Efficient Creation of Statistics over Query Expressions

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

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Abstract

Query optimizers use base-table statistics to derive statistics on the sub-plans that are enumerated during optimization. In practice, traditional optimizers rely on a number of simplifying assumptions, which can compromise the accuracy of cardinality estimates. To address this limitation, we had earlier introduced SIT*s, which are statistics built over query expressions, and we explained how a traditional optimizer can judiciously use* SIT*s to sidestep the problem of inaccurate estimates. A significant challenge that was not addressed was how to build* SIT*s efficiently in a database system. In this paper we present a family of techniques to create* SIT*s. These techniques differ from each other in the trade-off they present between accuracy and efficiency of creation. We also present techniques to efficiently create multiple* SIT*s by taking advantage of the commonalities among their generating query expressions.*

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. Therefore, 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 estimation, and rely on a number of simplifying assumptions while propagating these base-table statistics through query plans. It is widely recognized that such cardinality estimates may be off by orders of magnitude [17], and thus might lead the query optimizer to choose significantly low-quality execution plans.

In [2] we introduced the concept of SITs, which are statistics built over the result of query expressions, to alleviate this problem. SITs can be used to directly model the distribution of tuples on *intermediate* nodes in query execution plans. For instance, suppose we build a SIT over attribute $S.a$ on the result of evaluating $R \bowtie_{x=y} S$. Whenever the optimizer needs to estimate the cardinality of some sub-plan of the form $\sigma_{S.a<c}(R \bowtie_{x=y} S)$, it can use directly such SIT, without performing the error-prone propagation of base-table histograms over $R.x$, $S.y$ and $S.a$. When optimizers have appropriate SITs available during query optimization, the resulting query plans may be drastically improved [2].

Despite the conceptual simplicity of SITs, a significant challenge that still needs to be addressed is providing mechanisms to efficiently create SITs in a database system. The most accurate procedure to build a SIT is to execute its associated query expression and then build a histogram over this temporary result. In this way, the resulting SIT accurately reflects the true distribution without relying on any simplifying assumption. However, such an approach also results in a high cost for building SITs. In this paper, we investigate approximated variants of SITs that can be built far more efficiently while giving up some of the accuracy of traditional SITs. In order to develop such alternatives, we critically examine the main simplifying assumptions used by current optimizers while estimating cardinalities, which are the *containment* and *independence* assumptions. Furthermore, in many systems statistics are built using *sampling*. We lay out a spectrum of alternatives between using traditional estimation and approximated SITs that are obtained by eliminating one or more of the simplifying assumptions listed above (containment, independence, and sampling). These alternatives differ in trade-offs between accuracy and efficiency of construction. In particular, we study in detail one of these alternatives, which we call *Sweep*. *Sweep* eliminates the independence assumption, but does rely on the containment assumption and uses sampling. We show that *Sweep* presents a compelling trade-off: it is efficient while resulting in accurate approximated SITs.

We also recognize that while trying to create several SITs at once, commonalities between their generating queries need to be leveraged. We formalize the problem of optimally creating a set of SITs and show how this problem can be mapped to a generalized version of the Shortest Common Supersequence (SCS) problem. We modify the A* algorithm that solves SCS optimally to our framework, and also present more efficient greedy alternatives.

*Work done while the author was visiting Microsoft Research.

The rest of the paper is structured as follows. Section 2 reviews SITs and how they can be effectively incorporated into the optimization framework. Section 3 introduces *Sweep*, a novel family of techniques to efficiently create SITs. Section 4 introduces algorithms to find optimal strategies to create multiple SITs by sharing invocations of *Sweep*. Section 5 reports experimental results for the techniques of Sections 3 and 4. Finally, Section 6 reviews related work.

2 Background

In this section we briefly describe the query optimizer component in a RDBMS and how it takes advantage of statistical information over base tables to find good quality execution plans. We then motivate the introduction of SITs and review how SITs can be incorporated to existing optimizers.

2.1 Query Optimization

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 examine a large number of possible query plans and choose the best one in a cost-based manner. For each incoming query, the optimizer iteratively explores the set of candidate execution plans using a rule-based enumeration engine. After each candidate plan or sub-plan is generated, the optimizer estimates its execution cost, which in turn refines the exploration of further candidate plans. Once all "interesting" plans are explored, the most efficient one is extracted and passed to the execution engine.

The cost estimation module is critical in the optimization process, since the quality of plans produced by the optimizer is highly correlated to the accuracy of the cost estimation routines. The cost estimate for a sub-plan, in turn, crucially depends on cardinality estimations of its sub-plans. Traditionally, query optimizers use statistics (mainly histograms) that are built over base tables to estimate cardinalities. Histograms are accurate for estimating cardinalities of simple queries, such as range queries. For complex query plans, however, the optimizer estimates cardinalities by "propagating" base-table histograms through the plan and relying on some simplifying assumptions (notably the *independence* assumption between attributes). The following example shows how base-table histograms are used to estimate the cardinality of a `SELECT-PROJECT-JOIN` query.

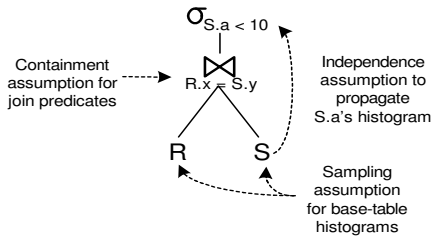


Figure 1. Simplifying assumptions in cardinality estimation.

Example 1 *Consider query* `SELECT * FROM R,S WHERE R.x=S.y AND S.a<10` *and the plan shown in Figure 1. Assume that histograms over $R.x$, $S.y$ and $S.a$ are available. To estimate the cardinality of the query, we first use histograms over $R.x$ and $S.y$ to estimate the cardinality of $R \bowtie S$, ignoring predicate $S.a < 10$. Then, the histogram over $S.a$ is propagated* [1] *through the join upwards in the tree. The propagated histogram is then used to estimate the cardinality of $S.a < 10$ over the intermediate result $R \bowtie S$, to finally obtain the cardinality of $\sigma_{S.a<10}(R \bowtie S)$.* ∎

Variations of the scheme above are used extensively in current optimizers. We identify the following simplifying assumptions during cardinality estimation of `SPJ` queries:

Independence assumption: When propagating histogram $H_{S.a}$ over $S.a$ through the join predicate $R \bowtie_{x=y} S$, bucket frequencies for $H_{S.a}$ are uniformly scaled down so that the sum of all frequencies in the propagated histogram equals the estimated cardinality of $R \bowtie_{x=y} S$. Implicit in this procedure is the assumption that distributions of attributes in R and S are independent.

Containment assumption: To estimate the cardinality of joins using histograms, the buckets of each histogram are aligned and a per-bucket estimation takes place, followed by an aggregation of all partial results. The *containment assumption* [16] dictates that for each pair of buckets, 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, if the number of distinct values in bucket b_R is 10, and the number of distinct values in bucket b_S is 15, the containment assumption states that each of the 10 groups of distinct valued tuples in b_R join with one of the 15 groups of distinct valued tuples in b_S.

Sampling assumption: Random sampling is a standard technique for constructing approximated base-table histograms. Usually the approximated histograms are of good quality regarding frequency distribution. However, estimating the number of distinct values inside buckets using sampling is provably difficult [3]. The *sampling assumption* states that the number of distinct values in each bucket predicted by sampling is a good estimator of the actual values.

