A Lightweight Online Framework For Query Progress Indicators

Chaitanya Mishra  
University of Toronto  
cmishra@cs.toronto.edu

Nick Koudas  
University of Toronto  
koudas@cs.toronto.edu

ABSTRACT

Recently there has been increasing interest in the development of progress indicators for SQL queries. In this paper we present a lightweight online framework for this problem. Our framework is online, in the sense that it refines its estimate of query progress based on feedback received during query execution. It is lightweight, since our techniques are designed to impose minimal overhead on query execution without sacrificing accuracy of estimates. Our framework can estimate progressively the output size of various relational operators and pipelines. These include binary and multiway joins as well as typical grouping operations and combinations thereof. We describe the various algorithms used to efficiently implement the estimators and present the results of a thorough evaluation of a prototype implementation of our framework in an open source data manager. Our results demonstrate the feasibility and practical utility of the approach presented herein.

1. INTRODUCTION

Recently there has been increasing interest in the development of progress indicators for SQL queries [18, 17, 9, 6]. A progress indicator aims to estimate precisely the value of a function that is related to the progress towards completion of a running query. Availability of such indicators can be of great help both to database administrators and end users. Given the complexity of queries in decision support applications, it is common for queries to take hours or days to terminate. Such indicators can greatly aid a user’s understanding of the progress of a query towards completion and allow the user to plan accordingly (e.g., terminate the query and/or change the query parameters). Similarly, from the point of view of administrators, unsatisfactory progress of queries may point to bad plans, poor tuning or inadequate access paths.

There are several important parameters associated with this problem. To begin with, one needs to specify the desired measure of query progress. Such a measure defines a model under which a progress indicator measures the amount of work done by the query, and predicts the total amount of work that will be done by it. The accuracy with which such measures can be approximated is crucial to the usefulness of the progress indicator. Moreover, the performance impact of any technique for approximating progress measures needs to be precisely analyzed and understood. Since a progress indicator runs while the query is being executed, it is essential that it imposes minimal overhead on the database engine.

Support for such functionality in current systems is fairly limited [1]. Recently [18, 17, 9, 6], a few solutions have been proposed which address this important problem. At a very high level, the approach taken is to decompose the query into a number of segments delimited by blocking operators. The papers define measures of work that are related to the cardinality of the operators in the query plan, and introduce techniques for approximating these measures over all segments of the plan. These techniques use both initial cardinality estimates from the optimizer, and information collected during query execution in order to arrive at an estimate of the progress of the query.

In this paper, we develop a lightweight online estimation framework that significantly benefits query progress indicators. Our framework can aid any measure of progress that relies on estimating intermediate cardinalities of operators in the query plan. By monitoring query execution at select points in the plan and collecting statistics from the tuples we’ve seen, we are able to not only accurately predict intermediate cardinalities, but also refine them as the query executes. Our estimates become increasingly more accurate, converging to the true values as the query executes and we are, under general distributional assumptions, able to analytically quantify their accuracy. We take special care to make our statistics collection and estimations lightweight. We experimentally demonstrate that our framework, when implemented in a real database management system, imposes minimal overhead on query execution time (for various types of queries), while enabling highly accurate estimation of query progress. More specifically in this paper we make the following contributions:

- We present our framework to estimate progressively the size of various types of relational (binary and multiway) joins. We present the types of guarantees one can obtain and show how it is can be efficiently realized and implemented inside a relational engine. We demonstrate how to tune the computation of our estimations in a way that imposes minimal overhead to query performance.

- We present our framework for estimating and progressively refining the estimates for the number of groups during GROUP-BY and related operations. Our estimates provably converge to the correct values. We identify the limitations of current state of the art estimators in this area and we present a new estimator that provides improved accuracy in certain cases. Based on this we propose techniques to choose the best estimator online for improved accuracy. We take special care
to make our techniques minimally intrusive to query performance in this case as well.

- We implement our framework in a real relational engine, reporting on our experience and the technical challenges. We also present our results of a thorough experimental evaluation, using benchmark as well as skewed data and report on the accuracy of our estimations as well as the overheads of our techniques.

In section 2 we review work related to this problem. Section 3 presents some background material necessary for the remainder of the paper. Section 4 presents our online estimation framework for various operators and query types. In section 5 we experimentally evaluate the performance, overheads and accuracy of our framework when implemented inside a real database management system. Section 6 concludes the paper and points to important directions for extending this work.

2. RELATED WORK

Progress indicators have been studied in various contexts (e.g., HCI [21] and file downloads) but there exists limited work on this topic in a data management context.

Recent work on query progress indicators [18, 17, 9, 6] introduced several novel ideas. Chaudhuri et al. [9] introduced the idea of decomposing a query plan into a number of segments (pipelines) delimited by blocking operators. Focusing inside each segment, they use the total number of getnext() calls made over all operators during the run of the query as an indicator of query progress. They use the progress at the driver node (the node that feeds tuples into the pipeline) as an indicator of the progress made in the pipeline, and use relational confidence intervals to refine optimizer estimates for cardinalities of pipelines that are yet to begin. Subsequent work [6] extended this approach proposing estimators that are worst case optimal, and showed that it is not possible to provide non trivial guarantees on progress estimation in the general case.

An independent but related approach was presented by Luo et al. [18]. This approach also makes use of segments for query partitioning. The model of work adopted is bytes processed at the input and output of segments. Since the bytes processed at these points is directly related to the cardinality of the operators which demarcate the segment, this measure of progress is analogous to the driver node estimator and the getnext() model of progress of Chaudhuri et al. [9]. In subsequent work [17], they increase the coverage of progress indicators to a broader class of SQL queries, and introduce refinements that capture the work done within operators in a more fine-grained manner. This work was also extended to handling multiple queries [19]. Although we phrase our presentation in terms of the getnext() model, our techniques are equally applicable to this model as well.

Haas and Hellerstein [11] presented a framework for online aggregation. In this work, aggregate values are progressively approximated in an online fashion as the query executes. Accuracy guarantees are provided at any point of the execution and the confidence to the final estimate improves as execution progresses. We would like to provide similar behaviour and guarantees for query progress estimation.

Recent work on adaptive query processing and optimization [16, 20, 2] makes use of statistics collection at various points in the query plan for reoptimization purposes. Specifically, Markl et al. [20] collect the number of tuples output by an operator and compare this number with a precomputed range of values (a check point) that guarantees the optimality of the remaining plan. Similarly, Kabra and Dewitt [16] also collect the number of tuples processed by an operator. In both cases however, the number is compared with existing estimates or validity ranges. In contrast, Babu et al. [2] use random sampling in order to re-optimize proactively during query execution. However, the estimation framework is quite limited as it addresses only the case of primary-foreign key hash joins.

Sampling based techniques for estimation of join result sizes [10, 7] aim to address issues related to skew in the underlying data distributions, but they cannot easily adapt to an online scenario in which data progressively arrive. Such techniques are not designed to operate incrementally, but rather return an estimate after a sample of a specific size, obtained in a way specific to the problem, is obtained. Moreover their runtime overhead is relatively large as they have to essentially join part of the relations in memory. We seek techniques to obtain such estimates while imposing minimal overhead at query execution time.

