On the Estimation of Join Result Sizes

Arun Swami[1] and K. Bernhard Schiefer[2]

[1] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099
[2] IBM Toronto Lab, 895 Don Mills Road, North York, Ontario, Canada M3C 1W3

Abstract. Good estimates of join result sizes are critical for query optimization in relational database management systems. We address the problem of incrementally obtaining accurate and consistent estimates of join result sizes. We have invented a new rule for choosing join selectivities for estimating join result sizes. The rule is part of a new unified algorithm called Algorithm **ELS** (**E**quivalence and **L**argest **S**electivity). Prior to computing any result sizes, equivalence classes are determined for the join columns. The algorithm also takes into account the effect of local predicates on table and column cardinalities. These computations allow the correct selectivity values for each eligible join predicate to be computed. We show that the algorithm is correct and gives better estimates than current estimation algorithms.

1 Introduction

The join is an important operation in relational database management systems. When a user-generated query involves multiple joins, the cost of executing the query can vary dramatically depending on the query evaluation plan (QEP) chosen by the query optimizer. The join order and access methods used are important determinants of the lowest cost QEP.

The query optimizer estimates the eventual result size, or cardinality, of joining the specified tables and uses this information to choose between different join orders and access methods. Thus, the estimation of join result sizes in a query is an important problem, as the estimates have a significant influence on the QEP chosen.

A survey of the use of statistics and estimation techniques in query optimization is given in [7]. There has been a lot of work done on estimating selectivities, e.g., ([1, 10, 8]). In [6], a number of new selectivity estimation methods were proposed for highly skewed distributions such as Zipf distributions [17, 3]. Errors in the statistics maintained by the database system can affect the various estimates computed by the query optimizer. An analytical model was used in [4] to study the propagation of errors in the estimated size of join results as the number of joins increases in queries with a single equivalence class.

When more than two tables are involved, the query optimization algorithm often needs to estimate the join result sizes incrementally. Incremental estimation is used, for example, in the dynamic programming algorithm [13], the **AB** algorithm [15] and randomized algorithms [14, 5]. The optimizer first determines the join result size from joining the first two tables, then determines the join result size from the join with a third table, and so on. Thus, the query optimizer

incrementally estimates the final join result size from the result sizes of the intermediate tables produced.

We have identified a number of problems with current practice in join size estimation. The problems include dealing with local predicates correctly, correct handling of multiple columns in a single table that belong to a single equivalence class, and correct incremental estimation of join result sizes. In this paper, we describe an algorithm called **ELS** that solves these problems and give arguments for the correctness of the solutions.

In the next section, we give a brief overview of the terminology and known results in join size estimation that are used in this paper. The reader knowlegeable in query optimization can skim Section 2. Section 3 describes some of the problems in current join size estimation practice. Algorithm **ELS** is outlined in Section 4. In [16] we also show how one can handle the case of multiple local predicates on a single column. In Sections 5, 6, and 7, we describe the techniques used in Algorithm **ELS** to address the problems described in Section 3. We illustrate the value of our algorithm in terms of execution time in Section 8. In Section 9 we summarize the contributions of this paper and indicate some directions for future work.

2 Background

The following assumptions are made by most query-processing cost models in practice and this paper:

1. *Independence*: Within each relation, values chosen from distinct columns that are involved in join predicates are independent.
2. *Uniformity of Attribute Values*: The distinct values in a join column appear equifrequently in the column.
3. *Containment*: When joining two tables, the set of values in the join column with the smaller column cardinality is a subset of the set of values in the join column with the larger column cardinality.

The implications of these and other assumptions are discussed in depth in [2].

Note that we need the uniformity assumption only for the join columns, i.e., we can use data distribution information for local predicate selectivities. Some work has been done ([1]) to relax some of these assumptions. However, this requires assuming a certain parametric model of the data (nonuniform Zipf distribution) and determination of the parameter values, or keeping a large amount of metadata (correlation statistics). There is a tradeoff between implementation complexity and possible gains in accuracy of estimates. Most query optimizers choose to work with the assumptions listed above. As in other work, we focus on *conjunctive* queries where the selection condition in the WHERE clause is a conjunction of predicates. These queries constitute the most important class of queries.

A table may participate in several join predicates. Let a table R be joined with another (possibly intermediate) table I. At this point in the join ordering, the query optimizer only needs to consider the predicates that link columns in

table R with the corresponding columns in a second table S that is present in table I. These join predicates are termed *eligible* join predicates.

The query may include predicates that involve only a single table. Such predicates are called *local* predicates. The join result size is typically estimated as the product of the cardinalities of the operand tables after applying the local predicates (if any) and the join predicates [13]. When queries contain equality predicates (either local or join predicates), it is possible to derive additional implied predicates using transitive closure. (Similar derivations are possible for nonequality predicates too, but equality predicates are the most common and important class of predicates that generate implied predicates.) Transitive closure can be applied to two join predicates that share a common join column in order to obtain another join predicate.

