On Random Sampling over Joins

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Rajeev Motwani
Stanford University
rajeev@cs.stanford.edu

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Abstract

A major bottleneck in implementing sampling as a primitive relational operation is the inefficiency of sampling the output of a query. It is not even known whether it is possible to generate a sample of a join tree without first evaluating the join tree completely. We undertake a detailed study of this problem and attempt to analyze it in a variety of settings. We present theoretical results explaining the difficulty of this problem and setting limits on the efficiency that can be achieved. Based on new insights into the interaction between join and sampling, we develop join sampling techniques for the settings where our negative results do not apply. Our new sampling algorithms are significantly more efficient than those known earlier. We present experimental evaluation of our techniques on Microsoft's SQL Server 7.0.

1 Introduction

Data warehouses based on relational databases are becoming popular. The investment in data warehouses is targeted towards developing decision support applications that leverage the massive amount of data stored in data warehouses for a variety of business applications. On Line Analytical Processing (OLAP) and data mining are tools for analyzing large databases that are gaining popularity. Many of these tools serve as middleware or application servers that use a SQL database system as the backend data warehouse. They communicate data retrieval requests to the backend database through a relational (SQL) query. On a large database, the cost of executing such ad-hoc queries against the relational backend can be expensive. Fortunately, many data mining applications and statistical analysis techniques can use a sample of the data requested in the SQL query without compromising the results of the analysis. Likewise, OLAP servers that answer queries involving aggregation (e.g., "find total sales for all products in the North-West region between 1/1/98 and 1/15/98") can significantly benefit from the ability to present to the user an approximate answer computed from a sample of the result of the query posed to the relational database. It is well-known that for results of aggregation, sampling can be used accurately and efficiently. However, it is important to recognize that whether for data mining, OLAP, or other applications, *sampling must be supported on the result of an arbitrary SQL query*, not just on stored relations. For example, the preceding example of the OLAP query uses a star join between three tables (date, product, and sales).

This paper is concerned with supporting random sampling as a primitive operation in relational databases. In principle, this is easy — introduce into SQL an operation $\text{SAMPLE}(R, f)$ which produces a uniform random sample S that is an f-fraction of a relation R. While producing a random sample from a relation R is not entirely trivial, it is a well-studied problem and efficient strategies are available [10]. However, these techniques are not effective if sampling needs to be applied to a relation R produced by a query Q rather than to a base relation. It seems grossly inefficient to evaluate Q, computing the entire relation R, only to throw away most of it when applying $\text{SAMPLE}(R, f)$. It would be much more desirable and efficient to *partially* evaluate Q so as to generate only the sample of R.

For this purpose, it suffices to consider the case where we are given a query tree T with $\text{SAMPLE}(R, f)$ only at the root. More general cases, where the sample operation appears arbitrarily in the query tree can be reduced to the above case. In this setting, it seems plausible that tremendous gains in efficiency can be achieved by "pushing" the sample operation down the tree towards the leaves, since then we would be feeding only a small (random) fraction of the relations (stored as well as intermediate relations) into the query tree and thereby minimizing the cost of query evaluation. To this end, we need to be able to "commute" the sample operation with standard relational operations.

In our work, we consider only the problem of sampling the result of a join tree, since an efficient strategy for this special case is prerequisite to realizing the general goal of implementing sampling as a primitive relational operation. Furthermore, we believe that techniques required to completely solve the join tree problem would be an important step in dealing with the more general problem. In fact, we focus primarily on developing a technique for commuting sampling with a *single* join operation, since we could apply this technique repeatedly to push down the sample operator from the root to the leaves of a join tree. However, we cannot assume that the single join is applied to stored base relations, since its operands could be the output of subtrees of a join tree. We establish that it is not possible to produce a sample of the result of even a single join from random samples of the two relations participating in the join. Fortu-

nately, we are able to devise a technique for circumventing this negative result. Our key observation is that given some partial statistics (e.g., histograms) on the first operand relation, we can use the statistics to *bias* the sampling from the second relation in such a way that it becomes possible to produce a sample of the join. We devise a variety of sampling schemes based on these observations, improving the state-of-the-art for join sampling. In the context of a join tree, our work shows that it is possible to push down the sampling operation to *one* of the two operand relations. At the same time, our negative results show that it is inherently difficult to achieve greater efficiency by pushing sampling down to *both* operands of a join in a query tree.

There has not been much past work on supporting sampling as an operation for the end-user of a database system. While random sampling has been proposed and used in many different ways in databases [10], the main focus has been on the use of random sampling for the purposes of estimating query result size, aggregate values, and parameters for query optimization [11, 8, 5, 6, 3, 2, 1].

A notable exception is the work of Olken and Rotem [9] and Olken [10]. They focus on the issue of whether the sample operation can be commuted with standard relational operations. The easy case is that of selection, which can be freely interchanged with sampling. They point out that the situation is much more difficult with respect to projection and join. For projection, the issue is that of duplicate removal which skews the probability distribution; without duplicate removal, it is possible to commute sampling with projection. In the case of join, the problem is that the join of random samples of the operand relations does *not* give a random sample of the join of the operand relations. While Olken et al did suggest a technique for sampling the result of a join, it is far from satisfactory in terms of efficiency. Moreover, it requires that the relations being operated on should be *base* relations and have *indexes*. This limits applicability in the general setting where we are trying to push down sampling in a query tree, since the intermediate relations would typically not be materialized and indexed.

We emphasize that known techniques for estimating the size of a join have little or no bearing on our problem — our goal is to *create a sample* of the join that satisfies *precisely* the semantics of the SAMPLE operation, while the earlier estimation techniques apply to determining an *approximation* to the *size* of the join. Note that the recent work by Hellerstein, Haas, and Wang [4] explores the issue of supporting sampling interactively. However, they do not address the issue of commuting join and sampling; rather, they focus on the choice of join methods that perform progressive approximate sampling and support interactivity.

2 Summary of Results

We begin in Section 3 with a discussion of three possible semantics for the sample operator: *with-replacement*, *without-replacement*, and *coin-flip*. We also present some observations concerning our ability to switch between the various semantics. Then, in Section 4, we turn to the issue of an efficient implementation of the sample operator in isolation. While efficient implementations were known earlier for without-replacement and coin-flip semantics, we had to develop new techniques for the case of with-replacement semantics. Furthermore, it will turn out that weighted (or non-uniform) sampling is essential for dealing with join sampling, so we present extensions of all strategies to weighted sampling, which may be of independent interest. Most of our results apply to all three semantics for sampling, but we discuss mainly the case of WR semantics.

In Section 5, we tackle the problem of efficiently sampling the result of a single join operation. We begin with a discussion of the reasons why sampling does not commute with join: first, the join of random samples from R_1 and R_2 does not give a random sample of their join[1] $J = R_1 \bowtie R_2$; moreover, the projection of a random sample of J onto the attributes of (say) R_1 does not yield a *uniform* random sample of R_1 but instead gives a *biased* random sample where the probability of a tuple depends on the number of tuples in R_2 that join with it. This indicates that it is essential to use frequency statistics for join attribute values in R_2 to generate a suitable sample from R_1. Based on the latter insight, we are able to identify possible approaches to circumventing the difficulty of join sampling. Efficiency of sampling depends on whether the operand relations are materialized or merely streaming by, and the kind of information (indexes and statistics) available for them. We divide our analysis into three broad cases: **Case A** where no information is provided for either relation, **Case B** where information is available for only one relation, and **Case C** where information is available for both relations.

The *naive* strategy for sampling from the join J is to compute the full join and then sample therein. The question is whether we can improve efficiency by avoiding the need to compute the full join. This seems impossible for Case A where we do not have any frequency statistics for join attribute values in R_2 to help guide the sampling from R_1. This leaves Cases B and C, of which Case B has not been considered in the past. The earlier work of Olken et al [9, 10] considered Case C and proposed that the following strategy be applied repeatedly to generate a *with-replacement* sample: choose a random tuple from R_1, join it with all matching tuples in R_2, sample a single tuple from the result, and reject the sample appropriately to ensure a uniform probability distribution over all tuples in J. In Section 6 we propose a new strategy which has two major advantages: using our results for weighted sampling, we avoid the requirement of an index for R_1 or even that it be materialized; and, we improve efficiency by avoiding the need for rejection of the sampled tuples to ensure uniformity of the sample. Thus, not only does our strategy apply to Case B, it is also more efficient.