A large body of work exists on estimation of the distinct number of values of an attribute (e.g., see [12, 5] and references therein). The estimator (GEE) proposed in [5] has nice analytical properties with optimality guarantees. However, it may lead to severe under/over estimates depending on the distribution. A heuristic estimator (AE) proposed in [5] combines features of various estimators, but the overhead of computing the right parameters to obtain the estimate is rather high since requires use of numerical methods to solve equations. For progress estimation, it is imperative to minimize runtime overhead since the estimation takes place continuously as the query execution evolves.

3. MODEL AND ASSUMPTIONS

In this section, we describe our computational model, the measure of progress we adopt, and provide definitions necessary for the remainder of the paper.

In accordance to previous work [18, 17, 6, 9] we assume query plans consisting of a tree of physical operators. The set of operators we consider are scan, π, ∃I NL, ∃I INL, ∃I hash, ∃I merge, sort, γ (group by). A query plan consists of one or more pipelines. Pipelines are defined as maximal subtrees of concurrently executing operators. For example, in Figure 1, the shaded region consists of one pipeline (a merge join and the index scans feeding it), and the non shaded region consists of another (a hash join, and its probe input). We refer to [9, 18] for a discussion of the various pipelines related to each blocking operator.

We adopt the model of progress introduced in [9] which is essentially based on the number of getnext() calls invoked in a query tree. Throughout this text, the terms getnext() model, and gnm are used interchangeably to refer to this model. The notion of query progress introduced by Luo et al. [18] is similar to gnm and our discussion applies to progress estimators using this model as well.

Suppose that the execution of a query Q involves n operators. If for all operators Qi, we denote the total number of getnext() calls over the execution of the query as NQi, and the number of getnext() calls made thus far as KQi, then the gnm measure of progress is \[ \frac{\sum_{i=1}^{n} K_i}{\sum_{i=1}^{n} N_i} \]. We denote the current number of getnext() calls
made as $C(Q) = \sum_{i} m_i K_i$, and the total number of $getnext()$ calls over the lifetime of the query as $T(Q) = \sum_{i} m_i N_i$.

Two observations are in order here. First, $C(Q)$ can be computed easily by just observing and counting the number of tuples produced by all operators in the plan. Secondly, estimating $T(Q)$ is equivalent to the problem of cardinality estimation. This problem, despite the large volume of research devoted to it, due to its complex nature (e.g., see [15]), remains an open challenge for general and fairly complex query plans. Notice that if one has accurate values for $T(Q)$, the problem of progress estimation can be easily solved, under the $getnext()$ model of query progress. Thus, under this model, the problem of progress estimation reduces to the problem of estimating and refining $T(Q)$ as the query executes. Previous work on progress estimation [18, 17, 6, 9] split this problem into the problem of refining the estimate of cardinalities of operators in the currently executing pipeline, and the problem of refining the estimate of cardinalities for operators in future pipelines. Our focus in this work is on faster and more accurate estimations for the currently executing pipeline.

Our framework does not require, but can make use of base table statistics. Such statistics are commonly histograms of the attribute value distribution of single base table attributes. We also assume knowledge of the size of base tables, which is usually available in the system catalogs. In order to provide confidence guarantees on our estimates, we require that table scans on base relations obtain on demand a (or have access to a precomputed) random sample of a specific size from disk. Our techniques make use of such samples of a specific size to derive statistically precise estimates of various quantities. Once such estimates are obtained, base tables can be read (in the order determined by the plan), while excluding tuples that were already in the sample. This is similar to the requirements of previous work on online aggregation [11] and reoptimization [2]. We analyze the sizes of samples required, and associated overheads in Section 5. For ease of exposition, we assume randomly ordered tuple streams in the description that follows. We make assumptions of independence between columns of the same relation in order to simplify the framework presented here. We note that this assumption is not necessary, however dealing with attribute correlations will impose additional overheads. Future work will address issues and complexities involved with attribute correlations.

4. OUR APPROACH

In this section, we present our framework for progress estimation of operators and query segments using the $getnext()$ model of query progress. We present our techniques for cardinality estimation for joins, aggregation, selections. We then explain how our estimators fit in the $getnext()$ model.

4.1 Joins

Consider a stream $T$ of tuples at the output of some relational operator. Assume that the size of this stream is either known exactly or an estimate exists for its size, say $|T|$. For an attribute $A$ of this tuple stream, assume that attribute values are observed in random order. For each attribute value $v$ we like to estimate the fraction of tuples with this specific attribute value in the stream. Let $p_t(v)$ be the true fraction for attribute value $v$ (we drop the superscript and use it only when required in the sequel). Thus $p_t$ is the probability with which a newly observed value is equal to $v$. We would like to obtain an estimate of $p_t$ which we refer to as $\hat{p}_t$. We assume that $p_t$ is independent and of all other $p_{t,j}$, $i \neq j$.

We will determine a $p_t$ such that $\hat{p}_t \in (p_t - \frac{\beta}{2}, p_t + \frac{\beta}{2})$ with probability greater than $\alpha$. Assume we can maintain each attribute value $i$ observed, along with the count of attribute values $i$ seen after $t$ tuples have been observed from $T$. We refer to this count using $N^R_i$. This quantity is updated incrementally for each $i$ as $t$ increases. Since $N^R_i$ is a binomial random variable with probability of success $p_i$, from the standard approximation of the binomial with a normal distribution we have:

$$\sqrt{\frac{N^R_i}{t} - p_i} = \Phi(0, p_i(1 - p_i))$$

where $\Phi(a,b)$ is the normal distribution with mean $a$ and variance $b$. From this, we can derive an expression for the confidence $\beta$ with which $\hat{p}_t$ approximates $p_t$. The maximum likelihood estimate for $\hat{p}_t$ is $\frac{N^R_i}{t}$. As a result, the $\alpha$ percentile confidence interval is provided as: $\hat{p}_t \pm Z_\alpha \sqrt{\frac{\hat{p}_t(1 - \hat{p}_t)}{t}}$. $Z_\alpha$ can be obtained from standardized normal tables, for the $\alpha$ percentile confidence interval (e.g., for $\alpha = 99.99\%$, $Z_\alpha = 4$). Since $p_t$ is not known, we can obtain a bound on its variance by observing that $p(1 - p)$ is maximized at $p = \frac{1}{2}$. Thus, since $\hat{p}_t = \frac{N^R_i}{t}$, making the right substitutions, we obtain that $\beta = \frac{\beta}{2^2}$. This provides us with an expression on how the confidence of our estimate improves with desired probability as we observe more elements of the tuple stream.

Now consider two tuple streams, $R$ and $S$ with known or estimated sizes $|R|$ and $|S|$ respectively. After $t$ tuples have been observed from both streams, assume we can afford to maintain precise values for $N^R_i$ and $N^S_i$ for attribute values $i$ appearing in attributes $A$ of $R$ and $B$ of $S$. Our estimate for the equijoin size of $R$ and $S$ on attributes $A$ and $B$ after seeing $t$ tuples, denoted $D_t$ can be obtained as:

$$D_t = \sum_{\forall i \in A \cap B} (p^R_i \pm \frac{\beta}{2}) |R|[p^S_i \pm \frac{\beta}{2}] |S|$$

$$= |R||S| \sum_{\forall i \in A \cap B} \frac{N^R_i N^S_i}{t^2} \pm \sqrt{\frac{N^R_i + N^S_i}{2t} \beta \pm \frac{\beta^2}{4}}$$

As $\beta$ becomes smaller while $t$ increases, the estimate converges to the correct value of the join size. This basic formula can be easily adjusted for the case of join conditions involving disjunctions and conjunctions of multiple attributes, using standard probabilistic techniques (details omitted due to space limitations).