Example 1a Suppose the following SQL statement is entered as a query [3]:

SELECT R_1.a
FROM R_1, R_2, R_3
WHERE (R_1.x = R_2.y) AND (R_2.y = R_3.z)

We can use the principle of transitivity with the join predicates J1: (R_1.x = R_2.y) and J2: (R_2.y = R_3.z) to obtain the join predicate J3: ($R_1.x = R_3.z$).

Performing this predicate transitive closure gives the optimizer maximum freedom to vary the join order and ensures that the same QEP is generated for equivalent queries independently of how the queries are specified. In the above example, without predicate J3, most query optimizers would avoid the join order beginning with ($R_1 \bowtie R_3$) since this would be evaluated as a cartesian product, which is very expensive[4]. Since join predicates can share columns, the effects of applying them may not be independent. In the above example, once join predicates J1 and J2 have been evaluated, J3 has in effect been evaluated, and hence evaluating it separately can have no further effect on the size of the join result. Equivalence classes of join columns can be used to capture these dependencies.

Initially, each column is an equivalence class by itself. When an equality (local or join) predicate is seen during query optimization, the equivalence classes corresponding to the two columns on each side of the equality are merged. In the example above, given join predicate J1, columns x and y are in the same equivalence class. When join predicate J2 is seen, the equivalence classes of columns y and z are merged. Thus columns x, y and z are now in a single equivalence class. As we will see later, we can use the equivalence class structure to correctly take into account the effect of dependencies between join predicates. We will often use the terminology "x and y are *j-equivalent* (columns)" to denote that x and y belong to the same equivalence class.

Multiple equivalence classes can be handled using the independence assumption as follows. Since the join columns in different equivalence classes are assumed satisfy the independence assumption, the predicates involving columns

[3] In the rest of the paper, we show only the WHERE clause of SQL queries.

[4] We use $\bowtie$ as the symbol for the join operation

in one equivalence class reduce the join result size independent of the predicates involving columns in another equivalence class. Thus, in the rest of the paper, we can focus on the case of a single equivalence class.

Two kinds of statistics are typically important in query optimization. One statistic is the number of tuples contained in a single table. This value is known as the *table cardinality* and is denoted by $\|R\|$, wherein R is the table. The second statistic is the number of distinct values present in a column. This value is known as the *column cardinality* and is denoted by d_x, wherein x is the column. These two statistics are important because they are used to estimate the size of the results from different operations on the table data, which in turn helps to determine the cost of these operations.

Consider a join predicate (J:($R_1.x_1 = R_2.x_2$)), where the column cardinality of x_1 (x_2) of table R_1 (R_2) is d_1 (d_2). The size of the join result can be estimated [13] as:

$$\|(R_1 \bowtie R_2)\| = d_i \times \frac{\|R_1\| \times \|R_2\|}{d_1 \times d_2}, \text{ where } d_i = \min(d_1, d_2) \qquad \text{(Equation 1)}$$

Since $\min(d_1, d_2)/d_1 \times d_2 = 1/\max(d_1, d_2)$ Equation Equation 1 can be rewritten as:

$$\|(R_1 \bowtie R_2)\| = \|R_1\| \times \|R_2\| \times S_J, \quad \text{where } S_J = \frac{1}{\max(d_1, d_2)} \qquad \text{(Equation 2)}$$

where S_J is termed the *selectivity* of the join predicate J. In order to enable a more intuitive understanding of Equation 2, we will derive Equation 1 using the assumptions listed above. For each distinct value, there are $\|R_i\|/d_i$ tuples in R_i having that value in the join column (from the uniformity assumption). Using the containment assumption, we get that the number of distinct values that appear in the join columns of both the tables is $\min(d_1, d_2)$. But these are precisely the values for which the join takes place. Hence, the number of tuples in the intermediate result $(R_1 \bowtie R_2)$ is:

$$\|(R_1 \bowtie R_2)\| = \min(d_1, d_2) \times \frac{\|R_1\|}{d_1} \times \frac{\|R_2\|}{d_2} = \|R_1\| \times \|R_2\| \times \frac{1}{\max(d_1, d_2)}$$

In [12], Rosenthal showed that Equation 1 holds even if the uniformity assumption is weakened to require expected uniformity for only one of the join columns.

