Estimating Progress of Execution for SQL Queries

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Ravishankar Ramamurthy[1]
University of Wisconsin, Madison
ravi@cs.wisc.edu

ABSTRACT

Today's database systems provide little feedback to the user/DBA on how much of a SQL query's execution has been completed. For long running queries, such feedback can be very useful, for example, to help decide whether the query should be terminated or allowed to run to completion. Although the above requirement is easy to express, developing a robust indicator of progress for query execution is challenging. In this paper, we study the above problem and present techniques that can form the basis for effective progress estimation. The results of experimentally validating our techniques in Microsoft SQL Server are promising.

1. INTRODUCTION

Decision support applications typically include long-running queries. For such queries, the ability to estimate the progress of query execution could be very useful. Progress estimation could help DBAs as well as end users or applications help decide whether to terminate the query or allow it to finish. Such feedback could qualitatively improve the experience for any database user. However, today's database systems only provide rudimentary feedback to users about progress of query execution. This feedback is limited to the query optimizer generated execution plan and its cost, as well as the number of tuples returned by the query during its execution. Beyond this, to the best of our knowledge, there is no prior published work on the problem of progress estimation for SQL query execution.

The most useful measure of progress would report to the user at any point during the query's execution, the amount of *time* required for the query to complete execution. However, any method that provides such a measure would be subject to uncertainty arising from concurrent execution of other queries. Due to this difficulty, we focus on the problem of estimating the *percentage remaining* (or equivalently completed) of the query, at any point during its execution, i.e., reporting a "progress bar" for query execution. Such an estimator is simpler than estimating time remaining since it is independent of other queries. In effect, this measure estimates the time remaining on an *isolated system* where only the given query is executing.

Effective progress estimation for query execution requires us to accurately estimate the total "work" required to execute the query. Queries in modern database systems are quite complex involving joins, nested sub-queries and aggregation. Any measure of work for a query that is independent of the intermediate cardinalities of such operators is likely to be too simplistic. For example, consider a metric that reports progress as the percentage of query results that have been returned thus far. Let us assume that we could accurately estimate the total number of rows that a query will return in its result. To see why such a metric for progress could be really inaccurate, consider an execution plan consisting of a very expensive join followed by an inexpensive Sort operation. Since Sort is a blocking operation, query results are not returned until the Sort starts outputting rows. Therefore, until such time, the above metric would report no progress irrespective of how much work was done in the join. As another illustration of why the problem is difficult, consider a metric that reports the percentage of nodes (i.e., operators) in the execution plan that have *completed*. However, if a query is just a single pipeline of operators, for almost the entire duration of the execution of the query, *all* the operators in the plan are active i.e., not yet completed. Thus the above metric will not report any progress until near the very end of query execution .

We note that a query optimizer already uses a model of work done by a query (based on estimated CPU and I/O costs). While leveraging this model for progress estimation may be possible, in this paper, we ask whether an even simpler model would suffice for the purposes of progress estimation. The motivation for this simpler model is the ease of incorporation into existing query execution engines. We model work done by a query as a function of the number of rows output by *each* operator in the query execution plan.

While this model does inherit the known difficulties of cardinality estimation faced by a query optimizer, we use two key ideas to help mitigate the impact of inaccurate cardinality estimation on *progress* estimation. First, we observe that it is possible to estimate the cardinalities of certain operators e.g., Table Scans or Index Scans which we refer to as *driver* nodes (formally defined in Section 2) much more accurately than other intermediate nodes in a pipeline e.g., a Filter or Hash Join. We show that in many cases estimating the overall query progress by only monitoring progress of these driver nodes can greatly improve accuracy. Second, *during query execution* we leverage runtime execution information to refine cardinality estimation. We take a conservative approach (based on maintaining and refining upper and lower bounds on cardinalities of operators in the plan) that is guaranteed not to introduce additional inaccuracies as a result of such refinement. Our solution is applicable to arbitrary SQL queries and can be implemented at low overhead in existing database systems. We have implemented our techniques inside Microsoft SQL Server and the initial results of experimentally

SIGMOD 2004, June 13–18, 2004, Paris, France.

[1] Work done while author was visiting Microsoft Research.

evaluating our estimator on the TPC-H benchmark [12] queries (10 GB version on uniform as well as skewed data distributions) are promising.

The rest of the paper is structured as follows. Section 2 describes the problem and presents our model of work done by a query. Given this model, we propose in Section 3, an estimator for the progress of a query whose execution consists of a single pipeline. Section 4 presents our solution for the general case of a query involving multiple pipelines. Experimental validation of our prototype on decision support queries is presented in Section 5. Section 6 discusses the desirable property of *monotonicity* of progress estimation and its relationship to accuracy of estimation. We present extensions to our model of work to be more robust to runtime conditions in Section 7, and discuss related work in Section 8. We conclude with a brief discussion on interesting areas of future work.

