Exploiting Statistics on Query Expressions for Optimization

Nicolas Bruno*
Columbia University
nicolas@cs.columbia.edu

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Statistics play an important role in influencing the plans produced by a query optimizer. Traditionally, optimizers use statistics built over base tables and assume independence between attributes while propagating statistical information through the query plan. This approach can introduce large estimation errors, which may result in the optimizer choosing inefficient execution plans. In this paper, we show how to extend a generic optimizer so that it also exploits statistics built on expressions corresponding to intermediate nodes of query plans. We show that in some cases, the quality of the resulting plans is significantly better than when only base-table statistics are available. Unfortunately, even moderately-sized schemas may have too many relevant candidate statistics. We introduce a workload-driven technique to identify a small subset of statistics that can provide significant benefits over just maintaining base-table statistics. Finally, we present experimental results on an implementation of our approach in Microsoft SQL Server 2000.

1. INTRODUCTION

Most query optimizers for relational database management systems (RDBMS) rely on a cost model to choose the best possible query execution plan for a given query. Thus, quality of the query execution plan depends on the accuracy of cost estimates. Cost estimates, in turn, crucially depend on cardinality estimations of various sub-plans (intermediate results) generated during optimization. Traditionally, query optimizers use statistics built over base tables for cardinality estimates, and assume independence while propagating these base-table statistics through the query plans (see Section 2 for a detailed discussion). However, it is widely recognized that such cardinality estimates can be off by orders of magnitude [21]. Therefore, the traditional propagation of statistics that assumes independence between attributes can lead the query optimizer to choose significantly low-quality execution plans.

In this paper, we introduce the concept of *SIT*s, which are *statistics built on attributes of the result of a query expression* [1]. Thus, SITs can be used to accurately model the distribution of tuples on *intermediate* nodes in a query execution plan. We will show that in some cases, when optimizers have appropriate SITs available during query optimization, the resulting query plans are drastically improved, and their execution times are tens, and even hundreds of times more efficient than those of the plans produced when only base-table statistics are used.

Despite the conceptual simplicity of SITs, significant challenges need to be addressed before they can be effectively used in existing RDBMS. First, we must show how query optimizers can be adapted to exploit SITs for choosing better execution plans. Next, we must address the problem of identifying appropriate SITs to build and maintain. The latter is a nontrivial problem since for a moderate schema sizes, there can be too many syntactically relevant SITs. Finally, we need to address the issue of efficiently building and maintaining SITs in a database system.

In this paper, we take the first steps towards meeting these challenges. While we briefly comment on the last issue, we primarily focus on the first two issues referred to above. We explain how a traditional relational query optimizer can be modified to take advantage of SITs. Identifying whether or not a SIT is applicable for a given query can leverage materialized view matching technology. But, as we will discuss, specific SITs applications have no counterpart in traditional materialized view matching. Another desirable goal is to ensure that the cardinality estimation module of an optimizer is modified as little as possible to enable the use of SITs. We have implemented such an optimizer by modifying the server code of Microsoft SQL Server 2000. However, the ideas introduced in this paper are general in the sense that the proposed algorithms do not depend on the specific structure of statistics used in a RDBMS (e.g., type of histogram).

We show how an appropriate set of SITs may be chosen to maximize the benefit to the query optimizer. We recognize that usefulness of SITs depends on how their presence impacts execution plans for queries against the system. Therefore, it is necessary to take into account *workload* information while selecting SITs. However, evaluating the effectiveness of SITs for queries in the workload leads us to a "chicken and egg" problem, as it is hard to determine effectiveness of a SIT until it has been built. In this paper, we present a novel technique to identify useful SITs based on information on *workload* analysis that has desirable property that we *do not necessarily need to build a SIT* to evaluate its effectiveness. Our technique can be seen as a non-trivial generalization of the MNSA algorithm [6], which selects statistics on stored tables only. We demonstrate experimentally that the plans produced using the set of SITs chosen by our algorithm is close in quality to the plans produced using all possible SITs, and considerably better than the plans obtained using statistics only on stored tables.

*Work done in part while the author was visiting Microsoft Research.

[1] The obvious acronym SQE (Statistics on Query Expressions) is not quite as nice as SIT(Statistics on Intermediate Tables). So, we decided to be safe and pick a nicer acronym rather than being technically accurate.

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

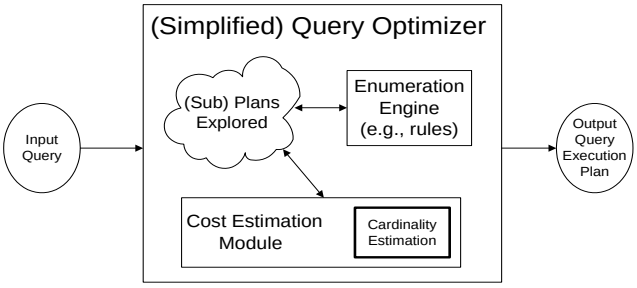


Figure 1: Simplified Optimizer's Architecture.

This work contributes to the broader goal of automating statistic management for RDBMSs and was done as part of the *AutoAdmin* project at Microsoft Research [2]. The goal of this project is to reduce the total cost of ownership of relational database systems by making them self-tuning.

The rest of the paper is structured as follows. In Section 2 we give a brief overview of cost and cardinality estimation in current query optimizers. In Section 3 we introduce the concept of SITs and a framework to incorporate them in existing query optimizers. In Section 4 we propose a workload-driven technique to efficiently select a good subset of SITs that can significantly improve quality of execution plans produced by the optimizers.

2. BACKGROUND

The query optimizer is the component in a database system that transforms a parsed representation of an SQL query into an efficient execution plan for evaluating it. Optimizers usually examine a large number of possible query plans and choose the best one in a cost-based manner. To efficiently choose among alternative query execution plans, query optimizers *estimate* the cost of each evaluation strategy. This cost estimation needs to be accurate (since the quality of the optimizer is correlated to the quality of its cost estimations), and efficient (since it is invoked repeatedly during query optimization). In this section we describe the components of a generic query optimizer and show how statistical information can be used to improve the accuracy of cost estimations, which in turn impacts the whole optimization process.

2.1 Architecture of a Query Optimizer

There are several optimization frameworks in the literature [10, 11, 12, 14, 20] and most modern optimizers rely on the concepts introduced by those references. Although the implementation details vary among different systems, all optimizers share the same basic structure [4], shown in Figure 1. For each incoming query, the optimizer maintains a *set of sub-plans* already explored, taken from an implicit search space. An *enumeration engine* navigates through the search space by applying rules to the set of explored plans. Some optimizers have a fixed set of rules to enumerate all interesting plans (e.g., System-R) while others implement extensible transformational rules to navigate through the search space (e.g., Starburst, Cascades). All systems use dynamic programming or memoization to avoid recomputing the same information during query optimization. For each discovered query plan, a component derives different properties if possible, or estimates them otherwise. Some properties (e.g., cardinality and schema information) are shared among all plans in the same equivalence class, while others (e.g., estimated execution cost and output order) are tied to a specific physical plan. Finally, once the optimizer has explored all interesting plans, it extracts the most efficient plan, which serves as the input for the execution engine.

A useful property of a query plan from an optimization perspective is the estimated execution cost, which ultimately decides which is the most efficient plan. The estimated execution cost of a plan, in turn, depends heavily on the cardinality estimates of its sub-plans. Therefore, it is fundamental for a query optimizer to rely on accurate and efficient cardinality estimation algorithms.

EXAMPLE 1. *Consider the query in Figure 2(a) and suppose that* $|R| \approx |S| \approx |T|$. *If the query optimizer has knowledge that* $R.a < 10$ *is much more selective than* $T.b > 20$ *(i.e., just a few tuples in* R *verify* $R.a < 10$ *and most of the tuples in* T *verify* $T.b > 20$*), it should determine that plan* P_1 *in Figure 2(b) is more efficient than the plan* P_2 *in Figure 2(c)* [3]. *The reason is that* P_1 *first joins* R *and* S *producing a (hopefully) small intermediate result that is in turn joined with* T. *In contrast,* P_2 *produces a large intermediate result by first joining* S *and* T. ∎

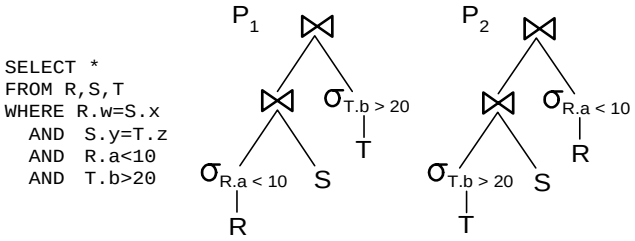


Figure 2: Query plans chosen by query optimizers depending on the cardinality of intermediate results.