Equation 1 can be generalized for n relations to get Equation 3 below. Let R_1^n denote the result of joining tables $R_1, R_2, \cdots, R_n$ on columns $x_1, x_2, \cdots, x_n$ respectively. Here there are equality predicates between each pair of columns and in the terminology of equivalence classes, all the $x_i s$ are in a single equivalence class, i.e., are j-equivalent. Denote the column cardinality of x_i by d_i for $i \in 1 \cdots n$. Let $d_{(1)}, d_{(2)}, \cdots, d_{(n)}$ be a rearrangement of $d_1, d_2, \cdots, d_n$ in increasing order as $d_{(1)} \leq d_{(2)} \leq \cdots \leq d_{(n)}$. Then, such that $d_{i_1}, d_{i_2}, \cdots, d_{i_n}$ are ordered in an ascending sequence. Then,

$$d_{(1)} = \min(d_{(1)}, d_{(2)}, \cdots, d_{(n)}) = \min(d_1, d_2, \cdots, d_n)$$

Then, using the assumptions listed above, it can be shown that the size of the intermediate result R_1^n is:

$$\|R_1^n\| = d_{(1)} \times \frac{\|R_1\|}{d_1} \times \frac{\|R_2\|}{d_2} \times \cdots \times \frac{\|R_n\|}{d_n}$$
$$= \frac{\|R_1\| \times \|R_2\| \times \cdots \times \|R_n\|}{d_{(2)} \times d_{(3)} \times \cdots \times d_{(n)}}, \quad \text{where } d_{(1)} = \min(d_1, d_2, \ldots, d_n) \quad \text{(Equation 3)}$$

that is, all column cardinalities except for the smallest one are present in the denominator.

Example 1b Continuing with Example 1a, let the statistics for the tables be as follows:

$$\|R_1\| = 100, \|R_2\| = 1000, \|R_3\| = 1000, d_x = 10, d_y = 100, d_z = 1000$$

The join selectivities from Equation 2 are:

J1: $S_{J1} = 1/\max(d_1, d_2) = 1/\max(10, 100) = 0.01$
J2: $S_{J2} = 1/\max(d_2, d_3) = 1/\max(100, 1000) = 0.001$
J3: $S_{J3} = 1/\max(d_1, d_3) = 1/\max(10, 1000) = 0.001$

If R_2 is joined with R_3, the size of the intermediate result table can be estimated using Equation 2:

$$\|R_2 \bowtie R_3\| = \|R_2\| \times \|R_3\| \times S_{J2} = 1000 \times 1000 \times 0.001 = 1000$$

Using Equation 3, the size of $(R_1 \bowtie R_2 \bowtie R_3)$ is estimated to be:

$$\|R_1 \bowtie R_2 \bowtie R_3\| = \frac{100 \times 1000 \times 1000}{100 \times 1000} = 1000$$

which is the correct answer.

3 Problems in Join Size Estimation

There are a number of problems with current practice in join size estimation. The problems include dealing with local predicates correctly, correct handling of multiple columns in a single table that belong a single equivalence class, and correct incremental estimation of join result sizes.

3.1 Effect of Local Predicates

If a local predicate is present, the local predicate may reduce the cardinality of the table. This reduced cardinality is called the *effective* cardinality of the table. It can be used in other cardinality calculations, such as for estimating join result sizes. When the local predicate is on a join column, the predicate can also reduce the column cardinality of the join column. Thus, it is clear that local predicates will affect join result sizes, since the predicates affect both the number of participating tuples and, possibly, the column cardinality of the join columns. It is evident that an algorithm is needed for taking into account the

effect of local predicates when estimating join result sizes. We do not know of any algorithm that *correctly* takes into account both local predicates and join predicates. In Section 5, we describe how to correctly take into account the effect of local predicates in join size estimation.

3.2 Single Table J-Equivalent Columns

During the incremental calculation of join result sizes, it can happen that two or more of the eligible join predicates involve join columns of the next table that are j-equivalent. For example, consider a table R_1 joining with another table R_2 using the following join predicates: $(R_1.x = R_2.y)$ and $(R_1.x = R_2.z)$. By transitive closure, we get the predicate $(R_2.y = R_2.z)$. When R_1 is joined with R_2, all three predicates are eligible and involve columns y and z of R_2 that are j-equivalent by virtue of the predicate $(R_2.y = R_2.z)$.

Current query optimizers do not treat this as a special case and therefore, by default, the join selectivities of all such join predicates are used in calculating the join result sizes. However, using them produces an incorrect result because the join columns are not independent. In Section 6, we describe an algorithm that correctly handles the case when two or more of the join columns of a relation are j-equivalent.

3.3 J-Equivalent Columns from Different Tables

Another problem that arises in incremental computation is that the effect of all the eligible predicates involving different tables is not independent. In Example 1b, once join predicates J1 and J2 have been evaluated, J3 has in effect been evaluated and hence, evaluating it separately can have no further effect on the estimated size of the join result. Yet, in current practice, its selectivity will be included in the result size estimate.