2. PROBLEM DESCRIPTION

2.1 Definitions

A progress estimator uses an *execution plan* that is chosen by the query optimizer for the given query. An execution plan is a tree where the nodes of the tree are physical operators. For example, Figure 1 shows the execution plan for a query.

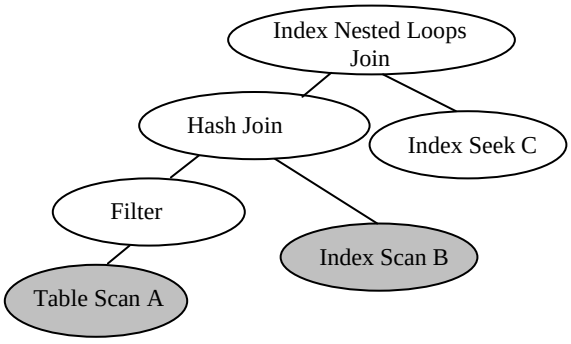

Figure 1. Example of an execution plan for a query

A physical operator is referred to as a *blocking* operator if it does not produce any outputs until it has consumed at least one of its inputs completely. For example, suppose Table Scan A with Filter is the *build* relation of the Hash Join and Index Scan B is the *probe* relation. The Hash Join operator in Figure 1 is blocking since it must consume all rows from the build relation before it produces any output. Another example of a common blocking operator is Sort.

The overall execution of a query is staged into multiple *pipelines.* We now define the notion of pipelines for an execution plan consisting of common physical operators such as Table Scan, Index Scan, Index Seek, Filter, Hash Join, Merge-Join, Index Nested Loops (INL) Join, NL Join, Group-By (Hash-based) and Sort. The definition is procedural and proceeds inductively in a bottom up manner over the nodes of an execution plan. A leaf node of the plan (Table Scan, Index Scan, Index Seek) starts a pipeline. A Filter node is part of the pipeline that its child operator belongs to. For a Hash Join, the join operator is included in the pipeline of the probe child, and the build child is the root of another pipeline. For a Merge-Join, the pipelines containing its children and the Merge Join operator itself are union'ed to create a single pipeline. For a Nested Loops or Index Nested Loops Join operator, the outer child, the join operator and its entire inner subtree are part of a the same pipeline as the outer child node. Both Sort and Group-By (hash-based) operators, which are blocking, start a new pipeline of their own. For the example in Figure 1, the pipelines are: $\mathbf{P_1}$ = {Table Scan A, Filter}, $\mathbf{P_2}$ = {Index Scan B, Hash Join, Index Nested Loops, Index Seek C}. In principle the above definition of can be extended to other physical operators as well. Thus, intuitively, a pipeline can be thought of as a maximal subtree of concurrently executing operators.

Every pipeline has a set of *driver* nodes, i.e., operators that are the sources of tuples operated upon by remaining nodes in the pipeline. More precisely, we define the driver nodes of a pipeline as the set of all leaf nodes of the pipeline, except those that are in the inner subtree of a Nested Loops/ Index Nested Loops join. For example, in Figure 1, the shaded nodes are driver nodes – Table Scan A is the driver node for the pipeline $\mathbf{P_1}$ and Index Scan B is the driver node for pipeline $\mathbf{P_2}$. Note that Index Seek C is not a driver node since it is a leaf node of the inner subtree of an Index Nested Loops Join. We observe that it is possible for a pipeline to contain more than one driver node, e.g., in a Merge-Join of two sorted relations, both the input relations to the Merge-Join are driver nodes.

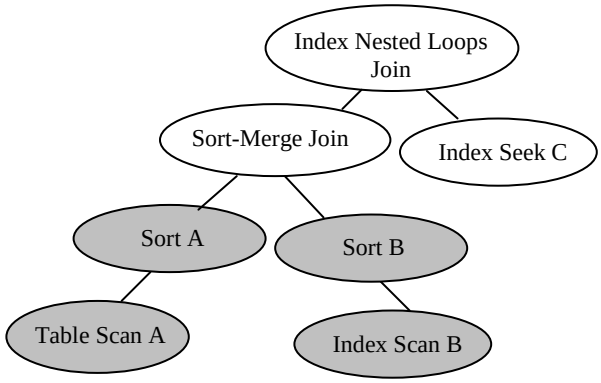

Figure 2. Execution plan with Sort-Merge Join.

This is illustrated in Figure 2. The pipelines identified for this query would be $\mathbf{P_1}$ = {Table Scan A}, $\mathbf{P_2}$ = {Index Scan B} $\mathbf{P_3}$ = {Sort A, Sort B, Merge Join, Index Nested Loops, Index Seek C} and the driver nodes (the shaded nodes) would be respectively {Table Scan A}, {Table Scan B}, {Sort A, Sort B}. Thus there are two driver nodes for the last pipeline. We note that unlike a Hash Join, for a Sort-Merge Join, the scans of both inputs do not necessarily need to complete for the Sort-Merge Join to complete.