The overhead of such a scheme is very high however. One has to maintain and incrementally adjust $N^R_i$ and $N^S_i$ which imposes memory and processing overhead. Moreover, such counts have to be correlated to obtain an estimate, thus imposing additional processing overhead. We show how to overcome the limitations of this basic scheme next.

4.1.1 Binary Hash Joins

Consider a hash join between 2 relations $R$ and $S$ with $|R|$ and $|S|$ tuples respectively. Let $R$ be the build input, and $S$ be the probe input of the join. Assume we compute an equijoin on attributes $A$ and $B$ of $R$ and $S$. Suppose that while partitioning $R$, we count the number of times each value in the join attribute is seen. This operation can be interleaved with the actual partitioning to keep overheads low (we provide implementation details in Section 5). In other words, we build a histogram that maintains a count $N^R_i$ for each value $i$ in $R$. Once the build input has been partitioned, the hash join reads in the probe input. In a similar fashion, we can count the occurrence of each value in the probe input as it is being.

Additional techniques to strengthen the utility of limit theorems can be easily adapted for improved estimation accuracy e.g. See [3]
read. Since \( N_i^R \) counts are exact, after \( t \) tuples of \( S \) are read, \( D_t \) becomes:

\[
D_t = \sum_{i \in A \cap B} \frac{N_i^R \cdot N_i^S}{t} |S| \pm \sqrt{\frac{N_i^R |S| \beta}{2}}.
\]

This estimator has zero error in expectation and the confidence bounds shrink with increasing number of tuples read. However, if we implement such an estimator in a hash join in its current format, we would have to build histograms on both the inputs during the partitioning phase and multiply the counts of corresponding buckets at regular intervals. Instead, it is possible to express \( D_{t+1} \) in terms of \( D_t \). If the \( (t+1) \)th tuple has the value \( i \) on its join attribute, we have

\[
D_{t+1} = \frac{D_t \cdot t + N_i^R |S|}{t+1}.
\]

Notice that this formula does not require us to build a histogram on the probe input. In addition, it also saves us from the expensive operation of multiplying corresponding buckets. For each tuple of \( S \), we probe the histogram built on \( R \) to get \( N_i^R \) and update our estimate as per the above formula. The computational overhead of this operation is negligible; the only overhead is storage for the histogram on \( R \) (see Section 5.2.2).

An important advantage of using this estimator is that it converges to the exact cardinality of the join by the end of the first pass on the probe input. The hybrid hash join and the grace hash join algorithms partition the probe input on the first pass into buckets, and leave most (or all in the case of grace hash join) join processing to the second pass. In contrast, the driver node estimator (\textit{dne}) of [9] and the estimator of Luo et al., [18] do not guarantee convergence until the entire join has been processed. Also, since we do estimation in the first pass of the join, we do not face the problems caused by reordering of the probe input. Recall, that due to hash partitioning all tuples with the same value are clustered into the same partition. The actual join is done by reading data partitionwise, and so the \textit{dne} [9] estimate of the output may fluctuate significantly if certain partitions contain tuples that join with many tuples of the build relation while others contain tuples that join with very few tuples. Similar problems can be observed with the estimator of Luo et al., [18]. Our estimator sidesteps this problem of reordering of probe input since it conducts estimation in the hash partitioning pass itself.

It is evident that we do not really need the probe input to the hash join to be the output of a table scan. As long as the input to a hash join (or part of it) can be considered to be randomly ordered, our estimation framework should work, irrespective of whether the probe input is the output of a table scan, or some other operator. We describe our techniques for handling some common cases of non-randomly ordered probe input in Section 4.1.4.

Finally, although the above discussion applies to an equijoin between 2 relations, similar estimators can be constructed for other kinds of join predicates (e.g., \( R.x > S.y \)). Moreover, although we omit details due to lack of space, we mention that similar estimators can be constructed for semijoins and various kinds of outerjoins as well.

### 4.1.2 Binary Sort-Merge Joins

The estimator described above can be used for the case of sort-merge joins as well. Consider a sort-merge join between two base relations \( R \) and \( S \). Both \( R \) and \( S \) are scanned in random order and thus before the join begins, both the relations need to be sorted. Suppose \( R \) is sorted first. In the sort operator, every tuple of \( R \) is seen at least once before any output is produced. Thus, it is possible to build a histogram on the join attribute of \( R \) which maintains the number of times each value is seen in \( R \). Note that this can be done even if the sorting is done within the sort-merge join and not in some separate sort operator. Now when \( S \) is being sorted, we can probe the histogram on \( R \) to get the estimate of the cardinality of the join in the same manner as in the case of binary hash joins. In this manner, we get the exact cardinality estimate of the join at the end of the sort of \( S \). In contrast, the \textit{dne} [9] and the estimator of Luo et al., [18] conduct cardinality refinement while the join is being processed. In addition, since a sorted input stream is necessarily clustered, their estimates fluctuate in the case of skewed data. If the relations are already sorted, it is not possible to push down estimation to a preprocessing phase. In such cases we default to the usual \textit{dne} estimate.

#### 4.1.3 Nested-Loops Joins

In hash joins and sort-merge joins, both inputs are read completely before any significant join processing is done. By building a histogram on one of the input relations, we are able to simulate the join in the preprocessing phase itself and obtain an estimate that converges progressively to the exact value. In the case of a nested-loops join, there is no such preprocessing of the outer input; it is joined as it is read in. Assume a nested-loops join of two relations \( R \) (outer) and \( S \) (inner) on attributes \( x \) and \( y \) respectively. A histogram on \( B \) may be available or can be obtained precisely after a single pass of \( S \). Building a histogram incrementally on \( x \) as we scan \( R \) however, will not aid estimations. All one can do is to scale the contribution of \( i \in x \) to the join by the size of \( R \). This however is exactly what the \textit{dne} estimator does. Thus, in the case of nested-loops join our estimation is equivalent to that of the \textit{dne} estimator.

In practice, nested-loops joins are often optimized by building temporary indices on the inner input (in the absence of a permanent index), and by pre-sorting the outer input to improve locality in memory access. In the presence of such preprocessing phases, we can construct estimators similar to the incremental estimator for hash joins, and get early estimates.

#### 4.1.4 Join Pipelines

We now consider the case of multiple join operators on a pipeline. A common construction is a pipeline of hash joins, each taking the output of a lower hash join as its probe input. In such pipelines, it is possible that the join columns of a lower join are involved in the upper join as well. We refer to such cases as \textit{joins on the same attribute}. We refer to cases where the join columns for the joins in the pipeline are different as \textit{joins on different attributes}. Note that this definition does not have anything to do with the names of the joining columns. If the lower join in Figure 2 (a) had the condition \( B.x = C.y \), and the upper join had the condition \( A.z = B.x \), it would be still considered a case of \textit{joins on the same attribute} since the join column of the lower join is involved in the upper join as well.