Very often, some simplifying assumptions described above (or all of them) do not hold. For instance, many attributes are actually correlated and the independence assumption is often inaccurate. Therefore, the optimizer might rely on wrong cardinality information and therefore choose low quality execution plans. More complex queries (e.g., n-way joins) only exacerbate this problem, since estimation

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

errors propagate themselves through the plans. To address these limitations in current optimizers, we introduced in [2] the concept of SITs. In the next section we briefly review SITs and how they can be incorporated in existing relational optimizers to increase the quality of the resulting plans.

2.2 SITs: Statistics on Query Expressions

SITs are statistics built over the results of query expressions, and their purpose is to eliminate error propagation through query plan operators. As a simple example, consider again Figure 1, and suppose that we build a histogram over the result of the query expression $\mathcal{RS} = R \bowtie_{x=y} S$, specifically on $\mathcal{RS}.a$. In this case, we can estimate the cardinality of the original query plan by simply estimating the cardinality of the equivalent plan $\sigma_{\mathcal{RS}.a<10}(\mathcal{RS})$, and thus avoid relying on the independence and containment assumptions [2]. We now formalize the concept of SITs.

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 over attribute A on the result of executing query expression Q. We call Q the* generating query expression *of* SIT$(R.A|Q)$.

Although SITs can be defined using arbitrary generating queries, in this paper we will focus on the important family of join generating queries, which in turn are enough to cover optimization of `SPJ` queries. Therefore, we will consider SITs with the form SIT$(R_k.a|R_1 \bowtie \ldots \bowtie R_n)$.

SITs are only useful if the optimizer is able to incorporate them during query optimization. In [2] we enabled the 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. Each time the query optimizer invokes the modified cardinality estimation module with a query plan, we transform this input plan into an equivalent one that exploits SITs. Identifying whether or not a SIT is applicable for a given query plan (and later rewriting the plan to incorporate such SIT) leverages materialized view matching technology [3]. After transforming the input plan, we forward the transformed plan to the original cardinality estimator, and obtain a more accurate cardinality estimation for the original plan. In this way, the transformed query plan is a temporary structure used by the modified cardinality estimation module, but is *not* used for query execution. We refer to [2] for more details.

In [2] we also showed how an appropriate set of SITs may be chosen to maximize the benefit to the query optimizer. We recognized that usefulness of SITs depends on how their presence impacts execution plans for queries against the system, and we introduced a workload-based technique that identify useful SITs without the need to build them a-priori to evaluate their effectiveness. However, we did not study efficient mechanisms to create a given set of SITs. The obvious approach to build SITs is to execute the generating query associated with each SIT, and build the desired statistics over the temporary results. Once statistics have been computed, the result of the generating query expression can be discarded. Of course, this strategy is not desirable in general, since executing generating queries can be very expensive. The focus of the rest of this paper is to introduce efficient algorithms to create SITs.

3 Creating SITs

For a large class of query expressions, we can design efficient techniques inspired from the wide body of work in *approximate query processing* to create SITs. This is possible because we might be satisfied with approximate statistical distributions. In Section 3.1 we introduce *Sweep*, a novel technique to efficiently build accurate approximations of SITs that does not rely on the independence assumption between attributes. However, for efficiency purposes, *Sweep* does rely on the containment and sampling assumptions. As we will show experimentally in Section 5, the independence assumption causes most of the estimation errors, so *Sweep* is generally both efficient and accurate. In Section 3.1.2 we describe how to trade accuracy for efficiency and introduce variations on the basic formulation of *Sweep*. Finally, in Section 3.2 we show how to extend *Sweep* (which is designed for single-join generating queries) to deal with arbitrary acyclic-join generating queries.

3.1 *Sweep*: A Novel Approach to Create SITs

To create SIT$(R.a|Q)$, *Sweep* attempts to efficiently generate a sample of $\pi_{R.a}(Q)$ *without* actually executing Q, and then use existing techniques for building histograms over this intermediate result. Consider the example shown in Figure 2, and suppose that we want to create SIT$(S.a|R \bowtie_{x=y} S)$. We start, in step 1, by performing a sequential scan over table S [4]. For each tuple (y_i, a_i) scanned from S, in step 2 we estimate the number of tuples $r \in R$ such that $r.x = y_i$. We denote this value the *multiplicity* of tuple y_i from table S in table R, and we call *m-Oracle* the procedure that calculates multiplicity values. Clearly, the multiplicity of y_i in R is the number of matches for tuple (y_i, a_i) in the join $R \bowtie_{x=y} S$. If the estimated number of matches for y_i is n, we append n copies of a_i to a *temporary table* in step 3. It is important to note that we do not materialize the temporary

[2] We can always avoid using the sampling assumption by creating the SIT using a full scan instead of sampling.

[3] However, some specific SITs applications have no counterpart in traditional materialized view matching [2].

[4] If a multi-column index covering attributes $\{S.y, S.a\}$ is available, we replace the sequential scan by an appropriate index scan.

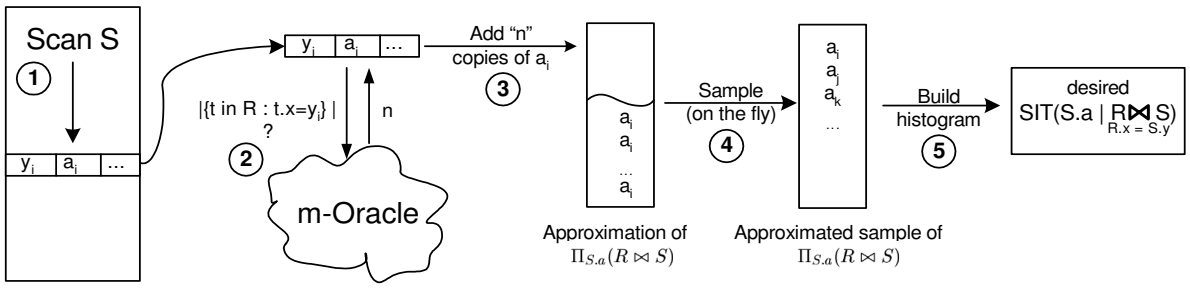


Figure 2. Creating SIT($S.a|R \bowtie_{R.x=S.y} S$) using *Sweep*.

table, but we treat it as a stream of values that approximate $\pi_{S.a}(R \bowtie_{x=y} S)$. Then, in step 4, we apply a one-pass sampling algorithm over the streaming table, such as reservoir sampling [19]. The result of this step is an approximated sample of $\pi_{S.a}(R \bowtie_{x=y} S)$, which is precisely the distribution we want to build a SIT over. Finally, in step 5, we use a traditional histogram technique to build SIT($S.a|R \bowtie_{x=y} S$) from the sample. Below we address the missing piece in the description of *Sweep*, i.e., estimating multiplicities for arbitrary values y_i from S in R.

3.1.1 Using an Approximating m-Oracle

A crucial routine to create SIT($S.a|R \bowtie_{x=y} S$) using *Sweep* is to obtain, both efficiently and accurately, the multiplicity of arbitrary tuples from S in R (step 2 in Figure 2). We now introduce a histogram-based technique that results in accurate and efficient approximations of multiplicity values. The idea is to use histograms over $R.x$ and $S.y$ (denoted h_R and h_S) to provide the multiplicity of tuples from S in R. For a given value y from S, we first identify the buckets $b_{R,y}$ and $b_{S,y}$ from histograms h_R and h_S that contain value y. Then, we calculate the *expected* number of tuples from R that join with y *under the containment assumption for join predicates*.

