When Can We Trust Progress Estimators for SQL Queries?

Surajit Chaudhuri Raghav Kaushik Ravishankar Ramamurthy
Microsoft Research
{surajitc, skaushi, ravirama}@microsoft.com

ABSTRACT

The problem of estimating progress for long-running queries has recently been introduced. We analyze the characteristics of the progress estimation problem, from the perspective of providing robust, worst-case guarantees. Our first result is that in the worst case, no progress estimation algorithm can yield anything even moderately better than the trivial guarantee that identifies the progress as lying between 0% and 100%. In such cases, we introduce an estimator that can optimally bound the error. By placing different types of restrictions on the data and query characteristics, we show that it is possible to design effective progress estimators with small error bounds. We show where previous solutions lie in this spectrum. We then demonstrate empirically that these "good" scenarios are common in practice and discuss possible ways of combining the estimators.

1. INTRODUCTION

For long-running decision support queries, the ability to estimate the progress of query execution could be very useful, for instance, to help end users or applications decide whether to terminate the query or allow it to complete. The problem of estimating the execution progress of long-running queries has been recently studied [5, 13]. The authors observed that naive approaches to progress estimation, such as returning the fraction of operators that are completed, are inadequate for complex query plans. Instead, a novel model for the work done by a query ("progress") is proposed and estimation techniques are developed under this model. Their proposed estimators fared well in the empirical studies that they reported. Recent work in [14] increases the coverage of the previous techniques by including a wider class of SQL queries.

Database technology prides itself for its robustness and therefore a natural question that arises is whether the proposed estimators are indeed resilient for varying execution plans and data distributions. Ideally, a progress estimator should be able to provide guarantees to the user, irrespective of the nature of the query and data distributions under consideration. Unfortunately, prior work does not dwell on this key question and does not offer any explanation as to when and why the proposed estimators will be reliable.

We start with the model of work proposed in [5] which uses a simple aggregate measure based on the iterator model of query execution (Section 2). The model of work proposed in [13] is very similar and the results in this paper would extend to this model as well. Progress of query execution is defined as fraction of "work" done. We study the progress estimation problem with the goal of bounding the ratio error of estimating this fraction.

We begin by showing that even for the simple case of a single join query, in the worst case, it is not possible to do much better than a trivial progress estimator that returns the interval $(0, 1)$ at every instant (Section 3). This result applies to any typical database system that uses statistics on base relations, allowing a broad class of statistics that covers both histograms and pre-computed samples. Since an effective strategy for this case is a prerequisite to realizing the general goal of providing guarantees on the estimated progress for complex SQL queries, this result shows that providing robust guarantees for the problem of progress estimation is impossible within the framework we consider.

Our worst case result shatters the hope that the proposed estimators can be used universally and is in apparent contrast with the empirical results demonstrated in prior work. However, we are able to characterize when and why we can expect the previously proposed solutions to yield low errors.

We next ask the question whether there are other estimators that can serve settings where prior estimators can have high errors. We show that for the class of queries where the average work done per "input" tuple (tuples at the leaf of a query plan) is small, we can build a progress estimator (pmax), which (1) is guaranteed to be an upper bound on the progress, and (2) yields a low bound on the ratio error. We empirically demonstrate through measurements on benchmark and real data sets, that such cases are fairly common in practice. We also propose an estimator (safe) that is guaranteed to be worst-case optimal (i.e., no algorithm is guaranteed to be more accurate in this setting). We show through experiments that both the above estimators can substantially improve the effectiveness of progress estimation where the previously proposed techniques fall short.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA Copyright 2005 ACM 1-59593-060-4/05/06 $5.00"

Given this tool-kit of progress estimators, we then ask if it is possible to combine them effectively so that we can cover the settings addressed by each of them. We show that, unfortunately, it is not possible to detect which setting we are in and use the "right" estimator. Hence, it is not possible to combine the above estimators to provably improve the error bounds. In Section 6, we reflect on the question of how we can effectively use these estimators in practice given the empirical study that indicates that the "good" scenarios are likely to be common.

An interesting observation is that the effectiveness of progress estimation can depend on the set of *physical* operators under consideration. By restricting the class of physical operators to exclude those that perform nested iteration, the effectiveness of all the above estimators provably improves in this setting. We also show this result empirically.

Finally, we explore the connection between the problem of progress estimation and related problems in query optimization, including cardinality estimation, join sampling, and query reoptimization. We discuss why it is possible to provide reasonably robust solutions for progress estimation without requiring solutions for cardinality estimation, and the potential implications of this on query optimization.

2. PRELIMINARIES

In this section we present our basic framework, in particular we formalize the notion of a progress estimator and concretely define information such estimators may be allowed to use including optimizer statistics and observing execution feedback. The high level relationship between the different components is illustrated in Figure 1. We also define different metrics in order to evaluate the "accuracy" of a progress estimator.

2.1 Queries

As indicated in Figure 1, one of the inputs to a progress estimator is a query plan which is a tree of physical operators. The set of operators we consider include scan, index-seek, $\sigma, \pi, \bowtie^{\text{NL}}, \bowtie^{\text{INL}}, \bowtie^{\text{hash}}, \bowtie^{\text{merge}}$, sort, γ(group by).

2.2 GetNext Model

We use the model of work proposed in [5] which is based on the number of *getnext* calls invoked in a query tree. The notion of work used in [13] (based on the number of bytes processed in the query tree) is very similar and the results in this paper would be equally applicable to the other model. For ease of exposition, we present the results for only the *getnext* model in this paper.

The execution of a query Q is modeled as a sequence of *getnext* calls across all operators, $Seq(Q) = (g_1, g_2, \ldots, g_q)$, corresponding to the standard iterator model of query execution of Q in a typical database system. Assume that $total(Q)$ returns the total number of *getnext* calls issued by query Q.

For any prefix s of $Seq(Q)$, we define $progress(s) = |s|/total(Q)$, where we use $|s|$ to denote the cardinality of s. In addition to the *getnext* calls, we assume that the progress estimator can observe any intermediate tuple values generated during partial query execution.

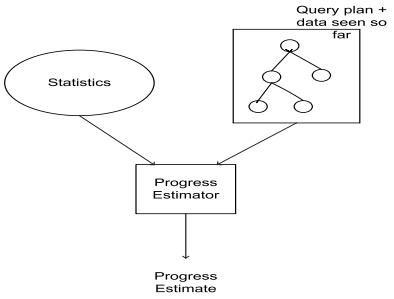

Figure 1: Progress Estimator

2.3 Database Statistics

A progress estimator can also use any available database statistics in order to aid its estimation. A *single-relation statistics generator* $\mathcal{SG}$ takes an instance of relation R as input and produces a single-relation statistic $\mathcal{SG}(R)$. Single-column histograms are examples of a single-relation statistic. The database statistics are obtained by using the single-relation statistics generator over each relation in the database separately to produce a union of single-relation statistics. We note that most commercial database systems maintain statistics only on single tables without capturing inter-table correlations. In this paper, we only focus on single-relation statistics, noting that our results also carry over in the presence of a bounded number of statistics over views [1, 2].