If the probe input itself is the output of a hash join on the same attributes, the probe input cannot be considered random anymore. In Figure 2 (a), suppose the joins are implemented as hash joins with the output of the lower hash join (\( JL \)) being the probe input to the upper hash join (\( JU \)). The probe input (\( C \)) of the lower join is joined with the build input (\( B \)) in a partitionwise manner. Therefore, the output of the join is clustered on attribute \( x \) as well, and we cannot assume that it is a randomly ordered input to the upper join. A similar problem arises if the joins for Figure 2 (a) are implemented as sort-merge joins. The output of the lower join is sorted on attribute \( x \), and does not need to be sorted again by a sort...
operator before it is processed by the upper join. In this case too, the randomness assumption is violated. Handling such clustering of output of a hash or sort-merge join is necessary if we have a pipeline of joins on the same attribute.

For ease of exposition we present our handling of both cases for pipelines of two hash joins (illustrated in Figure 2) (a) and (b), before we present the general case. We identify the following two cases:

4.1.4.1 Hash Joins On The Same Attribute.

In this case we have a pipeline with joins on the same attribute. It is possible to push the cardinality estimation for the upper join down to the lower join. If the joins in Figure 2 (a) are implemented as hash joins, A is the build input for the upper hash join while B is the build input for the lower hash join. We maintain counts $N^A_i$ and $N^B_i$ for each value i seen in the build phase for both joins. As we read the probe input, each tuple with value i in the join attribute will contribute $N^A_i \cdot N^B_i$ tuples to the output of the upper join. Thus the cardinality estimate for the upper join can keep getting refined as:

$$D_{i+1} = \frac{D_i t + N^A_i N^B_i |C|}{t + 1}$$

By the end of the first pass of the lower join, our estimator gives the correct estimate of the cardinality at the upper join as well. Such push-down estimation generalizes to any number of hash joins on the same join attribute. The central idea is to extract the value of the join attribute from the probe input at the lowermost join and to probe all the histograms of the build inputs in the pipeline with this value as illustrated in Figure 2 (a). Multiplying the counts returned by each of these histograms gives us the number of tuples of output that the given probe input will produce.

4.1.4.2 Hash Joins on Different Attributes.

In this case too, it is possible to push down cardinality estimation of all joins to the lowest join in the join tree. Doing so allows us to reach correct estimates for all joins in the pipeline at the end of the first pass on the probe input at the lowermost join.

Consider the join tree shown in Figure 2 (a). If the joins were on different attributes, the join condition on the upper join $JU$ could be either of the form $A.y = C.y$ or $A.y = B.y$, i.e., the join attribute of the probe input of the upper join could be an attribute of either the probe input or the build input of the lower join. We now describe these cases in detail.

- **Case 1** $A.y = C.y$: We build histograms on the upper and lower build inputs to the joins on their join columns. With histograms on $A.y$ and $B.x$ built during the build phase, we can start reading $C$. Now, any tuple of $C$ with column values $(x = x_1, y = y_1)$ will produce $N^A_{y_1} N^B_{x_1}$ tuples in the output of the upper join. Since this information is readily available in the histograms, we can do estimation as before. By the end of the first pass on the probe input of the lower join, we know the cardinality of the output of the upper join precisely.

- **Case 2** $A.y = B.y$: This case cannot be handled in the same manner as above. If we built histograms on columns $A.y$ and $B.x$ at the upper and lower joins, we would not be able to probe the upper histogram using the probe input of the lower join since there is no column $C.x$. To get around this problem, we use the fact that it is possible to simulate the join of relations $A$ and $B$ on column $y$ while building the hash partition for $A$. Suppose we have built a histogram on $A.y$ while reading in relation $A$. While reading in $B$, for each tuple with values $(x = x_1, y = y_1)$, we modify the histogram on the $A$ attribute of $B$, incrementing the count of the bucket corresponding to the value $x_1$ by 1. In addition, we modify another histogram representing the distribution of values in column $x$ of $A \bowtie_y B$ and increment the count of the bucket corresponding to $x_1$ by $N^A_{y_1}$. At the end of the build pass of relation $B$, we have two histograms representing the distribution of $x$: one on relation $A$, and the other on $A \bowtie_y B$. When the probe input to the lower join is read, we can probe both histograms using the value of the attribute $C.x$ and get an estimate of the cardinality of the output of the upper $A \bowtie_y (B \bowtie_z C)$ and lower $B \bowtie_z C$ joins as before. This example is illustrated in Figure 2 (b).

The two cases described above provide the basic intuition on how to handle a pipeline containing a chain of multiple hash joins. Algorithm 1 describes the procedure in more detail. Structure Node is the structure associated with a hash join operator. Associated with each Node is a list of histograms Node.histList which is a linked list containing pointers to histograms in the subtree below the node that are relevant for cardinality estimation at the node. The other important field is Node.joinList, which is a list of joins further up the pipeline that involve the build input to the node. Other fields in this structure are self explanatory. Algorithm 1 describes the estimation and histogram maintenance procedure as getNext() calls are made on the nodes of the join pipeline. We do not present the pseudo code for the buildHistograms() and probe() procedures due to space restrictions, but describe their functionality in the explanation that follows.

A hash join node is initially in state INIT. On the first getNext() call to the node, it initializes its histogram and labels it with the $(Relation, Attribute)$ pair which is to be used to probe it. It also checks if its build relation is involved in a join further up the join tree. This is done by the function makeJoinList() which checks the labels of the histograms at nodes in the tree above. If it finds a histogram in the joinList of a parent node with a $(Relation, Attribute)$ label that matches its build input, it adds an entry for the $(Parent, Attribute)$ pair to the joinList of the node. This list contains all the histograms
which have to be constructed as per Case 2 above. The node then moves to the BUILD state in which it reads in its build input and partitions it. In this phase, it builds a histogram on the build input, and also constructs a histogram for all entries in the joinList data structure. This functionality is provided by the call to buildHistograms(). Finally, the node moves into the PROBE state and reads in its probe input. We push down all estimation to the lowermost node in the join tree; the probe() call is made only at the lowest and node. This function contains the code for incrementally obtaining estimates for node cardinality.

The following example runs through the algorithm for a 4-way join pipeline.

**Algorithm 1** Estimation for hash join pipelines

```
proc getNext(Node node)
if node.state == INIT then
node.histList.add(node.probeRelation,node.joinAttribute)
end if
if node.state == BUILD then
while (tuple = getNext(node.buildChild))!=NULL do
buildHistograms(node,tuple)
end while
node.state = PROBE
end if
if node.state == PROBE then
probe(node,tuple)
end if
end if

proc makeJoinList(Node node,Node Parent)
for all attribute C in node.buildRelation do
if Parent.histList[node.buildRelation,attribute]!=NULL then
node.joinList.add(Parent.attribute)
end if
end for
Parent.histList.add(node.probeRelation,node.joinAttribute)
if type(Parent.parent)==hashjoin then
makeJoinList(node,Paren.Parenent)
end if
end if
```

4.1.4.3 Other Join Algorithms.

For the case of sort-merge joins related issues arise. In particular a sequence of sort-merge joins on the same attribute can be handled in exactly the same way as a pipeline of hash joins. When the first relation is sorted, we can build a histogram on the sort column. Subsequent sorts build histograms on their own value frequency while probing corresponding histogram buckets to incrementally obtain the join estimate. For the case of a sequence of sort-merge joins on different attributes, we no longer have a pipeline. This is due to the fact that there must be a blocking sort operator between the output of a lower join and an upper join if they are on different attributes. As a result, in this case, our discussion on binary sort-merge joins readily applies.