An execution plan can be viewed as a *partial order of pipelines* since, in general, for certain pipelines to start executing, one or ore other pipelines need to complete. For example in Figure 1, execution of $\mathbf{P_1}$ must precede $\mathbf{P_2}$. Similarly in Figure 2, execution of $\mathbf{P_1}$ *and* $\mathbf{P_2}$ must precede $\mathbf{P_3}$, but the order between $\mathbf{P_1}$ and $\mathbf{P_2}$ is arbitrary.

2.2 Desirable Properties of a Progress Estimator

Accuracy: The *estimated* percentage of work completed by the query at any point during its execution should be close to the *actual* percentage of work completed by the query at that point.

Fine granularity: It follows from the above accuracy requirement that the estimator should be able to provide estimates at sufficiently *fine granularity* over the duration of the query's execution. Thus, for example, an estimator that only provides accurate estimates at 0% and 100% completion would not be useful.

Low overhead: An essential requirement for a progress estimator to be practical is that it should impose low overhead on the actual execution of the query.

Leveraging feedback from execution: As query execution progresses, more information based on (intermediate) results of execution can become available. Ideally, an estimator should be able to take full advantage of such information.

Monotonicity: Since the actual execution of the query progresses monotonically, ideally, the estimated progress should be also be monotonically increasing from the start of query execution to its finish.

We observe that in today's database systems feedback on query progress during execution does not satisfy one or more of the above requirements. While the optimizer estimated cost of a query can be obtained at low overhead and progress estimation based on this cost is trivially monotonic (since the estimated cost does not change over the lifetime of the query's execution), it can potentially be inaccurate and it does not leverage any feedback from execution. Similarly, the number of tuples returned by a query during its execution (while low overhead and monotonic) has the major drawback that it can be inaccurate and lacking in granularity – as illustrated in the introduction. Moreover, it only takes limited advantage of execution feedback. Finally, we note that in general, there is a trade-off between guaranteeing monotonicity and achieving accuracy of progress estimation (we discuss this further in Section 6).

2.3 The GetNext() Model of Work

As described in the introduction, our goal is to estimate progress of a query on an isolated system, i.e., on a system where there is no other activity besides the execution of this query. Any progress estimator requires a model of work done by a query as the basis of its estimation. In this section we present such a model of work. One approach for modeling the work done by a query could have been to use the cost model used by query optimizer's for comparing different execution plans for a query. Query optimizers typically model the work done by the query as a function of CPU, random I/O and sequential I/O costs. Thus, to use such a model for progress estimation, we would need to measure the CPU, random and sequential I/O's performed by the query during its execution. In this paper we investigate whether an even simpler model of work would be adequate for the purposes of progress estimation. The main motivation for a simpler model is the ease with which it can be incorporated into today's database systems. The reason we expect that a simpler model may be adequate for progress estimation is that unlike the query optimizer that needs to distinguish between multiple plans for a given query using its cost model, we only need to be able to estimate the percentage of work done for a given query execution plan.

We note that operators in a query execution plan are typically implemented using a demand driven iterator model [5], where each physical operator in the execution plan exports a standard interface for query processing (including Open(), Close() and GetNext()). We propose to model the work done by a query as the *total number of GetNext()* calls issued throughout the duration of the query's execution over all operators in the execution plan. In essence, we are counting each GetNext() call as a primitive operation of query processing and modeling the total work done by the query by the total number of GetNext() calls. Note that all CPU instructions, I/Os etc. performed by the query occurs as a result of GetNext() calls. Thus, this model assumes that the total time required to execute the query is amortized across multiple GetNext() calls, and therefore the percentage of GetNext() calls done thus far is a good estimator of the time taken by the query (on an isolated system).

It should be noted that the GetNext() model of work is inadequate for the purposes of query optimization. As a simple example of why this is the case, consider two plans for the same query: one involving a non-clustered index Index Seek and another involving a Table Scan. With the above GetNext() model of work, the Index Seek would *always* be considered cheaper (i.e., less work) by the query optimizer since the number of rows it returns can never exceed that of the Table Scan.

Progress Estimation Based on GetNext() model

We now define progress estimation based on the GetNext() model of work. Suppose the execution plan has a total of *m* operators. Let the total number of tuples that flow out of operator Op_i (i.e., number of GetNext() calls invoked on that operator) at the end of query execution be N_i (i= 1..*m*). At any point during query execution, let the number of tuples that have flowed out of every operator thus far be K_i (i = 1..*m*). Thus, the ideal estimator under the GetNext() model of work (we call it ***gnm***) would estimate progress at that point during the query's execution as:

$$gnm = \frac{\sum_i K_i}{\sum_i N_i}$$

Note that while accurate K_i values can be obtained as the query is executing, the exact N_i values are available only at the end of query execution. Thus, the estimator *gnm* is not directly implementable as stated above since N_i's are not known exactly while the query is executing.