Cardinality estimation uses statistical information about the data that is stored in the database system to provide estimates to the query optimizer. Histograms are the most common statistical information used in commercial database systems. In the next section we show how they are currently used to estimate the cardinality of complex query plans during optimization.

2.2 Cardinality Estimation using Histograms

A histogram on attribute x consists of a set of buckets. Each bucket b_i represents a sub-range r_i of x's domain, and has associated two values. The frequency f_i of bucket b_i corresponds to the number of tuples t in the data set for which $t.x \in r_i$, and the value dv_i of bucket b_i represents the number of distinct values of $t.x$ among all the tuples t for which $t.x \in r_i$. The main assumption is that the distribution of tuples inside each histogram bucket is uniform. We use the uniform spread model inside buckets, in which each bucket b_i is composed of dv_i equidistant groups of f_i/dv_i tuples each. We define the *density* of a bucket as $\delta_i = f_i/dv_i$, i.e., the number of tuples per distinct value (assuming uniformity) that are represented in the bucket. Other techniques for modelling bucket contents can also be used, such as the continuous or randomized models. We now describe how histograms are used to estimate the cardinality of queries.

2.2.1 Selection Queries

The uniformity assumption inside histogram buckets suggests a natural interpolation-based procedure to estimate the selectivity of range and join predicates. The situation is particularly simple for range predicates. Consider the query $\sigma_{R.a<20}(R)$ and suppose we have a histogram on $R.a$. To estimate the cardinality of such a

[2] http://research.microsoft.com/dmx/autoadmin.

[3] We assume that no indexes, thus no interesting orders, are available.

query, we consider, one at a time, all histogram buckets that are completely or partially covered by the predicate, and then we aggregate all intermediate results. This procedure is illustrated below.

EXAMPLE 2. *Consider the four-bucket histogram on attribute $R.a$ of Figure 3. Bucket b_1, for instance, covers $0 \leq x < 10$ and has a frequency of 100 (i.e., it represents 100 tuples in the data set). Similarly, buckets b_2, b_3, and b_4 represent $50, 80$, and 100 tuples, respectively. Suppose we want to estimate the cardinality of the range predicate $p = R.a < 20$. Since p completely includes bucket b_1, we can guarantee that the 100 tuples in b_1 verify p. Also, p is disjoint with buckets b_3 and b_4, so no single tuple in b_3 or b_4 verifies p. Finally, p partially overlaps with bucket b_2 (in particular, p is verified by 50% of b_2's uniformly spread distinct values). Using our assumption, we estimate that 50% of the tuples in b_2 verify p. In summary, the number of tuples verifying predicate $p = R.a < 20$ is estimated as $100 + 50/2 = 125$.* ∎

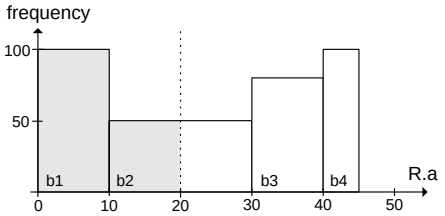


Figure 3: Range selectivity estimation using histograms.

In general, selection queries may have multiple predicates on different attributes of the table. For example, consider the query:

```
SELECT * FROM R
WHERE R.a>10 AND R.b<100
```

and assume that we have histograms on $R.a$ and $R.b$ available. If s_a is the selectivity for $R.a > 10$ and s_b is the selectivity for $R.b < 100$, the selectivity for the whole predicate is estimated, assuming independence, as $s_a \cdot s_b$. Multidimensional histograms [3, 13, 15, 17] have proved to be accurate in modeling attribute's correlation. It should be noted, however, that these novel estimation techniques are not widely used in commercial databases yet.

2.2.2 Join Queries

Let us consider the join predicate $R \bowtie_{x=y} S$. We can use histograms on $R.x$ and $S.y$ (if available) to improve the accuracy of the cardinality estimation. Consider histograms $H_{R.x}$ and $H_{S.y}$ in Figure 4, where each bucket is delimited by square brackets. The procedure to estimate the cardinality of the join predicate using histograms $H_{R.x}$ and $H_{S.y}$ is composed of three steps.

In the first step, we *align* the histogram buckets so that their boundaries agree (usually splitting some buckets from each histogram). For instance, buckets b_2 and b_2' in the figure share the same left boundary. However, bucket b_2 spans beyond bucket b_2''s right boundary. Therefore, we split bucket b_2 into two sub-buckets. The left sub-bucket boundary agrees with that of bucket b_2'. The right sub-bucket starts at the same position as bucket b_3' but ends before b_3' does. Then, bucket b_3' is split in the same way, and this procedure continues until all original buckets are aligned (see Step 1 in Figure 4). This step is guaranteed to at most double the total number of buckets in both histograms.

In the second step, we analyze each pair of aligned buckets and do *per bucket estimation of join sizes*. There is no well-founded approach towards doing it and here we sketch one of the approaches. First, using the *containment assumption* [20], it is concluded that each group of distinct valued tuples belonging to the bucket with the minimal number of different values joins with some group of tuples in the other bucket. For instance, in Step 2 of Figure 4, the three group of tuples from the upper bucket are assumed to match with three of the five group of tuples in the lower bucket. We can model the result of joining the pair of buckets as a new bucket with three distinct values and density $40 = 2 \cdot 20$. That is, each distinct value in the resulting bucket represents 40 tuples, which is the product of the original bucket densities. Therefore, the frequency of the new bucket is $120 = 3 \cdot 40$.

After applying the same procedure to each pair of aligned buckets, the third and last step consists of aggregating the partial frequencies from each resulting bucket to get the cardinality estimation for the whole join.

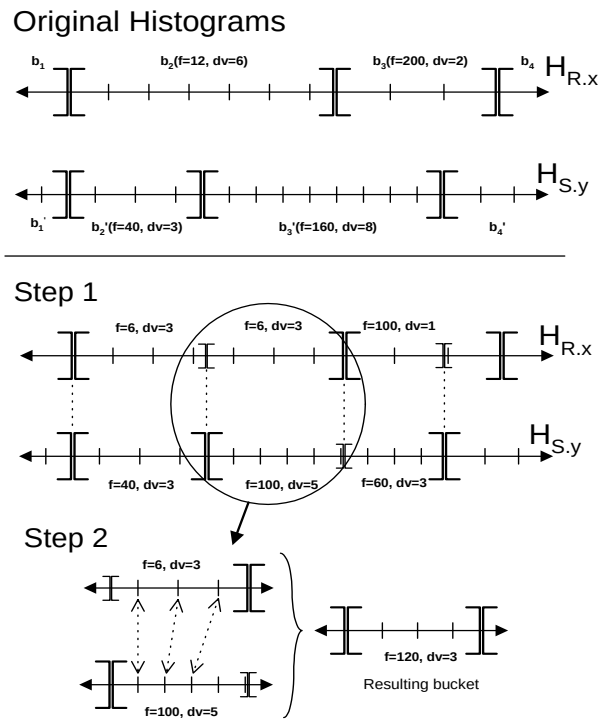


Figure 4: Join selectivity estimation using histograms.

2.2.3 Select Project Join (SPJ) Queries

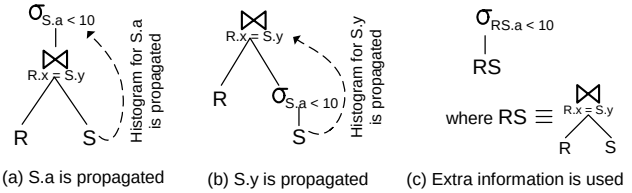
The techniques in the previous section are used when the predicates are directly applied to the base tables that hold the histograms involved. When considering arbitrary SPJ queries, we face the additional challenge that cardinality estimation requires *propagating statistics* through predicates. Consider the query:

```
SELECT * FROM R,S
WHERE R.x=S.y AND S.a<10
```

and assume that we have histograms on $R.x$, $S.y$ and $S.a$ available. There are conceptually two ways to estimate the selectivity of the whole expression. On one hand (Figure 5(a)), histograms for $R.x$ and $S.y$ are used to estimate the selectivity of $R \bowtie S$ ignoring the predicate $S.a < 10$. Then, *assuming independence* between $S.y$ and $S.a$, the histogram for $S.a$ is propagated [4] through the join

[4]The histogram propagation step just scales the bucket frequencies so that they reflect the new selectivity information. In this case, the frequency values for histogram $S.a$ are scaled so that the sum of all frequencies in the histogram equals the estimated number of tuples in $R \bowtie S$.

upwards in the tree. The propagated histogram is then used to estimate the selectivity of $S.a < 10$ over the result from $R \bowtie S$, to finally obtain the selectivity of $\sigma_{S.a<10}(R \bowtie S)$. On the other hand (Figure 5(b)), the histogram for $S.a$ is used first to estimate the selectivity of $\sigma_{S.a<10}(S)$. Then, assuming independence between $S.y$ and $S.a$, the histogram for $S.y$ is propagated through the selection operator and used together with the histogram of $R.x$ to estimate the selectivity of $R \bowtie (\sigma_{S.a<10}(S))$. It is important to note that, although the two methods above estimate the same expression, i.e., $R \bowtie (\sigma_{S.a<10}(S)) \equiv \sigma_{S.a<10}(R \bowtie S)$, the resulting estimations can be slightly different.