Next, we show how to handle more general settings closer to Case A, by reducing the dependence on the availability of statistics and indexes for R_2. To this end, we propose a suite of strategies based on the following observations. The main source of inefficiency in the naive strategy for Case A is the presence of join attribute values of high frequency (multiplicity). We develop hybrid strategies which use naive sampling for low-frequency values, and we provide new approaches for the high-frequency case.

The preceding discussion was concerned primarily with sampling from a single join. In Section 7 we discuss the application of our results to the problem of performing sampling from the output of a join tree, which involves pushing down the SAMPLE operation in the tree. We present some theoretical results showing that the natural approach for commuting sampling with join will not work. In this approach, the basic step is that of *oblivious* sampling: given random samples $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$, construct a random sample $S \subseteq R_1 \bowtie R_2$. Given this, the idea would be to start

[1] Throughout this paper, unless specified otherwise, by "join" or $R_1 \bowtie R_2$ we mean an equi-join of the relations R_1 and R_2 with respect to an attribute A.

with samples of the base relation at the leaves, and then repeatedly perform oblivious sampling to proceed bottom-up and obtain the final sample at the root. We show that it is *impossible* to preserve the semantics of random sampling when using oblivious sampling. This justifies our approach of *non-oblivious* sampling, where we use the frequency statistics (e.g., histograms) for R_2 to bias the sampling from R_1. For the non-oblivious case, we establish lower bounds on the *size* of the samples S_1 and S_2 required for producing a sample S of a given size.

We implemented our techniques on Microsoft SQL Server 7.0. Section 8 presents our experiments that involved varying the skew of the data distribution, index structures, as well as sampling fractions. We demonstrate that the methods proposed in this paper are efficient and consistently outperform earlier techniques. Note that where index/statistics are not available for both operand relations, the only technique known earlier was naive sampling. We conclude in Section 9 with a summary of our results and a discussion of future work.

3 Semantics of Sample

Consider the operation SAMPLE(R, f) which is supposed to produce a uniform random sample of R that contains an f-fraction of the tuples in R. This definition does not uniquely specify the semantics of the SAMPLE operation. In fact, there are at least three distinct interpretations of this definition as described below. Let n be the number of tuples in R.

Sampling with Replacement (WR): Sample fn tuples, uniformly and independently, from R. The sample is a bag (multiset) of fn tuples from R, as specific tuples could be sampled multiple times.

Sampling without Replacement (WoR): Sample fn *distinct* tuples from R, where each successive sample is chosen uniformly from the set of tuples not already sampled. The sample is a set of fn distinct tuples from R.

Independent Coin Flips (CF): For each tuple in R, choose it for the sample with probability f, independent of other tuples. This is like flipping a coin with bias f for each tuple in turn. The sample is a set of X distinct tuples from R, where X is a random variable with the binomial distribution [2] $B(n, f)$ and has expectation fn.

We make some observations concerning the conversion of one type of sampling into another.

1. Given a WR sampling process, we can convert it to WoR sampling by checking each new sampled tuple to see if it has already been generated and rejecting when that happens. There is only a minor loss in efficiency.

2. Given a CF sampling process, we can convert it to a WoR sampling by: sampling a slightly larger fraction f' to ensure that we get at least an f-fraction, as can be shown by the Chernoff bound [7]; and, then rejecting an appropriate number of the samples to ensure that we get exactly an f-fraction. The latter roughly corresponds to a WoR sampling of the CF sample.

3. Given a WoR sample, we can get a WR sample by sampling with replacement from the WoR sample, taking care to use the correct duplication probabilities.

4. To get a CF sample from either a WR or WoR sample is *impossible* since in the CF semantics we have a small but non-zero probability of sampling the entire relation. Therefore, any proper subset of the input relations cannot suffice to give CF semantics.

[2] In effect, this is the distribution of a random value generated counting the total number of heads when flipping n independent coins, each of which has probability p of turning up heads.

4 Algorithms for Sequential and Weighted Sampling

One important issue in the choice of sampling semantics is whether we can perform the sampling on a relation as it is streaming by, i.e., sampling in a single pass, or whether it requires some kind of random access to a materialized relation. Stream sampling is critical for efficiency even when the relation is materialized on disk since it permits sampling in a single pass, but it is even more important in situations where the relation is being produced by a pipeline (as in a query tree) and we *do not* wish to materialize it at all as it may be fairly large. We refer to such sampling as *sequential sampling*. Another important issue is whether the sampling is *unweighted* or *weighted*. In unweighted sampling, each element is sampled uniformly at random, while in weighted sampling each element is sampled with a probability proportional to its weight, for some pre-specified set of weights.

In this section we discuss strategies for performing the most general kind of sampling — weighted and sequential. The case of non-sequential sampling is comparatively easier and well-studied in the database literature [10].

Observe that CF semantics is particularly easy to work with in a streaming situation. Suppose that we wish to obtain SAMPLE(R, f) with CF semantics by making a single pass over R. We simply flip a coin for each tuple (with probability f for heads) as it goes by, adding the tuple to the random sample as if the corresponding coin flip turns up heads. There is also the standard "reservoir sampling" strategy for producing a sample without replacement from a relation in a single pass [12]. These two strategies have two key features: they do not need to know the size of the relation in advance; and, they produce samples in the same relative order as in the original relation. The former is useful for sampling a relation as it streams through a pipeline without any materialization on disk, and the latter preserves properties such as sortedness. The reservoir sampling algorithm has the disadvantage that no samples are produced until the entire process has terminated. However, when scanning a relation on disk, it can be made efficient by reading only those records that get into the reservoir, by generating random intervals of records to be skipped.

This leaves open the issue of performing unweighted sequential sampling for WR semantics. In Section 4.1, we propose two different strategies. In Section 4.2, we turn to the issue of *weighted* sequential sampling and generalize the unweighted sequential sampling WR algorithm to the weighted case. We can extend these algorithms to the case of weighted CF or WoR sampling, but the details are omitted. We will refer to the sampling algorithms as "black-boxes" since that is how they will later be used for join sampling.

4.1 Unweighted Sequential WR Sampling

The following black-box picks $r = fn$ tuples uniformly at random and with replacement from a relation with n tuples. Let $B(n, p)$ denote the binomial distribution with parameters n and p; there are standard algorithms for generating random values from this distribution.

Black-Box U1: Given relation R with n tuples, generate an unweighted WR sample of size r.

1. $x \leftarrow r$; $i \leftarrow 0$.
2. while tuples are streaming by and $x > 0$ do begin
   (a) get next tuple t;
   (b) generate random variable X from $B(x, \frac{1}{n-i})$;
   (c) output X copies of t;
   (d) $x \leftarrow x - X$;
   (e) $i \leftarrow i + 1$
   end.

One disadvantage of U1 is the need to know the size n of the relation being sampled. (This is in Step 2(b); in Step 1, we only need to know r which is specified as a part of the input.) Depending on the application, this may not be a big issue since: if the stream is a base relation, we already know the size of the relation; else, it is a random sample from a lower part of the tree in which case we have specified the size of the sample as a part of the sampling semantics, and so we know the size. On the other hand, this strategy has several advantages: it scans the relation (or has it streaming by in a pipeline) and produces the sample online in the same relative order as in the original relation; further, it does not need any significant auxiliary memory.

Theorem 1 *Block-Box U1 gives a WR sample of size r of a relation of size n in time $O(n)$ using $O(1)$ auxiliary memory.*

We remark that Black-Box U1 can be efficiently extended to block-level sampling on disk. Further, we can use the technique due to Vitter [12] of skipping over a random set of tuples (those for which X would have been 0), thereby improving efficiency. Similar comments apply to our second strategy which extends reservoir sampling to WR semantics.

Black-Box U2: Given relation R with n tuples, generate an unweighted WR sample of size r.

1. $N \leftarrow 0$.
2. Initialize reservoir array $A[1..r]$ with r dummy values.
3. while tuples are streaming by do begin
   (a) get next tuple t;
   (b) $N \leftarrow N + 1$;
   (c) for $j = 1$ to r do set $A[j]$ to t with probability $\frac{1}{N}$
   end.

Note that U2 does not need to know the size n of the relation being sampled. Moreover, it can be modified to produce the sample in the same order as in the original relation. Its drawbacks are the need to maintain the reservoir in memory (or in disk at additional I/O cost), and that it does not produce any tuples till the algorithm ends.