Thus, the key challenge for any progress estimator **E** that uses the above model of work is to *estimate* $\sum N_i$ as accurately as possible *while the query is executing*. Note that the problem of estimating the number of GetNext() calls for an operator in the query execution plan is the cardinality estimation problem faced by query optimizers. The only difference is that unlike a query optimizer, which can only use pre-computed database statistics (e.g., histograms), the estimator **E** can potentially also observe feedback from query execution for use in its estimation.

We observe that the fine granularity requirement (see Section 2.2) should typically be satisfied by an estimator using the GetNext() model since for a long running query, a large number of GetNext() calls are made during its execution. Another desirable property of an estimator is small runtime overhead. For example, an estimator that actually executes the query in order to obtain the total number of GetNext() calls ($\sum N_i$) would be unacceptable. Thus, we require that the information used by any estimator be limited to a small amount of aggregated information either in the form of pre-

computed database statistics or statistics computed on observed feedback from query execution. Although this restriction by itself is not sufficient to guarantee low overhead, it appears to be necessary for an estimator to be practical. The estimator that we present in this paper uses feedback from query execution (see Section 4) to refine estimates of N_i. Observe that since $\sum K_i$ is monotonically increasing as the query executes, the monotonicity of the estimator depends on how the estimates of $\sum N_i$ are changed by the estimator as the query executes. We comment on the monotonicity property of our estimator based on the GetNext() model in Section 6. We note a couple of additional properties of the GetNext() model of work: (1) It can be applied to modern database systems since they typically employ a demand driven iterator model for query execution. (2) It has the property that it is invariant across multiple runs of the same query.

3. DRIVER NODE ESTIMATOR: SINGLE PIPELINE QUERIES

In this section, we outline our solution for the progress estimation problem for the class of queries that consist of a *single* execution pipeline. We show how our solution extends to the general class of arbitrary query execution plans (consisting of multiple pipelines) in Section 4.

For simplicity, we consider a query whose execution plan is a single pipeline consisting of a chain of *m* (non-blocking) operators: Op_1 -> Op_2 …. -> Op_m and having a *single* operator Op_1 as its driver node (see Section 2.1 for definition of a driver node). Typically, such a pipeline consists of a single driver node (e.g., Table Scan or Index Scan) followed by a sequence of non-blocking operators such as Filter and Index Nested Loops (INL) join. As described earlier, they key challenge for any estimator using the GetNext() model (i.e., trying to estimate *gnm*) is to accurately estimate $\sum N_i$, the total number of GetNext() calls that will be performed over all nodes in the query. In an ideal world, the optimizer's estimates of N_i (and hence the progress estimator, which can use such estimates) would be accurate. But cardinality estimation usually involves simplifying assumptions (particularly on the correlation between data values) and consequently is prone to estimation errors. For example, it is known that estimation errors propagate exponentially as a function of the number of joins in the query [8]. Our focus in this paper is *not* on developing techniques for better cardinality estimation for the purpose of query optimization. Rather, we develop additional techniques that could mitigate the impact of errors in cardinality estimation on *progress estimation*.

Our estimator (called the Driver Node Estimator, *dne* for short) for single pipeline queries having exactly one driver node is defined as:

$$dne = \frac{K_1}{N_1}$$

where K_1 is the number of GetNext() calls done on the driver node of the pipeline, Op_1, thus far; and N_1 is the estimated total number of GetNext() calls for Op_1. Therefore, underlying *dne* is the hypothesis (we refer to it as the *driver node hypothesis*) that overall query progress can be estimated by the progress of only the *driver* node of the pipeline, i.e.,:

$$\frac{K_1}{N_1} \approx \frac{\sum_i K_i}{\sum_i N_i}$$

There are a few important reasons why the estimator *dne* can work well in practice. First, note that inaccuracies in *gnm* arise due to inaccurate N_i estimates. Since a driver node in a pipeline is the source of tuples that are operated upon by other nodes in the pipeline, prior to start of execution of that pipeline, the cardinality of the driver node is typically known accurately. For example, for many pipelines, driver nodes are typically Table Scans or Index Scans, and the estimates of N_i for such driver nodes can be obtained (almost exactly) from the database system catalogs. While the estimates may not be as accurate in the case of the driver node being an Index Seek operator, any histograms on the predicate columns can be leveraged. In such cases, the estimate of N_i for the driver node can still be quite accurate. On the other hand, accurately estimating N_i for a Filter node that references a UDF, or a Nested Loops Join node are usually more inaccurate due to the inherent difficulties in selectivity estimation and errors in propagation to intermediate nodes [8]. Thus, using only driver nodes for progress estimation can often result in better accuracy.

Second, when cardinality of the driver node N_1 dominates N_i's of other operators in the pipeline we can expect the estimator *dne* to be close to *gnm*. This is not uncommon in decision support queries such as TPC-H [12] where the driver node cardinalities are large (e.g. large Table/Index Scans), and where operators such as Filter and Group-By can greatly reduce the cardinality of non-driver nodes.