(a) S.a is propagated (b) S.y is propagated (c) Extra information is used

Figure 5: Histogram propagation in complex queries.

In this paper we focus on the propagation of statistics through predicates, and study the sensitivity of the query optimizer to the quality of statistics in intermediate nodes of query execution plans. For that purpose, we introduce in the next section the notion of SITs and evaluate the impact of this structure in the optimizer's choice of query execution plans.

3. SIT: STATISTICS ON QUERY EXPRESSIONS

In this section we introduce the concept of *statistics on query expressions*, or *SIT*s, which help eliminate the propagation of errors through the query plan operators. As an example, consider again Figure 5, and assume that we build statistics on the result of the *query expression* $RS = R \bowtie S$, specifically on $RS.a$. In this case (see Figure 5(c)), we can estimate the cardinality of the original query plan by simply estimating the cardinality of the equivalent plan $\sigma_{RS.a<10}(RS)$. Thus, we avoid propagating estimation errors through the join predicate. For complex query plans, the beneficial effect of having statistics on *query expressions that match intermediate subexpressions* of the query is magnified since we can avoid the propagation of errors through a sequence of operators. We now formalize the concept of statistics over query expressions.

DEFINITION 1. *Let R be a table, A an attribute of R, and Q an SQL query that contains $R.A$ in the* `SELECT` *clause. We define SIT$(R.A|Q)$ as the statistic for attribute A on the result of executing query expression Q. We call Q the* generating query expression *of SIT$(R.A|Q)$.*

The above definition is easily extended for multi-attribute statistics. Furthermore, the definition can be used as the basis for extending the `CREATE STATISTICS` statement in SQL where instead of specifying the table name of the query, more general query expression (i.e., a table valued expression) can be used. We omit the specifics of such formulation in this paper.

How to build and update SITs is an interesting problem in itself. The obvious approach to build SITs is executing the query expression associated with the SIT and building the necessary statistics on the result of the query. Once the statistics has been computed, the result of the query expression can be discarded. When explicitly requested or triggered by the system, updating of the statistics can be accomplished by recomputation and rebuilding of the statistics. However, for a large class of query expressions, more efficient techniques drawn from the wide body of work in *approximate query processing* can be used. This is possible because we only need to obtain statistical distributions instead of exact results. For instance, the construction of SITs with generating queries consisting of foreign-key joins can be efficiently performed by using sampling. Furthermore, existing indexes and statistics can also be leveraged for efficient computation of SITs. We defer a comprehensive study of these alternatives to future work.

After introducing SITs, two important questions need to be answered. The first one is how to leverage an existing query optimizer so that it incorporates SITs during query optimization. We address this issue in the next section, by presenting a general framework that can be incorporated into existing query optimizers. The second question is how to select *which* SITs to build. We address this complementary issue in Section 4.

3.1 A Framework to Exploit SITs

SITs are only useful if the optimizer is able to incorporate them during query optimization. We enable use of SITs by implementing a wrapper on top of the original cardinality estimation module of the RDBMS. During the optimization of a single query, the wrapper will be called many times, once for each different query sub-plan enumerated by the optimizer (see Figure 1). Each time the query optimizer invokes the modified cardinality estimation module with a query plan, we transform this input plan into another one that exploits SITs. Then, we obtain a potentially more accurate cardinality estimation for the modified query plan, and return it to the query optimizer. We should note that the transformed query plan is simply a temporary structure used by the modified cardinality estimation module, and is *not* used for query execution. In summary, the modified cardinality estimation module should verify the following properties to be effectively integrated into an existing query optimizer:

1. The transformed plan should exploit applicable SITs, so that its estimated cardinality is potentially more accurate than the original one.

2. The original cardinality estimation module should be able to take the transformed plan as an input with only a few changes.

3. The transformation should be efficient, since it will be used for several sub-plans during a single query optimization.

As we will see, in the simplest case the original (unmodified) cardinality estimation module could be used with the transformed plan. For more complex transformations, we might need to slightly modify the cardinality estimation module. This issue is further explored in Section 3.2.3. We now discuss cardinality estimation using SITs in detail.

3.2 Cardinality Estimation using SITs

For the rest of this paper, we will make some simplifying assumption about incoming queries (to be optimized) as well as on the class of query expressions used to generate SITs. We assume that incoming queries are `SELECT-PROJECT-JOIN (SPJ)` queries where the filter expression is a conjunction of simple predicates. We assume that SITs are constrained to belong to the above class of queries as well. The four steps in the modified cardinality estimation module are summarized as follows: (1) analyze the input query plan, (2) identify and apply relevant SITs, and (3) estimate

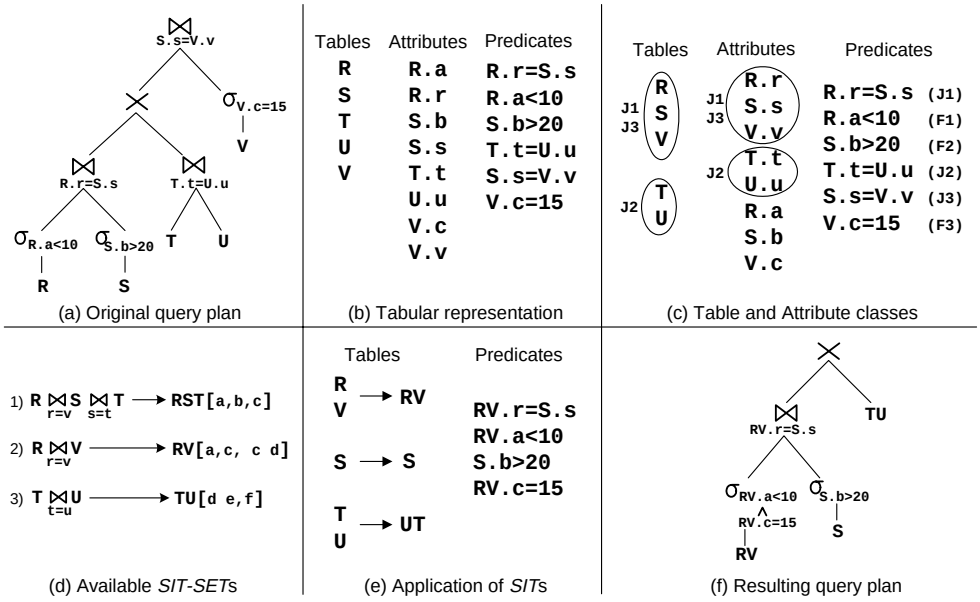


Figure 6: Example of the transformation algorithm in the modified cardinality estimation module.

and return the cardinality of the transformed query plan. In the remainder of this section, we explain these steps in detail using as a running example the query in Figure 6(a).

3.2.1 *Analysis of the Input Query Plan*

In this first step we perform simple structure analysis on the input query q that will later help to identify and apply SITs to q. For instance, in our running example we first identify the tables and columns referenced in the query, and the list of conjunctive predicates (see Figure 6(b)). Next, we classify predicates as either *filter predicates* or *join predicates*. We use the equality join predicates to generate column equivalence classes and also to get the set of table subsets that are joined. Figure 6(c) shows the results of this step in the running example. The filter predicates are marked with an F label, and the join predicates with a J label. Tables R, S and V are joined using predicates J_1 and J_3, and tables T and U are joined using join predicate J_2. In a similar way, columns $R.r$, $S.s$ and $V.v$ form one equivalence class, columns $T.t$ and $U.u$ form another equivalence class, and the remaining columns form singleton classes. More complex analysis can be performed in this step, depending on the set of rewriting transformations that we apply later.