Theorem 2 *Black-Box U2 gives a WR sample of size r of a relation of size n in time $O(n)$ using $O(r)$ auxiliary memory.*

4.2 Black-Boxes for Weighted Sequential Sampling

In many applications, it is necessary to perform a weighted sampling instead of a uniform sampling of the tuples in a relation. We will soon see that weighted sampling is critical to some of our join sampling algorithms. As in the previous section, we wish to ensure that the sampling algorithm operates on a pipeline or a streamed relation. We describe extensions of the earlier black-box for WR semantics described to the weighted case.

First, let us specify the precise semantics of weighted WR sampling. We are given a relation R with a total of n tuples, where each tuple t has a specified weight $w(t)$. A weighted WR sample is obtained by repeating fn times the following: choose a tuple from R at random such that any tuple t is chosen with probability proportional to $w(t)$. That is, we perform a set of fn independent random selections from R, such that each random selection picks a tuple t with probability proportional to $w(t)$. The following is an equivalent definition for the case where $w(t)$ are non-negative integers.

Definition 1 *Assuming that $w(t)$ are non-negative integers, a* weighted WR sample *from R is the same as an unweighted WR sample from a modification of the relation R to a relation R^w in which there are $w(t)$ copies of each tuple $t \in R$.*

The motivation for this definition is the following. When sampling from the join of R with another relation S, we set the weights $w(t)$ to be the number of tuples in S that join with t; then, a WoR sample of the join of R and S will correspond to a sample from R with the same distribution as in the preceding definition. Sequential sampling for the weighted case should work as follows: a relation R of size n is streaming by such that each tuple t comes in with an associated weight $w(t)$, and we would like to perform a weighted WR sample of R to produce a total of fn samples. Consider the following extensions of the earlier black-boxes.

Black-Box WR1: Given relation R with n tuples, generate a weighted WR sample of size r.

1. $x \leftarrow r$; $D \leftarrow 0$.
2. $W \leftarrow \sum_{t \in T} w(t)$.
3. while tuples are streaming by and $x > 0$ do begin
   (a) get next tuple t with weight $w(t)$;
   (b) choose random variable X from $B(x, \frac{w(t)}{W-D})$;
   (c) output X copies of t;
   (d) $x \leftarrow x - X$;
   (e) $D \leftarrow D + w(t)$
   end.

Observe that just as U1 needs to know the size n of R, WR1 needs to know the total weight W of tuples in R; in general, WR1 has all the features of U1.

Theorem 3 *Black-Box WR1 gives a weighted WR sample of size r from a relation of size n in time $O(n)$ using $O(1)$ auxiliary memory.*

Similarly, we can extend U2 to the weighted case.

Black-Box WR2: Given relation R with n tuples, generate a weighted WR sample of size r.

1. $W \leftarrow 0$.
2. Initialize reservoir array $A[1..r]$ with r dummy values.
3. while tuples are streaming by do begin
   (a) get next tuple t with weight $w(t)$;
   (b) $W \leftarrow W + w(t)$;
   (c) for $j = 1$ to r do set $A[j]$ to t with prob. $\frac{w(t)}{W}$
   end.

Theorem 4 *Black-Box WR2 gives a weighted WR sample of size r from a relation of size n in time $O(n)$ using $O(r)$ auxiliary memory.*

We omit the definitions of the semantics and the adaptations of Black-Boxes WR1 and WR2 to the case of weighted sequential sampling for WoR and CF semantics.

5 The Join Sampling Problem

In this section we examine the problem of efficiently computing SAMPLE($R_1 \bowtie R_2, f$) in a variety of settings which differ in terms of materialization and indexing of the operand relations. We highlight the obstacles to solving this problem and identify possible approaches for circumventing the obstacles. We place previous work in this context and set the stage for describing our strategies in Section 6.

Assume that the two relations R_1 and R_2 are of size n_1 and n_2, respectively, and that we are interested in an equijoin with respect to an attribute A. We denote the domain of the attribute A by D. For each value $v \in D$, let $m_1(v)$ and $m_2(v)$ be the number of distinct tuples in R_1 and R_2, respectively, that contain value v in attribute A. Clearly,

$$\sum_{v \in D} m_1(v) = n_1 \quad \text{and} \quad \sum_{v \in D} m_2(v) = n_2.$$

Let $J = R_1 \bowtie R_2$ and define $n = |J| = |R_1 \bowtie R_2|$; clearly,

$$n = \sum_{v \in D} m_1(v) m_2(v).$$

For a tuple $t \in R_1$, let

$$J_t(R_2) = \{t' \in R_2 \mid t'.A = t.A\}$$

be the set of tuples in R_2 that join with t; further, define $t \bowtie R_2$ as the set of tuples in $R_1 \bowtie R_2$ obtained by joining t with the tuples in $J_t(R_2)$. Observe that $|J_t(R_2)| = |t \bowtie R_2| = m_2(t.A)$. Similarly, define for each $t \in R_2$ the sets $J_t(R_1)$ and $R_1 \bowtie t$, each being of size $m_1(t.A)$.

5.1 The Difficulty of Join Sampling

The following example will help illustrate some of the subtleties of the problem.

Example 1 Suppose that we have the relations

$$R_1(A, B) = \{(a_1, b_0), (a_2, b_1), (a_2, b_2), (a_2, b_3), \ldots, (a_2, b_k)\},$$

$$R_2(A, C) = \{(a_2, c_0), (a_1, c_1), (a_1, c_2), (a_1, c_3), \ldots, (a_1, c_k)\}.$$

That is, R_1 is defined over the attributes A and B; amongst its $n_1 = k+1$ tuples, one tuple has the A-value a_1 and k tuples have the A-value a_2, but all have distinct B-values. Similarly, R_2 is defined over the attributes A and C; amongst its $n_2 = k+1$ tuples, k tuples have the A-value a_1 and one tuple has the A-value a_2, but all have distinct C-values. Observe that their join over A, $J = R_1 \bowtie R_2$, is of size $n = 2k$ and has k tuples with A-value a_1 and k tuples with A-value a_2.

Assume that we wish to choose a random sample with WR semantics; our discussion below applies to the other two semantics as well. Consider a random sample $S \subseteq J$. We expect that roughly half of the tuples in S have A-value a_1, and roughly half of the tuples in S have A-value a_2.

Suppose we pick random samples $S_1 \subseteq R_1$ and $S_2 \subseteq R_2$. It is quite unlikely that S_1 will contain the tuple (a_1, b_0), or that S_2 will contain the tuple (a_2, c_0). Thus, given the samples S_1 and S_2, it is impossible to generate a random sample of $J = R_1 \bowtie R_2$ for any reasonable sampling fraction or under any reasonable sampling semantics. Note that this conclusion holds even if we allow (say) S_2 to be all of R_2 but require that S_1 be a *proper* subset of R_1. In fact, in all these cases we would expect $S_1 \bowtie S_2$ to be empty.

The problem is that the projection (after duplicate removal) of J onto attributes A and B does not give a uniform random sample of R_1. In fact, it gives a *weighted* sample of R_1 where each tuple of R_1 is sampled with probability dependent on the number of tuples in R_2 that join with it; specifically, in R_1 the tuple (a_1, b_0) is sampled with probability $1/2$ while the remaining tuples are sampled with probability $1/2k$ each, while in R_2 the tuple (a_2, c_0) is sampled with probability $1/2$ while the remaining tuples are sampled with probability $1/2k$ each. It is duplicate removal which causes the skewness of the resulting distribution. The extremely high skew in the relations R_1 and R_2 prevents samples of these relations from capturing attribute values that appear frequently in the join output. ∎

Thus, SAMPLE(R_1, f_1) $\bowtie$ SAMPLE(R_2, f_2) cannot generate SAMPLE($R_1 \bowtie R_2, f$) for any reasonable values of f_1 and f_2, when $f > 0$. In other words, SAMPLE does not commute with join. In fact, SAMPLE(R_1, f_1) $\bowtie$ SAMPLE(R_2, f_2) may not even contain any non-trivial size subset of J, and so further computation or sampling from it cannot be used to extract a sample of J. We will formalize later (in Section 7) the negative results implicit in this example.