Third, observe that the driver node hypothesis implies:

$$\frac{\sum_i N_i}{N_1} \approx \frac{\sum_i K_i}{K_1} \approx \frac{\sum_i N_i - \sum_i K_i}{N_1 - K_1}$$

where $\sum_i N / N_1$ can be thought of as the work done per tuple output by the driver node. Therefore, when the total number of GetNext() calls made over all nodes in the pipeline does not vary significantly over the lifetime of the pipeline, monitoring progress of only driver node is sufficient. Although this condition does not hold for arbitrary pipelines, we show below an important class of pipelines (in which the output cardinality of each operator is no larger than its input cardinality) for which *dne* still yields progress estimates that are within a constant factor of *gnm*.

Finally, since a driver node is the source of tuples processed by other operators in the pipeline, it typically provides sufficiently fine granularity of progress estimation. This property may not hold in general for other operators (such as Filter or NL Join) in a pipeline, since their cardinalities may be arbitrarily small.

Guarantee of *dne* for monotonically decreasing pipelines:

We discuss an important class of single pipeline queries where *dne* is guaranteed to be accurate within a constant factor of *gnm* (the constant factor is the number of operators *m* in the pipeline). Consider pipelines having the logical property that no operator in the pipeline can increase its incoming cardinality. Thus, at any point during the query's execution, $K_i \geq K_{i+1}$ and $N_i \geq N_{i+1}$. We refer to such a pipeline as a *monotonically decreasing* pipeline. Some of common physical operators that could be part of a monotonically decreasing pipeline are Table Scan, Filter and

streaming aggregate operators. INL Join would also satisfy the above property when the join looks up a key value (i.e., a foreign key – key join).

Claim: For a monotonically decreasing pipeline with *m* operators, the estimator *dne* is guaranteed to be accurate within a constant factor *m* of the ideal estimator *gnm*, i.e.

$$\frac{gnm}{m} \leq dne \leq m \,.\, gnm$$

Proof: See **Appendix A**.

Note that for the case of a single pipeline consisting of a Table Scan, Filter and aggregation operator (similar to the class of queries studied in the online aggregation work e.g., [7]), if the input tuples to such a pipeline P are read in random order, then the driver node hypothesis will hold, i.e., the expected value of K_1/N_1 is $\sum_P K / \sum_P N$ for that pipeline. Finally, for the case of a single pipeline that is not monotonically decreasing, the above guarantee does not hold, and *dne* is a heuristic. Intuitively, if an intermediate operator (e.g., a non foreign-key Nested Loops Join) can increase its incoming cardinality arbitrarily, then the distribution of work done in the pipeline can be skewed so that progress at the driver node may not be indicative of overall progress of the pipeline.

4. SOLUTION FOR GENERAL CASE

In this section, we extend our solution for an arbitrary SQL query execution plan that consists of multiple pipelines. As described in Section 2.1, we model an arbitrary execution plan as a partial order of pipelines and extend the ideas of Section 3 of using only driver nodes for each currently executing pipeline. Therefore, in our approach, the key issues are: (1) explaining how to use the Driver Node Estimator (*dne*) for a single pipeline to obtain an overall progress estimate for entire query execution plan (Section 4.1), and (2) initializing and refining the cardinality estimates based on feedback from query execution (Section 4.2).

4.1 Estimator for Arbitrary Query Execution Plan

As per our definition of *gnm* (Section 2.3), for a query execution plan with *s pipelines*, our estimator for the entire query can be re-written equivalently as follows:

$$gnm = \frac{\sum_{P1} K + \ldots \sum_{Ps} K}{\sum_{P1} N + \ldots \sum_{Ps} N}$$

where each summation term denotes the sum over all nodes in the corresponding pipeline. As discussed previously, the key challenge for a pipeline P is estimating the $\sum_P N$ for that pipeline. We note that in a query execution plan that involves multiple pipelines, we know that each pipeline must be in one of the following states: (a) Completed. (b) Currently executing. (c) Not yet started executing. For any pipeline that has completed execution we have the exact values of the number of GetNext() calls done on all operators in that pipeline, and thus $\sum_P K = \sum_P N$ for such a pipeline. For a currently executing pipeline, we use *dne* to estimate $\sum_P N$. Specifically, it follows directly from the driver node hypothesis that $\sum_P N = \sum_P K$ / *dne*. For a pipeline that has not yet started executing ($\sum_P K = 0$), and we use the optimizer's estimates for $\sum_P N$. In fact, it is for this case where we expect a significant opportunity to improve estimate of $\sum_P N$ using feedback from query execution.