In [13], the selectivities of all the eligible join predicates are multiplied together along with the product of the individual effective table cardinalities (as reduced by the selectivities of local predicates). We call this the *multiplicative* rule ($\mathcal{M}$). This rule can compute an incorrect join result size because all the join selectivities are used without accounting for dependencies. Rule $\mathcal{M}$ can dramatically underestimate the join result size as shown in Example 2.

Example 2 As in Example 1b, let R_2 and R_3 be joined first, and then R_1. From Example 1b, we have $\|R_2 \bowtie R_3\| = 1000$. When R_1 is joined, Rule $\mathcal{M}$ estimates the join result size to be:

$$\begin{aligned}\|R_2 \bowtie R_3 \bowtie R_1\| &= \|R_2 \bowtie R_3\| \times \|R_1\| \times S_{J1} \times S_{J3} \\ &= 1000 \times 100 \times 0.01 \times 0.001 = 1 \quad \text{(correct answer is 1000)}\end{aligned}$$

The problem of underestimation of join result sizes by Rule $\mathcal{M}$ suggests dividing the eligible join predicates into groups, with join predicates involving j-equivalent columns being grouped together, and then choosing for each group a single join selectivity. The problem appears to be analogous to the problem of picking among multiple local predicates on a single column [16]. Thus, one would expect to pick the *smallest* join selectivity value for each group. We refer to this

idea as the smallest selectivity rule ($\mathcal{SS}$). Though this seems to be the intuitive choice, this choice of join selectivity can compute an incorrect join result size as shown in Example 3.

Example 3 As in Example 1b, let R_2 and R_3 be joined first, and then R_1. From Example 1b, we have $\|R_2 \bowtie R_3\| = 1000$. When R_1 is to be joined, Rule $\mathcal{SS}$ puts predicates J1 and J3 in a single group, since columns x, y and z are j-equivalent. Since $S_{J3} < S_{J1}$, it uses only predicate J3 in calculating the join result size. Thus, Rule $\mathcal{SS}$ estimates the join result size as follows:

$$\begin{aligned}\|R_2 \bowtie R_3 \bowtie R_1\| &= \|R_2 \bowtie R_3\| \times \|R_1\| \times S_{J3} \\ &= 1000 \times 100 \times 0.001 = 100 \quad \text{(correct answer is 1000)}\end{aligned}$$

A third procedure has been proposed but, to our knowledge, has not been implemented in any optimizer. The proposal is to assign a *representative* join selectivity to each equivalence class and to use that selectivity whenever join predicates in that equivalence class are being used. The problem with this proposal is that there is no certainty that a correct value for this representative join selectivity exists that will work in all cases. In our example query, if the representative selectivity is 0.01, the estimate for the final join result size will be 10000, which is too high. If the representative selectivity is 0.001, the estimate for the final join result size will be 100, which is too low. In Section 7, we describe a new rule for correctly choosing the join selectivities when undertaking an incremental estimation of join result sizes.

4 Algorithm ELS

We describe an algorithm called **ELS** (**E**quivalence and **L**argest **S**electivity) that solves the problems described in the previous section. Algorithm **ELS** consists of two phases. The first is a preliminary phase that is to be performed before any join result sizes are computed. In this phase, the predicates implied due to transitive closure are generated and join selectivities are calculated. The preliminary phase consists of steps 1 through 5. The second phase computes, incrementally, the join result sizes. The processing for this final phase is outlined in step 6.

1. Examine each given predicate.
   - Remove any predicate that is identical to another predicate, so that queries involving duplicate predicates such as (R_1.x > 500) AND (R_1.x > 500) can be handled.
   - Build equivalence classes for all columns that are participating in any of the predicates.
2. Generate all implied predicates using transitive closure. There are five variations:
   a. Two join predicates can imply another join predicate.
      (R_1.x = R_2.y) AND (R_2.y = R_3.z) $\Longrightarrow$ (R_1.x = R_3.z)
   b. Two join predicates can imply a local predicate.
      (R_1.x = R_2.y) AND (R_1.x = R_2.w) $\Longrightarrow$ (R_2.y = R_2.w)

c. Two local predicates can imply another local predicate.
$(R_1.\mathrm{x} = R_1.\mathrm{y})$ AND $(R_1.\mathrm{y} = R_1.\mathrm{z}) \Longrightarrow (R_1.\mathrm{x} = R_1.\mathrm{z})$

d. A join predicate and a local predicate can imply another join predicate.
$(R_1.\mathrm{x} = R_2.\mathrm{y})$ AND $(R_1.\mathrm{x} = R_1.\mathrm{v}) \Longrightarrow (R_2.\mathrm{y} = R_1.\mathrm{v})$

e. A join predicate and a local predicate can imply another local predicate. Here *op* denotes a comparison operator, and c is a constant.
$(R_1.\mathrm{x} = R_2.\mathrm{y})$ AND $(R_1.\mathrm{x}\ op\ c) \Longrightarrow (R_2.\mathrm{y}\ op\ c)$