To estimate the multiplicity of y in R we consider two scenarios. Let $dv_{R,y}$ and $dv_{S,y}$ be the number of distinct values in buckets $b_{R,y}$ and $b_{S,y}$, respectively. In the case that $dv_{S,y} \leq dv_{R,y}$, i.e., the number of distinct values from h_R is larger than that of h_S, we can guarantee (under the containment assumption) that value y, which belongs to one of the $dv_{S,y}$ groups in $b_{S,y}$, would match some of the $dv_{R,y}$ groups in $b_{R,y}$. Since we use the uniform spread assumption inside buckets, the multiplicity for y in this situation would be $f_{R,y}/dv_{R,y}$, where $f_{R,y}$ is the frequency of bucket $b_{R,y}$. However, if $dv_{S,y} > dv_{R,y}$, we can no longer guarantee that value y joins with some of the $dv_{R,y}$ buckets in h_R. If the tuples that verify the join are distributed uniformly, the probability that y is in one of the $dv_{R,y} < dv_{S,y}$ groups in $b_{R,y}$ that match some group in $b_{S,y}$ is $dv_{R,y}/dv_{S,y}$. In that case, the multiplicity would be $f_{R,y}/dv_{R,y}$. Otherwise (y does not match with any value in R), the multiplicity would be 0. In conclusion, when $dv_{S,y} > dv_{R,y}$, the expected multiplicity of y in R is $(f_{R,y}/dv_{R,y}) \cdot (dv_{R,y}/dv_{S,y}) + 0 \cdot (1 - dv_{R,y}/dv_{S,y}) = f_{R,y}/dv_{S,y}$.

Putting both results together, we obtain that the expected multiplicity of y from S in R is given by $f_{R,y}/\max(dv_{R,y}, dv_{S,y})$. Since we can locate the bucket that contains a given tuple very efficiently in main memory, *getMultiplicity* is extremely efficient. In Section 5 we will show experimentally that *getMultiplicity* is also an accurate estimator of multiplicity values.

3.1.2 Accuracy/Efficiency Tradeoffs

As mentioned in Section 2, the main assumptions when estimating cardinalities of SPJ queries are the independence assumption between attributes, the containment assumption for join predicates, and the inaccuracies introduced by using sampling when building histograms. When creating SIT($S.a|R \bowtie_{x=y} S$), *Sweep* does not rely on the independence assumption, since it works over the multidimensional distribution $(S.a, S.x)$ during the sequential scan in step 1. However, *Sweep* does rely on the containment assumption (when estimating multiplicity values in step 2), and on the sampling assumption (when building the SIT by using the sample obtained in step 4). We now show how we can avoid relying on these remaining assumptions, and therefore obtain more accurate SITs, at the expense of introducing execution overheads to the original formulation of *Sweep*.

SweepIndex (Containment assumption): If an index over attribute $R.x$ is available in step 2, we can issue repeated index lookups to find exact multiplicity values. Since index lookups are more expensive than histogram lookups, *SweepIndex* is more expensive than *Sweep*.

SweepFull (Sampling assumption): We can omit step 4 in *Sweep* and build the SIT directly from the temporary table. Of course, that might require materializing the temporary table on disk, since it can be too large to fit in main memory. For that reason, *SweepFull* is more expensive than *Sweep*.

SweepExact (Containment and Sampling assumptions): By combining ideas from *SweepIndex* and *SweepFull* we avoid relying on any assumption whatsoever, and the result of *SweepExact* is the same as if we evaluate the SIT's query and then build a histogram over the result. This technique returns the most accurate histogram for a given SIT, but it is also the most expensive.

In the rest of the paper, we focus on the original formulation of *Sweep*. Later, in Section 5, we compare the accuracy of the alternatives introduced in this section.

3.2 Acyclic-Join Generating Queries

The basic formulation of *Sweep* handles SITs with single-join generating queries. We now show how we can extend *Sweep* to handle arbitrary acyclic-join queries. A given query is an *acyclic-join query* if its corresponding join-graph is acyclic [5]. In the remaining of this section we look at a restricted class of acyclic join queries and assume that for every pair of tables t_1 and t_2 in the generating query q, there is at most one predicate in q joining t_1 and t_2. In the general case, e.g., for $R \bowtie_{R.w=S.x \wedge R.y=S.z} S$, we can still extend the techniques in this section by using multidimensional histograms. The details of these extensions, however, are omitted for lack of space.

As a first case, we extend *Sweep* to handle *chain-join* generating queries. A chain-join query can be expressed as $(R_1 \bowtie \ldots \bowtie R_n)$, where the i-th join ($1 \leq i \leq n$-1) connects tables R_i and R_{i+1}, i.e., the corresponding join-graph is a chain. From the description of *Sweep* in Section 3.1, it is clear that to approximate SIT($S.a|R \bowtie_{x=y} S$) we only need to perform the following operations: (i) a sequential (or index) scan covering attributes $\{S.y, S.a\}$ in table S, and (ii) histogram lookups over attributes $R.x$ and $S.y$. To approximate a SIT over a chain-join query, we left associate the query joins and unfold the original SIT into a set of single-join SITs, as illustrated below.

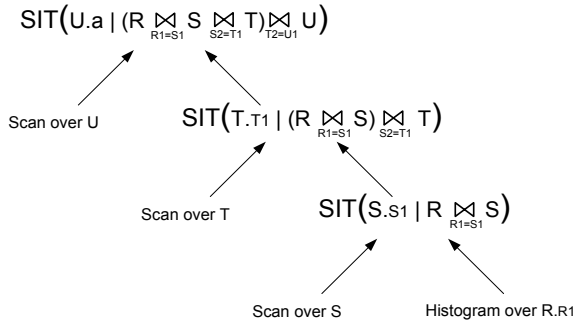


Figure 3. Using *Sweep* to approximate a chain-join query.

Example 2 *Consider* SIT($U.a|\ R \bowtie_{r1=s1} S \bowtie_{s2=t1} T \bowtie_{t2=u1} U$)*, shown in Figure 3. We can rewrite such* SIT *as* SIT($U.a|RST \bowtie_{t2=u1} U$)*, where RST is defined as $R \bowtie_{r1=s1} S \bowtie_{s2=t1} T$. To approximate this equivalent* SIT*, we need to perform a sequential scan over table U, and we need to access histograms over $U.u1$ and $RST.t2$. However, $RST.t2$ is just* SIT($T.t2|R \bowtie_{r1=s1} S \bowtie_{s2=t1} T$)*. We then calculate* SIT($T.t2|R \bowtie_{r1=s1} S \bowtie_{s2=t1} T$)*, and finally create the original* SIT *we were interested in. The base case corresponds to approximating* SIT($S.s2|R \bowtie_{r1=s1} S$)*, since both join operands are base tables, and Sweep works in this case. Figure 3 illustrates this procedure.* ∎

[5] We obtain the join-graph of query q by associating nodes with each table in q and edges that correspond to the join predicates in q.

We next show how to extend *Sweep* to handle more general kinds of acyclic-join generating queries. For that purpose, we first convert the (acyclic) join-graph into the join-tree that has as the root the table holding the SIT's attribute. As an example, Figure 4 shows a SIT with an acyclic-join generating query and its induced join-tree. Suppose that the height of the join-tree is one, i.e., the join-tree consists of a root R and children $S_1, \ldots, S_n$, and we want to get SIT($R.a|R \bowtie_{r1=s1} S_1 \bowtie_{r2=s2} S_2 \bowtie \ldots \bowtie_{rn=sn} S_n$). In this case we can extend *Sweep* as follows. We first build base-table histograms for each attribute $S_i.si$ and, similar to the single-join case, we perform a sequential scan over R. To obtain the multiplicity of tuple $r = (a, r_1, \ldots, r_n)$ from R in the multi-way join, we first get each partial multiplicity of r_i in S_i (denoted m_i) and then we multiply the partial results. The multiplicity of tuple r from R in the join is then $\Pi_i(m_i)$. We note that this multiplicity value does not assume independence between join predicates. In fact, each tuple r from R joins with m_i tuples from each S_i, and since the join-graph is acyclic, the result contains all possible join combinations between r and each qualifying tuple in S_i. After obtaining the multiplicity values, we proceed with *Sweep* as before.