3.2.2 *Application of Relevant* SITs

For ease of notation, we represent the set of available SITs by using *SIT-Sets*, which basically group SITs by their generating query expressions. Suppose we have SIT$(Q|a_1), \ldots,$ SIT$(Q|a_n)$. These SITs can be compactly represented by the following SIT-Set:

$$Q \rightarrow \mathcal{S}[a_1, \ldots, a_n]$$

where $\mathcal{S}$ is the SIT-Set identifier that holds the set of statistics $\{$SIT$(Q|a_1), \ldots,$ SIT$(Q|a_n)\}$. Consider a query (or its plan) q and a SIT-Set $\mathcal{S}$ defined by the generating query expression Q. SIT-Set $\mathcal{S}$ is potentially useful for cardinality estimation of q if some attributes a_i are referenced in the selection condition of q and there is an "occurrence" of Q in q [5]. To verify the latter, we can use well-known algorithms for materialized view matching (e.g., [5, 9, 19]). Thus, we check if q can be rewritten using $\mathcal{S}$, and if so generate a new query expression (e.g., see Figure 5(c)). In such cases, we can simply forward the rewritten query expression to a traditional cardinality estimator module. During cardinality estimation, $\mathcal{S}$ will be treated as a base table having statistics on columns a_i. This requires a change in the system metadata so that $\mathcal{S}$ is treated as a *hypothetical* table [8]. Furthermore, note that such rewriting is exclusively used for cardinality estimation and *not* for plan generation.

Exploiting Multiple SIT-Sets. In general, more than one SIT-Set may be applicable to a query expression. Consider Figure 6(d), which lists the SIT-Sets that are available. Consider the second SIT-Set that uses as generating expression the join $R \bowtie_{r=v} V$ [6]. This SIT-Set can be used for the query in Figure 6(a). Likewise, the third SIT-Set can be applied in conjunction with the first one (see Figure 6(e)). The resulting query plan is shown in Figure 6(f), for which traditional cardinality estimation can be used as described in the previous paragraph. Note that no SIT is used for attribute $S.b$, so a base-table statistic (if available) will be used in that case.

The above example illustrates the case where using one SIT-Set does not interfere with the use of another. We now discuss examples where application of SIT-Sets may not be compatible. For example, assume that a fourth SIT-Set, $R \bowtie S \rightarrow RS[a, b]$, is added in Figure 6(d). Whenever SIT-Set RST is applicable for a query, then so is RS. However, RST will be favored over RS. The reason is that while estimating the cardinality of the query transformed using RST we make *fewer independence assumption* compared to using RS (as explained in Section 2.2.3, use of the independence assumption is responsible for error propagation). Note that RS may be applicable in cases where RST is not. These considerations are similar to the case of materialized view matching.

A more complex scenario occurs when the use of one SIT-Set results in a "rewriting" that excludes the use of other SIT-Sets which can still be useful to improve cardinality estimation. This is illustrated in the example below.

[5] We do not give a formal definition for "occurrence", as it is a well understood concept in materialized view matching.

[6] This SIT-Set shows the use of single-column SITs on attributes a and c, and a multi-column SIT on attributes (c, d). The statistical object associated with a multi-column SIT will have the same structure as any multi-column statistics on base tables.

EXAMPLE 3. *Consider the following SIT-Sets:*

$$\begin{array}{lcl} S \bowtie_{s=t} T & \rightarrow & \mathcal{ST}[b,c] \\ R \bowtie_{r=s} S \bowtie_{r=t} T & \rightarrow & \mathcal{RST}[a] \\ R \bowtie_{r=c} T & \rightarrow & \mathcal{RT}[a,c] \end{array}$$

and suppose we want to estimate the cardinality of the query:

```
SELECT * FROM R,S,T
WHERE R.r=S.s AND S.s=T.t AND
      R.a<10  AND T.c>20
```

SIT-Set $\mathcal{RST}$ can be applied to the given query. (Note that the join predicate $R.r = T.t$ in $\mathcal{RST}$'s generating query is equivalent to the join predicate $S \bowtie_{s=t} T$ in the given query modulo column equivalence.) We then apply $\mathcal{RST}$, replacing $R \bowtie_{r=s} S \bowtie_{s=t} T$ in the query with the SIT-Set $\mathcal{RST}$. In this case, SIT$(a|\mathcal{RST})$ will be used for the filter condition $R.a < 10$, but SIT$(c|\mathcal{RST})$ is not available. However, we can use SIT$(c|\mathcal{ST})$ from SIT-Set $\mathcal{ST}$ to avoid assuming independence between $T.c$ and $T.t$ (note, however, that we are implicitly assuming independence between $T.c$ and $R \bowtie \mathcal{ST}$). We are allowed to use SIT$(c|\mathcal{ST})$ because $\mathcal{ST}$ is compatible with $\mathcal{RST}$'s generating query. Note that in this case, we cannot use SIT$(c|\mathcal{RT})$ from SIT-Set $\mathcal{RT}$ due to the join predicate $R.r = T.c$ in its generating query. ∎

The example above underscores the point that simple materialized view rewriting is not sufficient in some cases, since such rewriting cannot account for the use of statistics such as SIT$(c|\mathcal{ST})$ in the example. Therefore, when considering application of any given SIT-Set $\mathcal{S}$ to a query q, the following steps are taken. First, we verify that $\mathcal{S}$'s generating query is applicable to q and determine a "rewriting" that uses the SIT-Set. Then, for each attribute of q that potentially affects cardinality estimation but is not covered by $\mathcal{S}$ (i.e., it occurs in one or more predicates of q but it is not among the attributes for which $\mathcal{S}$ provides statistics), we look for a SIT that would provide the best alternative for estimation. Such SIT *must* come from a SIT-Set whose generating query is subsumed by the original SIT-Set's generating query, or the result might not be correct. In particular, if many options exist, we select the SIT that would require the fewest number of independence assumptions when we estimate the cardinality of the resulting query. Our attempt to minimize the number of applications of the independence assumption is justified since precisely independence assumptions are the source of error propagation for cardinality estimation. We refer to such additional SITs as *auxiliary SITs due to application of SIT-Set $\mathcal{S}$ to query q*. In some cases, no such auxiliary SITs may be necessary.

In order to minimize the number of applications of independence assumptions in the resulting query, we have adopted a greedy heuristic to determine the SIT-Sets and auxiliary SITs that should be applied for a given input query. For each SIT-Set $\mathcal{S}$, we consider rewriting the query with $\mathcal{S}$ and at the same time identify the set of auxiliary SITs that are applicable. Next, we count the number of independence assumptions that must be made by a traditional cardinality estimator if we apply the given SIT-Set and its auxiliary SITs to the input query. This provides a *score* for each SIT-Set, and we select the SIT-Set with the lowest score. After applying the selected SIT-Set, we repeat the procedure until no new SIT-Sets qualify. This is summarized in the pseudo-code below.

```
01 while more SIT-Sets can be applied to the query q
02   Select the SIT-Set compatible with q that
       minimizes the number of applications of the
       independence assumption
03   Apply the selected SIT-Set and auxiliary SITs
```

As an example, assume that all SIT-Sets' generating queries consist only of joins (no selections), and the attributes in the predicates of the input query plan are $\{a_1, \ldots, a_k\}$. It is not difficult to see that the number of independence assumptions is minimized when each attribute uses a SIT with the maximal number of joined tables in its generating query. In such scenario, we need to find the SIT-Set (and auxiliary SITs) that maximize the value $\sum_{i=1}^{k} |Ant_i|$, where $|Ant_i|$ is the number of joined tables in the generating query expression that provides the SIT for attribute a_i. (The value of $|Ant_i|$ for an attribute that does not use a SIT is set to one if such attribute has a base-table statistic available, or zero otherwise.)

3.2.3 Actual Estimation

In this last step, we get the estimated number of tuples in the transformed query and return this value to the optimizer. It is important to note again that we do not use the transformed query *outside* the modified cardinality estimation module (otherwise it would cause problems since some tables are hypothetical and do not really exist in the system).

As discussed in the previous section, for some simple query transformations the original cardinality estimation module does not need to change at all except for the need to use hypothetical tables for cardinality estimation. For more complex query transformations, however, we would need to do some modifications to the cardinality estimation module. For instance, to handle the auxiliary SITs of Section *3.2.2*, we would need to augment the cardinality estimation module with *statistical hints*, which detail specifically which statistic in the system to use for specific attributes. A full discussion of these details is beyond the scope of this paper.