For the same reasons raised in our discussion on nested-loop joins, estimation in the case of multiple nested-loops joins reduces to the dne estimate [9].

4.2 Aggregation

In database systems, aggregation is typically implemented by sorting or hashing. In a hash based aggregation, the input is read and partitioned using a hash function and then data is grouped by reading in the partitions. In sort-based aggregation, the input is first sorted on the group-by attribute and aggregates are computed by reading in the sorted input. In both cases we can get perfect cardinality estimates of the output by counting the number of groups seen in the hashing/sorting phase. Doing this requires maintaining a data structure for storing the set of grouping values that have been seen.

Although counting the number of groups seen in the preprocessing phase gives us the cardinality of the output of the aggregation operator, we would like to be able to estimate this cardinality even before the entire input has been seen. This is the distinct value estimation problem and it has been well studied in the database and statistics literature (see [14, 5, 13] and references therein). However, previous work on distinct value estimation has been conducted without taking the runtime overhead to obtain the estimate into account. For selectivity estimation used in query optimization, this makes sense since optimization is a one time overhead and high accuracy is a prime objective. For progress estimation however, the overheads of obtaining estimates are of equal importance. During progress estimation one has the ability to examine progressively more tuples and refine the estimate. As a result, accurate but low overhead estimation techniques for distinct values are important.

In the literature, the state of the art estimator for the number of distinct values is the GEE estimator (Guaranteed Error Estimator) [5]. Let $T$ be a random stream of attribute values of size $|T|$. The GEE estimator has guaranteed accuracy as it can provide a lower and upper bound guarantee for its estimate. It operates based on the principle that data can be split into low-frequency and high-frequency groups and that any reasonably sized sample should contain all the high frequency groups. It treats the singleton values in the sample as a sample of the low frequency values and scales them up to minimize the error. If $f_i$ is the number of groups that occur exactly i times in the sample of size $t$ from $T$, the GEE estimate for the number of distinct values in a relation of size $|T|$ is

$$D_t = \sqrt{\frac{|T|}{t} f_1} + \sum_{j=2}^{t} f_j$$

The estimate follows by taking the geometric mean of the lower and upper bound estimate for the number of singleton groups ($f_1$ and $\frac{1}{|T|} f_1$ respectively). Assuming that we accumulate the frequencies with which values appear (i.e., construct a histogram of the attribute value frequencies observed in the sample of $t$ values of $T$) we can compute the GEE estimate by a single pass on the histogram. However, we observe, that it is also possible to compute
this estimate by incrementally maintaining the histogram as \( t \) increases in size. Consider what happens when a new value \( i \) is seen. If \( i \) has never been seen before, it contributes to the first term in the GEE estimate. If \( i \) has been seen before it contributes to the second term. If \( i \) had been seen only once before, its contribution to the first term must be deducted. If \( S_1 \) and \( S_n \) represent the number of groups seen once and more than once respectively, the estimator can be updated using Algorithm 2.

**Algorithm 2 Updating the GEE Estimator**

```plaintext
while getNext(T) do
    i = value of the new tuple on its grouping attribute
    if \( N_i = 0 \) then
        \( S_1 = S_1 + 1 \)
    else if \( N_i = 1 \) then
        \( S_1 = S_1 - 1 \)
        \( S_n = S_n + 1 \)
    end if
    \( N_i = N_i + 1 \)
    \( t = t + 1 \)
    \( D_t = \sqrt{\frac{t}{2}}S_1 + S_n \)
end while
```

As the original paper on GEE attests, the estimator works best for data with high skew. However, it tends to overestimate the number of groups when the sample size is small. It also doesn’t perform well on low skew data that have a large number of distinct groups. For this reason, a heuristic is proposed to obtain analytically an estimate for the number of singleton groups. There are two problems with this approach: (a) obtaining the analytical estimate involves solution of equations using numerical software that imposes a very large overhead per solution and (b) the problem still persists, because like singleton groups, groups of size two or three may constitute a large fraction of the input. In such cases the estimate will continue to be highly inaccurate.

We observe that the cases in which the GEE estimator fails are exactly the cases in which a large number of groups appear in the input having relatively low variance in their frequencies. Our goal is to obtain an improved estimate in the cases where the GEE fails, but at the same time keep the runtime overhead of obtaining such an estimate low to make it suitable for progress estimation. Assume that the random stream \( T \) consists of a number of groups \( g \). Moreover assume that the fraction of each group \( i \) in the stream is \( p_i \). After observing \( t \) values of \( T \) our expectation for the number of new (currently unseen) groups that will be identified if we read \( t \) more values is:

\[
\sum_{i=1}^{g} (1-p_i)^t (1-(1-p_i)^t) = \sum_{i=1}^{g} (1-p_i)^t - \sum_{i=1}^{g} (1-p_i)^{t+1}
\]

Using this formula as an estimator is not possible, however, since \( g \) and the \( p_i \)’s are unknown in \( T \). However, if the \( p_i \)’s have low variance, the maximum likelihood estimators (MLE) for both \( g \) and the \( p_i \)’s will be good estimators, since \( T \) is random and the \( p_i \)’s stationary. Our estimate \( \hat{g} \) for \( g \) is \( \hat{g} = \sum_{i=1}^{M} f_i \) where \( M \) is the maximum frequency count for any group. Similarly, our MLE estimate \( \hat{p} \) for a group that has been observed \( \hat{j} \) times in \( t \) values from \( T \) is \( \hat{p} \). As a result, the expectation for the number of groups to be observed if we read \( t \) more values of \( T \) would be:

\[
D_t = \hat{g} = \sum_{i=1}^{M} f_i (1 - \frac{i}{t})^t - \sum_{i=1}^{M} f_i (1 - \frac{i}{t})^{t+1}
\]

This estimate is monotonic and converges to the correct value, as \( t \) increases. Moreover, through an asymptotic analysis, it can be shown that such MLE estimators, have bounded variance [4]. This estimator rarely overestimates the number of groups, although it is prone to underestimation. Unlike the GEE estimator, as will be demonstrated in Section 5.1.4, it works best when the data has low skew.

The MLE estimator cannot be incrementally maintained like the GEE estimator and so it must be recomputed regularly. Setting a constant interval for recomputing the estimate is not a good idea since we would like to refine our estimates more often when they are changing frequently. Algorithm 3 presents our method for refining our estimates. We define a lower \( (l) \) and upper \( (u) \) threshold for the interval of recomputation \( (I) \) and start off recomputing the estimate every \( I \) tuples. If the new estimate is within \( k\% \) of the old estimate, we double our interval, provided it is less than \( u \); otherwise we set our interval back to \( l \). This method ensures that we recompute the estimate more frequently when it is needed.

**Algorithm 3 Interval setting for MLE**

```plaintext
getMLE(): function to compute the MLE estimate
\[ I = l \]
while getNext(T) do
    \( l = l + 1 \)
    if \( (t \mod I) = 0 \) then
        \( E_{old} = E \)
        \( E = \text{getMLE}() \)
        if \( 1 - k < E_{old}/E < 1 + k \) then
            \( I = I \times 2 \)
        end if
        else
            \( I = l \)
        end if
end while
```

The GEE estimator is inexpensive, but it can also cause severe overestimates for small sample sizes. It works best for data with high skew. The MLE estimator is somewhat more expensive but works best when data has low skew. It is natural to make a choice between the two based on the skew of the data and the cost of computing the estimate.