Observe that the impossibility of commuting SAMPLE with join does not preclude the possibility of somehow obtaining SAMPLE($R_1 \bowtie R_2, f$) from *non-uniform samples* of R_1 and R_2. To better understand this point, consider the tuple $t = (a_1, b_0) \in R_1$ and its influence on $R_1 \bowtie R_2$. While $m_1(a_1) = 1$, the set $J_t(R_2)$ has size $m_2(a_1) = k$. Thus, even though a random sample of R_1 is unlikely to pick up the tuple with A-value a_1, half of the tuples in the join J have A-value a_1. This suggests that we sample a tuple $t \in R_1$ of join attribute value v with probability proportional to $m_2(v)$, in the hope that the resulting sample is more likely to reflect the structure of J. This is the basic insight behind most of our strategies given in Section 6.

5.2 The Role of Statistics in Join Sampling

The preceding discussion suggests that we sample tuples from R_1 based on frequency statistics for R_2. This requires that R_2 be materialized and indexed appropriately. This leads us to the following classification of the problem.

Case A: No information is available for either R_1 or R_2.

Case B: No information is available for R_1, but indexes and/or statistics are available for R_2.

Case C: Indexes/statistics are available for R_1 and R_2.

Observe that any sampling strategy for an earlier case will also apply to a later case (where more information is available). When no statistics are available for a relation, our strategies are such that we may as well as assume that the relation is not materialized and is being produced as a stream by a pipeline process. On the other hand, when information is available for a relation, a lot depends on whether merely statistical summaries or full indexes are available. We remark that the sampling strategy due to Olken et al [9, 10] applies only to Case C since it repeatedly samples tuples from R_1 using an index, and it also assumes full statistics and random access into R_2, which requires an index.

5.3 Previous Sampling Strategies

We conclude this section with a brief description of sampling strategies suggested in the literature.

In Case A, we do not have any frequency statistics for join attribute values in R_2 to help guide the sampling from R_1, and vice versa; therefore, the only possible approach appears to be the naive one of computing the full join $J = R_1 \bowtie R_2$, followed by rejection sampling where we reject each output tuple with probability $1/|J|$. There is one minor improvement we can give using our black-box for WR sampling. The idea is to avoid materializing the join J by performing the sampling sequentially, as shown below. To handle WoR or CF semantics, we use unweighted sequential sampling black-boxes for those semantics.

Strategy Naive-Sample:

1. Compute the join $J = R_1 \bowtie R_2$.
2. As the tuples of J stream by, use Black-Box U1 or U2 to produce SAMPLE($R_1 \bowtie R_2, f$).

Olken's sampling strategy, described below, applies only to the most restrictive case, Case C, since it requires a random access to and statistics for R_2, as well as the ability to repeatedly sample tuples from R_1. While it may appear that we could use sequential sampling for R_1 and thereby avoid the need for an index on it, this is not possible because the rejection process makes the number of samples required from R_1 a random variable whose distribution depends on the distribution of values in R_2. The situation is similar for the other sampling semantics.

Strategy Olken-Sample:

1. Let M be an upper bound on $m_2(v)$ for all $v \in D$.
2. repeat
   (a) Sample a tuple $t_1 \in R_1$, uniformly at random.
   (b) Sample a random tuple $t_2 \in R_2$ from among all tuples $t \in R_2$ that have $t.A = t_1.A$.
   (c) Output $t_1 \bowtie t_2$ with probability $m_2(t_2.A)/M$, and with the remaining probability reject the sample.

   until r tuples have been produced.

Theorem 5 (Olken [10]) *Strategy Olken-Sample produces a WR sample of $R_1 \bowtie R_2$ and requires $\frac{Mn_1}{n}$ iterations for each output tuple.*

6 New Strategies for Join Sampling

We now turn to the description of our new sampling strategies. The only non-trivial sampling strategy known earlier is Olken's strategy which applies only to Case C; also Case A does not permit any improvement over the naive sampling strategy. All our strategies apply to Case B, and therefore trivially to Case C as well. First, we improve upon Olken's strategy by describing Strategy Stream-Sample which does not require any information about R_1 and avoids the inefficiency of rejecting samples from R_2.

We develop another strategy, called Strategy Frequency-Partition-Sample, based on the insight that the naive sampling strategy performs badly only when the average frequency of attribute values is high enough to make the join size significantly larger than the size of the operand relations. The idea is to partition the operands into two subrelations, one with the high-frequency values and the other with the low-frequency values. For the low-frequency values, we can use the naive sampling strategy, but for the high-frequency values we need to develop more refined approaches as this is precisely the set of values for which computing the full join is expensive. For Frequency-Partition-Sample, we devise a high-frequency sampling strategy called Group-Sample. An important payoff of our hybrid sampling technique is that we do not need a full index on R_2 but merely some partial statistics (on the high frequency values). We also give some variants of our strategies that either promise higher efficiency or require less by way of information about the operands.

We summarize in Table 1 the information about R_1 and R_2 required by the previously-known and our new strategies. This illustrates an additional advantage of our strategies over Strategy Olken-Sample, beyond the sheer improvement in efficiency over both Naive-Sample and Olken-Sample.

Sampling Strategy	R_1 Info.	R_2 Info.
Naive-Sample	—	—
Olken-Sample	Index	Index/Stats.
Stream-Sample	—	Index/Stats.
Group-Sample	—	Statistics
Frequency-Partition-Sample	—	Partial Stats.

Table 1: Summary of information about R_1 and R_2 required by old and new sampling strategies.

We give details of our sampling strategies only for the case of sampling $r = fn$ tuples of J with WR semantics. To obtain WoR samples we employ the conversion trick discussed in Section 3, although most of the strategies below can be directly adapted to produce samples with WoR semantics. Our techniques also generalize to CF semantics.

6.1 Strategy Stream-Sample

We are now in the case where no information is available for R_1 and in fact we assume that it is only available as a stream from a pipeline process. We first describe a more sophisticated and efficient version of Strategy Olken-Sample that performs only a sequential sampling from R_1. Another key difference with respect to Strategy Olken-Sample is that we do not generate excess tuples only to reject them later, leading to greater efficiency. As in Olken's strategy, we assume availability of frequency statistics and an index for R_2.

Strategy Stream-Sample:

1. Use Black-Box WR1 or WR2 to produce a WR sample $S_1 \subseteq R_1$ of size r, where the weight $w(t)$ for a tuple $t \in R_1$ is set to $m_2(t.A)$.
2. while tuples of S_1 are streaming by do begin
   (a) get next tuple t_1 and let $v = t_1.A$;
   (b) sample a random tuple $t_2 \in R_2$ from among all tuples $t \in R_2$ that have $t.A = v$;
   (c) output $t_1 \bowtie t_2$

   end.

In Step 1, we sample each tuple t in R_1 with weight proportional to the frequency of its join attribute value in R_2. Several tuples in R_1 may have the same join attribute value and hence the same weight.

Theorem 6 *Stream-Sample gives a WR sample of $R_1 \bowtie R_2$ and requires only one iteration for each output tuple.*

6.2 Strategy Group-Sample

We now outline a strategy which works in the following scenario: no information is available for R_1 and the relation is merely streaming by; and, for R_2 we have frequency statistics specifying $m_2(v)$ for each $v \in D$. Notice that this is a special case of Case B. As will be shown in Section 6.3, this strategy can be adapted to be applicable even when only partial frequency statistics (e.g., end-biased histograms) are present.

Strategy Group-Sample:

1. Use Black-Box WR1 or WR2 to produce a WR sample $S_1 \subseteq R_1$ of size r, where the weight $w(t)$ for a tuple $t \in R_1$ is set to $m_2(t.A)$.
2. Let S_1 consist of the tuples $s_1, \ldots, s_r$. Join S_1 with R_2 to produce a relation S_2 whose tuples are grouped by S_1's tuples $s_1, \ldots, s_r$ that generated them.
3. From S_2, pick one tuple at random from each of the groups corresponding to S_1's tuples $s_1, \ldots, s_r$. For this, we use r separate invocations of Black-Box U1 or U2 to sample exactly one tuple from each group.

The cost of this scheme depends on the fraction of the full join computed as an intermediate result in Step 2. The following theorem gives the expectation α of this fraction. For relations without skew, where each join attribute value has frequency m, we have $\alpha = \frac{r}{md}$ where d is the number of distinct values common to the operand relations. We see significant efficiency improvement for small r. The efficiency of Group-Sample should be compared to that of Naive-Sample, which is the only strategy known earlier for this scenario.