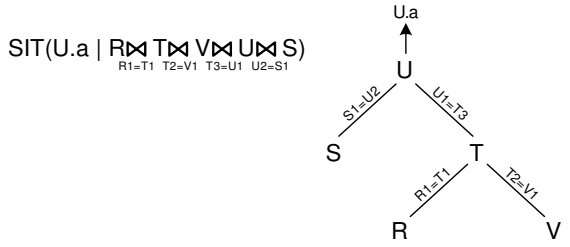


Figure 4. SIT with acyclic-join query and join-tree for $U.a$.

For an arbitrary acyclic-join generating query, we perform a post-order traversal of the join-tree. At each leaf node we build (if not present already) the base-table histogram for the attribute that participates in the join with the parent table in the join-tree. For each internal node, we use the children's SITs produced earlier to compute the corresponding SIT for the attribute that participates in the join predicate with the parent (or the final attribute for the root node). As an example, for the SIT of Figure 4, we first get the base-table histograms for $S.s1$, $R.r1$, and $V.v1$. When we process node T, we calculate SIT($T.t3|R \bowtie_{r1=t1} T \bowtie_{t2=v1} V$). Finally, when we process node U we calculate the desired SIT($U.a|R \bowtie_{r1=t1} T \bowtie_{t2=v1} V \bowtie_{t3=u1} U \bowtie_{u2=s1} S$).

In summary, in this section we introduced *Sweep*, a technique to create SITs with arbitrary acyclic-join generating queries. *Sweep* requires a sequential scan over each involved table (except for the root of the join-tree) and some additional amount of extra processing to build intermediate SITs.

In this section we assumed that the only available procedure to build SITs is *Sweep*. More complex scenarios are possible. For instance, we can materialize some portions of the generating query first, and then apply the techniques of this section. These extensions, however, are beyond the scope of this work.

4 Multiple SIT Creation

Sweep is a technique to build SITs with arbitrary acyclic-join generating queries. If we are interested in creating a single SIT, we can directly apply *Sweep* to obtain the desired histogram. However, in many cases we want to create several SITs at once. For instance, the techniques in [2] return a set of candidate SITs to create for a given workload. In such situations, many commonalities between SITs might exist, and we could be able to "share" the same sequential scan to build more than one SIT. For that reason, the obvious one-at-a-time approach for building SITs is often suboptimal, as the following example illustrates.

Example 3 *Suppose we want to create the* SIT*s below:*

$\text{SIT}(T.a|R \bowtie_{r1=s1} S \bowtie_{s3=t3} T)$

$\text{SIT}(S.b|R \bowtie_{r2=s2} S)$

A naive approach would be to apply Sweep to each SIT *separately. In that case, we would use one sequential scan over tables S and T (to build the first* SIT*), and a second sequential scan over table S (to build the second* SIT*). However, the following strategy is more efficient:*

1. *Perform a sequential scan over table S to get both* $\text{SIT}(S.b|R \bowtie_{r2=s2} S)$ *and* $\text{SIT}(S.s3|R \bowtie_{r1=s1} S)$*. This can be done by sharing the sequential scan over S (on attributes $S.s2$, $S.b$, $S.s3$, and $S.s1$) and using histograms over $R.r2$ and $S.s2$ for the first* SIT*, and histograms $R.r1$ and $S.s1$ for the second to obtain the required multiplicity values.*
2. *Perform a sequential scan over T, and using the previously calculated* $\text{SIT}(S.s3|R \bowtie_{r1=s1} S)$*, obtain the required* $\text{SIT}(T.a|R \bowtie_{r1=s1} S \bowtie_{s3=t3} T)$*.*

The second strategy requires a single sequential scan over table S, as opposed to two scans for the naive strategy. Of course, the memory requirements for the second strategy are larger than those for the first, since we need to maintain two sets of samples in memory: one for $\pi_{S.b}(R \bowtie_{r2=s2} S)$, and another for $\pi_{S.s1}(R \bowtie_{r1=s1} S)$. ∎

These concepts are similar to the ideas in multi-query optimization [15], which aims to exploit common subexpressions among the input queries to reduce the overall execution cost. However, there is a difference between multi-query optimization and our problem. In our scenario, the execution plan for each individual SIT is known in advance (it corresponds to the unique sequence of applications of *Sweep* described in Section 3.2). Therefore, the search space is much smaller than in the general case, and we can provide a tailored solution to find the optimal strategy. We now formalize the optimization problem to create a given set of SITs:

> Given a set of SITs $\mathcal{S} = \{S_1, \ldots, S_n\}$, a sampling rate s [6], and the amount of available memory M, find the optimal sequence of applications of the *Sweep* algorithm (sharing sequential scans as explained above) such that: (i) at any time the total amount of memory used for sampling is bounded by M, and (ii) the estimated execution cost for building $\mathcal{S}$ is minimized.

To address this optimization problem, we first review the *Shortest Common Supersequence* (SCS) problem in Section 4.1, and a previously proposed A*-based technique to solve it in Section 4.2. Then, we show how to map our SIT creation problem to a generalized version of SCS in Section 4.3. We extend the A*-based technique to solve our problem optimally and finally introduce two greedy alternatives.

4.1 Shortest Common Supersequence

The Shortest Common Supersequence [8], or SCS, is a well known problem used in text editing, data compression, and robot assembly lines, among other applications. We now define the SCS problem.

Definition 2 *Let $R = x_1 \ldots x_n$ be a sequence of elements (individual elements of R can be accessed using array notation, so $R[i] = x_i$). Given a pair of sequences R and R', we say that R' is a* subsequence *of R, if R' can be obtained by deleting zero or more elements from R (we then say that R is a* supersequence *of R'). A sequence R is a* common supersequence *of a set of sequences $\mathcal{R} = \{R_1, \ldots, R_n\}$ if R is a supersequence of all $R_i \in \mathcal{R}$. A* shortest common supersequence *of $\mathcal{R}$, denoted SCS($\mathcal{R}$), is a common supersequence of $\mathcal{R}$ with minimal length.*

Example 4 *Consider $\mathcal{R} = \{abdc, bca\}$. Some common supersequences of $\mathcal{R}$ are $abdcbca$, $aabbddccbbcaa$, and $abdca$. We know that SCS($\mathcal{R}$) = $abdca$, since no sequence of size four is a common supersequence of both $abdc$ and bca.* ∎

Finding the SCS of a set of sequences is an NP-complete problem, and can be solved using dynamic programming in $\mathcal{O}(l^n)$ for n sequences of length at most l, by formulating SCS as a shortest path problem in an acyclic directed graph with $\mathcal{O}(l^n)$ nodes [18]. For a given set of sequences $\mathcal{R} = \{R_1, \ldots, R_n\}$, the graph is constructed as follows. Each node in the graph is a n-tuple $(r_1, \ldots, r_n)$, where $r_i \in \{0..|R_i|\}$ indexes a position in R_i. Node $(r_1, \ldots, r_n)$ will encode a solution for the common supersequence of $\{S_1, \ldots, S_n\}$, where $S_i = R_i[1]R_i[2] \ldots R_i[r_i]$, i.e., the r_i-prefix of R_i. We insert an edge from node $(u_1, \ldots, u_n)$ to

[6] The sampling rate can be specified as a percentage of the table size, an absolute amount, or a combination or both (depending of the table size).

node $(v_1, \ldots, v_n)$ with label θ if the following properties hold: (i) $u_i = v_i \vee u_i + 1 = v_i$, (ii) at least one position u_j verifies $u_j + 1 = v_j$, and (iii) for every position v_j such that $u_j + 1 = v_j$, we have that $R_j[u_j] = \theta$. Informally, an edge labelled θ connects nodes u and v if we can reach the state represented by v from the state represented by u by adding θ to the common supersequence encoded at u.