Since the set of statistics is typically a small fraction of the original relations, there is an inherent "loss" in information, in other words, one can always modify a relation R such that $\mathcal{SG}(R)$ remains the same. This notion is formalized below. A single-relation statistics generator is defined to be *lossy* if for any sufficiently large relation size N and any tuple position i, there are instances R and R', each with N tuples, such that R' can be obtained from R by changing the i^{th} tuple $t \in R$ with values not currently present in the relation, and both R and R' produce the same statistic. Observe that (single or multi-dimensional) histograms are lossy, since it is possible to change the individual values in some bucket without affecting the overall histogram. In this paper, we only consider single-relation statistics that are lossy.

We allow the statistics generator to be deterministic (covering single-table histograms) or randomized (covering precomputed samples). For ease of exposition, however, we present results only for the case when the statistics generator is deterministic. We note that all our results also apply to randomized statistics generators (with a high probability requirement).

2.4 Progress Estimators

We now formalize the notion of a progress estimator. Fix any query Q and let s be a prefix of $Seq(Q)$ (representing the partial execution of Q). A progress estimator is a function that takes as input Q, s along with the data returned so far as a consequence of s, and the database statistics, to return

an estimate of $progress(s)$.

Note that we do not restrict the time taken by the progress estimator. However, the only part of the data the estimator is allowed to access is the part returned by the sequence of *getnext* calls so far. In particular, the estimator is not allowed to run the query on the data instance (which would make the problem trivial). Hence, if the instance changes without changing the statistics and the sequence of *getnext* calls seen so far, the progress estimator is constrained to return the same value.

2.5 Guarantees

In order for a progress estimator to be fully "accurate", it needs to obtain the correct value for $total(Q)$ before the query starts executing. Observe that it is possible to compute this value if a solution to the cardinality estimation problem exists. Thus the progress estimation problem (under our model of work) is trivially solved if a solution to the cardinality estimation problem exists. In general, cardinality estimates are known to be erroneous [11]. To focus our attention on such cases, we consider two weaker forms of guarantees.

The first is the *ratio-error* requirement: for $e \geq 1$, a progress estimator is said to yield a ratio error of e if the estimate is always within a factor of e of the real progress. The goal of a progress estimator is to yield as small a ratio error as possible. Of course, if cardinality estimates have a low error, this requirement can be met. This could happen in cases when accurate statistics are available on the data, and propagation errors are low, for example, when the joins are key lookups.

A weaker form of guarantee relaxes the notion of a ratio error and looks at providing an interval which bounds the actual progress. Of course, a trivial guarantee that can be provided by any progress estimator is that the progress is in the interval $(0, 1)$. The bare minimum requirement for any reasonable progress estimator is to better the trivial guarantee. One way to achieve this is by requiring the progress estimator to return whether the progress is above or below a certain threshold, say τ. For instance, an interface that tells the user if the query progress is greater or less than 50% could certainly be useful. In addition, we allow a little leeway, captured through a "grey area" indicator δ. A progress estimator is said to satisfy the *threshold* requirement with threshold τ and error δ if the following holds: at any instant, (1) if the progress is less than $\tau - \delta$, then it returns an estimate in the interval $(0, \tau)$, (2) if the progress is more than $\tau + \delta$, it returns an estimate in the interval $(\tau, 1)$. If the progress is between $\tau - \delta$ and $\tau + \delta$, the estimate is allowed to be above or below the threshold. To illustrate, let us consider for example $\tau = 0.5$, and $\delta = 0.05$. What the threshold requirement says is that we must be able to identify whether the progress is below or above 50%, give or take the grey area of 5% on either side.

Notice that the threshold requirement is weaker than the ratio error requirement. A progress estimator that satisfies the ratio error requirement satisfies the threshold requirement with arbitrary τ and $\delta = \tau.\max(1 - 1/e, e - 1)$.

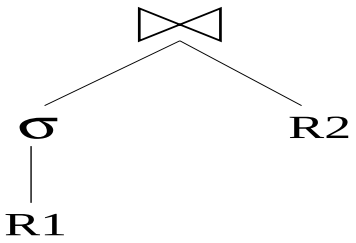

Figure 2: Single Join Query

3. DIFFICULTY OF PROGRESS ESTIMATION

In this section, we analyze what worst-case guarantees are achievable for the progress estimation problem. We consider the simplest scenario of a single join with selections. An effective strategy for this case is a prerequisite to realizing the general goal of providing guarantees for complex SQL queries. Unfortunately, it turns out that in this basic setting, satisfying even the threshold requirement (Section 2.6) is impossible. Consider the restricted class of *linear* joins, where the size of the output is bounded by the larger of the two inputs. Notice that this subsumes the common class of key-foreign key joins. We illustrate the difficulty of progress estimation through the following example.

EXAMPLE 1.: *Fix a progress estimator P. Consider the query shown in Figure 2, where the selection σ is $R_1.A = x$ or y and the join condition is $R_1.A = R_2.B$. Assume that the join algorithm used is index nested loops join and the appropriate index on relation R2 exists. In addition assume that $|R_2| = 10|R_1|$.*

Since the database statistics are lossy, there is an instance of R_1 such that some tuple t appearing after 90% of the tuples in the relation has value x in column A and changing this value to y does not affect the statistics. Suppose we fix R_2 so that it has a large number of y values (say $9|R_1| + 9$) in its (single) column B. Notice that we can do this without changing the statistics for R_1 since we only consider single-relation statistics. The amount of work done for the query could significantly vary depending on which of the two values is actually present in relation R_1. Depending on whether $t.A$ is x or y, the total number of getnext *calls performed varies from $|R_1| + 1$ to $10|R_1| + 10$.*

Consider the behavior of P at the instant before t is retrieved from R_1. P has to base its estimate on whether $t.A$ is y or not, if it hopes to be accurate. But the only information P has is the execution trace seen so far, and the database statistics. The execution trace seen so far does not reveal anything about whether the next tuple to be retrieved has the value x or y. Owing to our choice of x and y, the database statistics also do not reveal whether $t.A$ is going to be x or y. At this point P has to guess one way or the other and there will always be a case where it's estimate is inaccurate. Thus, if P biases its decision on the assumption that t could have the value x, it must suffer from a large error should the value turn out to be y, and vice versa.

For the above example, before t is retrieved, if the estimate

returned by P is less than 0.1, the error is high when $t.A$ is x (in which case the progress is around 0.9). On the other hand, if the estimate is more than 0.9, then the error is high when $t.A$ is y (where the progress is less than 0.1).

Hence, P cannot address the ratio error requirement with bound 9, nor even the threshold requirement with $\tau = 0.5$ and δ as large as 0.4.

The above argument is formalized below.