Theorem 7 *Strategy Group-Sample gives a WR sample of $J = R_1 \bowtie R_2$ and computes a subset of J of expected size $\alpha \times |J|$, where*

$$\alpha = r \times \frac{\sum_{v \in D} m_1(v) m_2(v)^2}{\left(\sum_{v \in D} m_1(v) m_2(v)\right)^2}.$$

We note that in the case of a foreign-key join, where the join attribute is a key for R_2, the value of $m_2(v)$ is either 0 or 1 for each $v \in D$, and the bound on α can be improved.

6.3 Strategy Frequency-Partition-Sample

While having full statistics for R_2 can be advantageous, in general these may not be available without the presence of an index. However, it is reasonable to assume that we have a histogram for R_2 which gives the frequency statistics for all values with high frequency, say higher than t. We now show how to create a hybrid[3] strategy which involves logically partitioning the domain into two sets of values, high-frequency (in R_2) and low-frequency (in R_2), using Strategy Group-Sample on the former values and Strategy Naive-Sample on the latter values. Note that since a skew in frequency is exactly the problem illustrated in Example 1, it makes sense to handle high- and low-frequency values differently in a sampling strategy. A more crucial insight behind the hybrid strategy is that the size of the join is large precisely for the high-frequency values. Therefore, the key to better efficiency is to avoid computing the full join for that set of values. Working in our favor are the properties that there cannot be too many values of high frequency and that these are precisely the set of values for which we can maintain frequency statistics without incurring a high cost.

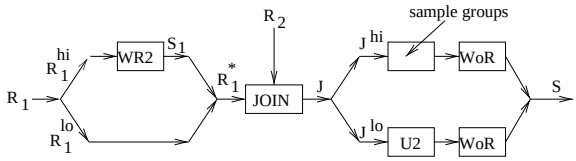

Figure 1: *Block Diagram for Frequency-Partition-Sample*

The join attribute values need not be of high frequency simultaneously in both operand relations, but a good approximation to the set of values which create a large number of tuples in the join are the values which have high frequency in only R_2. Another issue is to determine the distribution of the sample between the high-frequency and low-frequency subdomains. To this end, we sample r ruples from each subdomain, in the process determining the relative size of the join in the two subdomains, and later rejecting an appropriate number of tuples in each of the two samples. The number of samples from each subdomain can be determined by flipping a coin r times, where the probability of heads is the fraction of the tuples lying in the high-frequency subdomain. This gives us Strategy Frequency-Partition-Sample; refer to Figure 1 for a schematic description of this strategy.

Strategy Frequency-Partition-Sample:

1. Select a frequency threshold t for the domain D of the join attribute. Determine D^{hi} as the set of values in D that have frequency exceeding t in R_2. This requires having available an end-biased histogram[4] on $R_2.A$, containing frequencies of all values that occur t or more times, and let D^{lo} be the remaining values in D. This induces a partition of R_1 into $R_1^{hi} = R_1 \mid_{D^{hi}}$ and $R_1^{lo} = R_1 \mid_{D^{lo}}$, and R_2 into $R_2^{hi} = R_2 \mid_{D^{hi}}$ and $R_2^{lo} = R_2 \mid_{D^{lo}}$, where $R \mid_{D'}$ denotes selection of tuples from R with join attribute value in D'.
2. As the tuples from R_1 stream by, let the tuples from R_1^{lo} go through untouched. Filter the tuples of R_1^{hi} through Black-Box WR1 or WR2 employing frequency statistics from R_2^{hi} as weights to create r samples $S_1 \subseteq R_1^{hi}$ that are merged back into the stream. In the process, collect frequency statistics for R_1^{hi} and, combining with the frequency statistics for R_2^{hi}, determine the net size n_{hi} of $J^{hi} = R_1^{hi} \bowtie R_2^{hi}$. Denote the output stream by $R_1^* = S_1 \cup R_1^{lo}$.
3. Compute the join $J^* = R_1^* \bowtie R_2$. As J^* is being produced, determine the total number n_{lo} of the tuples in J^* that contain D^{lo} values, i.e., the size of $J^{lo} = R_1^{lo} \bowtie R_2^{lo}$.
4. As J^* streams by, partition it into J^{lo} and J^{hi}. Feed J^{lo} into Black-Box U1 or U2 to produce r samples (unweighted WR sampling). In J^{hi}, select one tuple at random from each of the r groups corresponding to the tuples $s_1, \ldots, s_r \in S_1$ (as in Strategy Group-Sample). Materialize both samples.
5. Flip r coins with heads probability proportional to n_{hi} and tails probability proportional to n_{lo}. Let r_{hi} be the number of heads and r_{lo} be the number of tails.

[3] For different reasons, Ganguly, Gibbons, Matias, and Silberschatz [2] also used a similar hybrid strategy for efficient estimation of join-size.

[4] Note that to ensure accuracy of Frequency-Partition-Sample, the histogram must be up to date. However, the cost of keeping the histogram updated may be traded-off by adjusting the parameter t.

6. Scan the two samples of size r from J^{lo} and J^{hi}. Feed the samples from J^{lo} into a black-box for sampling with WoR semantics to produce r_{lo} samples. Similarly, feed the samples from J^{hi} into a black-box for sampling with WoR semantics to produce r_{hi} samples.
7. Combine the r_{lo} samples from J^{lo} and the r_{hi} samples from J^{hi} to get the overall r samples.

A key advantage of this scheme is that it requires neither an index nor complete frequency statistics for R_2; rather, it merely requires summary statistics in the form of a histogram. For this reason, its efficiency should be compared to that of Strategy Naive-Sample which is the only strategy known earlier for this scenario. The performance is elucidated in the following theorem.

Theorem 8 *Frequency-Partition-Sample gives a WR sample of $J = R_1 \bowtie R_2$ and only needs to compute a subset of J whose expected size is an α-fraction of J's size, where*

$$\alpha = \frac{\sum_{v \in D^{lo}} m_1(v)m_2(v) + r \times \frac{\sum_{v \in D^{hi}} m_1(v)m_2(v)^2}{\sum_{v \in D^{hi}} m_1(v)m_2(v)}}{\sum_{v \in D} m_1(v)m_2(v)}.$$

The two terms in the expression for α correspond to the low-frequency and high-frequency values, respectively. In case of operand relations with high skew, the second term dominates and has the same behavior as in the case of the fraction α for Strategy Group-Sample.

6.4 Other Strategies

In case an index is also available (or can be quickly constructed) for the high-frequency values in R_2^{hi}, in addition to the frequency statistics, we would apply a variant of Strategy Frequency-Partition-Sample that is more efficient. (This is only a temporary requirement and we will shortly see how to eliminate the need for this index without affecting performance.) This variant is called Strategy Index-Sample and it does not compute the full join of S_1 with R_2^{hi} in Step 3. Instead, it uses the idea in Strategy Stream-Sample to directly compute a sample of size m from J^{hi} by joining each tuple $s_i \in S_1$ with a random tuple from $J_{s_i}(R_2)$.

Theorem 9 *Index-Sample produces a WR sample of $J = R_1 \bowtie R_2$ and only needs to compute a subset of J whose expected size is at most an α-fraction of J's size, where*

$$\alpha = \frac{r + \sum_{v \in D^{lo}} m_1(v)m_2(v)}{\sum_{v \in D} m_1(v)m_2(v)}.$$

It may appear unrealistic to assume the existence of an index for R_2^{hi}. But it is not very hard to see that in Strategy Index-Sample we can replace the requirement for an index on R_2^{hi} by a scan of R_2^{hi}. The following strategy, called Count-Sample, accomplishes this and can replace the use of Stream-Sample in the Index-Sample technique.