3. Assign to each local predicate a selectivity estimate that incorporates any distribution statistics. In [16], we present a complete algorithm to handle the case of multiple local predicates on a single column. In essence, the most restrictive equality predicate is chosen if it exists, otherwise we chose a pair of range predicates which form the tightest bound.
4. For each table, compute an estimate of the table cardinality and the column cardinality of each column after all the local predicates have been included. We show that if these new estimates are correctly used in the computation of join selectivities and join result sizes, we do not need to concern ourselves with local predicates after this step. This is discussed further in Section 5.
5. Process each join predicate by computing its join selectivity. If two join columns from the same table are j-equivalent, special care is needed in the computation of the join selectivities. We describe an algorithm for handling this case in Section 6.
6. For each intermediate result of the join order, estimate the result size. We have invented a new rule, which is described in Section 7, to estimate the result size. We also prove that the rule is correct under the assumptions stated in Section 2.

5 Effect of Local Predicates on Join Result Sizes

In Section 3, we described the need to incorporate the effect of local predicates in the estimation of join result sizes. Let R be one of the tables participating in a join. Let a join column of R be x and let a local predicate involve column y of R. Here, y could be identical with x. Let $\|R\|$ be the cardinality of R before the local predicate is applied, and $\|R\|'$ be the cardinality of R after the local predicate is applied. Let d_x denote the column cardinality of join column x before the local predicate is applied, and d'_x denote the column cardinality of column x after the local predicate is applied. Similarly, we can define d_y and d'_y.

The estimates of $\|R\|'$, d'_x, and d'_y depend on the local predicate and the information available about the column y. For example, if the local predicate is of the form $y = a$ where a is a literal, we know that $d'_y = 1$. If we have distribution statistics on y, they can be used to accurately estimate $\|R\|'$. Otherwise, we can use the uniformity assumption (see Section 2) to estimate $\|R\|'$ (i.e., $\|R\|' = \|R\|/d_y$). For some other local predicate L involving y, if the local predicate selectivity S_L is known, we can estimate $\|R\|' = \|R\| \times S_L$ and $d_y{}' = d_y \times S_L$.

Now that we have estimated $\|R\|'$ and d'_y, we can estimate d'_x as follows. If y is identical with x, clearly $d'_x = d'_y$. If y and x are different columns, we use a simple urn model to estimate d'_x. The process is one of assigning $\|R\|'$

balls among d_x urns that are initially empty. Any of the urns can be chosen uniformly. The balls correspond to the selected tuples and the urns correspond to the distinct values in column x. The number of distinct values in x remaining after selection corresponds to the number of non-empty urns. We obtain the expected number of non-empty urns as follows, where $n = d_x$ and $k = \|R\|'$.

$$\begin{aligned}
&\text{Prob}\{\text{ball is put in urn } i\} && = 1/n \\
&\text{Prob}\{\text{ball is not put in urn } i\} && = (1 - 1/n) \\
&\text{Prob}\{\text{none of the } k \text{ balls is put in urn } i\} && = (1 - 1/n)^k \\
&\text{Prob}\{\text{at least one of the } k \text{ balls is put in urn } i\} && = (1 - (1 - 1/n)^k)
\end{aligned}$$

Then, the expected number of non-empty urns is given by $n \times (1 - (1 - 1/n)^k)$. Hence in the case where column x is distinct from column y, we have that

$$d'_x = \left\lceil d_x \times (1 - (1 - 1/d_x)^{\|R\|'}) \right\rceil$$

Note that if $\|R\|'$ is sufficiently large enough and approaches $\|R\|$, the term $(1-1/d_x)^{\|R\|'}$ will be close to 0 and $d'_x \approx d_x$. This estimate can be quite different from another common estimate $d'_x = d_x \times (\|R\|'/\|R\|)$. This can be seen from the following numerical example. Let $d_x = 10000$, $\|R\| = 100000$ and $\|R\|' = 50000$. Then the urn model estimate gives $d'_x = 9933$ whereas the other estimate gives $d'_x = 5000$. If $\|R\|' = \|R\|$, the estimate according to the urn model is $d'_x = 10000$.

At this point, the table and column cardinality estimates have already incorporated all local predicates. The rest of the estimation algorithm only has to deal with join predicates. We have thus taken into account the effect of local predicates and simplified subsequent processing. We can also take advantage of distribution statistics on columns that are involved in local predicates. This enables us to obtain more accurate estimates of join result sizes. The original, unreduced table and column cardinalities are retained for use in cost calculations before the local predicates have been applied, for example, when estimating the cost of accessing the table.