THEOREM 1.: *For any $0 < \tau < 1$ and $0 < \delta < 1$, such that $0 < \tau - \delta$ and $\tau + \delta < 1$, no progress estimator can meet the threshold error requirement with threshold τ and error δ.*

Proof: We give the proof for the deterministic case. The proof for the randomized progress estimatoris analogous. Fix a single-relation-statsprogress estimator$\mathcal{P}$ as required in the statement above. Let R_{11} and R_{12} be two instances of relation schema R_1 where R_{12} is obtained by changing exactly one tuple t in R_{11}, such that (a) their single-relation synopsis is the same, (b) t appears after a fraction f_2 of the rows in R_{11}. Let t be changed in column A from value v to v', where v' is not present elsewhere in the relation. Let the first column in R_2 be B. Fill R_2 with $(\frac{f_2}{f_1} - 1)|R_{11}|$ rows where each row has the value v' in column B.

Now, consider the query $\mathsf{scan}(R_1) \bowtie^{\text{INL}}_{R_1.A=R_2.B} R_2$. Consider the output of $\mathcal{P}$ when the first $f_2|R_{11}|$ tuples in R_{11} have been read. It must return a value $\geq f_2$, since R_{11} does not join with any tuple in R_2. However, when the first $f_2|R_{12}|$ tuples in R_{12} have been read, it must return a value $\leq f_1$, since t joins with the whole of R_2. But the first $f_2|R_{12}|$ tuples in R_{12} are the same as the first $f_2|R_{11}|$ tuples in R_{11}, a contradiction, since the single-relation synopsis of both R_{11} and R_{12} is the same. □

COROLLARY 2.: *For any $e \geq 1$, no progress estimator can meet the ratio error requirement with error bound e.*

Several observations are in order. First, while the above example is presented for linear joins, the same result would hold when $R_1.A$ is a key column and the index nested loops join looks up $R_2.B$ to find matching foreign keys.

Second, this "adversarial" scenario is not only of theoretical interest, it could occur in practice, for example, in a star schema when rows selected from a small dimension table after a selection are used to lookup the much larger fact table. If the foreign key column in the fact table has a zipfian distribution (known to be common in real data sets [16]), we have the kind of situation described in the above example — it is hard to decide if the key matching the high frequency values in R_2 will fail the selection or not.

Third, note that the argument in the example is carried out for $\delta = 0.4$ which is a high value for the "grey area". In fact, for this value of δ coupled with $\tau = 0.5$, the threshold requirement boils down to the following: at any given instant, if the progress is less than 0.1, return $(0, 0.9)$ and if the progress is more than 0.9, return $(0.1, 1)$. This is only slightly better than the trivial guarantee of returning the interval $(0, 1)$. What our result shows is that given our framework, in the worst case, it is impossible to achieve anything even slightly better than the trivial guarantee of returning the interval $(0, 1)$.

Fourth, we can make several observations about the data and query in the above argument: (1) the mean number of tuples with which a tuple in R_1 joins can be really high if $t.A$ has y, (2) there is an "offending" tuple, i.e., a tuple with a high join skew, and the progress estimator has no way to detect (either through database statistics or execution feedback) if the "offending" tuple will participate in this query, and (3) the join is an index nested loops join. The question that arises is whether any of these properties of the above argument can be relaxed to make progress estimation effective. We address this question in the following sections. We begin in the next section with the previously proposed technique [5, 13] where empirical results demonstrate cases that are amenable to effective solutions.

4. CHARACTERIZING PRIOR TECHNIQUE

As discussed above, empirical results in [5, 13] suggest that several common cases in practice are amenable to reasonable solutions. In this section, we characterize when the previously proposed technique yields low errors. The estimators proposed in [5, 13] are quite similar, and measure the progress of a query by focusing on certain nodes in the query plan, referred to as driver/dominant nodes. We begin by reviewing this technique (referred to henceforth as the dne estimator), and move on to our analysis, relating it to the lower bound argument in Section 3.

4.1 Review of dne

The dne estimator is based on the observation that an execution tree can be decomposed into a set of *pipelines* that execute in a partial order. The solution primarily focuses on estimating the progress within a single pipeline, which is then extended to complex queries.

In a serial execution of the query tree, a pipeline is a set of concurrently executing operators (we refer the reader to [5, 13] for details). Common examples of pipelines are a table scan followed by a sequence of filters, and a table scan followed by index nested loops joins or a simple nested subquery. The example shown in Figure 2 is an example of a single pipeline query.

The approach taken in [5, 13] is to identify the operator node that acts as input to the pipeline (referred to as a segment in [13]) [1]. The pipeline is intuitively "driven" by the tuples returned by this input node. As a result, this input node is referred to as the *driver* node (the "dominant" node in [13]). The dne estimator is based on using the fraction of the tuples read at the input node as an estimate of the progress.

[1]In general, multiple nodes could be inputs to the pipeline. We do not address that case in this paper

DEFINITION 1.: *Consider any instant during the execution of a single pipeline. The* dne *estimator returns the fraction of the tuples read at the input node.*

Thus, for example, in Figure 2, the input node is the table scan of the outer R_1. At any instant, dne estimates the progress to be the fraction of R_1 scanned.

4.2 Effect of Input Order

We now proceed to analyze the behavior of this approach. Since the core of the dne estimator addresses the case of a single pipeline, we restrict ourselves to single pipeline queries in this section.

We illustrate the intuition behind the dne estimator through the single pipeline query shown in Figure 2. As mentioned above, the input node in this case is the table scan of R_1, and the estimate of progress at any instant is the fraction of R_1 scanned. For each tuple in R_1, the number of *getnext* calls performed is 1 in order to scan the tuple, plus 1 plus the number of tuples with which it joins in case it passes through the selection. Let μ be the number of *getnext* calls performed on the average per tuple in R_1. Consider an instant during the query execution. Let μ_{curr} be the average number of *getnext* calls performed on the average per tuple of R_1 *seen so far*. Intuitively, the dne estimator assumes that μ_{curr} is μ, from which it follows that the progress is the fraction of the driver node scanned so far. Observe that this assumption holds good when the tuples arrive at the driver node in a "uniform" manner, and when the variance in the per-tuple number of *getnext* calls is low. We formalize this intuition as follows.

Let D be the input node of the pipeline under consideration. We refer to the number of *getnext* calls performed for a given tuple of D as the work done for that tuple. Let μ be the average work done per tuple retrieved from D, and let *var* be the variance in the work done. We have the following result.

THEOREM 3.: *Consider an instant during query execution when k tuples have been processed out of the N tuples to be retrieved from the input node D. Let prog be the progress at this instant. Let $err = |prog - \text{dne}|$ be the error yielded by* dne*. If tuples are retrieved from D in random order, then $E(err) = 0$. In other words, if tuples are retrieved in random order,* dne *is expected to be accurate.*

Proof: Since the tuples arrive in a random order, the subset of tuples retrieved from D constitutes a random sample of all the tuples to be retrieved from D. The result follows from the Central Limit Theorem. □

Hence, if the tuples are retrieved from the input node in random order, then dne is expected to yield the correct value of the progress. We note that previous work on online aggregation [6, 7, 8, 9] assumes that data is retrieved in a random order, and so the random order assumption made by the dne approach holds whenever online aggregation techniques apply. A purely random order, while sufficient, is however not necessary. Instead, it is enough if the order in which tuples are retrieved from the driver node is independent of the number of *getnext* calls per tuple. For example, in Figure 2, it is enough if the ordering of the tuples of R_1 is independent of the number of tuples in R_2 with which it joins. This condition can be met if the storage of R_1 is determined purely by its own properties.