Strategy Count-Sample:

1. Use Black-Box WR1 or WR2 to produce a WR sample $S_1 \subseteq R_1$ of size r, where the weight $w(t)$ for a tuple $t \in R_1$ is set to $m_2(t.A)$.
2. Materialize S_1 and in the process, determine for each value $v \in D$ the number of tuples $s_1(v)$ in S_1 with join attribute value v.
3. Sample r tuples $S_2 \subseteq R_2$ such that the number of tuples with join attribute value v is exactly $s_1(v)$. This may be done as follows: while scanning R_2, for each tuple t with $t.A = v$ feed t into a copy of the Black-Box U1 with r replaced by $s_1(v)$ and n replaced by $m_2(v)$. That is, for each value v, a different black-box produces $s_1(v)$ samples *with replacement* from the $m_2(v)$ tuples of that value in R_2.
4. As each tuple of S_2 is produced, select a random tuple with the same join attribute value from S_1 and join the two tuples, placing the result in the output. Delete (or mark) the tuple of S_1 so that it is not selected again in the future; effectively, we are doing a sampling *without replacement* from S_1 at this stage. Alternately, S_2 can be materialized and randomly matched to S_2.

Unlike Strategy Stream-Sample, the above strategy does not need an index on R_2; instead, it merely scans R_2 once. Therefore, we can substitute Count-Sample for the use of Stream-Sample in Strategy Index-Sample. This yields a variant of Frequency-Partition-Sample and Index-Sample; we refer to this variant as Strategy Hybrid-Count-Sample. We plan to compare the performance of the new strategy with Frequency-Partition-Sample.

7 Extensions and Negative Results

The previous section was primarily concerned with the issue of sampling from a single join operation. We now turn to the question of implementing sampling as a primitive relational operation in the context of a join tree. As remarked earlier, the problem of dealing with a query tree with an arbitrary mixture of join and sampling can be reduced to the case where the sampling is applied only once to the relation R produced by a join tree. If we could devise an effective technique for pushing the sampling all the way down to leaves in this particular case, then we could repeatedly apply this technique to push all sampling operations to the leaves and thereby solve the more general problem.

Suppose we have a query Q generating a join tree T with a sampling operation applied to its result R. We are interested in the special case of *linear* join trees, which is a left-deep join tree where R_1 is joined to R_2, the result is joined to R_3, and so on; e.g., $R = ((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$ gives a left-deep tree with three joins in it. The strategies presented in Section 5 apply only to single joins. In this section we expose the inherent limitations on our ability to improve the efficiency of these strategies for single joins, and discuss their application to sampling from join trees.

We have already given some indication of the reasons why join sampling is a difficult problem even in the case of single joins, especially when full information is not provided for the operand relations. In Section 7.1, we show that sampling cannot be commuted with join in the natural way. This explains why our results in Section 5 apply only to Cases B and C where we are not restricted in our access to the operand relations to merely sampling uniformly therein, but are also allowed to use non-uniform sampling with cross-dependence on frequency statistics. Then, in Section 7.2, we consider the implications of our work for linear and arbitrary join trees. We argue that our results go a long way towards handling linear join trees efficiently, although they fall short of effectively dealing with arbitrary join trees.

7.1 The Inherent Difficulty of Join Sampling

Suppose we are given samples $S_1 = \text{SAMPLE}(R_1, f_1)$ and $S_2 = \text{SAMPLE}(R_2, f_2)$ under some choice of sampling semantics. We emphasize that S_1 is a uniform random sample of R_1 chosen independently of R_2, and S_2 is a uniform random sample of R_2 chosen independently of R_1. Given only S_1, S_2, and any desired set of statistics for R_1 and R_2, but without direct access to the tuples in R_1 and R_2, we are required to produce a sample $S = \text{SAMPLE}(R_1 \bowtie R_2, f)$ for some value of $f > 0$. The following results address the question of when it is possible to produce the sample S and how efficiently. All these results are independent of the sampling semantics.

We begin with a formal statement of the extremely negative result implicit in Example 1 — even when we are given *arbitrarily large samples* from R_1 and R_2, as well as *arbitrarily detailed statistics*, it is still not possible to generate any *non-empty*random sample of $R_1 \bowtie R_2$ under any of the sampling semantics.

Theorem 10 *Suppose that at least one of f_1, f_2 is strictly less than 1. Then, it is not possible to generate the sample $S = \text{SAMPLE}(R_1 \bowtie R_2, f)$ from S_1 and S_2 for any $f > 0$.*

This negative result prohibits commuting sampling with join by pushing down the sampling to both operand relations, regardless of the degree of inefficiency we are willing to tolerate. The proof idea is to consider R_1 and R_2 which both have high skew but on different values of the join attribute. Of course, one might argue that in practice we expect the high skew in both relations to be on the same values of the join attribute. Unfortunately, even there we can give fairly strong negative results, even for operand relations *without any skew*. That is, we now consider only the special case where each value $v \in D$ has frequency at most m_1 in R_1 and at most m_2 in R_2; call this the *uniform case*.

Theorem 11 *In the uniform case, it is not possible to generate $S = \text{SAMPLE}(R_1 \bowtie R_2, f)$ from S_1 and S_2, unless:*

- $f_1 \geq \frac{fm_2}{2}$ and $f_2 \geq \frac{fm_1}{2}$, *if* $f \leq \frac{1}{m}$,
- $f_1 \geq \frac{1}{2}$ and $f_2 \geq \frac{1}{2}$, *if* $f \geq \frac{1}{m'}$,

where $m = \max\{m_1, m_2\}$ and $m' = \min\{m_1, m_2\}$.

The preceding theorems show why it is necessary to perform non-oblivious and non-uniform sampling, where the probabilities of sampling tuples in R_1 depends on the frequencies of join attribute values in R_2. The next theorem shows that there are lower bounds on the sizes of the samples from R_1 and R_2 that are required to generate a sample from their join, even if the samples of R_1 and R_2 are generated in a non-oblivious manner.

Theorem 12 *It is not possible to produce the sample $S = \text{SAMPLE}(R_1 \bowtie R_2, f)$ from S_1 and S_2, unless $f_1 \times f_2 \geq f$.*

This theorem implies, for example, that when $f = 0.01$ and $f_1 = f_2$, then it must be the case that $f_1 \geq 0.1$ and $f_2 \geq 0.1$.

7.2 Dealing with Join Trees

The negative results in Section 7.1 state that it is essentially impossible to push down the sampling operation to both operands of a join operation, shedding some light on the difficulty of dealing with linear and arbitrary join trees. But can we at least push down the sampling operation to one of the two operands, say R_1, in some efficient manner? Note that this would be extremely useful for linear join trees where we could keep pushing down the sampling operator to the left operand; in fact, it would also give a partial solution for arbitrary join trees. The problem is that, as we argued in Section 5 and as formalized in Theorem 10, picking a sample from R_1 independent of the statistics of R_2 is no use in sampling the join. Consider sampling from $R_1 \bowtie R_2 \bowtie R_3$ — we cannot just select a uniform random sample of $R_1 \bowtie R_2$ but have to pick a non-uniform sample whose distribution depends on the statistics of R_3. But then, what about pushing the sampling further down the tree to R_1? Now, we will have to sample from R_1 using statistics for both R_2 and R_3. In principle, this can be done, since the operand relations are all base relations and their statistics can be precomputed. We defer a detailed analytical and experimental study of the efficiency of this strategy to later work.

From the point of view of efficiency, we would like to push down the sampling operator to *both* operand relations of the join node below the sample node. There are two major problems in doing so. First, this would require statistics on both operands, so as to permit appropriate sampling from the two operands. Such statistics will not be available in the first place unless both operands are base relations, which is not the case for any non-trivial join tree. Furthermore, even if we somehow managed to generate the two samples, it appears to be difficult to construct a uniform sample $S \subset J$ from the samples S_1 and S_2 of R_1 and R_2, respectively. To see this, consider the "projection" of the sample S onto R_1 and R_2, yielding two samples $S_1^* \subset R_1$ and $S_2^* \subset R_2$. Of course, the distribution of S_1 (respectively, S_2) is identical to that of S_1^* (respectively, S_2^*), since that was the whole point of the cross-dependent sampling strategy. But note that S_1^* and S_2^* are *highly correlated* — in particular, the total number of tuples with join attribute value v must be identical in the two samples, for any $v \in D$. This correlation is troublesome even for the case of a single join, and will be difficult to track as we keep pushing the sampling operations down different branches of the join tree.

8 Implementation and Experimental Evaluations

We implemented the following strategies for join sampling on Microsoft SQL Server 7.0 — Naive-Sample, Olken-Sample, Stream-Sample, and Frequency-Partition-Sample. In addition to modifications of existing join operators, these algorithms require the use of the black-boxes U1 and WR1 for producing unweighted and weighted sequential sampling with replacement. We implemented each of these black-boxes as operators. Because of the object-oriented design of code in SQL Server 7.0, adding an operator to the query execution tree only requires creating a derived class of a base operator class and implementing the necessary methods (e.g., *Open*, *Close*, and *GetRow*).