3.3 An Illustrative Experiment

In this section we show with a simple example the effectiveness of using SITs during query optimization. For that purpose, we used the popular TPC-H benchmark schema [22]. One of the requirements of the benchmark, however, is that the data is generated from a *uniform* distribution. Likewise, there is a constraint in the number of foreign key joins per tuple (e.g., each `order` tuple has associated n `lineitem` tuples, where n is a random integer between one and seven). Our techniques are meaningful in the very common case of skewed data distributions (where the simple histogram propagation mechanisms tend to introduce large estimation errors). For that reason, we extended the TPC-H generation program to support data generation with varying degree of *skew*. In particular, the generator produces data for each column in the schema from a zipfian distribution (similar to the modifications proposed in [6]). Zipfian distributions are also applied to foreign key joins, so for instance the number of tuples in `lineitem` that join with each tuple in `orders` follows a zipfian distribution.

We generated the TPC-H data sets using a skew factor $z = 1$ and a resulting size of 100MB. Consider the following SQL query, which asks for information about the most expensive orders (those with a total price greater than 1,000,000):

```
SELECT * FROM lineitem, orders, part, supplier
WHERE l_orderkey = o_orderkey and
      l_partkey = p_partkey and
      l_suppkey = s_suppkey and
      o_totalprice > 1000000
```

In our database $|\sigma_{\texttt{o\_totalprice}>1000000}(\texttt{orders})| = 120$, i.e., 120 out of 750,000 tuples in `orders` verify the filter condition (the selectivity is lower than 0.02%). However, precisely those tuples are joined with a very large number of tuples in `lineitem` (that is the reason they are so expensive). In fact, we have that

$\left|\sigma_{\texttt{o\_totalprice}>1{,}000{,}000}\ (orders \bowtie lineitem)\right| = 971,851$ out of 2,943,815 tuples (the selectivity is around 33%). Clearly, if we simply propagate the histogram for `o_totalprice` through the join `lineitem` ⋈ `orders`, we will incur in large estimation errors, which in turn will affect the optimizer's choice of an execution plan.

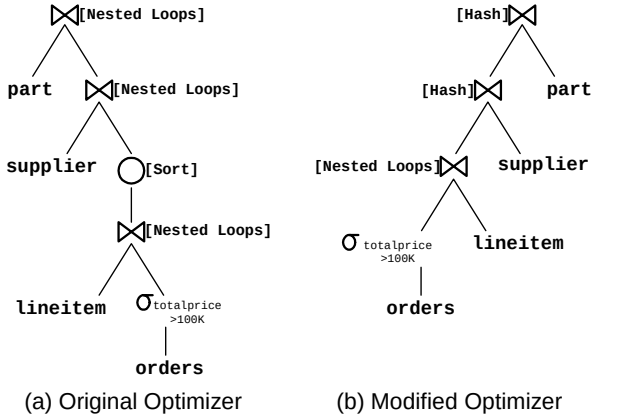


Figure 7: Query execution plans.

We optimized the query above using the original query optimizer and the one that incorporates the framework of Section 3.1. We made available to the query optimizer all possible SITs. When we optimized the query using the original optimizer, we obtained the query execution plan in Figure 7(a). In this scenario, the optimizer estimates that the result size of the subquery `lineitem` ⋈ $\sigma_{\texttt{o\_totalprice}>1{,}000{,}000}(\texttt{orders})$ is small (only 713 tuples), therefore chooses to sort this intermediate result before pipelining it to the next nested loop join with `supplier`. Since the estimated intermediate result is still small, another nested loop join is used with `part` to obtain the final result. In contrast, the modified query optimizer (Figure 7(b)) accurately estimates that the number of tuples in `lineitem` ⋈ `orders` is large (970,627 tuples) and chooses a different set of operators. In particular, the expensive sort operation is removed and the nested loop joins are replaced with the (more efficient) hash joins (in some cases, the inner/outer role of the tables is reversed). Figure 8 shows the execution time of both query plans broken down in CPU time and I/O time (the shown times are averaged over five independent executions). The actual elapsed time of the original plan in Figure 7(b) was 419 seconds. In contrast, the plan produced by the modifier optimizer incurred in an elapsed time of only 23 seconds (less than 6% of the time spent by the original plan). In this example, the modified optimizer that uses SITs dramatically reduces the execution time of the given query.

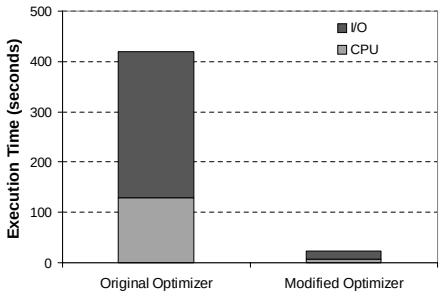


Figure 8: Elapsed execution times.

4. AUTOMATED SELECTION OF SITS

In Section 3.3 we showed that we can substantially improve the quality of execution plans of existing query optimizers if statistical information about intermediate nodes in the query sub-plans is made available. However, building SITs for all possible intermediate results is not viable even for moderate schema sizes: loading many statistics and incrementally maintaining them can be very expensive. Therefore, an important problem is to select a small subset of SITs that are sufficient to increase the quality of the query plans produced by the optimizer. One approach to address this problem is to take into consideration workload information. In other words, the problem statement becomes: given a query workload and a space constraint, find the set of SITs that fits in the available space so that the actual cost from answering queries in similar workloads is minimized (or at least substantially reduced). Note that other criteria besides space, such as update cost, could be relevant for such selection.

In this section we present a novel algorithm to choose a small subset of SITs in such a way that it does not compromise the quality of plans chosen by the optimizer. We will consider in turn each attribute a_i that occurs in the filter predicates of the input queries, and obtain the optimized query plans assuming that attribute a_i has different skewed hypothetical distributions [7] (see Section 4.2). Intuitively, for a given attribute a_i, if the estimated difference in cost of the obtained query plans (assuming different distributions for a_i) is close to zero, the introduction of more detailed information (SITs) on a_i will result in little effect, if any, on the quality of plans chosen by the optimizer. In contrast, if the cost difference is significant, chances are that a SIT over attribute a_i can provide relevant information and help the optimizer to choose the correct query plan. Our technique can be seen as a very significant generalization of the *Magic Number Sensitivity Analysis* (MNSA) technique [6] that is able to consider SITs (see Section 4.1 for a description of MNSA). However, even if we determine that the presence of a SIT on attribute a_i could improve the quality of plans chosen by the query optimizer, we still need to identify *which* generating query should we use for attribute a_i. We address this issue in Section 4.3. Although the main concepts in our techniques can be applied to general queries, in the rest of the section we focus on a workload consisting of `SPJ` queries.

4.1 Magic Number Sensitivity Analysis

The workload-based MNSA technique [6] significantly reduces the set of base-table statistics that need to be created in a database system without sacrificing the quality of generated query plans. A relaxed notion of plan equivalence is exploited to make this selection. In particular, two plans p_1 and p_2 are *t-Optimizer-Cost equivalent* if the query optimizer predicts that the execution costs of p_1 and p_2 are within t percent of each other, where t reflects the degree of rigor used to enforce equivalence.

For a given a workload, the MNSA algorithm incrementally identifies and builds new statistics over the base tables until it determines that no additional statistic is needed. To test whether the current subset of statistic is enough for estimation purposes, MNSA considers how the presence of such statistics would impact optimization of queries without building statistics first. For this purpose, MNSA replaces the magic selectivity numbers, which are used by the optimizer in absence of statistics, with extremely small and large values (in practice, ε and $1-\varepsilon$, with $\varepsilon = 0.0005$). It then verifies whether the optimized query plans are insensitive, i.e., t-Optimizer-Cost equivalent, to those changes. Under reasonable

[7] This step is analogous to the magic number replacements in MNSA.

assumptions, if the query plans obtained by using these extreme predicted selectivities for all attributes without statistics are cost equivalent, then all actual plans for which the actual selectivities lie between those extremes will be t-Optimizer-Cost equivalent as well, and therefore the impact of materializing new statistics will be rather limited.

In our scenario, we assume that all needed base-table statistics were already materialized, either by using MNSA or some other equivalent procedure. However, we cannot apply directly MNSA to the problem of selection of SITs since the query optimizer does not rely on magic numbers for cardinality estimation of non-leaf expressions, i.e., simple variations of MNSA are not suitable for this generalized scenario. To overcome this limitation, in the next section we generalize the main ideas of MNSA by introducing novel estimation strategies that propagate cardinality information through query plans by making extreme assumptions about the distribution of attribute values.