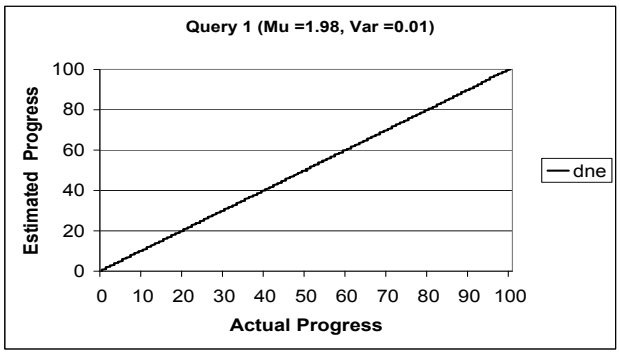

Figure 3: The dne estimator for TPCH Query 1

In addition, by the Central Limit Theorem, Var(err) is proportional to $\frac{var}{N}$. Hence, the error of the dne estimate drops as more of the tuples from the driver node are examined, approaching zero as k approaches N. Indeed, if *var* is low, then the dne estimate is tight and converges to the correct value of the progress very soon. The value *var* can be low when we are dealing, for example, with pipelines that only consist of filters and index nested loop joins that look up keys. In such a setting, where the difference between the maximum and minimum amount of work done per tuple is small, the dne estimator is likely to be very effective.

We next empirically examine whether the above characterization holds. The graph in Figure 3 shows the performance of the dne estimator for Query 1 of the TPCH decision support benchmark [17]. We generated a 1 GB TPCH database with a skew factor of 2 [18]. The graph plots the progress as estimated by the dne estimator on the y axis and the actual progress (according to the *getnext* model) on the x axis. As the graph shows, the dne estimator is almost exactly accurate. For this pipeline, it turns out that μ is 1.98 and *var* is 0.01, which is extremely small. This confirms our intuition that the dne estimator would work well in such scenarios. Note that this behavior is in spite of errors in cardinality estimates (recall that the data has high skew).

On the other hand, a low value of *var* is required for fast convergence (as confirmed by experiments in Section 5). We next ask the question what happens when both assumptions fail, i.e., *var* is high and the tuples retrieved from the driver node aren't in random order. Note that the lower bound argument sketched in Example 1 exploits this property: there is a tuple that causes a high join skew and it appears late. This question is pertinent since the data layout is typically fixed before query execution, and hence, tuples are retrieved in that order.

In this case, we examine if it is possible to bound the error *after* a fraction of the tuples from the driver node is consumed. As discussed above, dne requires that the work done per tuple seen so far reflect the overall average work

done per tuple. We introduce the notion of *predictive* orders where this property holds after a certain point, fixed at 50%. An order of retrieval of tuples from the driver node D is *c-predictive* if after half the tuples have been retrieved from D, the average work done per tuple so far is within a factor of c of μ. In a predictive order, the average work done per tuple is within a factor of the overall average after some point in the execution. Several questions arise. First, what is the effectiveness of dne given a predictive order. Second, how frequent is a predictive order. We address these questions below. First, we observe that:

PROPERTY 2. : *Given a c-predictive order of D,* dne *yields a ratio error of c after half the tuples in D have been retrieved.*

We next ask what fraction of orders are predictive. Fortunately, it turns out this is a reasonable fraction, irrespective of μ and var.

THEOREM 4. : *At least 1/2 the orders of D are 2-predictive.*

Coupled with Property 2, this implies that dne yields a ratio error of at most 2 after half the tuples from the driver node are retrieved, for at least 50% of all orders.

All the above results indicate that there are reasonable cases where dne guarantees bounded error, at least after some fraction of the input is consumed. This provides some intuition behind the empirical results presented in [5, 13].

Among the factors influencing the lower bound argument, the ones relaxed in this discussion was the variance in the amount of work done per tuple and the input order. This raises the question whether there are other scenarios not addressed by dne where we can design progress estimators that bound the ratio error. This is the question we address next.

5. OPPORTUNITIES TO DO BETTER

We showed that the dne does well for single pipelines where the variance in costs is small and the input order is favorable. These relax only some of the properties of the argument presented in Section 3. In this section, we ask where we can do better. We describe two progress estimators —pmax and safe— that use bounds on the cardinalities to perform better than the dne estimator in certain scenarios. We then discuss how the problem of progress estimation can be simplified when certain physical operators, involving those that perform some form of nested iteration, are ignored.

5.1 Bounds on Cardinalities

Recall that our goal is to provide progress estimation with formal guarantees on the error. In order to estimate $total(Q)$ for a query Q, we consider maintaining lower and upper bounds on $total(Q)$, instead of using the query optimizer's estimates which do not come with error intervals.

We maintain cardinality bounds for each operator in the execution. For example, a table scan has lower and upper bounds equal to the cardinality of the base relation, which is accurately available from the database catalogs. As execution proceeds, these bounds can be refined. For example, the lower bound for σ is at least the number of tuples returned so far ("at least" since we could use histograms to yield better bounds). For linear operators such as σ, π and γ, the upper bound is at most that of its child node. In addition, if we know that any of the join operators is linear, that is returns at most as many tuples as the larger of its children, we can set its upper bound to be the larger of the upper bounds of its children. This is the case, for example with foreign key joins. We refer the reader to [5] for more details on how these bounds can be maintained as the execution proceeds.

5.2 The PMax Progress Estimator

The first estimator we describe uses the above bounds as follows. At any point in the query execution, let *Curr* be the current number of *getnext* calls across all operators in the operator tree. Let *LB* be the sum of the lower bounds for the total number of *getnext* calls across all nodes in the operator tree. The pmax estimator assumes that only the least amount of work is going to be done in the future.

DEFINITION 3.: *Consider an instant during query execution. The* pmax *estimator returns* $\frac{Curr}{LB}$.

Since pmax uses a lower bound on the actual cardinalities, we have the following result (also the intuition behind the name pmax).

PROPERTY 4.: *Consider any instant during query execution. Let prog be the progress. We have that:* $prog \leq$ pmax.

We now explain the properties of the pmax estimator through an example.

EXAMPLE 2.: *Consider the index nested loops join example in Figure 2. Suppose both R_1 and R_2 have 100,000 tuples each. Assume that the only tuple to pass through the selection in R_1 joins with 10,000 tuples in R_2 and that the other tuples fail the selection. Notice that for this query, total(Q)* $= 100,000 + 10,000 + 1 = 110,001$.