In order to get a measure of skew, we compute the squared coefficient of variation \( \gamma^2 \) of the frequencies of the groups seen. The quantity \( \gamma^2 \) is a measure of the variance of the frequencies of the groups, and a low value reflects that the data has low skew. In addition, it is inexpensive to compute since it can be incrementally maintained as new tuples are seen. We set a threshold \( \tau \), and use the MLE estimator if \( \gamma^2 < \tau \), and the GEE estimator otherwise. (See Section 5.1.4 for experiment details)

With these estimators in place, we still need to handle cases where the input to an aggregation operator cannot be treated as a randomly ordered input. Such cases are similar to the ones outlined for joins and can occur when a hash or sort-merge join is an input to an aggregation operator on the same attribute as the join. As in the case of join pipelines, it is possible to push down the estimation for the aggregation into the join. In addition to computing the estimate of the cardinality of the output of the join, we also build a histogram storing the frequency distribution of the output. The GEE or MLE estimators can now be run using this histogram as input in order to estimate the cardinality of the output of the aggregation operator.

### 4.3 Selections

---

(Details omitted due to space limitations; the main idea if to decompose the coefficient of variation formula to elements – prefix sums and prefix sums of squares – that can be maintained incrementally.)
As in the case of nested-loops joins, selection operators do not have any preprocessing phases in which the entire input is seen before the actual selection take place. Selection conditions are pushed as close to the leaves of the query plan as possible, and so it is not possible to further push down estimation. Since we assume that part of the input is seen in random order, the dne estimator has zero error in expectation [6], and so we use the dne estimator to handle these cases.

4.4 Progress Estimation

We now have a set of techniques which allow us to refine cardinality estimates of operators in a query plan. Our techniques hinge on the fact that operators for joins and aggregations often have preprocessing phases where they see the entire input and partition it either by sorting or hashing. We have shown that it is possible to push down cardinality estimation for operators into these preprocessing phases. Pushing down estimation allows us to not only get correct estimates, but also to take advantage of the random order of the tuple stream before it gets partitioned/ordered by hashing or sorting. We revert to the dne estimate for operators that do not have any preprocessing phase, such as certain forms of nested-loop joins and selections.

A query Q consists of multiple pipelines. If we label the pipelines as (p1 . . . pn), the gnm estimate of progress is

\[ gnm = \frac{C(Q)}{T(Q)} = \sum_{i \in P1} \frac{K_i}{N_i} + \sum_{i \in Pn} \frac{K_i}{N_i} \]

For a pipeline p, let C(p) be the number of getnext() calls made over all operators in it i.e. C(p) = \sum_{i \in P} K_i. Similarly let T(p) be our estimate of the number of getnext() calls that will be made over the run of the pipeline i.e. T(p) = \sum_{i \in P} N_i. Of these pipelines, some would have already finished executing. For these, we know C(p) and T(p) precisely. For pipelines, that are currently executing, we know C(p) precisely, while we estimate T(p) using our estimation techniques. For pipelines that are yet to begin, C(p) = 0, and we refine the optimizer estimates for T(p) using upper and lower bounds as in [9]. Summing these values over all pipelines gives us C(Q) and T(Q), and we estimate the progress of the query as C(Q)/T(Q).

Since our techniques require only a random sample of the input, for each pipeline, we keep obtaining estimates until the random sample is read. We use a simple inter-operator communication mechanism, similar to the punctuations used in [2] in order to notify the operator when the random sample is over. After this point, we have an approximately correct estimate (T(p) for pipeline p), which may be adjusted as the operators execute.

5. EVALUATION

In this section, we evaluate the accuracy of the estimations introduced in the previous sections and also quantify the overheads of our estimation framework.

Implementation: We implemented our estimation framework in the PostgreSQL 8.0 database engine. As a first step, we implemented the gnm model along with the dne estimation framework [9]. This was done by modifying the central control function for query execution in PostgreSQL, which acts like a wrapper for all operators. We also changed the data structures for operators and added counters for the number of tuples emitted by the operator, the estimated cardinality of the operator, and upper and lower bounds for the cardinality.

We then implemented our proposed estimators within each operator. The modifications consisted of adding calls to functions for histogram construction and cardinality estimation using the techniques mentioned in the previous sections. We also modified the table scan operators to first read in a precomputed block-level random sample of the base tables before scanning the rest of the table. A simple antijoin operation on block-ids of tuples is done while scanning the base table to exclude the tuples that were already read in the sample.

Experiment Design: We performed all experiments on a machine running Fedora Core 3 with a 2.80 GHz processor and 1 GB RAM. We utilized the TPC-H schema as our template and used the publicly available tool [8] to generate skewed data. We modified this tool in order to be able to vary the number of distinct values in a table column. In what follows, we first describe the experiments for evaluating the accuracy of our estimation methods, and then move on to measuring the overheads.

5.1 Evaluating Accuracy

We define the ratio error (R) of an estimator as the ratio of the estimated cardinality of the output of the operator to the final cardinality of the output. Our presentation of the accuracy of our estimators is in terms of the ratio error. The sooner the estimator reaches a ratio error of 1, the better it is. This definition of ratio error generalizes in a natural way to complete query plans. For a query, if T′(Q) is the current estimate of the number of getnext() calls over the query, and T(Q) is the actual number of getnext() calls that will be made, then as per gnm:

\[ \frac{Actual\ Progress}{Estimated\ Progress} = \frac{C(Q)}{T(Q)} \cdot \frac{T(Q)}{T'(Q)} = R \]

5.1.1 Methodology

Our experiments on the accuracy of our estimators were all conducted on tables complying with the customer and nation schemas of the TPC-H specification and for ease of presentation we restricted them to just the nationkey attribute of these tables. Nationkey is a primary key of the nation table, and a foreign key to the customer table; it assumes values from [1 . . . 25] in the TPC-H specification. In order to have flexibility to test our estimators on a variety of data sets and domain sizes, we modified the generating function for the nationkey column. For the accuracy experiments we present, we keep the number of rows of the table equal to that resulting by running TPC-H data generation with a scaling factor of 1 (150K rows).

In the discussion that follows, \( C_{[i..n]} \) denotes a customer table having a Zipfian distribution with skew z (Z = z) on the nationkey column, taking values from the set \([1 . . . n]\). Thus, for example \( C_{[2,125K]} \) would denote a customer table with Zipfian skew of 2 on the nationkey column which takes values between 1 and 125000. To simulate join operations between columns in which key value frequency exhibits different correlation patterns, we generate skewed distributions in which the peak value frequency corresponds to different values. We use superscripts to distinguish between tables that have the same skew, but different distributions of frequencies across the domain of values. So \( C_{2,125K,125K} \) all have Zipfian skew of 2 on the nationkey column, but the values with a high frequency in one table may have a low frequency in another table. In a join between two columns, if the high frequency values in one column are also the high frequency values in the other column, then estimating the size of the join is easy since most of the output of the join is due to the join of these high frequency values. In most of the experiments, we try to capture distributions that have different high frequency values. This is the
estimate when only a small percentage of the probe input has been
joined. Figure 4 (a) corresponds to a join between the
ures, our estimator has already converged to the correct cardinality
on the percentage of the probe input that has been
most of the probe input has been joined. In both fig-
and present three curves one for each value
of the probe input as described in Section 4.1.1. Similarly, Figure 4 (b) corresponds to a primary-key foreign-key join between a
customer table \( C_{1,125K} \) and its corresponding nation table with a selection condition \( \text{nationkey} < 50000 \). Even for this scenario, the
byte and dne estimates are highly inaccurate and remain inaccurate until most of the probe input has been joined.