4.2 Extreme Cardinality Estimation

We now introduce two new strategies to estimate cardinalities of SPJ query plans. As explained in Section 4, these estimation strategies make use of extreme hypothesis on the attribute distributions, and are the building blocks of our main algorithm for selecting a small set of SITs. In particular, we will focus on SPJ input queries and *histograms* as the choice for SITs, but the general ideas can be extended to other queries and statistical structures as well.

As explained in Section *2.2.2*, cardinality estimation routines assume independence between attributes and propagate statistics through query plans. We now illustrate this technique using the following query:

```
SELECT * FROM R,S
WHERE R.r=S.s AND S.a<10
```

Suppose that the cardinality of predicate $S.a < 10$ is estimated before the cardinality of the join (as in Figure 5(b)). In this case, histogram $S.s$ is *uniformly* scaled down so that the total number of tuples equals to the estimated cardinality of $S.a$. That is, if N is the number of tuples in table S, and N_a is the number of tuples that verify predicate $S.a < 10$, each bucket frequency from $S.s$'s histogram is multiplied by the factor $\frac{N_a}{N}$. After this transformation, $R.r$ and $S.s$'s histograms are used to estimate the cardinality of the join, as explained in Section *2.2.2*. We call this default estimation strategy *Ind with respect to* $S.a$ since we use the independence assumption for attribute $S.a$. In this section we introduce two new estimation techniques, *Min* and *Max* (with respect to some attribute), which make "extreme" assumptions about the statistical distribution of such attribute. In particular, instead of uniformly reducing the frequency of all tuples in histogram $S.s$, we selectively choose the N_a tuples in $S.s$ that survive the filter condition, so that the resulting join cardinality is the smallest (largest) possible *under the containment assumption*, as illustrated in the following example.

EXAMPLE 4. *Consider the already aligned histograms on attributes* $R.r$ *and* $S.s$ *for the query above, which are denoted in Figure 9 as* H_R *and* H_S*, respectively. For instance, there are three groups of* 20 *tuples each in the first bucket of histogram* H_S*, and two groups of* 10 *tuples each in the first bucket of histogram* H_R*. At the bottom of the figure we show the number of tuples that can be joined from each pair of buckets. For instance, the expression* $40S \times 10R$ *below the first pair of buckets specifies that* 40 *tuples in* S *(two groups of* 20 *tuples each) can be joined with* 10 *tuples in* R *each. In the same way, the expression* $20S \times 0R$ *specifies that for* 20 *tuples in* S *(the remaining group of tuples) there is no tuple in* R *that matches them. Now suppose that we know that only* 30 *tuples in* S *verify the filter predicate* $S.a < 10$*. Using the Max strategy, we choose the* 8 *tuples in* H_S*'s third bucket (since each tuple in that bucket joins with the largest number of tuples in* R*) and* 22 *out of the* 40 *tuples in* H_S*'s first bucket that join with* 10 *tuples in* R*. The estimated cardinality for the join is then:* $8 \cdot 50 + 22 \cdot 10 = 620$*. In contrast, using the Min strategy, we choose the* 20 *tuples in* $S.s$*'s first bucket that do not join with any tuple in* R*, and* 10 *out of the* 200 *tuples in* $S.s$*'s middle bucket. The estimated cardinality for the join is:* $20 \cdot 0 + 10 \cdot 5 = 50$*. For completeness, the Ind strategy scales down the densities for* $S.s$ *by the factor* $30/268$ *(*268 *is the cardinality of* S*), and therefore the estimated cardinality is* $2.23 \cdot 10 + 11.19 \cdot 5 + 0.44 \cdot 50 = 100$. ∎

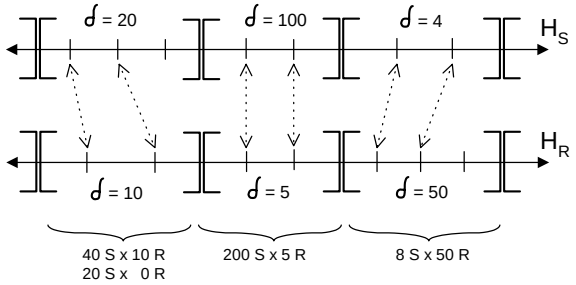


Figure 9: Extreme cardinality estimation routines selectively choose the matching tuples.

As hinted in the previous example, a simple procedure to select the appropriate tuples for strategy *Min* (*Max*) is to sort the list of pairs at the bottom of Figure 9 by increasing number of tuples in R, and select the first (last) N_a tuples in S from that sorted list. It can be proved that this procedure effectively chooses the set of tuples in S that minimize (maximize) the number of tuples in the join. These strategies are not limited to just one join predicate, but they can be easily extended to cope with multiple joins. Since both the *Min* and *Max* strategies return a cardinality value, we can use the output cardinality of one join as the input to the next join, in the same way as the traditional *Ind* strategy, to get an extreme cardinality estimation for the complete join. Consider the 5-way join represented in Figure 10, where each edge represents a join predicate between two tables, and suppose we want to get the *Max* cardinality estimation with respect to attribute $U.a$. To do so, we first get the cardinality of $\sigma_{U.a<10}$ using traditional techniques (suppose N_1 is such cardinality). We then apply the *Max* strategy for the join $T \bowtie U$, selecting the N_1 tuples in U so that the number of tuples in the result is maximized (suppose the new cardinality estimation for $T \bowtie U$ is N_2). We repeat the procedure by selecting the N_2 tuples in $(T \bowtie U)$ that maximize the cardinality result of $S \bowtie (T \bowtie U)$. We continue in this way (joining the accumulated result first with R and finally with V) to obtain the extreme cardinality estimation for the whole join. Of course, instead of the join order used in this example, any order that is consistent with the topological order in the join graph is possible.

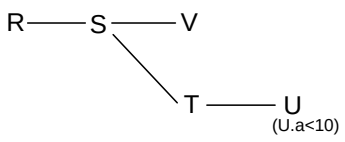


Figure 10: Chained extreme cardinality estimation.

The scheme described above works for join queries with a single filter predicate. In the general case, consider a SPJ query of the form $\sigma_{p_1 \wedge \ldots \wedge p_k}(R_1 \bowtie \ldots \bowtie R_n)$, and suppose we assign to each column attribute $a_1, \ldots, a_k$ an estimation strategy (*Min*, *Max*, or *Ind*). We can get the final cardinality estimation as follows:

```
01 Get the cardinality C of the join sub-query
     (R1 ⋈ ... ⋈ Rn)
02 For each filter pi with attribute ai, get the
     "partial" extreme selectivity si of query
     σpi(R1 ⋈ ... ⋈ Rn) as explained above.
03 Assuming independence multiply all "partial"
     selectivities with the join cardinality and
     return C · Πi si.
```

Note that in step 3 above we assume independence, but that is the best we can do in the absence of multi-column statistics. In the next section, we use these techniques in our algorithm for selecting SITs.

4.3 Selecting SITs

In this section we present our algorithm to choose a small subset of SITs in such a way that it does not compromise the quality of plans chosen by the optimizer. In particular, we will consider in turn each attribute a_i present in a query filter predicate, and obtain the estimated execution costs when a_i propagates through the query plan using the *Min* and *Max* strategies, and the remaining attributes use the *Ind* strategy (see Section 4.2). Intuitively, if for attribute a_i the difference in estimated cost between the two extreme strategies is close to zero, the introduction of any SIT on a_i will result in little or no effect on the quality of plans produced by the optimizer. In contrast, if the cost difference is significant, chances are that a SIT over attribute a_i can provide relevant information and help the optimizer to choose better quality query plans. Besides, this very difference in estimated execution cost is a good estimator of the relative importance of the different attributes, and can be used to rank the candidate SITs.

However, once we identified a promising attribute to build a SIT on, we still need to choose *which* generating query to use for such SIT. As an example, consider again Figure 10, and suppose we obtain a large difference in estimated execution cost for the *Min* and *Max* strategies with respect to attribute $U.a$. This difference in estimated execution cost might come from correlation between attribute $U.a$ and another attribute in an intermediate join. In other words, we need to determine which SIT over $U.a$ to build among several candidates, such as SIT($U.a|T \bowtie U$) or SIT($U.a|S \bowtie T \bowtie U$), among others.