To implement Naive-Sample on the result of a join, we modified the execution tree generated by SQL Server optimizer by adding the U1 operator (unweighted WR sampling) as the root of the execution tree.

For Olken-Sample, we needed to sample tuples at random from the outer (left) relation R_1 in the join. For simplicity, we simulated such an access by first creating a uniform random sample of the key values of R_1 and put these values into a temporary table T_1. When T_1 is joined with R_1 (on the key value), it has the same effect as choosing random tuples from R_1. We also had to modify each join method in SQL Server, i.e., Nested Loops, Hash Join, and Merge Join,

so that the tuple from the outer relation is joined with a random tuple from R_2 among all tuples in R_2 that would have joined. For example, in the Nested Loops join method, for a given value v in a tuple from R_1, we pick a random number k between 1 and $f(v)$. Then, we skip $k-1$ matches of the join, and accept the kth matched tuple with probability $m_2(v)/M$, where M is an upper bound on $m_2(v)$ for all v. We then proceed to the next random value from R_1.

For Stream-Sample, we inserted the WR1 operator as a child of the join operator, i.e., between the scan operator on the outer-relation and the join operator. In the *GetRow* method of the WR1 operator, for the next tuple, the operator determines the number of copies that must be generated and produces that many copies. The statistics on the join attribute of R_2 are read from a file and stored in a work table. This table is indexed on the join attribute so that it is efficient to look up the frequency of a given value of the join attribute. We note that when the data distribution was moderately to highly skewed, the work table was usually small enough to fit into memory. Finally, we modified the join operator so that for each tuple sampled from R_1, we output exactly one tuple at random from among all the tuples that join with R_2.

For Frequency-Partition-Sample, we implemented a modified version of the WR1 operator for producing a random sample from R_1 of the tuples belonging to the high-frequency partition. The low-frequency tuples simply passed through this black-box. We simulated the effect of an end-biased histogram that keeps statistics on high-frequency values using the same mechanism as in Stream-Sample, i.e., the statistics were read in from a file and stored in a work table. We modified the join operator to additionally perform the Group-Sample operation to produce a sample of the join of high-frequency values (J^{hi}). We also added a WR1 operator on top of the join operator to produce the sample from J^{lo}. Finally, we added as the root of the query tree, an operator for (a) materializing the samples from J^{hi} and J^{lo}, and (b) picking samples without replacement of the required size from each materialized sample to produce the final output.

8.1 Experimental Setup

We generated four tables with 100K tuples each, and four tables with 1 million tuples each. Each table had an RID column which consists of unique randomly generated number between 1 and the number of tuples in the relation. The second column

was an integer column generated from a Zipfian distribution [13]. The second column in each of the four tables differed in the value of the Zipf parameter z used, which was set to 0, 1, 2, and 3, respectively. The data values in the second column were generated so that the most frequent value was picked in the same order in each case. The third column was used as padding (character field of 32 bytes) to ensure a reasonable record size. The queries that were run were of the form SELECT * FROM t1, t2 WHERE t1.col2 = t2.col2, where t1 was one of the 100K-tuple tables, and t2 was one of the million-tuple tables. In other words, the join attributes from both tables were the second column. This allowed us to study the effect of varying skew on the join columns on both the outer (left) and the inner (right) relation. The parameters varied in our experiments were:

- skew in data distribution of the join columns on the outer and inner relations,
- fraction to be sampled from the join,
- threshold used for statistics on the inner relation (relevant only for Frequency-Partition-Sample),
- indexes (i.e., the availability of indexes on the inner relation for Frequency-Partition-Sample).

8.2 Experimental Results

We summarize below the results of our experiments. The overall trend is that our new strategies are consistently better than earlier techniques. When indexes/statistics are not available on both operands, Frequency-Partition-Sample significantly outperforms Naive-Sample, the only other strategy known earlier in this situation. When indexes/statistics are available on both operands, Stream-Sample beats Olken-Sample. Moreover, Stream-Sample is also applicable (but Olken-Sample is not) when indexes/statistics are available only on the inner relation.

Varying the Sampling Fraction: We varied the sampling fraction for four different values: 100 tuples, $\sqrt{n}$, 1%, and 5%. We present the results for two cases of skew: (a) when skew in both columns are low, i.e., $z = (0,0)$; and, (b) when both skews are high, i.e., $z = (2,3)$. For Frequency-Partition-Sample, the threshold[5] for statistics was set at 5%. An index was present on the inner (right) relation. Figure A shows that at low skew values, Stream-Sample outperforms the other strategies at all sampling fractions. Further, the improvement with Olken-Sample drops more rapidly as the sampling fraction increases, since the number of random accesses to the outer relation increases. As expected, Frequency-Partition-Sample provides little improvement over naive sampling since the fraction of low-frequency values in R_2 is very high. However, as shown in Figure B, when the data in the join columns is highly skewed, both Stream-Sample and Frequency-Partition-Sample consistently outperform Olken-Sample at higher sampling fractions, i.e., 5% and 10%. This experiment demonstrates the superiority of Stream-Sample over the other strategies.

Varying the Skew: In this experiment we varied the skew on the inner relation over the values 0, 1, 2, and 3, while keeping the skew of the outer relation fixed. We measured the performance improvement of each strategy relative to that of Naive-Sample. An index was present on the inner relation, and for Frequency-Partition-Sample the statistics threshold was set to 5%. Figures C and D show that although for $z = 0$, Frequency-Partition-Sample is slower than the other strategies, for all other z values it is faster than Olken-Sample and Stream-Sample. Again, Stream-Sample is the best over a wide range of data distributions.

Performance of Frequency-Partition-Sample with no Index on Right: In this experiment, we study the performance of Frequency-Partition-Sample when there is no index on the inner relation, as the skew is varied. This experiment shows the generality of Frequency-Partition-Sample since it is the only applicable strategy when there is no index on the inner relation. Figure E shows that the strategy gives significant improvement in running time compared to the naive sampling, particularly when the data distribution on the join columns is skewed.

[5] A threshold of k% means that frequency counts are kept for all values which occur k% of the time or more in the relation.

Varying the Partition Threshold: In our next experiment, we varied the threshold for deciding which statistics to keep on the inner relation (this is relevant only to Frequency-Partition-Sample). We varied k over the values 0.1%, 0.5%, 1%, 2%, 5%, 10%, and 20%. Figure F shows that for highly skewed data the performance is not sensitive to the threshold parameter. This is because the number of distinct values in the relation is relatively small for highly skewed data and hence the cost of looking up the statistics is small. On the other hand, as Figure E shows, for data with low skew picking the right threshold is important. If a low threshold is picked, then too many infrequent values are part of the statistics, thereby increasing the overhead of looking up the statistics. On the other hand, with a high threshold, information about frequent values may not be available, resulting in a large number of values being unnecessarily joined. We conclude that a statistics threshold of 2% gives good performance over a wide range of data distributions.

9 Conclusions and Future Work

In this paper we have conducted a detailed study of some of the issues involved in introducing sampling as a primitive relational operation. We identified the inability to commute sampling with join as the major bottleneck in this regard. This led us to focus on the problem of efficiently sampling the result of a join operation without computing the entire join in the first place. We shed some light on the intrinsic difficulty of this problem and that suggested a possible approach for the case of a single join. One key understanding was that of the complications introduced by the presence of skew in the data, particularly in terms of requiring the dependence of the sample from one operand relation on the frequency statistics of the other operand relation of the join operation. This in turn led us to classify the problem into three cases based on the information available for the operand relations. We presented a series of sampling strategies that provide enhanced efficiency in a variety of settings with varying assumptions about the available information. Experimental results were provided to demonstrate the increased efficiency resulting from employing our sampling strategies. We also considered the more general problem of sampling the result of a join tree. While our techniques apply to the case of linear join trees, more work is required to fully validate this claim. On the other hand, our techniques are not entirely satisfactory for the case of arbitrary join trees. But perhaps this is inevitable, as we have provided strong negative results for the natural approaches to dealing with arbitrary join trees. In the process, we provided new schemes for sequential random sampling for uniform and weighted sampling distributions, which may be of interest in their own right.