The purpose of Figure 4 is to stress on the importance of reordering caused by hash partitioning in a hash join. Similar effects are
cased by sorting before a sort-merge join. Despite the fact that
tables are retrieved in random order from disk, the \( \text{byte} \) and \( \text{dne} \) estimates do not converge smoothly and quickly to correct cardinality estimates because they do estimation after the data has been hashed or sorted.

5.1.2 Joins

To evaluate the accuracy of our estimators for joins, we generated customer tables with different domain sizes and varying skew
on the nationkey attribute. In each case, we evaluate joins between
two tables with the same domain size and skew, but different distributions i.e a high frequency value in one table may be a low frequency value in the other.

We present results for the ratio error on small domain sizes (5K elements) in Figure 3(a), and large domain sizes (125K elements) in Figure 3(b). In these graphs, the value of \( Z \) corresponds to the skew parameter of the Zipfian distribution. In each case two Zipfian distributions with the same skew and different peak frequency values are joined. In all cases, as is evident in the figures, our estimators converge to an approximately correct ratio error estimate while having seen only a fraction of the probe input. It is evident that only a small percentage sample of the input is required in each case, to accurately obtain estimates and trace progress precisely.

Figure 4 shows a comparison of our estimator (labeled as once for online cardinality estimation) with the dne estimator and the estimator of Luo et al. [18] (labeled as \( \text{byte} \) since their framework is based on counting bytes). Since the other two estimators see data as it is output by the join, the graphs are parameterized with respect to the percentage of the probe input that has been joined. In both figures, our estimator has already converged to the correct cardinality estimate when only a small percentage of the probe input has been joined. Figure 4 (a) corresponds to a join between the customer ta-
bles \( C_{1,125K} \) and \( C_{1,125K}^1 \) on the nationkey attribute. In this case, the PostgreSQL cardinality estimates are off by about a factor of 13. The \( \text{byte} \) estimator imposes a weighted average operation involving the original cardinality estimate, and so it converges slowly to the correct answer. The \( \text{dne} \) estimator, disregards the original optimizer estimate as soon as the pipeline starts executing. However, it underestimates the number of tuples to be produced by the operator because of the effect of reordering caused by the partitioning of the probe input as described in Section 4.1.1. Similarly, Figure 4 (b) corresponds to a primary-key foreign-key join between a
customer table \( C_{1,125K} \) and its corresponding nation table with a selection condition \( \text{nationkey} < 50000 \). Even for this scenario, the
byte and dne estimates are highly inaccurate and remain inaccurate until most of the probe input has been joined.

The purpose of Figure 4 is to stress on the importance of reordering caused by hash partitioning in a hash join. Similar effects are
cased by sorting before a sort-merge join. Despite the fact that
tables are retrieved in random order from disk, the \( \text{byte} \) and \( \text{dne} \) estimates do not converge smoothly and quickly to correct cardinality estimates because they do estimation after the data has been hashed or sorted.

5.1.3 Join pipelines

We now present accuracy results for pipelines of joins. We present both cases namely the case where the join column of a lower join is also the join column of the upper join and the case where it is not as discussed in Section 4.1.4.

Figure 5 shows the behaviour of our estimator for the case where the join column is the same. In this experiment, we use \( C_{1,5K} \), \( C_{1,5K}^1 \) and \( C_{1,5K}^2 \) and present three curves one for each value of \( z \) (\( z = 0, 1, 2 \)). In each experiment the three relations joined are generated using the same \( z \) value. Figure 5 (b) shows the estimate of the cardinality of the lower join as its probe input is seen while Figure 5 (a) shows the estimate of the cardinality of the upper join, also with respect to the percentage of the lower join seen. Notice that for the \( Z = 2 \) case, the estimate of the cardinality of the upper join is inaccurate in between before converging to the correct value. This point corresponds to a value of the lower join, joining with a
high frequency value of the upper join. The reason why it is so prominent is that in this case, only a few values actually contribute to the join.

Figure 6 shows the behaviour of our estimator in a two join pipeline where the join columns of the upper and lower joins are different. The join attribute of the probe input of the upper join can come from either the probe input of the lower join or the build input of the lower join (Case 1 and Case 2 respectively in Section 4.1.4.2). Here too we want to recreate the worst case scenario for joins as described in Section 5.1.1, and this time for two different join columns. To do so, we replace the primary key column custkey for the customer relation with a skewed distribution on a domain with 25K elements. We also set the domain of the nationkey column to 25K elements. The lower join is kept fixed between nationkey columns with the same skew, and the upper join is between custkey columns with varying skew. All join conditions are on columns which have the same skew, but different distribution of frequencies across the domain of values. We describe a subset of our experiments here.

Figure 6 (a) shows the behaviour of our estimator for Case 1. Here we keep the skew of the join columns for the lower join fixed as 2. In the join of the upper relation, the join is with the custkey column of the probe relation of the lower join. We vary the skew of the columns for the upper join. The reason why there is no graph for $z = 2$ for the upper join is that the join produced no tuples. Similar plots are observed for other values of $z$ for the lower join, and for varying domain sizes.

Figure 6 (b) shows the behaviour of our estimator for Case 2, i.e., when the upper join involves a join between relations that are build inputs in the pipeline. In this case, the skew of the lower join is kept fixed between columns with $z = 1$ and we vary the skew of the columns for the upper join. In both figures we can clearly see the fast convergence of our estimators as the probe input of the lower join is read. Similar behaviour is observed when other $z$ values are used in the lower joins as well.

Note that in these cases, the byte and dne estimators would not have seen many tuples at the upper join by the time the probe input at the lower join has been read, since only a fraction of it would have been joined. Therefore they give inaccurate estimates.

### 5.1.4 Aggregation

We also evaluate the accuracy of the two estimators for aggregates. To choose which estimator to use we observe the value of the squared coefficient of variation ($\gamma^2$) of the group-by column. Table 1 contains an evaluation of the GEE and MLE estimates for tables with varying number of distinct values and skew on the TPC-H customer table with scale factor 1. The # Values column contains the maximum number of distinct values possible in that table column. The actual number of values may be much less since several values get discarded when generating highly skewed data. The $\gamma^2$ column shows the value of $\gamma^2$ when 10% of the input data has been seen. We choose 10% since our final choice of the estimator is made by this point. The GEE and MLE columns show the number of input rows seen by the estimators before they reach within 10% of the correct answer. The “All Seen” column shows the number of input tuples read before all grouping values have been seen. The estimator that reaches within 10% of the correct answer by observing the smallest number of tuples is better. As the table shows, GEE usually does better than MLE on high skew data and when there exists a large number of low frequency values in the input. In other cases, MLE is a better estimator. The results validate our reasoning behind the choice of two estimators.