We have assumed that if the effect of a local predicate on the table cardinality and the column cardinalities of join columns is taken into account, then the local predicate is, in effect, applied before or together with the join predicates on this table. This being true, then the following argument shows that our approach to size estimation is correct. Consider all the local and join predicates being used to form an intermediate result. For the estimation of the result size, it does not matter in which order the predicates are applied. We can assume that the local predicates are applied first, evaluate their effects on the table statistics, and then compute the size obtained by applying the join predicates. Hence, in the following discussion, we will focus on join predicates and assume that local predicates can be handled as described above.

6 J-Equivalent Join Columns in a Single Table

Algorithm **ELS** needs to deal with the special case described in Section 3.2. Consider the following example query, involving columns from the same table that are j-equivalent.

WHERE $(R_1.\mathrm{x} = R_2.\mathrm{y})$ AND $(R_1.\mathrm{x} = R_2.\mathrm{w})$

After transitive closure of predicates, we obtain the following tranformed query:

WHERE $(R_1.\mathrm{x} = R_2.\mathrm{y})$ AND $(R_1.\mathrm{x} = R_2.\mathrm{w})$ AND $(R_2.\mathrm{y} = R_2.\mathrm{w})$

where the implied local predicate $(R_2.\mathrm{y} = R_2.\mathrm{w})$ is added using rule 2.a from Section 4. Let the statistics for the tables in the query be as follows: $\|R_1\| = 100$, $\|R_2\| = 1000$, $d_x = 100$, $d_y = 10$, $d_w = 50$. After transitive closure, $R_2.\mathrm{y}$ participates in every join in which $R_2.\mathrm{w}$ participates and vice versa. Hence, for computing join selectivities and join result sizes, we have to know the number of R_2 tuples that can qualify under the local predicate and also the number of distinct values that the qualifying tuples contain. We wish to quantify the effect of the local predicate on the effective cardinality of R_2 and the column cardinalities prior to evaluating the effect of the join predicates on the result sizes. To do so, we use a probabilistic argument as follows.

Without loss of generality, let $d_w >= d_y$. Let $\|R_2\|$ denote the table cardinality prior to join predicates being applied. We use the containment assumption for join columns (see Section 2) to infer that all the d_y distinct values are contained in the d_w distinct values. Now, consider any tuple of R_2. It has some value, say $\mathbf{q}$, in column $R_2.y$. Assuming the independence and uniformity assumptions for columns $R_2.w$ and $R_2.y$, the probability that $R_2.w$ has the same value $\mathbf{q}$ is $1/d_w$. Since this is the case for every tuple of R_2, the effective cardinality of R_2 is given by $\|R_2\|'$, where

$$\|R_2\|' = \left\lceil \frac{\|R_2\|}{d_w} \right\rceil$$

Once this selection has been performed, only one of the columns needs to be joined since the evaluation of the local predicate has made the other join redundant. Using a simple urn model as in Section 5, we deduce that the column cardinality value that should be used for join selectivity computations is:

$$\left\lceil d_y \times (1 - (1 - 1/d_y)^{\|R_2\|'}) \right\rceil$$

that is, we pick the column with the most restrictive effect on the column cardinality. In our example query, $\|R_2\|' = \|R_2\|/d_w = 1000/50 = 20$. We pick $R_2.\mathrm{y}$ and the effective column cardinality in joins is hence,

$$\left\lceil d_y \times (1 - (1 - 1/d_y)^{\|R_2\|'}) \right\rceil = \lceil 10 \times (1 - (1 - 1/10)^{20}) \rceil = 9$$

The formulas given above can be generalized to the case of three or more join columns, from the same table, that are j-equivalent. Let $1, 2, \cdots, n$ be the equivalent join columns of table R, and let the column cardinalities be $d_1, d_2, \cdots, d_n$. Let $d_{(1)}, d_{(2)}, \cdots, d_{(n)}$ be a rearrangement of $d_1, d_2, \cdots, d_n$ in increasing order as $d_{(1)} \leq d_{(2)} \leq \cdots \leq d_{(n)}$. Then,

$$d_{(1)} = \min(d_{(1)}, d_{(2)}, \cdots, d_{(n)}) = \min(d_1, d_2, \cdots, d_n)$$

Using an argument similar to the one above, it follows that

$$\|R\|' = \left\lceil \frac{\|R\|}{d_{(2)} \times d_{(3)} \times \cdots \times d_{(n)}} \right\rceil$$

and that the effective column cardinality in joins is

$$\left\lceil d_{(1)} \times (1 - (1 - 1/d_{(1)})^{\|R\|'}) \right\rceil$$

7 Incremental Computation of Result Sizes

We saw in Section 3 that the "intuitive" rules currently used in query optimizers (Rule $\mathcal{M}$ and Rule $\mathcal{SS}$) do not correctly choose the join selectivities in the incremental estimation of join sizes. We present a new rule (called $\mathcal{LS}$) for picking one of the join selectivities for an equivalence class.