It is fairly easy to show that any path from node $O = (0, \ldots, 0)$ to node $F = (|R_1|, \ldots, |R_n|)$ in the graph corresponds to a common supersequence of $\mathcal{R}$. In particular, any shortest path from O to F corresponds to a shortest common supersequence of $\mathcal{R}$. Therefore, to solve SCS we materialize the induced graph and use any algorithm to find the shortest path between O and F.

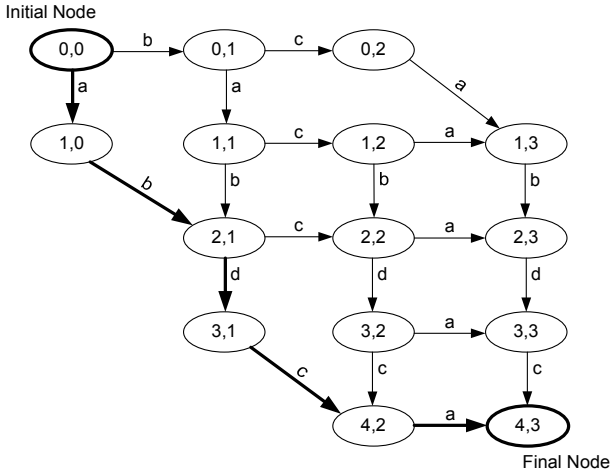


Figure 5. The graph induced by $\mathcal{R} = \{abdc, bca\}$.

Example 5 *Figure 5 shows the graph induced by the set of sequences $\mathcal{R} = \{abdc, bca\}$. For instance, $(2, 1)$ is the final node for the subproblem $\mathcal{R}' = \{ab, b\}$ (the 2- and 1-prefixes of the original sequences, respectively). By adding an edge c to any common supersequence of $\{ab, b\}$ we obtain common supersequences for $\{ab, bc\}$ and therefore we insert edge c between nodes $(2, 1)$ and $(2, 2)$. The shortest path from the initial node $O = (0, 0)$ to the final node $F = (4, 3)$ is $\{(0, 0), (1, 1), (2, 1), (3, 1), (4, 2), (4, 3)\}$, which corresponds to SCS$(\mathcal{R}) = abdca$.* ∎

In the next section we review an algorithm to solve SCS [9] that generally results in fast executions and does not need to materialize the whole induced graph in advance.

4.2 A*-based Approach to Solve SCS

Algorithm A* [10] is a heuristic technique to efficiently find shortest paths in graphs that are inductively built (i.e., graphs in which we can generate the set of successors of any given node). The key idea in [9] is to apply A* to the SCS problem and materialize only a portion of the graph induced by the input set of sequences. A* searches the input graph outwards from the starting node O until it reaches the goal node F, expanding at each iteration the node that has the most chances to be along the best path from O to F. The application of A* is based on the possibility, for each node u in the induced graph, of estimating a lower bound of the length of the best path connecting O and F through u (we denote this value $f(u)$). At each step in the search, we choose the most promising node, i.e., the node for which $f(u)$ is the smallest among those for the nodes created so far. Then, we *expand* the chosen node by dynamically generating all its successors in the graph. Typically, the cost function $f(u)$ is composed of two components, $f(u) = g(u) + h(u)$, where $g(u)$ is the length of the shortest path found so far between O and u, and $h(u)$ is the expected remaining cost (heuristically determined) to get from u to F. If the heuristic function $h(u)$ is always an *underestimate* of the actual length from u to F, A* is guaranteed to find the optimal solution. However, if $h(u)$ is too optimistic, A* will expand too many nodes and may run out of resources before a solution is found. Therefore, it is critical to define $h(u)$ as tight as possible. Also, if for any pair of nodes u and v that are connected by an edge in the graph, we have that $h(u) - h(v) \leq d(u, v)$ where $d(u, v)$ is the cost of going from u to v, the following property holds: whenever a node u is expanded, a shortest path from O to u is already known. This property allows efficient implementations of A*.

For the SCS problem, an estimate on the length of the shortest path from u to F, i.e., $h(u)$, is equivalent to an estimate of the shortest common supersequence of the suffixes of the original sequences not yet processed in state u. A good value for $h(u)$ can then be calculated as follows. If we denote $o(u, c)$ to the maximum number of occurrences of c in some suffix sequence in state u, a lower bound $h(u)$ is then $\sum_c o(u, c)$, since every common supersequence must contain at least $o(u, c)$ occurrences of c. For instance, consider the node $(2, 1)$ in Figure 5, for which we processed the two first elements of $abdc$ and the first element of bca. The remaining suffixes are dc and ca, respectively. In this case, $h((dc, ca)) = o((dc, ca), d) + o((dc, ca), c) + o((dc, ca), a) = 1 + 1 + 1 = 3$. Putting all pieces together, we present below the A* algorithm to solve SCS:

Algorithm A* for SCS ()
```
01 OPEN={O}; CLOSED=∅; g(O)=0; f(O)=0;
02 repeat
03  bestN=n ∈ OPEN s.t.  f(n) is minimal
04  OPEN=OPEN - {bestN}
05  CLOSE=CLOSE ∪ {bestN}
06  for each successor s of bestN do
07    gNew=g(bestN) + 1  // d(bestN, s)==1
08    if (s ∉ OPEN ∪ CLOSE) ∨
            (s ∈ OPEN ∧ gNew < g(s)) )
09      g(s)=gNew; f(s)=g(s)+∑_c o(u, c)
10      OPEN=OPEN ∪ {s}
11  until (bestNode=F)
```

It should be noted that A* does not affect the size of the graph, but usually results in faster executions since it does not need to generate the whole graph in advance, but only explores a small fraction guided by the heuristic function h.

4.3 Exploiting SCS to Create Multiple SITs

In this section we show how we can reduce our problem of optimally creating a set of SITs into a generalized version of SCS. Then, we show how to adapt the A* technique of Section 4.2 to solve this problem, and present a greedy algorithm to quickly approximate a close-to-optimal solution in a small amount of time.

We recall from the description of *Sweep* (see Section 3) that creating a given SIT requires performing sequential scans over the set of tables referenced in the SIT's generating query (with the exception of the root table in the join-tree). Moreover, the sequential scans must follow the order given by some post-order traversal of the join-tree (see Section 3.2). As an example, suppose we want to create a SIT over attribute R.a with the acyclic-join generating query of Figure 6(b). Clearly, sequential scans over tables S and U (which return SIT($S.s1|S \bowtie_{s2=t1} T$) and SIT($U.u1|U \bowtie_{u2=v1} V$)) must precede sequential scan over table R, since the latter uses SITs produced by the former. However, no ordering is required between scans of S and U.

We can concisely specify these order restrictions by using a set of *dependency sequences*. A dependency sequence is simply a sequence of tables $(R_1, \ldots, R_n)$, such that for all $1 \leq i, j \leq n$, the sequential scan over table R_i must precede the sequential scan over R_j. For chain-join queries, a single dependency sequence is needed, which is obtained by traversing the chain of joins starting from the table that originally hosts the SIT's attribute, and omitting the last table. In general, for a acyclic-join query we need one dependency sequence for each root-to-leaf path in the join-tree (omitting leaf nodes). Figure 6 shows two join queries and their corresponding dependency sequences.