4.2 Exploiting Execution Feedback for Refining Estimates

A key challenge arises from estimating cardinality of nodes of pipelines that start with intermediate blocking nodes e.g., Sort nodes and hash based Group-By nodes. For nodes of such pipelines there is an opportunity to get better cardinality estimates by using feedback from query execution.

Consider the query execution plan shown in Figure 3. Suppose A is the build relation and B the probe relation of the Hash Join. The pipelines for the query are P_1 = {Table Scan A, Filter, Hash Join}, P_2 = {Table Scan B}, P_3 = {Group-By} and P_4 = {Sort}, which are executed in the order P_1, P_2, P_3, P_4. The driver nodes for the query are shaded in the figure. To estimate the cardinality of the Sort operator (in pipeline P_4), we would need to have accurate estimates on the filter, join and group-by operators of the first two pipelines. This is in fact the traditional cardinality estimation problem and is error prone. Hence, our initial estimate of work done by the Sort could be inaccurate, potentially leading to overall incorrect progress estimation. Here, execution feedback can be leveraged to improve estimate of the Sort node cardinality. For example, when pipeline P_2 completes, we in fact have the exact value of the cardinality of the result of the join. Similarly, when the Group-By, completes, we have exact cardinality (no uncertainty) of the input to the Sort. In the rest of this section, we describe a general framework for refining cardinality estimates of a given execution plan based on execution feedback.

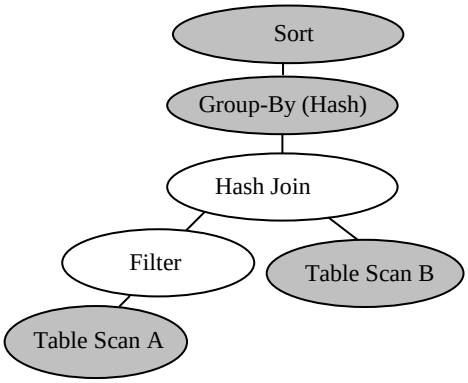

Figure 3. Execution plan with multiple blocking operators

While several techniques are possible, in this paper we follow a conservative approach that ensures that we never introduce any additional inaccuracies due to the refinement process. Thus, we refine the current N_i estimate of any node only if we are certain that the refinement will make the estimate more accurate. We achieve this as follows: For each node in the execution plan, we track two additional values UB_i and LB_i, which are respectively, the upper and lower bounds on the cardinalities of the rows that can be output from that node. These bounds are based solely on the algebraic properties of the operator and observed cardinalities from execution, and are guaranteed to be actual bounds on N_i. In particular, this means that $LB_i \leq N_i \leq UB_i$. We adjust these lower

and upper bounds as we get more information from query execution using the techniques described below. The invariant that we maintain at all times is that $LB_i \leq$ current estimate of $N_i \leq UB_i$, i.e., if we find that the current estimate of N_i lies outside the bounds, then we correct its value to the appropriate bound. The effectiveness of such refinement based on bounds depends on how much and how quickly these bounds can be refined based on execution feedback.

When an upper (or lower) bound for a particular node is refined, this could potentially help refine the upper (or lower) bound of other nodes *above* it in the execution tree. We propagate these bounds using algebraic properties of operators. For example, in Figure 3, suppose that at some point in time T during the query's execution, we were able to conclude that the upper bound for the Hash Join can be reduced from 1 million rows to 0.5 million rows. Suppose the upper bounds for the Group By and Sort nodes were 0.8 million rows. Then, based on the algebraic properties of Group-By and Sort nodes, we can also conclude that each of their upper bounds cannot exceed 0.5 million rows. The lowering of the upper bound could help refine the estimates of N_i at one or both of these nodes at time T. Note that although *dne* uses only the driver node cardinalities for the currently executing pipeline, it is necessary to refine cardinalities of all nodes in the pipeline, since it could influence the N_i estimates for nodes in a pipeline that is yet to start executing. In our implementation, we propagate bounds a few times per second (at roughly the granularity at which feedback is necessary to the user/application).

Refining lower and upper bounds

The refinement of lower and upper bounds for an operator Op_i at query execution time uses the following information: (1) The observed input and output cardinalities of the operator (i.e., the K_i of the operator as well as its input operators) (2) Algebraic properties of the operator. For example, for Filter and Group-By operators, we know that the cardinality cannot exceed its input cardinality. (3) The current state of the operator. This refers to the state of internal data structures used by the operator. For example, the current number of entries in the hash table of a Group-By operator.

For refining lower bounds, K_i (the actual number of rows output from the operator thus far) is itself a correct lower bound for any operator. An example of where the algebraic property of the operator is useful for refining lower bounds is Sort. Since Sort has the property that it does not change its input cardinality, in fact, K_{i-1} (i.e., cardinality of the input operator to the Sort) is a valid lower bound. Thus, the cardinality of the Sort operator (which is always the start of a new pipeline) can be refined when the previous pipeline is executing. An example of where the current state of the operator is useful in refining lower bounds, consider the Group-By (hash based) operator. If we can count the number of distinct values observed during the operator's execution thus far (say *d*), then the lower bound can be refined to *d* at that point in time. This could be done, for example, by tracking the internal hash table used by the operator.