The maximum error faced by pmax *is when its estimate of the total number of* getnext *calls is farthest from the true value. The LB used by the* pmax *estimator is at least 100,000 since the outer relation has to be scanned once to evaluate this query. Thus, the ratio error for the* pmax *estimator is at most 1.1 (irrespective of the ordering of relation R_1).*

To put this in perspective, we first contrast this setting with the one in Example 1. Recall that in Example 1, the cardinality of R_2 is 10 times that of R_1. As a result, if there is a tuple in R_1 that joins with a large number of tuples in R_2, it will dominate the cost, since there are not enough tuples in R_1 to "compensate". On the other hand, in the current example, there are *enough tuples of R_1 to be processed so*

that even in the presence of join skew, pmax *is able to effectively estimate the progress. This is the intuition captured by a small value of μ.*

On the other hand, this is a data set where the variance in the per-tuple work is high, since there is one tuple that joins with 10,000 tuples of R_2 whereas no other tuple even passes through the selection. As a result, dne *could yield high errors for certain orderings of relation R_1. For instance, if the tuple that satisfies the predicate happens to be the first tuple of R_1, the correct value of progress (after the first tuple is processed) is 10,000/110,001 whereas* dne *would calculate the progress as 1/100,000, which would result in a huge ratio error.*

The above example highlights the intuition behind when pmax is likely to be effective for progress estimation. We formalize this intuition as follows. For a query tree, let $\mathcal{L}_s$ be the set of leaves that are scanned (exactly once). We know then that $LB \geq \Sigma_{i \in \mathcal{L}_s} L_i$ where L_i is the cardinality of leaf i.[2]

Define $\mu = total(Q)/\Sigma_{i \in \mathcal{L}_s} L_i$. Intuitively, μ is the average number of *getnext* calls performed during the entire query execution per "input" tuple. For the single pipelines considered in Section 4, this reduces to the notion of μ introduced there. In particular, for the example in Figure 2, μ is the average number of *getnext* calls performed per tuple of R_1. We have the following:

THEOREM 5.: *Consider any instant during query execution. Let prog be the progress. Then: prog* $\leq$ pmax $\leq$ $\mu \cdot prog$. *In other words,* pmax *is within a factor of μ of the correct progress.*

Proof: The proof follows from the fact that the lower bound is at most a fraction of $1/\mu$ of the correct number of *getnext* calls. □

This result states that pmax is effective as a progress estimator when μ is small. Observe that the above guarantee depends only on μ and holds irrespective of the variance in the "per-tuple costs". In cases where μ is small and the variance is high, the pmax estimator is likely to substantially outperform the dne estimator.

We now illustrate this with an experiment performed on Microsoft SQL Server 2005 (Beta 2). We base our experiment on a synthetic data set that introduces a skew in the "work done" per tuple. For this purpose, we generate two relations $R_1(A)$ and $R_2(B)$, both containing 10,000,000 tuples. The tuples of $R_2(B)$ are generated using the zipfian distribution on the join attribute which is known to commonly occur in practice. The zipfian parameter z is set at 2. On the other hand, the values in $R_1(A)$ are unique. The variance in the per-tuple cost is achieved simply by joining R_1 with R_2 through index nested loops, using a plan similar to the one shown in Figure 2. The μ value for this query is 2.

[2]For a leaf operator that is a range scan on a clustered index, lower bounds can be obtained by looking at appropriate bucket boundaries in histograms

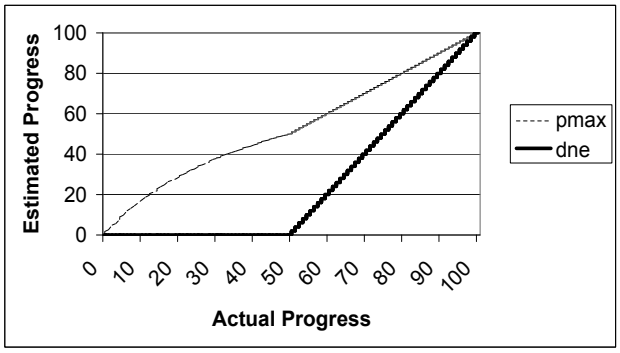

Figure 4: pmax **vs** dne

However, the variance in per-tuple costs is high since the join column in R_2 follows a zipfian distribution. Analogous to the discussion in Example 2, consider an ordering of R_1 where the elements that cause a high join skew appear at the very beginning. In this setting, as argued in Example 2, the dne estimator tends to substantially underestimate the progress. Figure 4 shows how these estimators perform for this query. As the graph indicates, the pmax estimator is more effective while the dne estimator substantially underestimates the progress.

We note here that while the above experiment is carried out for a single join query, the underlying behavior of the data and query, where the value of μ is small, but the variance in per-tuple costs is high, could easily be part of a more complex query. The pmax estimator is an effective choice for such cases. In Section 6, we investigate how small the values of μ are, over both benchmark and real data sets.

So far, we have shown that the problem of progress estimation can be effectively addressed in scenarios obtained by restricting some of the properties of the lower bound argument in Section 3. The lower bound results indicate that when these "interesting" subclasses don't apply, then the progress estimator cannot provide effective guarantees. In the next section, we revisit the worst case and ask if we can *limit* the error obtained.

5.3 The Safe Estimator

Let us revisit the scenario described in Example 1. The key problem that a progress estimator faces when it is about to retrieve the next tuple from R_1 is whether this tuple is going to cause a lot of *getnext* calls or very few *getnext* calls or somewhere in between. The dne estimator takes the approach of saying the present is an indicator of the future, and so the next tuple is going to produce as many *getnext* calls as the current per-tuple average. The pmax estimator, on the other hand, simply assumes that the next tuple produces minimal number of *getnext* calls. Both these estimators suffer when their prediction is way off the mark, as in the lower bound argument (Example 1), where the tuple t produces a huge number of *getnext* calls. The reason is that they do not account for the possibility that their assumptions about the future tuples may be totally wrong. In this section, we introduce the safe estimator that is prepared for the possibility that the future could be completely different from the execution trace seen so far.

The safe estimator also uses the bounds maintained on cardinalities discussed in Section 5.1. At any point in the query execution, let *Curr* be the current number of *getnext* calls across all operators in the operator tree. Let *LB* be the sum of the lower bounds for the total number of *getnext* calls across all nodes in the operator tree and *UB* be the corresponding sum of the upper bounds.

DEFINITION 5.: *Consider an instant during query execution. The* safe *estimator returns* $\frac{Curr}{\sqrt{LB \times UB}}$.

Observe that the ratio error yielded by the safe estimator is at most $\sqrt{\frac{UB}{LB}}$. Consider the lower bound argument in Example 1. The essence of the argument is that the presence or absence of a *single* tuple can determine whether the sum of all the operator cardinalities approaches the lower bound or the upper bound. It is impossible to detect the presence or absence of a single tuple from the database statistics and the execution feedback. The safe estimator hence plays it "safe" by assuming that either of the two bounds is attainable. The following result shows that this is indeed worst-case optimal.