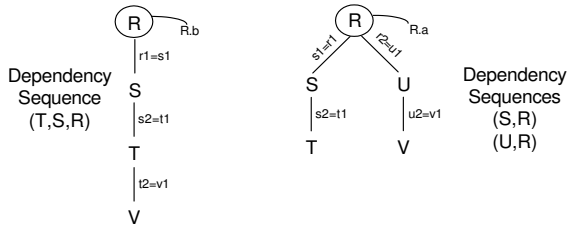


Figure 6. Dependency sequences for join generating queries.

To model the time and space required to execute *Sweep* over a single-join generating query, we associate with each table T the following values: *Cost*(T), which is the estimated cost to perform a sequential scan over T, and *SampleSize*(T, a), which specifies how much memory we allocate for a sample over attribute a of T. *SampleSize*(T, a) can be a constant value or depend on the specific table and attribute. Therefore, if we use *Sweep* to create SIT($S.a|R \bowtie S$), we estimate the cost of the procedure as *Cost*(S) and the memory requirements as *SampleSize*(S, a).

As illustrated in Example 3, we can share a sequential scan over table S to create any SIT of the form SIT($S.a|R \bowtie_{x=y} S$) for arbitrary table R and attributes a, x and y, provided that we have a histogram over $R.x$ available (recall that for acyclic-join generating queries, R could represent an intermediate join result). In this situation, the cost of executing *Sweep* remains fixed at *Cost*(T) since the sequential scan is shared. However, the space required for sampling increases to $\sum_i sampleSets(a_i) \cdot$ *SampleSize*(T, a_i), where $sampleSets(a_i)$ is the number of sample sets for attribute a_i required during the sequential scan over table S [7]. For instance, if we share the sequential scan over S to create SIT($S.a|R \bowtie_{x=y} S$), SIT($S.b|R \bowtie_{x=y} S$), and SIT($S.a|T \bowtie_{z=y} S$), the estimated cost will be *Cost*(S) and the memory requirements for sampling will be $2 \cdot$ *SampleSize*(S, a) + *SampleSize*(S, b).

If the amount of available memory is unbounded, the optimization problem can be very easily mapped to a weighted version of SCS, where the input sequences to the SCS problem are all the dependency sequences of the given SITs. All we need to change in the A* algorithm is the definition of the distance function between nodes to incorporate weights, and the heuristic function $h(u)$ (lines 6 and 8 in the A* algorithm of Section 4.2). In particular, we replace $d(u, v)$ from the constant 1 (no weight) to *Cost*(R), where R is the label of edge (u, v). The definition of $h(u)$ is changed accordingly to $h(u) = \sum_c$ *Cost*(c) $\cdot\, o(u, c)$.

Once we get the SCS, we can "execute" it as follows. We iterate through the elements (tables) of the SCS one at a time. When we process table T, we create (using *Sweep*) all SITs of the form SIT($T.a|S \bowtie_{s_i=t_j} T$) for which the histogram over $S.s_i$ is already built (or if S is a base table, we create the corresponding base-table histogram).

Example 6 *Consider the* SITs *of Figure 6, and assume that* *Cost*(R)=*Cost*(S) $= 10$, *Cost*(T)=*Cost*(U)=*Cost*(V)=20, *and SampleSize*(t, a) $= 10,000$ *for all tables and attributes. In such a case, a shortest weighted common supersequences with cost 60 is* (U, T, S, R). *The execution of schedule* (U, T, S, R) *proceeds as follows. We first perform a sequential scan over table* U *obtaining* SIT($U.u1|U \bowtie_{u2=v1} V$). *Then, with a sequential scan over* T *we obtain* SIT($T.t1|T \bowtie_{t2=v1} V$). *Then, we perform a sequential scan over* S *obtaining* SIT($S.s1|S \bowtie_{s2=t1} T$) *and* SIT($S.s1|S \bowtie_{s2=t1} T \bowtie_{t2=v1} V$) *(using* $2 \cdot$ *SampleSize*($S, s1$) *memory for samples). Finally, we do a sequential scan over table* R *and obtain, using* *SampleSize*(R, a) + *SampleSize*(R, b) *for samples, the re-*

[7] We would need slightly more space than *SampleSize*(T, a) to store the input histograms, but we omit this detail for simplicity.

quired SIT($R.b|R \bowtie_{r1=s1} S \bowtie_{s2=t1} T \bowtie_{t2=v1} V$) *and* SIT($R.a|R \bowtie_{r1=s1} S \bowtie_{s2=t1} T \bowtie_{r2=u1} U \bowtie_{u2=v1} V$) . ∎

The scenario considered above assumes that we can allocate any amount of memory to create SITs. When the amount of available memory M is bounded, we need to additionally modify the search space to solve a constrained, weighted SCS. For instance, if $2 \cdot \mathit{SampleSize}(S, s1) > M$ in the example above, we would not be able to share the sequential scan over S, and the optimal execution path would be different. We now show how to modify A* to obtain the optimal strategy to create multiple SITs given memory constraints.

4.3.1 The A* Algorithm for Multiple SIT Creation

The main implication of having a bounded amount of available memory is that some edges in A*'s search graph are not valid any longer. As explained in Section 4.2, the implicit meaning of an edge from node $u = (u_1, \ldots, u_n)$ to node $v = (v_1, \ldots, v_n)$ with label c is to "advance" one position *all* input sequences for which $R[u_i] = c$. Of course, while creating SITs, each position that was changed from u_i to $v_i = u_i + 1$ in transition (u, v) corresponds to an additional SIT to create and might increase the memory requirements above the given limit. For that reason, in general we can only advance subsets of all possible positions from node u using edge c. Moreover, we must try each possibility, or we might compromise optimality. We modified the procedure to get the successors of a given node at each iteration of A* to solve this constrained version of SCS. We note that the size of the search graph is still bounded by $\mathcal{O}(l^n)$, where n is the number of input SITs and l is the size of the largest dependency sequence among the input SITs. However, the number of edges typically increases. For many input SITs, or SITs with many joins, the A*-based technique may become too expensive. The modified procedure is described below.

generateSuccessors ($u = (u_1, \ldots, u_n)$: node,
$\mathcal{R} = \{R_1, \ldots, R_n\}$: sequences,
M : memory limit)

```
01 successors = ∅
02 for each table T in R do
03   candidates = {i : R_i[u_i] = T}
04   for each C ⊆ candidates such that
       Σ_{a_i} sampleSets(a_i) · SampleSize(T, a_i) ≤ M do
05     successors = successors ∪ (v_1, ..., v_n),
         where v_i = { u_i + 1   if i ∈ C
                     { u_i       otherwise
06 return successors
```

4.3.2 *Greedy*/*Hybrid* Alternatives

The A* algorithm is guaranteed to find an optimal scheduling to create a set of SITs that minimizes the total execution cost. However, the worst-case time complexity of the algorithm is $O(l^n \cdot 2^S)$ where l is the maximum length of any chain of joins, n is roughly the number of input SITs, and S is the maximum size of any candidate set. For small values of l and n, the A* algorithm is efficient, but if we increase n or l some executions of A* become prohibitively expensive. For that reason, we now propose a simple greedy variant of the A* algorithm and also a hybrid approach that balances efficiency and quality of the resulting schedule.

Greedy: The *Greedy* approach can be described as a simple modification of A*. At each iteration of A*, after we get the best node u, we empty the OPEN set before adding the successors of u. In this way, the *Greedy* variant simply chooses at each step the element that would result in the largest *local* improvement. In this case, the size of OPEN at each iteration is bounded by the maximal number of successors of any given node, and the algorithm is guaranteed to finish in at most $\sum_i |R_i|$ steps (since the induced search graph is always acyclic). However, due to the aggressive pruning in the search space, *Greedy* usually results in suboptimal schedules.