Our work sets up an agenda for future work. It is possible that even more efficient strategies can be developed for the case of a single join based on observations made here. More work is needed to to understand the impact of our ideas on the problem of sampling the result of join trees. Also, in order to handle general query trees, it is essential to extend our work to other relational operators.

Acknowledgements We thank Jun Rao for discussions that helped develop the Frequency-Partition-Sample strategy; he also contributed to an initial implementation on SQL Server.

References

[1] S. Chaudhuri, R. Motwani, and V. Narasayya. Using Random Sampling for Histogram Construction. In *Proc. ACM SIGMOD Conference,* pages 436–447, 1998.

[2] S. Ganguly, P.B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal Sampling for Skew-Resistant Join Size Estimation. In *Proc. ACM SIGMOD Conference,* pages 271–281, 1996.

[3] P.J. Haas, J.F. Naughton, and A.N. Swami. On the Relative Cost of Sampling for Join Selectivity Estimation. In *Proc. 13th ACM PODS,* pages 14–24, 1994.

[4] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online Aggregation. In *Proc. ACM SIGMOD Conference,* pages 171–182, 1997.

[5] W. Hou, G. Ozsoyoglu, and E. Dogdu. Error-Constrained COUNT Query Evaluation in Relational Databases. In *Proc. ACM SIGMOD Conference,* pages 278–287, 1991.

[6] R.J. Lipton, J.F. Naughton, D.A. Schneider, and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. *Theoretical Computer Science* 116(1993): 195–226.

[7] R. Motwani and P. Raghavan. **Randomized Algorithms.** Cambridge University Press, 1995.

[8] J.F. Naughton and S. Seshadri. On Estimating the Size of Projections. In *Proc. Third International Conference on Database Theory,* pages 499–513, 1990.

[9] F. Olken and D. Rotem. Simple random sampling from relational databases. In *Proc. 12th VLDB,* pages 160–169, 1986.

[10] F. Olken. **Random Sampling from Databases.** PhD Dissertation, Computer Science, University of California at Berkeley, 1993.

[11] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. ACM SIGMOD Conference,* pages 256–276, 1984.

[12] J.S. Vitter. Random sampling with a reservoir. *ACM Trans. Mathematical Software*, 11 (1985): 37–57.

[13] G.E. Zipf. **Human Behavior and the Principle of Least Effort.** Addison-Wesley Press, Inc, 1949.

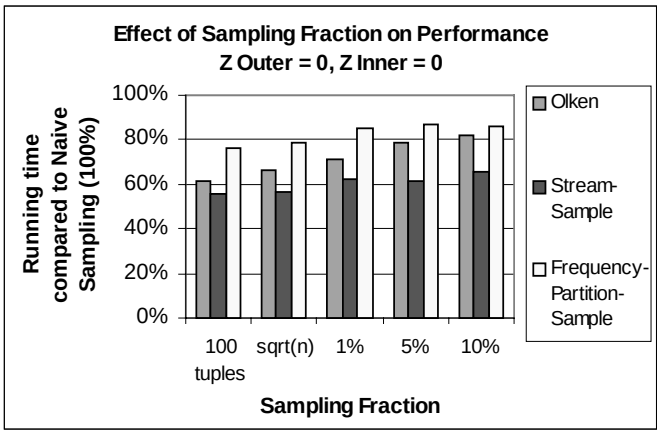

Figure A. Effect of Sampling Fraction on performance. Z=(0, 0).

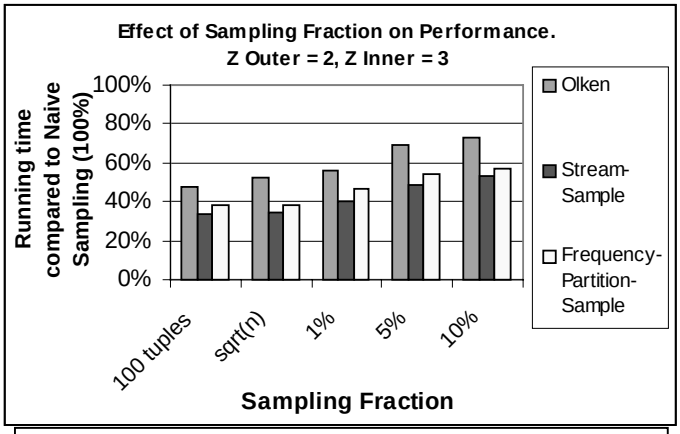

Figure B. Effect of Sampling Fraction on performance. Z=(2, 3).

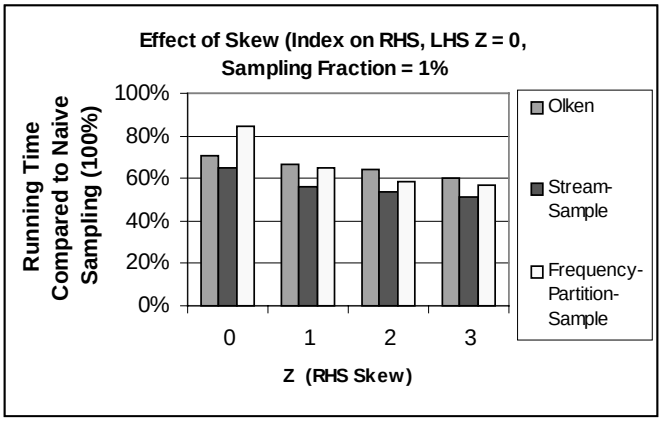

Figure C. Effect of skew on performance (LHS Z=0)

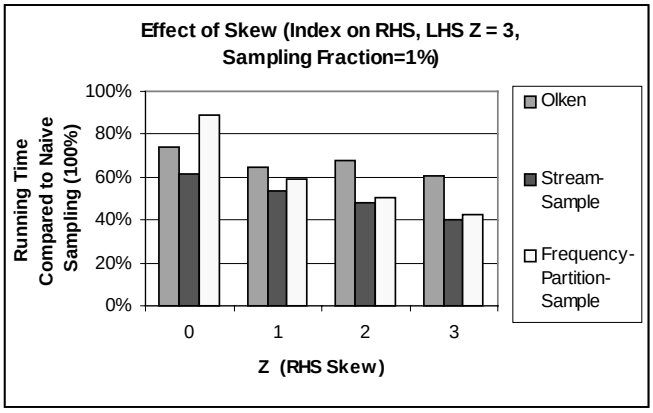

Figure D. Effect of skew on performance (LHS Z=3)

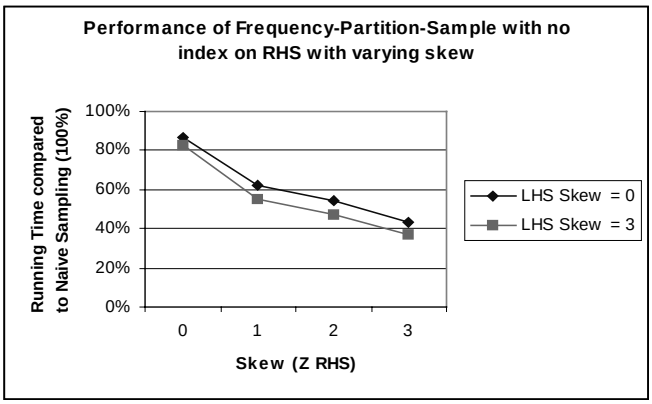

Figure E. Performance of Frequency-Partition-Sample with no index on inner relation in join.

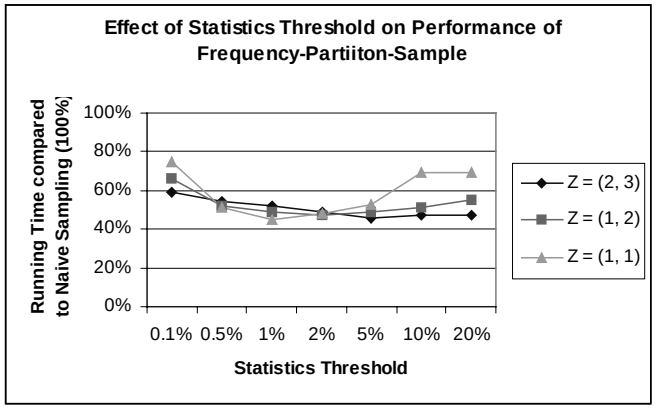

Figure F. Effect of varying Statistics Threshold on performance.