$\mathcal{LS}$: *Given a choice of join selectivities for a single equivalence class, always pick the* **largest** *join selectivity.*

Rule $\mathcal{LS}$ appears counter-intuitive and a proof is provided in [16].

Multiple equivalence classes can be accommodated by using the independence assumption. Since the join columns in different equivalence classes are assumed to satisfy the independence assumption, each equivalence class independently reduces the join result size. The algorithm then determines the join selectivity for each equivalence class and multiplies them together to obtain the effective join selectivity.

Example 3 As in Example 1b, let R_2 and R_3 be joined first, and then R_1. From Example 1b, we have $\|R_2 \bowtie R_3\| = 1000$. When R_1 is to be joined, Rule $\mathcal{LS}$ puts predicates J1 and J3 in a single group, since columns x, y and z are j-equivalent. Since $S_{J3} < S_{J1}$, it uses only predicate J1 in calculating the join result size. Thus, Rule $\mathcal{LS}$ estimates the join result size as follows:

$$\begin{aligned} \|R_2 \bowtie R_3 \bowtie R_1\| &= \|R_2 \bowtie R_3\| \times \|R_1\| \times S_{J1} = 1000 \times 100 \times 0.01 \\ &= 1000 \quad \textbf{(correct)} \end{aligned}$$

As before, denote the result of joining tables $R_1, R_2, \cdots, R_n$ by R_1^n. Let us say that we have calculated the size of R_1^n. We now wish to join table R_{n+1} to obtain the result R_1^{n+1}. We now prove that Rule $\mathcal{LS}$ is correct by showing that the algorithm calculates the join result size in agreement with the size calculated by Equation 3 for a single equivalence class. Recall that we have already discussed how to handle multiple equivalence classes. The proof is by induction on the number of tables.

8 Experiment

We have presented Algorithm **ELS** and shown that it correctly estimates of join result sizes. In this section, we illustrate that correct estimation of join sizes can make a significant difference in query execution times. We use a straightforward select-project-join query that is shown below. For this query, Algorithm **ELS** can result in an order of magnitude improvement in query execution time.

SELECT COUNT()
FROM S, M, B, G
WHERE s = m AND m = b AND b = g AND s < 100

The tables used in the query are S (*s*mall), M (*m*edium), B (*b*ig) and G (*g*iant). A single column from each table participates in the query and is denoted by the table name in lower case, e.g., column s from table S. The query after transitive closure is transformed as follows.

SELECT COUNT()
FROM S, M, B, G
WHERE s = m AND m = b AND b = g AND s = b AND s = g AND
m = g AND s < 100 AND m < 100 AND b < 100 AND g < 100

The table and column cardinalites are given below.

$\|S\| = 1000$, $\|M\| = 10000$, $\|B\| = 50000$, $\|G\| = 100000$
$d_s = 1000$, $d_m = 10000$, $d_b = 50000$, $d_g = 100000$

The query optimizer in the Starburst DBMS [9] was modified to implement the multiplicative rule (Rule $\mathcal{M}$), the smallest selectivity rule (Rule $\mathcal{SS}$), and our Algorithm **ELS** (with Rule $\mathcal{LS}$). Predicate transitive closure (PTC) was implemented as a query rewrite rule [11] so that we could disable it as necessary for the experiments. Note that both Rule $\mathcal{SS}$ and Rule $\mathcal{LS}$ are sensible only when predicate transitive closure has been applied. No other changes were made to the optimizer and the optimizer's entire repertoire was enabled (including the Nested Loops and Sort Merge join methods).

The standard algorithm most commonly in use in current relational systems computes join selectivities independent of the effect of local predicates. Let Algorithm **SM** denote using Rule $\mathcal{M}$ with the standard algorithm. Similarly, let Algorithm **SSS** denote using Rule $\mathcal{SS}$ with the standard algorithm.

We ran four experiments. The original query, before predicate transitive closure, was run using Algorithm **SM**. Next, the query after predicate transitive closure was applied, was run with Algorithm **SM** and Algorithm **SSS**. Finally, the original query was run with Algorithm **ELS**. The results are shown in the table below which identifies the query and rewrite rule used, the algorithm used, the join order chosen by the algorithm, the result sizes estimated after each join, and the elapsed time for the chosen query evaluation plan (QEP) in seconds. Note that all the QEPs were executed using the same buffer size and that the access methods and join methods did not differ between the QEPs. Both Nested Loops and Sort Merge join methods were used in the QEPs.