Hybrid: *Hybrid* is a combination of *Greedy* and A*, based on the observation that we can switch from A* to *Greedy* at any given iteration, by simply cleaning OPEN at the current and every subsequent iteration. *Hybrid* starts as A* and after a switch condition, greedily continues from the most promising node found so far. Several switching conditions can be used for *Hybrid*. For instance, we can switch after a pre-determined amount of time has passed without A* returning the optimal solution, or after $|\texttt{OPEN} \cup \texttt{CLOSE}|$ uses all available memory. In our experiments, we switch after one second without A* finding the optimal solution.

5 Experiments

In this section we present experimental results for a C++ implementation of the algorithms described in Sections 3 and 4. All experiments were conducted in a Pentium IV 2.2Ghz with 1GB of main memory. In Section 5.1 we compare the different variations of *Sweep* for creating a single SIT using a variety of data sets and generating queries (we focus on contrasting the accuracy of the techniques rather than their execution times). In Section 5.2 we report results for the algorithms of Section 4 to create sets of SITs.

5.1 Creating a Single SIT

In this section we compare different techniques to build a single SIT, using the following setting:

Data sets, generating queries, and histograms: We used a synthetic database consisting of 4 tables with 10,000 to 100,000 tuples. Each table is composed of three to five attributes. Some attributes are uniformly distributed and others follow a zipfian distribution (with parameter z varying from

0.1 to 1). For the SITs' generating queries, we used 2, 3, and 4-way chain-join queries. We used a variant of *MaxDiff* histograms [14] which are natively supported in Microsoft SQL Server 2000. We set the default number of buckets in the histograms as $nb = 100$.

Techniques compared: We compare *Sweep* (with a sampling rate of 10%) and its variations of Section 3.1.2. We also included *Hist-SIT*, the traditional technique used by current optimizers to estimate SITs by propagating base-table histograms. *Hist-SIT* is very efficient, since it operates on the histogram domain without accessing the actual data, but relies on all the simplifying assumptions of Section 2.

Evaluation metric: For each technique considered, we created the corresponding SIT and also materialized the generating query to obtain the actual result. Then, we issued 1,000 random range queries over the SIT domain (corresponding to SPJ queries over the actual data sets) and calculated the relative error between the actual and estimated cardinalities.

Results: Figure 7 shows the relative error for the different techniques and generating queries of increasing complexity. For these experiments, we used join attributes with skewed distributions ($z = 1$), so that the independence assumption is not valid. As expected, *Hist-SIT* is consistently much worse than the other techniques, due to all the simplifying assumptions involved in the estimation. Moreover, the gap between *Hist-SIT* and the other techniques increases with the number of joins in the generating query, due to the propagation of errors through the query plans. For instance, in Figure 7(c) the relative error is close to 200% independent of histogram sizes. In contrast, the variants of *Sweep* result in estimation errors that are close to those for the most accurate *SweepExact*. In particular, *Sweep* results in slightly worse estimations than *SweepFull* and *SweepIndex*, since it relies on additional simplifying assumptions. However, the difference is small, which suggests that the critical assumption is *independence*. Also, there is not a clear winner between *SweepFull* and *SweepIndex* in all situations.

We also conducted an experiment in which the join predicates were uniformly distributed and independent of the remaining attributes. In this case, the independence assumption holds, and all techniques result in very accurate estimations (with relative errors below 2%). In this situation, *Sweep* and *SweepIndex* result in slightly worse accuracy (around 2% versus 1% for the other techniques) due to sampling. A comprehensive study of which scenarios are estimated better by each technique is an important piece of future work.

5.2 Creating Multiple SITs

We now compare different techniques to schedule the creation of multiple SITs, using the setting below:

Data sets and generating queries: We generated data schemas and SITs using several parameters. In particular, we first created nt different table schemas [8]. The number of tuples in each table was taken from a zipfian distribution with parameter $z = 1$. We set $Cost(T) = |T/1000|$ units (since the cost of a sequential scan is proportional to the size of the input), and $SampleSize(T) = s \cdot |T|$, where s is the sampling rate. Finally, we created each input instance by randomly generating $numSITs$ dependency sequences, each one with length between 2 and $lenSITs$. By default we used the following values: $numSITs = 10$, $lenSITs = 5$, $nt = 10$ and $s = 10\%$. The combined size of all tables was 1,000,000.

Techniques compared: We compare the following techniques: *Naive* (which creates each SIT separately), *Opt* (which is the optimal strategy of Section 4.3 based on A*), *Greedy*, and *Hybrid*. By default, we specified the available amount of memory as $M = 50,000$.

Evaluation metric: For each experiment, we generated 100 different instances (sets of SITs) and optimized their evaluation strategies using each technique. We finally averaged, for each technique, the total optimization time used to find each schedule, and the estimated cost of such schedule.

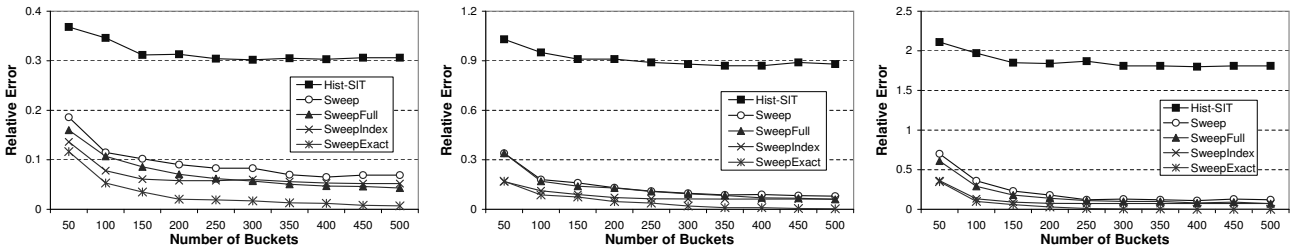
5.2.1 Varying $numSITs$ and $lenSITs$

Figure 8 shows the estimated cost and optimization time for the default setting and varying $numSITs$. Not surprisingly, *Naive* results in considerably slower schedules than the remaining techniques, since it does not take advantage of commonalities between SITs. Both *Greedy* and *Hybrid* result in close to optimal schedules for all scenarios (see Figure 8(a)). However, the optimization times of *Greedy* and *Hybrid* are substantially smaller than those of *Opt*, specially for larger instances. For instance, for sets with 20 SITs, *Opt* took over 36 seconds on average to find the optimal solution. In contrast, *Greedy* and *Hybrid* took 0.02 and 0.8 seconds, respectively, to produce slightly less efficient schedules (the average estimated costs were 2024, 2124 and 2278 units for *Opt*, *Greedy*, and *Hybrid* respectively). We also conducted experiments varying $lenSITs$ and obtained similar results, but we omit those for space constraints.