THEOREM 6.: *No progress estimator can guarantee a ratio error lower than the* safe *estimator in the worst case. In other words,* safe *is worst-case optimal.*

Proof: This result follows from the proof of Theorem 1 as follows. There is a query and the set of instances considered in that proof. On at least one of the instances, any progress estimator has a higher ratio error than safe. Hence, there is at least one query and one instance where any progress estimator yields higher error than safe. □

Note that the safe estimator deals with the kind of scenario described in Example 1 by assuming that both the extremes are possible in future, and taking a "middle" road to minimize the worst-case error. We now show experimentally how this estimator can substantially improve upon the dne estimator when the worst case behavior takes place. We use the same experimental setup as in the previous section (relations $R_1(A)$ and $R_2(B)$). Consider $(R_1 \bowtie R_2)$ where the join algorithm is index nested loops (with R_1 as the outer). Recall that while R_1 has unique values in its column A, the distribution of $R_2(B)$ is zipfian. Consider the ordering of relation R_1 where the element that joins with the most number of tuples occurs at the end. At the instant before this tuple is retrieved, the dne estimator forecasts that the query is almost finished thus overestimating the progress, whereas this tuple joins with a lot of tuples in R_2, thus causing a large number of *getnext* calls. The safe estimator, on the other hand accounts for this possibility and yields substantially lower error, as shown in Figure 5. We note that on this instance, the pmax estimator would behave almost identically to the dne estimator.

In this section, we presented an estimator (safe) which is worst-case optimal. We also showed empirically that this estimator could perform substantially better than the dne estimator for the worst case. We examine the lower bound argument again to note that the argument is carried out using the index nested loops join algorithm. Thus, one of the possibilities in addressing the lower bound in Section 3 is to simply ask if it is a property of the specific physical operator considered. We consider this option next.

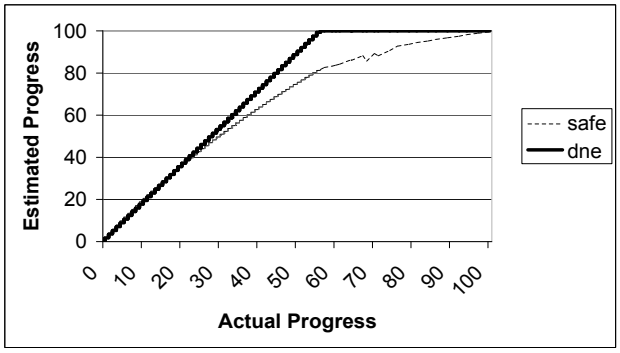

Figure 5: worst-case order

5.4 Scan Based Queries

One property of the lower bound argument in Section 3 is that the join method used is index nested loops. This gives rise to the question whether the crux of the argument is specific to this particular join method. Does the nature of the progress estimation problem change in the absence of this operator? We investigate this question in this section.

We begin with an example involving a hash join and study what happens.

EXAMPLE 3. : *Consider the two-way join* $(R_1 \bowtie^{Hash}_{R_1.A=R_2.B} R_2$, *where* R_1 *is the build side. Assume that the join is linear, that is that the output size is at most that of the larger input. Also assume that, as in Example 1,* $|R_2| > |R_1|$. *Query execution would start with a build phase where relation* R_1 *is scanned followed by a probe phase in which relation* R_2 *is scanned. There is a crucial difference in this setting contrasting it with the setting described in Section 3. The key is that* both R_1 *and* R_2 *are scanned. As a result, the total number of* getnext *calls lies between* $|R_1| + |R_2|$ *and* $2(|R_1| + |R_2|)$.

On this instance, safe *yields a ratio error of at most* $\sqrt{2}$, *while* pmax *yields a ratio error of at most 2, since* $\mu \leq 2$. *Hence, changing the physical join operator makes it possible to bound the error in progress estimation.*

The key in the above example is that both the inputs to the join are fully scanned. Indeed, if the join operator is a sort-merge join where each input is sorted, we obtain a similar result. We generalize this to the class of what we call *scan-based* queries.

A scan based query is defined to be defined to be operator trees without operators $\bowtie^{\text{NL}}$, $\bowtie^{\text{INL}}$ and index-seek i.e., operators performing some form of nested iteration. Scan-based operator trees have the property that each leaf node is scanned (exactly once). We assume *linear* operator trees, where each operator is linear, that is has the property that its result size is at most the size of the largest of its inputs.

Note that σ, π, γ, and foreign key joins are all linear operators. Scan-based queries are quite common in decision support environments, especially with adhoc queries. In fact, many of the benchmark queries in the TPCH benchmark produce plans that are scan-based.

Let the operator tree have m internal nodes. Let $L_i, 1 \leq i \leq l$ be the cardinalities of the leaf nodes in the operator tree. Observe that:

$$\begin{aligned} LB &\geq \Sigma_1^l L_i \\ UB &\leq \Sigma_1^l L_i + m.\max(L_i) \text{ by linearity} \\ \text{Setting } L_1 &= \max(L_i) \text{ and } \alpha = \Sigma_2^l L_i, \text{ we get} \\ UB &\leq L_1(m+1) + \alpha \leq (m+1)LB \end{aligned}$$

As a result, we have the following properties of scan-based queries.

PROPERTY 6.: *Consider a scan-based query with m internal nodes.*

1. $\mu \leq (m+1)$
2. *The* safe *estimator yields a ratio error of at most* $\sqrt{m+1}$

It follows that pmax yields a ratio error of at most $m+1$. By constraining dne to be within the upper and lower bounds on the progress, we find that dne also yields a ratio error of at most $m+1$.

Given that in the presence of nested iteration, we cannot even achieve even the threshold guarantee (Section 3), the above result shows that the problem becomes worst-case tractable in the absence of nested iteration. We also note that the above bounds are *worst-case*, and errors are likely to be much smaller in practice. Next, we empirically study the improvement obtained by restricting the class of physical operators.

Recall that our experiment in Section 5.3 involving relations $R_1(A)$ and $R_2(B)$ illustrates the worst case behavior of the dne estimator, and also the pmax estimator. This happens when the element that causes high join skew appears at the end of R_1 and both the dne and pmax estimators do not account for this. We study the effect of scan-based operators simply by reusing the same setup and repeating the experiment with a hash join instead of an index nested loops join. We then compare the effectiveness of each of the estimators under consideration between the two cases. Table 1 reports the results of our experiment. Clearly, there is a substantial improvement when we move to a scan based plan. We also note that this property holds for all the estimators we consider.

To summarize the results in this section, we began with the lower bound argument and found that by relaxing some of the properties of the argument, we are able to design effective progress estimators for different scenarios. We now move on to study whether it is possible to combine the progress estimators we have described so far to obtain one that addresses all the above scenarios.

Progress Estimator	Max Err (INL)	Max Err (Hash)	Avg Err (INL)	Avg Err (Hash)
dne	49.50%	19.20%	24.74%	7.37%
pmax	49.50%	19.20%	24.74%	9.04%
safe	25.2%	8.2%	14.8%	4.2%

Table 1: Impact of Scan-based Plan

6. WHICH ESTIMATOR TO USE?

Section 3 shows that no progress estimator can guarantee low error bounds across all data and query distributions. However, Sections 4 and 5 discuss different progress estimators that address various scenarios obtained by relaxing the properties of the argument in Section 3. Previous work has shown that there are common cases where the dne estimator is effective. We begin by empirically showing that "good" cases for the pmax estimator are also common in practice. On the other hand, the safe estimator is worst-case optimal. This leaves us with the question of which of these estimators to use which we elaborate in this section.