The table clearly shows a wide gap between $\gamma^2$ values for low skew and high skew data, and we can observe a correlation between the value of $\gamma^2$ and which estimator does better. In practice, we set a limit of 10 on $\gamma^2$, and use this as our threshold $\tau$; selecting MLE when $\gamma^2 < 10$ and GEE otherwise. We use this simple thresholding cutoff for (runtime) performance reasons and it performs well in practice. Using more sophisticated statistical tests is possible (e.g., chi-square) but that will impose additional overheads to the runtime performance of this decision.

<table>
<thead>
<tr>
<th># Values</th>
<th>Mem. Used</th>
<th>Mem. Alloc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>21.8 KB</td>
<td>24.6 KB</td>
</tr>
<tr>
<td>10000</td>
<td>205.1 KB</td>
<td>253.9 KB</td>
</tr>
<tr>
<td>100000</td>
<td>2.03 MB</td>
<td>2.09 MB</td>
</tr>
<tr>
<td>1000000</td>
<td>20.3 MB</td>
<td>25.2 MB</td>
</tr>
</tbody>
</table>

Table 2: Memory Overheads of Histograms

We now quantify the overheads imposed by our framework on the query engine. Our estimators introduce overheads in the form of extra memory consumption for maintaining histograms, and the cost of executing the extra code for probing the histograms and refining cardinality estimates. Our discussion here is in terms of the lineitem and orders tables of the TPC-H schema, since these are the largest tables as per the specification. Notice that overheads are a function of table sizes and not the table distribution. As a result, we only vary the scaling factor of the TPC-H database generated. A TPC-H database with a scaling factor of 1 has a 6 million row lineitem table and a 1.5 million row order table. A database with scaling factor 2 doubles these parameters, and one with scaling factor 0.5 is approximately half the size.

We first quantify the memory overheads, which are primarily in the form of histograms. We then present results detailing the runtime performance overheads to queries due to the estimators added. We used precomputed random samples in the experiments we present. We also did experiments with samples computed on the fly, with the results being consistent modulo the overhead of computing the sample.

#### 5.2 Runtime Estimation Overheads

We now quantify the overheads imposed by our framework on the query engine. Our estimators introduce overheads in the form of extra memory consumption for maintaining histograms, and the cost of executing the extra code for probing the histograms and refining cardinality estimates. Our discussion here is in terms of the lineitem and orders tables of the TPC-H schema, since these are the largest tables as per the specification. Notice that overheads are a function of table sizes and not the table distribution. As a result, we only vary the scaling factor of the TPC-H database generated. A TPC-H database with a scaling factor of 1 has a 6 million row lineitem table and a 1.5 million row order table. A database with scaling factor 2 doubles these parameters, and one with scaling factor 0.5 is approximately half the size.

We first quantify the memory overheads, which are primarily in the form of histograms. We then present results detailing the runtime performance overheads to queries due to the estimators added. We used precomputed random samples in the experiments we present. We also did experiments with samples computed on the fly, with the results being consistent modulo the overhead of computing the sample.

#### 5.2.1 Histograms

We implement histograms using the generic hash table structure
of PostgreSQL. As Table 2 shows, this corresponds to a space overhead of approximately 20 bytes per entry in the histogram. However, we store just 8 bytes per entry in the histogram; 4 bytes for the value, and 4 for its count. The extra overhead is due to the pointers maintained by PostgreSQL. A simpler hash table would reduce memory costs significantly. In this work, we do not investigate the use of approximate (i.e. bucketized) histograms instead of accurate ones. Investigating the accuracy-overhead tradeoff with regard to using such estimators is deferred for future work. However, we note that all the estimators described here can work with such approximate summary structures as well.

### 5.2.2 Joins

In order to evaluate the overheads of our estimators on simple binary joins, we generated `lineitem` and `order` relations from the TPC-H database with different scaling factors. We measured the time to join corresponding `lineitem` and `order` tables on the `orderkey` attribute. This is a primary-key foreign-key join. Thus, for example, a join between these tables with scaling factor 1, should produce a histogram of 1.5 million entries, consuming approximately 30 MB of memory. Table 3 shows the performance overhead experienced in both binary hash joins and sort-merge joins due to our estimators with varying sample sizes. These numbers are averages of multiple runs of the queries. As can be seen, the performance overhead of the framework is small and a small fraction of the overall query response time. This is primarily due to the fact that estimation takes place in the preprocessing phases and not during actual join processing. These phases involve heavy I/O and the overheads due to using slightly more memory are reasonable.

Table 4(a) describes the performance overhead due to our estimation framework on join pipelines involving joins between copies of `order` relations on the `orderkey` column. We duplicate the `orderkey` column in the tables in order to ensure that this corresponds to the case of joins on different attributes. The table shows overheads when the join column for the upper join involves the probe input of the lower join (Case 1) and when it involves the build input (Case 2). The sample size was set at 10%, and the overheads are slightly lower for smaller sample sizes. In both cases, our estimation framework does not introduce any major overheads and query times are increased imperceptibly.

### 5.2.3 Aggregations

Table 4(b) describes overheads for a group-by query on the `custkey` attribute of the `orders` table for varying scaling factors. The queries clearly show that neither the GEE nor the MLE estimators slow down aggregations appreciably. For the MLE estimator, we set the interval of recomputation as in Algorithm 3 by setting the lower bound $(t)$ to 0.1% of the input size, and the upper bound $(u)$ at 3.2% of the input; we double the recomputation interval if the difference between the old and new estimates is less than 1%. These values are chosen in order to provide fine granularity for the MLE estimator without having a high overhead and have been empirically validated for different data sets. This experiment was also conducted using 10% samples of the relations. From Table 4(b) it is evident that the maintenance procedures of the aggregate estimators proposed herein impose small performance overheads.

### 5.3 Progress Estimation

We now present the behaviour of a progress estimator which uses the estimation techniques we have described in this paper. Figure 8 shows the behaviour of our estimator and the `dne` estimator for TPC-H query 8 on a database populated with Zipfian skew 2 data, and scaling factor 1. This query involves a join of 8 tables which generates a busy plan tree followed by an aggregation. The main processing is done in a pipeline of 3 hash joins, the sizes of which are underestimated by the optimizer. We used 10% random samples in this experiment. When the pipeline begins, our estimation
framework pushes down estimation to get accurate cardinality estimates for all the joins in the pipeline. Due to this quick adjustment, it gives correct progress estimates during the rest of the query. The dne estimator does not adjust the cardinality estimate for the joins higher up in the pipeline until much later, and so it overestimates the progress for a long time. The behaviour of the byte estimator is similar and hence not shown.

### 6. CONCLUSIONS

In this paper, we have proposed an online framework for progress indication. We have proposed estimators to track and progressively refine the cardinality estimates for various pipelines in a way that it is easily integrated to models of progress indication. We have evaluated both the accuracy and performance overheads of our framework through experiments conducted in an operational data management system.

This work raises various avenues for future work in this area. In particular, we envision work in the following directions. First, it is possible to conduct further performance tuning and reduce the run time overheads even further by deploying approximations of the histograms we construct. Thus the classic accuracy performance trade-off can be explored via approximation. Although we make the independence assumption for attributes of the same relation, to simplify our framework, this is not necessary. An estimation framework can be constructed taking existing correlations into account (or even discovering them on the fly as more data is read in). This involves deployment of alternate statistical models, in order to obtain accuracy guarantees.

### 7. REFERENCES