For this purpose, we will exploit the *Min* and *Max* extreme cardinality estimation strategies as follows. Consider the query $q = \sigma_{U.a<10}(R \bowtie S \bowtie T \bowtie U)$. When estimating the cardinality of q using the *Max* and *Min* strategies with respect to $U.a$, we also get for free the partial approximate cardinalities of the intermediate queries $\sigma_{U.a<10}(U)$, $\sigma_{U.a<10}(T \bowtie U)$, and $\sigma_{U.a<10}(S \bowtie T \bowtie U)$ (this sequence is based on the join order used in the extreme cardinality estimation strategies). At no extra cost, we can also obtain the cardinality of the pure join queries $U, T \bowtie U, \ldots, R \bowtie S \bowtie T \bowtie U$. Combining these cardinalities, we obtain the minimal and maximal partial selectivities of the join predicates, which are graphically represented in the example of Figure 11 (each point in the x-axis corresponds to a different join, and we assume a fixed natural join order). For instance, for the base table U, both the minimal and maximal estimated selectivities are 0.55, since they are taken from the base-table statistic for $U.a$. However, each join increments the possible range of selectivities, and consequently, the propagated estimation error. The estimated selectivity for the whole join ranges between 0.25 and 0.85. However, most of this range is inherited from the previous join $S \bowtie (T \bowtie U)$. In effect, the last join does not introduce large variations in selectivity when using the *Min* and *Max* strategies.

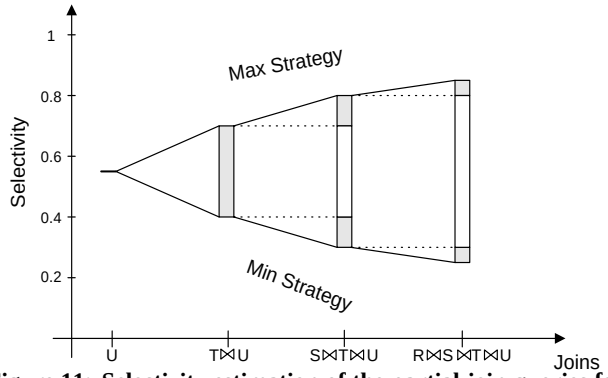


Figure 11: Selectivity estimation of the partial join queries for the *Min* and *Max* strategies.

We make the simplifying assumption that for a fixed attribute, the relative importance of a join query (and therefore the importance of a candidate SIT) is proportional to the increase of uncertainty of the selectivity estimation with respect to the previous join in the sequence. That is, if for some particular operator the minimal and maximal selectivity estimations change significantly, it is more likely that this particular operator has some correlation with the filter predicate for which we are considering building statistics. Using that assumption, the effect of building and exploiting SIT($U.a|R \bowtie S \bowtie T \bowtie U$) would be limited in Figure 11. In contrast, since $T \bowtie U$ substantially increases the range of possible selectivities for the query, so SIT($U.a|T \bowtie U$) should be one of the first candidates to build.

We now make these ideas concrete, starting with the simplest case of a single SPJ query, and then generalizing the results for workloads consisting of multiple SPJ queries. Consider the input query $q = \sigma_{p_1 \wedge \ldots \wedge p_k}(R_1 \bowtie \ldots \bowtie R_n)$ and assume that predicate p_i references attribute a_i. For simplicity, further assume that the attribute a_i we are interested in belongs to table R_1 and the join order that the *Min* and *Max* strategies consider is $R_1, \ldots, R_n$. In this case, the candidate SITs for attribute a_i are SIT($a_i|R_1$), ..., SIT($a_i|R_1 \bowtie \ldots \bowtie R_n$). We define the score of SIT($a_i|R_1 \bowtie \ldots \bowtie R_j$) relative to query q as:

$$Score_q\big(\text{SIT}(a_i|R_1 \bowtie \ldots \bowtie R_j)\big) =$$

$$= \begin{cases} 0 & \text{if } j = 1 \\ (E^{a_i}_{Max} - E^{a_i}_{Min}) \cdot \frac{\Delta^{a_i}_j - \Delta^{a_i}_{j-1}}{\Delta^{a_i}_n} & \text{otherwise} \end{cases}$$

where $E^{a_i}_{Min}$ and $E^{a_i}_{Max}$ are the estimated execution times for query q when using the *Min* (respectively, *Max*) strategy with respect to attribute a_i, and $\Delta^{a_i}_j = SelMax^{a_i}_j - SelMin^{a_i}_j$ is the difference in selectivity of $\sigma_{p_i}(R_1 \bowtie \ldots \bowtie R_j)$ when using the *Max* and *Min* strategies with respect to attribute a_i. The quantity $(\Delta^{a_i}_j - \Delta^{a_i}_{j-1})/\Delta^{a_i}_n$ varies from 0 to 1 and simply represents the fraction of selectivity, relative to the final selectivity range for the whole query, that is introduced by the $j-th$ join (the shaded regions in Figure 11). Clearly, the larger the score of a candidate SIT, the more likely that it makes a difference during query optimization.

Now we generalize this procedure to a workload that consists of several queries. In this situation, we maintain a hash table of SITs and we add to each SIT the partial scores obtained from each query

in the workload. Therefore, for a given workload W, the score $Score\big(\text{SIT}(a_i|Q)\big)$ is defined as $\sum_{q \in W} Score_q\big(\text{SIT}(a_i|Q)\big)$. After processing all queries, we select the top SITs according to the $Score$ value that fit in the available space. The following pseudocode summarizes these steps:

```
01 for each q in W and attribute a_i referenced in
     a filter condition p_i in query q
02   E_Min, E_Max = estimated cost for q using the
       Min, Max strategies with respect to a_i
03   Let R_1,... R_n be the join order used by the
       extreme strategies
04   SelMin_j^{a_i}, SelMax_j^{a_i} = selectivity of predicate
       σ_{p_i}(R_1 ⋈ ... ⋈ R_j) using Min, Max w.r.t.
       a_i for j ∈ 1...n (see pseudocode in Section 4.2)
05   for j = 2 to n
       Score[SIT(a_i|R_1 ⋈ ... ⋈ R_j)] +=
         (E_Max^{a_i} - E_Min^{a_i}) · (Δ_j^{a_i} - Δ_{j-1}^{a_i}) / Δ_n^{a_i}
           where Δ_j = SelMax_j^{a_i} - SelMin_j^{a_i}
06 Select the top statistics SIT(a_i|J_k) that fit in
     the available space
```

Discarding non-essential statistics. The algorithm that we describe above only *predicts* which statistics can be useful to the query optimizer. In practice, SITs with large scores can be false positives, i.e., the independence assumption might work fine. A post-processing step to discard SITs whose cardinality distributions are similar to those from which they were generated would be beneficial. In those cases, the independence assumption used by traditional optimizers is accurate, and we can use the resulting available space to build other (more useful) SITs. This task can be done with similar adaptations to our algorithm as in the MNSA/D technique [6], but in this paper we do not make use of such extensions.

5. EXPERIMENTAL STUDY

In this section, we present experimental results of an implementation of the framework proposed in this paper over Microsoft SQL Server 2000, and the algorithm for selecting a subset of SITs. We used as SITs the native statistics provided by Microsoft SQL Server for base tables: a variant of *MaxDiff* histograms which minimize intra-bucket frequency variance. The changes we made in the server were minimal: less than 20 lines in the optimizer's code needed to be modified to use our proposed wrapper. The wrapper itself is around 4,000 lines of code and is incorporated in the server as a new module. Finally, the algorithm to select SITs was implemented as a client connecting to the server via ODBC.

5.1 Setup

Database: We created a synthetic database with the star schema of Figure 12. Each node in the figure represents a table that consists of 500,000 tuples and each edge represents a foreign-key join. Each table is composed of four to eight attributes. Some attributes are uniformly distributed and others follow a zipfian distribution (with parameter z varying from 0.1 to 1). To verify the effectiveness of our algorithms, some attribute distributions are generated independently of the join attribute (so that the independence assumption is accurate), and others are correlated with the join attribute in a similar way as the `totalprice` attribute in Section 3.3 (so that the independence assumption could result in large estimation errors). We also used a database ten times larger than the original one, i.e., in which each table in Figure 12 contains 5,000,000 tuples. The results are almost identical to those we already presented, and are omitted for lack of space.

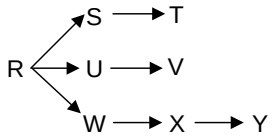


Figure 12: Star schema used in the experiments.