6.1 Scenarios in Practice

We first address how common are the scenarios that favor the estimators we discuss. Previous work [5, 13] has shown cases in practice where the dne estimator is effective. Recall that the safe estimator targets worst case scenarios. What about the pmax estimator? Are the cases where μ (recall that this is the average "work" performed per input tuple, as defined in Section 5) is small common in practice? We address this question in this section.

Intuitively, in adhoc decision support queries, we expect that a large amount of data is scanned in order to compute a relatively small number of aggregates. This leads us to expect relatively small values of μ for such data sets. We confirm this intuition with measurements over two data sets. One is the TPCH decision support benchmark [17], generated with a skew factor of 2 [18], and its associated benchmark queries. The other is the personal edition of the Sky Server database [4] which is a real-life astronomical database, which comes with a suite of 35 queries. The data sizes in both cases is 1GB.

Table 2 illustrates the μ values for the benchmark queries in the TPCH suite. Table 3 illustrates the same for the long running queries from the Sky Server data set. As the numbers indicate, there are many cases when the μ value is extremely small. For example, query 4 in the TPCH benchmark suite has a value of $\mu = 1.003$. For this query, the maximum ratio error yielded by the pmax estimator is 1.003.

Consider cases where the μ value is higher, for instance TPCH Query 21, a complex query with multiple subqueries. Even here, the continuous refinement of the bounds on cardinalities means that the pmax estimator catches up with the actual progress as execution proceeds. Figure 6 shows how the ratio error drops as query execution proceeds for Query 21. We can see that the error drops to a small value (around 1.5) after a reasonable fraction (around 30%) of the query is done, soon converging to 1 as the query executes further.

Query	μ Value
1	1.989
2	1.213
3	1.886
4	1.003
5	1.007
6	1.008
7	1.538
8	1.432
9	1.021
10	1.004
11	1.014
12	1.001
13	2.019
14	1.001
15	1.149
16	1.157
17	1.020
18	2.771
19	1.025
20	1.159
21	2.782

Table 2: μ values for TPCH

Query	μ value
3	1.008
6	1.428
14	1.078
18	1.79
22	1.246
28	1.044
32	1.253

Table 3: μ values for Sky Server

6.2 Is One Estimator Enough?

In this section, we ask the question if there is a clear winner among the three estimators we have discussed so far.

Our experiments in Section 5 show that there are scenarios likely to occur in practice where the pmax and safe estimators are substantially more effective than the dne estimator.

What about the pmax estimator? Consider TPCH Query 1, where we showed previously that the dne estimator is very effective (Figure 3), with an average error of less than 1%). On this query, the pmax algorithm results in an average error of around 11%. The fact that the variance in per-tuple "costs" is small is not exploited by the pmax estimator, whereas if the variance is indeed small as it is for this query, the dne estimator is very effective.

What about safe? It has the property of worst-case optimality. How bad can it be with respect to the dne in cases favorable to the dne estimator. We study this using the same setup used to show where safe is more effective than dne. Figure 5 illustrated the tradeoff between the dne and safe estimators for the worst case. Consider the same join query (between tables R_1 and R_2) with an additional predicate on relation R_1 that filters out the high skewed tuples in R_1. As a result, very few tuples will actually join; thus the variance in the per-tuple work is negligible. Figure 7 recasts the performance of the two algorithms (dne, safe) in this scenario; dne is almost exactly accurate in this case (as expected) while the safe estimator is off by 20% even at the end. This illustrates the point that while the safe estimator is worst-case optimal, it's accuracy could suffer in cases favorable to the other estimators we discuss.

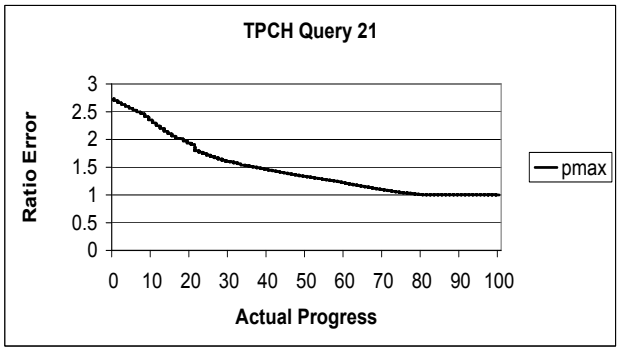

Figure 6: Ratio error of pmax over query execution

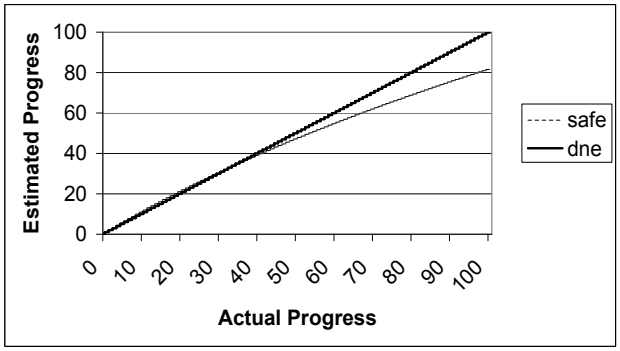

Figure 7: safe vs. dne

The main observation we make from this discussion is that there is no clear winner among the estimators we have discussed. Hence, we are left with a tool-kit of three estimators, which have the following properties: one is a "safe" estimator that is prepared for the worst case. The others make assumptions about the query and data characteristics, and are very effective when those assumptions hold. Our empirical evidence indicates that these assumptions often hold in practice. But, of course, both these estimators could yield large errors in the worst case. We also recall from Section 3 that the worst case behavior is quite likely in practice. How do we then fix an estimator? Can we at least detect when to switch among these estimators? We address this question next.

6.3 Detecting which estimator to use

Among the three progress estimators we discuss, the dne estimator is geared toward single pipelines where the variance in the per-tuple cost is low, or at least the order is predictive (as defined in Section 4) so that the estimator can converge after processing a fraction of the input. On the other hand,

the pmax estimator is geared toward cases where the mean per-tuple cost is small. The safe estimator does not assume anything about the data and accounts for the worst case.

In order to be able to use the above estimators for the specific settings they target, the least we must be able to do is to detect these settings. We show next that unfortunately, this is not possible. The intuition behind these results is similar to the argument in Section 3. Recall that in Example 1, a progress estimator is unable to decide whether the next tuple to be retrieved joins with a large number of tuples or not. Note that depending on whether this tuple joins with a large number of tuples or not, the value of the "per-tuple average" μ would undergo a large change too. We formalize this through the following result.