Query	Algorithm	Join Order	Estimated Result Sizes	Time
Original	**SM**	$S \bowtie M \bowtie B \bowtie G$	(100, 100, 100)	610
Orig. + PTC	**SM**	$S \bowtie B \bowtie M \bowtie G$	$(0.2, 4 \times 10^{-8}, 4 \times 10^{-21})$	472
Orig. + PTC	**SSS**	$S \bowtie B \bowtie M \bowtie G$	$(0.2, 4 \times 10^{-4}, 4 \times 10^{-7})$	472
Orig.	**ELS**	$B \bowtie G \bowtie M \bowtie S$	(100, 100, 100)	50

We see that there is an improvement in query execution time when predicate transitive closure is used (see the first two rows in the above table). Predicate transitive closure is important because the additional predicates permit early selection and greater flexibility in the selection of the join order. This may result in the generation of better QEPs.

However, using predicate transitive closure is not enough. After predicate transitive closure is used, the QEPs chosen differ in the join order. This is due to the different estimates obtained for the intermediate join result sizes. The QEP chosen by **ELS** runs 9 - 12 times faster than the other QEPs.

The correct join result size after any subset of joins has been performed can be shown to be exactly 100. We can see in the second and third rows of the table that Algorithm **SM** and Algorithm **SSS** greatly underestimate the join result sizes. The optimizer estimates that the size of the 3-table composite result involving tables S, M, and B is extremely small. This misleads the optimizer into believing that table G will be accessed very infrequently. Thus, the optimizer puts table G last in the join order, which in turn leads to a poor QEP.

9 Summary

Accurate estimation of join result sizes is crucial in query optimization because the estimates have a significant influence on the query plans chosen. This paper described an algorithm (called **ELS**) for the correct incremental estimation of join result sizes. For each table, estimates of the table cardinality and the column cardinality of each column are computed, while taking into account the effect of local predicates and the case when two or more join columns in a table belong to the same equivalence class. These estimates are then used in the computation of join selectivities. For any intermediate result, Algorithm **ELS** chooses a correct subset of the eligible join predicates to determine the result size, while taking into account any dependencies between eligible join predicates. We proved the correctness of Algorithm **ELS**. We have shown that using the algorithm can make a significant difference in query execution time (possibly an order of magnitude or more).

Future work involves relaxing the assumptions that are used in this paper and in most of the work on join size estimation. We have already relaxed the uniformity assumption in the case of local predicates. Relaxing the assumption in the case of join predicates would enable query optimizers to account for important data distributions such as the Zipfian distribution. The independence assumption needs to be relaxed. All published work on estimation has dealt with the case of a single SQL query block. Estimation in nested SQL queries is a much harder problem. Work also needs to be done to extend the results in this paper to queries involving disjunctions.

Acknowledgements

We wish to thank Paul Bird, Philip Blair, Peter Haas, Richard Hedges, Bruce Lindsay, Guy Lohman, Sheila Richardson, Pat Selinger and Lori Strain for their comments on various drafts of this paper.

References

1. S. Christodoulakis. Estimating Block Transfers and Join Sizes. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 40–54, 1983.
2. S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, June 1984.
3. C. Faloutsos and H. V. Jagadish. On B-tree Indices for Skewed Distributions. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 363–374, Vancouver, British Columbia, 1992. Morgan Kaufman.
4. Y. E. Ioannidis and S. Christodoulakis. On the Propogation of Errors in the Size of Join Results. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 268–277, Denver, Colorado, 1991.
5. Y.C. Kang. *Randomized Algorithms for Query Optimization.* PhD thesis, University of Wisconsin-Madison, October 1991. TR 1053.
6. C. A. Lynch. Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 240–251, Los Angeles, USA, 1988. Morgan Kaufman.
7. M. V. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Database Systems. *ACM Computing Surveys*, 20(3):191–221, September 1988.
8. M. Muralikrishna and D. J. Dewitt. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 28–36, Chicago, Illinois, 1988.
9. K. Ono and G. M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 314–325, Brisbane, Australia, 1990. Morgan Kaufman.
10. G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 256–276, 1984.
11. H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 39–48, San Diego, California, 1992.
12. A. Rosenthal. Note on the Expected Size of a Join. *ACM-SIGMOD Record*, pages 19–25, July 1981.
13. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 23–34, 1979.
14. A. Swami. *Optimization of Large Join Queries.* PhD thesis, Stanford University, June 1989. STAN-CS-89-1262.
15. A. Swami and B. Iyer. A Polynomial Time Algorithm for Optimizing Join Queries. In *Proceedings of IEEE Data Engineering Conference*, pages 345–354. IEEE Computer Society, April 1993.
16. A. Swami and K. B. Schiefer. On the Estimation of Join Result Sizes. Technical report, IBM Research Division, October 1993. IBM Research Report RJ 9569.
17. G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Reading, MA, 1949.