Workloads: For each experiment we generated two 100-query workloads, denoted *training* and *validation* workloads, taken from the same distribution. Each query in a given workload consists of three to seven joined tables and one to three filter predicates. The selectivity and attributes used in the filter predicates were randomly generated. For each experiment, we used our algorithm of Section 4.3 with the *training* workload, and built 100KB of SITs (that roughly corresponds to the top-25 SITs). We then optimized each query in the *training* workload using the following three optimization frameworks:

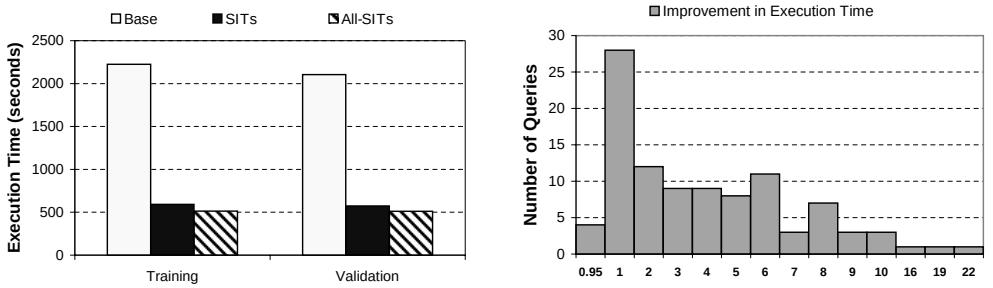
1. **Base**: The unmodified query optimizer that uses only base-table statistics.
2. **SITs**: The modified optimizer that also uses the SITs identified by our algorithm.
3. **All-SITs**: The modified optimizer for which we made available all possible SITs. This framework is used to evaluate the effectiveness of our algorithm for selecting SITs.

For each optimization framework described above, we evaluated the resulting query plans three times and averaged the elapsed times. Finally, we repeated the optimization and evaluation steps above for all the queries in the validation workload.

5.2 Results

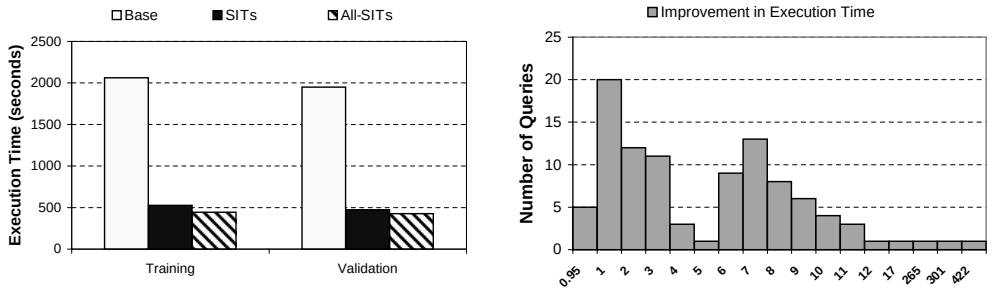
No indexes available: In this experiment we used the database described in Section 5.1 with no indexes available. Figure 13(a) shows the reduction in execution time for the whole workload when using SITs. The total execution time when using **SITs** is around 25% of that for **Base**. Also, when using **All-SITs** (in our case, more than 180 SITs were built) we obtain only marginal improvement in execution time (around 5%). This result validates our algorithm of Section 4.3 for selecting a subset of SITs. Figure 13(b) presents a histogram of the improvements in execution time for the queries in the validation workload. As an example, 12 queries reduced in half their execution times, and 6 queries had between 10- and 22-fold improvements. For the 4 queries that performed slightly worse in the **SITs** framework (with execution times less than 5% larger than those for **Base**), we checked the corresponding query plans and found out that they were the same for both the **Base** and **SITs** frameworks, so even in those situations **SITs** did not force the query optimizer to choose worse query execution plans. For two thirds of the workload there was a 2- to 22-fold improvement in execution time. In those situations, the chosen query plans varied considerably between the different optimization frameworks.

Indexes available. The Index Tuning Wizard in Microsoft SQL Server is a tool that automatically selects appropriate indexes for a given workload. We used such tool with our *training* workload and materialized the combination of indexes it suggested. We then repeated the experiment in the previous section but using the richer set of execution plans derived by using the new indexes. Figures 14(a) and 14(b) show the results for this case. We can see that



(a) Total execution time. (b) Improvement in execution time.

Figure 13: Using SITs in a database with no indexes.



(a) Total execution time. (b) Improvement in execution time.

Figure 14: Using SITs in a database with indexes.

for the whole workload, the relative improvements are similar to those of the no-indexes case. A closer inspection of Figure 14(b), however, shows that some queries had two orders of magnitude improvements when using **SITs**.

6. RELATED WORK

Virtually all optimization frameworks [10, 11, 12, 14, 20] rely on statistics over the tables in the database to choose the most efficient execution plan in a cost-based manner. There is a large body of work that studies representation of statistics on a given column [16, 17, 18] or combination of columns [3, 13, 15, 17]. In this paper we present techniques to effectively use SITs for query optimization, and we study the orthogonal problem of deciding *which* columns over intermediate query plans to build SITs on.

Reference [6] presents the MNSA technique to select which subset of base-table statistics needs to be built without sacrificing the quality of the generated query plans. In this paper we present a non-trivial generalization of the techniques in [6] for the case of statistics over intermediate nodes of query plans. Similar to our work in self-tuning statistics [1, 3], LEO (DB2's LEarning Optimizer) [21] is a framework that repairs incorrect statistics and cardinality estimates of a query execution plan. By monitoring previously executed queries, LEO computes adjustments to cost estimates and statistics that may be used during future query optimizations. In this work we take a closer look at the particular case of error propagation in query plans. We believe that some of our ideas (specifically those discussed in Section 3) are relevant for LEO's framework as well.

The idea of building statistics over non base-tables was introduced in [2] by using join synopses, which are precomputed samples of a small set of distinguished joins. The main focus of this work is approximate query processing, and the generating queries are restricted to be foreign-key joins. In contrast, we present an effective framework to incorporate SITs to existing query optimizers. Application of SITs to a given query leverages materialized view matching algorithms [5, 9, 19]. However, as pointed out in Section *3.2.2*, our need to detect auxiliary SITs differs from traditional view matching. Furthermore, note that we do not need to store and maintain materialized views, but instead we just need to build statistics over those views.

Finally, the identification and use of SITs has great relevance to the problem of selecting the right indexes of a database, e.g., [7]. Specifically, current index tuning tools use existing (and build new) statistics to determine the appropriate choice of indexes. Such tools will benefit from the techniques proposed in this paper.

7. CONCLUSIONS

In this paper, we showed how to extend a traditional query optimizer so that it exploits statistical information on query expressions. In many cases, the quality of the resulting plans could be much better than when only base-table statistics are available. Extending and evaluating our methodology for more complex queries (such as aggregations and nested queries) and more complex statistics (such as multidimensional histograms) is an important next step. We introduced a workload-driven algorithm to select conservatively a small subset of SITs to build that can significantly improve quality of query plans compared to using statistics on base-tables only. Our implementation and experimental evaluation on Microsoft SQL Server 2000 showed the promise of our techniques, but more extensive experimental study is necessary to validate our approach.

8. ACKNOWLEDGEMENTS

We thank Luis Gravano, Christian König, and Vivek Narasayya for their valuable feedback.

9. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the 1999 ACM International Conference on Management of Data (SIGMOD'99)*, June 1999.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proceedings of the 1999 ACM International Conference on Management of Data (SIGMOD'99)*, 1999.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM International Conference on Management of Data (SIGMOD'01)*, 2001.

[4] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, 1998.

[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, 1995.

[6] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. In *Proceedings of the Sixteenth International Conference on Data Engineering*, 2000.

[7] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the Twenty-third International Conference on Very Large Databases (VLDB'97)*, Aug. 1997.

[8] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, 1998.

[9] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the 2001 ACM International Conference on Management of Data (SIGMOD'01)*, 2001.

[10] G. Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, 18(3), 1995.

[11] G. Graefe and D. J. DeWitt. The exodus optimizer generator. In *Proceedings of the 1987 ACM International Conference on Management of Data (SIGMOD'87)*, 1987.

[12] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, 1993.

[13] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the 2000 ACM International Conference on Management of Data (SIGMOD'00)*, 2000.

[14] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proceedings of the 1989 ACM International Conference on Management of Data (SIGMOD'89)*, 1989.

[15] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *Proceedings of the 1988 ACM International Conference on Management of Data (SIGMOD'88)*, 1988.

[16] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM International Conference on Management of Data (SIGMOD'84)*, 1984.

[17] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the Twenty-third International Conference on Very Large Databases (VLDB'97)*, Aug. 1997.

[18] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM International Conference on Management of Data (SIGMOD'96)*, 1996.

[19] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, 2000.

[20] 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 the 1979 ACM International Conference on Management of Data (SIGMOD'79)*, 1979.

[21] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *Proceedings of the Twenty-seventh International Conference on Very Large Databases*, 2001.

[22] TPC Benchmark H. Decision support. *Available at* `http://www.tpc.org`.