THEOREM 7.: *Consider a query tree Q. For any $c \geq 1$, no progress estimator can estimate μ within a factor of c.*

Proof: Follows from the proof of Theorem 1. □

Similarly, recall that the dne estimator primarily focuses on single pipelines and relies on the input order having "predictive" properties. We next ask whether it is possible to detect whether the input order is predictive, in which case Property 2 tells us that the dne estimator is effective. Again, the same intuition used for the argument in Section 3 also explains why no progress estimator can detect whether the input order is predictive or not — if the next tuple joins with a large number of tuples, then the input order is not predictive, and if not, then the input order *is* predictive. We formalize this intuition through the following result.

THEOREM 8.: *Consider a single pipeline query. Fix constant $c > 1$. Even after half the tuples are retrieved from the input node, no progress estimator can determine whether the input order is c-predictive or not.*

Proof: Follows from the proof of Theorem 1. □

Given these results, the only option we are faced with is to identify heuristic approaches to combine these estimators. We next discuss some possible approaches that can be taken.

6.4 Combining the estimators

The previous sections have outlined a tool-kit of estimators that consist of one worst-case optimal "safe" estimator and two other estimators, each of which addresses scenarios that we have demonstrated occur often in practice. We also have results that show that no single algorithm can perform well across the spectrum and that, in particular it is not possible to formally improve effectiveness by switching among these algorithms at runtime. Hence the choice must be made heuristically based on query and data characteristics.

Using the execution of the query so far or over a sliding window to gather information is a useful direction to consider. This could be used to find out whether μ is small, whether the variance in per-tuple costs is small, and whether the input order is predictive (in the case of single pipelines). Based on the behavior of the query so far, we can think of ways of switching between the estimators. For example, one can even think of a hybrid algorithm that uses the safe estimator but switches to the pmax estimator for the current pipeline if the value of μ is small.

Other approaches could be based on the properties of the specific physical plan. For example, for queries involving queries involving simple filter predicates and key lookup joins, the variance in per-tuple costs is likely to be low. In such cases, the dne estimator is likely to be the best choice.

Another promising direction is to use inter-query feedback, either across different runs of the same query, or across runs of similar looking physical plans. This could be used to bound the values of μ, the values of the variance, or even to detect whether the tuple arrival order is predictive.

We intend to study the possibilities of some of the above approaches in detail as part of future work.

7. DISCUSSION

In this section, we explore the relationship between progress estimation and some of the traditional problems of query optimization [10] such as cardinality estimation, join sampling [3] and query reoptimization [12, 15].

The model of work used in this paper is based on the number of *getnext* calls. Notice that under this model, if a solution to cardinality estimation exists, the progress estimation problem is trivially solved (since accurate estimates at each individual node are available). We presented some interesting subclasses where effective estimators are available. It is important to note that the errors in cardinality estimation could *remain* in such cases. For instance, we showed in certain cases the μ values for the TPCH database (even with skew 2) could be small making it an effective case for progress estimation. However, since the skew factor is high, the errors in the cardinality estimates are off by orders of magnitude. A similar case arises for TPCH Query 1 (which has low variance), where the dne estimator can perform well in spite of the errors in cardinality estimation. The key difference is the fact that progress estimation (as defined by the *getnext* model) only needs accurate estimates for the *sum* of the cardinalities at all nodes and does not require accurate estimate at each individual node.

Interestingly, as seen from our discussion in Section 5, it turns out that the nature of join algorithm used in the final plan can strongly influence the robustness of progress estimation. This is not the case for problems such as cardinality estimation and join sampling, which are independent of the algorithm used by the join operator.

There has been work in the context of query reoptimization where the optimizer recomputes the cardinality estimates at runtime in order to change the current plan. If the recomputed cardinality estimates are accurate, it could obviously be used for progress estimation. Typically, such systems reinvoke the optimizer cost functions to derive the new estimates; as a result errors due to simplistic assumptions (e.g., independence between predicates) would remain. Moreover, the optimizer estimates come with no guarantees which is the focus of this work. The estimators proposed in this paper

relied on maintaining bounds on cardinalities at runtime. It would be interesting to examine the use of similar bounds for the purpose of query reoptimization.

Moreover, we are able to exploit in addition to everything else, the *order* in which tuples are processed, which is again in contrast with cardinality estimation and join sampling. As a result, if we assume that tuples arrive in a random order, it is possible to provide accurate solutions to progress estimation in certain cases. There has been prior work in the context of online aggregation which propose specialized operators (e.g., ripple joins) in order to provide a random order. The dne estimator is guaranteed to work well for such operators.

Query optimizers have traditionally worked on throughput or response time metrics. Some optimizers have the ability to optimize queries for the "first-tuple" metric. In this paper, we showed that the join algorithm could be an important factor in the robustness of a progress estimator. The interface to the user could be an important factor while choosing query plans. Exploring further connections between query optimization and progress estimation is future work.

8. CONCLUSIONS

The focus of this paper has been to characterize the query progress estimation problem introduced in [5, 13] so as to understand what the important parameters are and under what situations can we expect to have a robust estimation of such progress. We have good and bad news.

The bad news is that providing any nontrivial guarantee in a worst-case sense for even single join queries is not possible. We presented an algorithm (safe) that can provide the best possible error guarantees for the worst-case. The good news is that by placing appropriate restrictions on the query/data characteristics, we can design effective estimators. We show where previously proposed solutions (based on the concept of driver/dominant node) lie in this spectrum. We also illustrate the fact that these "good" cases are fairly common in practise (using real and benchmark data sets).

It turns out that there is an interesting tradeoff between these algorithms. While the estimators dne and pmax can have high errors in the worst case, the safe estimator can perform poorly in the "good" cases. Thus, there is no *single* estimator that can cover the whole spectrum. We also showed that it is not possible to choose one of these estimators dynamically. This leads to a tool-kit of estimators; we explored some initial ideas on choosing an appropriate estimator and intend to develop further heuristics based on query/data characteristics as part of future work.

9. REFERENCES

[1] S. Acharya, P. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, 1999.

[2] N. Bruno and S. Chaudhuri. Statistics on query expressions. In *SIGMOD*, 2002.

[3] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, 1999.

[4] S. Chaudhuri and V. Narasayya. The sky server database. `http://skyserver.sdss.org`.

[5] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of long-running queries. In *SIGMOD*, 2004.

[6] P. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, 1997.

[7] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

[8] P. Haas, J. Naughton, S. Seshadri, and A. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 1996.

[9] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *SIGMOD*, 1997.

[10] Y. Ioannadis. Query optimization. *ACM Computing Surveys*, 1996.

[11] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[12] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[13] G. Luo, J. Naughton, C. Ellmann, and M. Watzke. Towards a progress indicator for database queries. In *SIGMOD*, 2004.

[14] G. Luo, J. Naughton, C. Ellmann, and M. Watzke. Increasing the accuracy and coverage of SQL progress indicators. In *ICDE*, 2005.

[15] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[16] V. Poosala and Y. E. Ioannidis. Balancing histogram optimality and practicality for query result size estimation. In *SIGMOD*, 1995.

[17] The TPCH Benchmark. `http://www.tpc.org`.

[18] Program for TPC-D Data Generation with Skew. `ftp://ftp.research.microsoft.com /users/viveknar/tpcdskew`.