As far as the upper bound is concerned, for operators such as Filter and NL Join (foreign-key join), we can leverage their algebraic properties (the fact that they can never increase their input cardinality) and the K_i's to refine the upper bound to: $(UB_{i-1} - K_{i-1}) + K_i$. Another example where algebraic properties help refine upper bound is Sort, where UB_{i-1} (i.e., upper bound of input to Sort) is an upper bound for the Sort itself. An example of the use of current operator state for refining upper bounds is the Hash Join operator. Consider a Hash Join between two relations A (build side) and B (probe side). Assume A has already been hashed into buckets, and suppose S is the number of tuples of the *largest* bucket. We can exploit this information during the probe phase to obtain a tighter upper bound since we know that each row from B can produce at most S tuples after the join. We refer the reader to **Appendix B** for details of how upper and lower bounds can be refined for certain common physical operators. In the future, we intend to explore applicability of other rules that can yield tighter bounds based on execution feedback.

We observe that whenever an operator terminates, we know *exactly* the upper and lower bounds of that operator (which are identical at that point). Thus, e.g., for the query plan in Figure 3, when the final pipeline (P_4) starts executing, we know exactly the cardinality of its driver node (the Sort node). In general, when a pipeline starts executing, we know exactly the cardinality of its driver nodes.

In our experiments on TPC-H queries, we have found that both lower and upper bounds help refine N_i's of certain driver nodes significantly (e.g., by three orders of magnitude for Q21) for driver nodes of upper level pipelines (when the optimizer underestimates the cardinality e.g., of a Sort node). Interestingly, the impact of these refinements on the overall estimation errors (see Section 5) is typically much smaller (a few percent). This is because in these queries, the N_i's of driver nodes such as Table/Index Scan dominate the N_i's of other nodes. A more thorough evaluation of the effectiveness of these bounding techniques on other data sets/queries is part of our ongoing work.

Finally, we note that other techniques to leverage information from query execution are possible. For example, online estimation techniques based on observing intermediate results as in [7], refining statistics based on observed query results [1,11], and re-invoking optimizer for cardinality estimated based on observed cardinalities similar to [4]. Exploiting these ideas to augment our techniques is an important area of future work.

5. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

In this section, we first describe the implementation of our solution for estimating progress of SQL queries inside Microsoft SQL Server. We follow this with the results of an experimental evaluation of our solution for long running decision support queries on both the TPC-H benchmark [12] as well as an internal customer database.

5.1 Implementation

Our implementation inside Microsoft SQL Server consists of the following simple extensions to the existing query execution engine. We augment the data structure corresponding to a node in the query execution plan with counters for K_i (number of rows output by the node thus far), N_i (current estimate of total number of rows that will be output by node at completion), UB_i and LB_i (upper and lower bounds respectively of number of rows that can be output by node). After the query is optimized and an execution plan tree P has been generated for the query, we identify pipelines

in P and the driver node(s) for each pipeline. We initialize N_i for each node to the optimizer estimated cardinality (for leaf-level nodes such as Table/Index Scan this is the cardinality of the base table/index). We update and propagate the values of UB_i, LB_i and N_i for nodes using the K_i and algebraic properties of operators as described in Section 4.2.

For convenience of collecting the progress information of an executing query, we implement a background thread that wakes up periodically (approximately 4 times a second), traverses P, computes the progress, and logs the progress estimate and a timestamp to a file. The overheads of gathering this information at runtime are negligible relative to execution time of queries we considered. In general, we would expect that database servers will extend interfaces (e.g., via system stored procedures or functions) to allow clients to programmatically access progress information for an executing query by polling the server.

5.2 Experiments

Goal: The goal of the experiments is to:

- Evaluate the accuracy of our estimator (which is based on the GetNext() model of work presented in Section 2) on a set of long running and complex decision support queries.
- Evaluate robustness of our estimator when data skew is varied.
- Validate the driver node hypothesis for progress estimation of currently executing pipelines.

Setup: We conducted the experiments on a machine with a 2.8GHz CPU and 512 MB RAM.

Databases: We ran the experiments on the TPC-H 10GB database [12]. We chose the 10GB configuration because the queries are truly long running (typically 10s of minutes). For the evaluation with varying skew, we generated a TPC-H 10GB database with a Zipfian skew factor of 2 using the publicly available tool [3]. We also ran queries from a real data warehouse application used within the company to analyze sales (we refer to this as the SALES database – approx. 5GB in size).