5.2.2 Varying the Total Number of Tables.

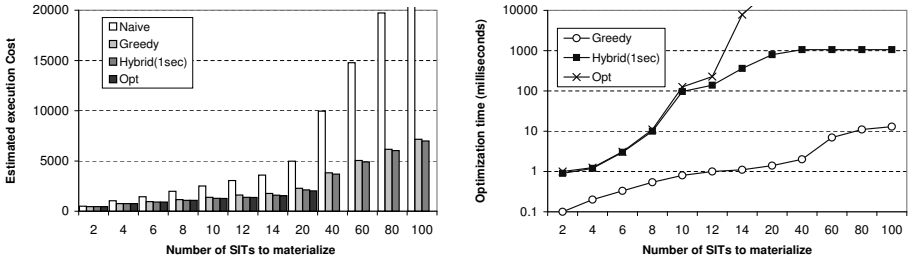
Figure 9 shows the average estimated cost to create set of SITs for varying number of tables. When we increase nt (keeping $numSITs$ fixed) the amount of overlapping between SITs is reduced. For that reason, all techniques tend to return similar schedules. In the extreme, if every SITs is composed of different tables, the *Naive* approach is optimal, since it performs just one sequential scan over each involved

[8] In this section we compare different algorithms that return schedules to create SITs, but we do not actually execute those schedules. Therefore, we do not populate the tables, but only specify their schemas and sizes.



(a) 2-way join query. (b) 3-way join query. (c) 4-way join query.

Figure 7. Creating SITs with skewed distributions in the join attributes of the generating queries.



(a) Estimated cost of the best plan. (b) Optimization cost to get the best plan.

Figure 8. Creating SITs with varying $numSITs$.

table. In real applications, however, the degree of commonality among SITs may be relatively high and thus necessitate the use of alternatives to *Naive*.

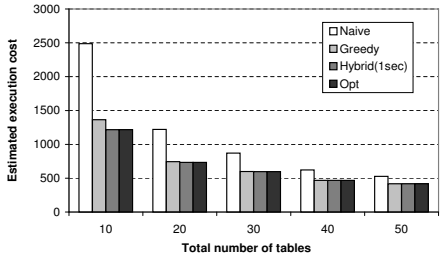


Figure 9. Creating SITs with varying number of tables.

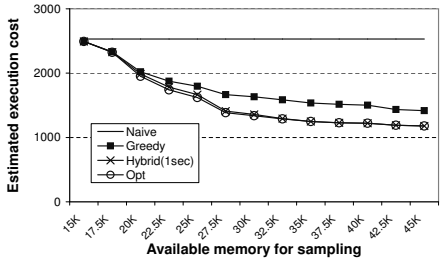


Figure 10. Creating SITs with varying memory limit.

5.2.3 Varying the Amount of Available Memory M

For this experiment, we varied the amount of available memory M from 15 Kb, which is the sample size of the largest table and therefore the minimal amount of memory for any algorithm, to 45 Kb which resulted in the same schedule as when M is not bounded (see Figure 10). *Naive* is not affected by memory constraints, since it maintains a single sample in memory at all times. The remaining techniques clearly benefit from extra memory and their schedules range from similar to twice as efficient as that of *Naive*.

6 Related Work

Virtually all relational optimizers [6, 7, 16] rely on base-table statistics to choose the most efficient execution plan in a cost-based manner. Histograms are the most common type of statistical information used in RDBMS, and there is a large body of work that studies histogram techniques on a given column [12, 13, 14]. In this paper we rely on existing histogram techniques and we focus on approximate attribute distributions over query expressions.

The idea of building statistics over non base-tables was introduced in [1] by using join synopses, which are precomputed samples of a small set of distinguished joins. The main focus of that work is approximate query processing, and the generating queries are restricted to be foreign-key joins. Reference [2] introduces the concept of SITs, which are statistics built over the result of executing query expressions, and presents an effective framework to incorporate them to existing query optimizers. It also introduces a workload-driven algorithm (based on the MNSA technique of [5]) to select conservatively a small subset of SITs that can significantly improve quality of query plans compared to using statistics on base-tables only. The present work studies the complementary problem of actually creating a given set of SITs.

There has been some work in the literature on trying to introduce sampling as a primitive relational operation [4, 11]. In particular, reference [4] studies some issues involved when trying to commute sampling and join operators. This work

introduces several algorithms to sample the result of a join operation without computing the entire join in the first place. The key difference with our approach is that in [4, 11] an actual *sample of the result* is needed. For that reason, all techniques process both tables and actually evaluate joins (at least in restricted ways), which is expensive. In our scenario, when creating SIT($S.a|R \bowtie S$) we are not interested in actual values from table R, since we only need an approximate distribution of $\pi_{S.a}(R \bowtie S)$. Therefore, we can apply efficient procedures that provide all required information.

Multi-query optimization aims at exploiting common subexpressions in the input queries to reduce evaluation cost. It is recognized that exhaustive algorithms for multi-query optimization are impractical, since they explore a doubly exponential search space. Reference [15] proposes cost-based heuristics that can be incorporated to existing optimizers, and shows that the algorithms provide significant benefits over traditional optimization with acceptable overhead. In this paper, we solve a more constrained optimization problem, since the execution plans for each individual SIT is known in advance (it corresponds to the unique sequence of applications of *Sweep* described in Section 3.2). For that reason, the search space is much smaller than in the general case, and we are able to adapt SCS to find the optimal global strategy. The Shortest Common Supersequence (SCS) is a well studied problem in the literature [8]. Finding the SCS of a set of sequences is NP-complete, and can be solved by dynamic programming in $\mathcal{O}(l^n)$ for n input sequences of length l [18]. Reference [9] introduces an A* algorithm to solve SCS which is generally more efficient than the classic dynamic programming technique. In this paper we modified such algorithm to fit our problem of finding an optimal schedule to create a set of SITs.

7 Conclusions

In this paper we studied the problem of efficiently creating statistics over query expressions, or SITs. We designed a family of techniques that balance accuracy and efficiency by approximating SITs. In particular, we discussed *Sweep*, which avoids relying on the independence assumption between attributes and requires a single scan over the tables involved in the SIT's generating query. Although *Sweep* uses sampling for its creation and does rely on the containment assumption, we showed experimentally that *Sweep* results in significant improvements in accuracy. We also studied the problem of scheduling basic applications of *Sweep* to create a set of SITs subject to space constraints. We proposed an algorithm for constructing the optimal schedule, and also presented greedy variants that are significantly more efficient without sacrificing the quality of the resulting schedules in most cases. An interesting problem that we defer to future work is to decide which SITs should be created using *Sweep*, and which ones would only be useful if built using more accurate (and therefore more expensive) techniques, such as the intermediate variants of Section 3.1.2.

Acknowledgements

We thank Luis Gravano for his valuable feedback.

References

[1] 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*, 1999.

[2] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings of the 2002 ACM International Conference on Management of Data*, 2002.

[3] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proceedings of the 1998 ACM International Conference on Management of Data*, 1998.

[4] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM International Conference on Management of Data*, 1999.

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

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

[7] 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*, 1989.

[8] D. Maier. The complexity of some problems on subsequences and supersequences. In *Journal of the ACM, 25*, 1978.

[9] G. Nicosia and G. Oriolo. Solving the shortest common supersequence problem. In *Operation Research. Springer-Verlag*, 2001.

[10] N. J. Nilsson. *Problem solving methods in artificial intelligence*. McGraw-Hill, 1971.

[11] F. Olken and D. Rotem. Random sampling from database files: A survey. In *Statistical and Scientific Database Management, 5th International Conference (SSDBM)*, 1990.

[12] 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*, 1984.

[13] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the Twenty-third International Conference on Very Large Databases*, 1997.

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

[15] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient algorithms for multi query optimization. In *Proceedings of the 2000 ACM International Conference on Management of Data*, 2000.

[16] P. G. Selinger et al. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM International Conference on Management of Data*, 1979.

[17] 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.

[18] V. G. Timkovskii. Complexity of common subsequence and supersequence and related problems. In *Journal of Cybernetics, 25*, 1990.

[19] J. S. Vitter. Random sampling with a reservoir. In *ACM Trans. Mathematical Software, 11*, 1985.