of the cardinalities of two row sets, multiplied by the selectivity of the join predicate.
Distinct cardinality	the number of distinct values in a column of a row set.
Group cardinality	the number of rows produced from a row set after the GROUP BY operator is applied.

REFERENCES

1. Banchong Harangsri, John Shepherd, Anne H. H. Ngu: *Query Size Estimation Using Systematic Sampling*; proceedings of the International Symposium on Cooperative Database Systems for Advanced Applications, 1996. Kyoto, Japan.
2. *Oracle 9i Database Performance Tuning Guide and Reference*. 2002: Oracle Corporation.
3. Note:35934.1: *Cost Based Optimizer - Common Misconceptions and Issues*. metalink.oracle.com
4. Wolfgang Breitling: *A Look under the Hood of Cbo: The 10053 Event*. www.centrexcc.com
5. E. F. Codd: *A Relational Model of Data for Large Shared Data Banks*. Communications of the ACM, 1970. 13(6): p. 377-387.
6. Cary Millsap: *When to Use an Index*. www.hotsos.com
7. P.G. Selinger, M.M. Astrahan, D.D. Chamberlain, R.A. Lorie, T.G. Price: *Access Path Selection in a Relational Database Management System*; proceedings of the ACM SIGMOD International Conference on Management of Data, 1979. Boston, USA.
8. George Kingsley Zipf: *Human Behaviour and the Principle of Least Effort: An Introduction to Human Ecology*. 1949, Cambridge: Addison-Wesley.
9. Note:68992.1: *Predicate Selectivity*. metalink.oracle.com
10. Note:1031826.6: *Histograms: An Overview*. metalink.oracle.com
11. Steve Adams: *Ixora News - April 2001*. http://www.ixora.com.au/newsletter/2001_04.htm
12. Yannis E. Ioannidis, Stavros Christodoulakis: *On the Propagation of Errors in the Size of Join Results*; proceedings of the ACM SIGMOD International Conference on Management of Data, 1991. Denver, CO.