Queries: For TPC-H we evaluate all the queries defined in the benchmark. We report numbers for all the long running queries in the benchmark (those that reference the *lineitem* table). For TPC-H queries, the joins are typically foreign-key joins, and thus most pipelines exhibit the property of being monotonically decreasing (Section 3). For the SALES database, we picked a few queries for evaluation. The queries against the sales database are aggregation queries that are joins of 7-10 tables, and have 8-10 grouping columns. The joins are non foreign-key joins, thus the property of monotonically decreasing pipelines does not hold.

Evaluation Metric: Our experiments are conducted a single query at a time, and on a machine on which only the database server is executing. In this setting, we expect the percentage work completed reported by any scheme to be a good estimator of the percentage time taken by the query. As described in Section 5.1 above, we record the fraction complete predicted by our solution at regular intervals throughout query execution. Assume the query starts executing at time t_0. Let f_i be the percentage of the query completed as reported by our estimator at time t_i ($i > 0$, $t_i > t_{i-1}$). Let t_n be the time at which the query completes. Then, at any point in time t_i, an estimator that has perfect knowledge of the future would report the actual percentage of the query completed as $100 . (t_i-t_0)/(t_n-t_0)$. Thus, we define the *estimation error* of an estimator at time t_i (denoted by e_i) as: $e_i = \left| \frac{100 \cdot (t_i - t_0)}{(t_n - t_0)} - f_i \right|$

Note also that since we take the absolute value of the difference, we do not distinguish between under estimates or over estimates. We report the overall estimation error for a query using three aggregate measures over all the e_i's collected for the query, the *average*, *standard deviation* and *max* over all e_i's.

5.2.1 TPC-H Benchmark Queries

The goal of this experiment is to evaluate the accuracy of our progress estimator (see Section 4.1), which is based on the GetNext() model of work. We evaluate the estimator on complex decision support queries of the TPC-H benchmark [12] on the 10GB database.

Table 2 shows the mean and maximum error (as defined above) for several long running TPC-H queries for the uniform data distribution case (Z=0) as well as the skewed data distribution case (Z=2). As we see from the table, for the Z=0 case, the *maximum* error for any query does not exceed 10%, and the average error is small (typically below 5%). The standard deviation was also small (at or below 5% in all cases). One interesting observation is that expensive Sort nodes at the top of a query execution plan can potentially be problematic (as in Q5 for Z=0), particularly when the query optimizer overestimates the cardinality of the Sort node. In such cases it is difficult to rectify errors based on execution feedback until the lower pipeline (that feeds into the Sort node) is almost complete. Thus, the error induced by the optimizer's estimates persists for almost the entire duration of the query.

For the Z=2 case, the maximum and mean errors are higher for certain queries e.g., Q8, Q18, and Q21. To understand the reasons for the errors better, refer to Figures 4 and 5, which show scatter plots of the actual percentage completed vs. estimated percentage completed for Q8 for Z=0 and Z=2 respectively. A perfect estimator would have all data points along the diagonal of the graph. For the Z=2 case (Figure 5), when the major pipeline in this query (involving scan of the *lineitem* table followed by a Merge Join and couple of Hash Joins (probes)) starts, the estimates of the cardinalities of the joins used by our estimator are significantly overestimated. However, shortly after the pipeline starts executing, (as explained in Section 4.2) we estimate the cardinalities using *dne* which is based on the progress of the driver node (Scan of lineitem). This results in quickly reducing the estimation error, and explains the discontinuity in progress estimation around 20% actual completion. In general, until a pipeline starts executing, our estimator is more susceptible to errors in cardinality estimation. For the case of Z=0, the cardinality estimates of this pipeline are quite accurate, and therefore we see lower errors. We observe similar behavior in queries Q18 and Q21. This experiment shows our estimator (based on the GetNext() model of work) results in fairly robust progress estimation, even in the presence of skewed data distributions.

Table 2. Estimation Errors TPC-H Benchmark Queries (10 GB database), Uniform and Skewed Data Sets

	Estimation Error (Z=0)		Estimation Error (Z=2)	
Query	**Mean**	**Max**	**Mean**	**Max**
Q1	0.9%	2.8%	0.2%	0.5%
Q3	1.1%	2.0%	3.4%	4.7%
Q4	0.5%	1.0%	0.6%	1.4%
Q5	7.3%	9.0%	3.7%	5.4%
Q6	1.2%	2.9%	2.8%	4.6%
Q7	2.3%	4.0%	3.8%	7.6%
Q8	0.8%	1.7%	5.2%	16.2%
Q9	2.7%	4.9%	2.9%	8.3%
Q10	0.4%	1.4%	1.6%	4.4%
Q12	1.0%	1.7%	0.9%	3.8%
Q14	0.5%	1.8%	1.5%	3.2%
Q15	0.6%	1.3%	1.6%	4.4%
Q17	1.7%	2.6%	0.7%	2.0%
Q18	5.9%	16.8%	14.2%	25.5%
Q19	0.5%	1.5%	1.8%	2.7%
Q20	3.0%	9.8%	3.7%	5.9%
Q21	0.9%	2.5%	15.7%	38.8%

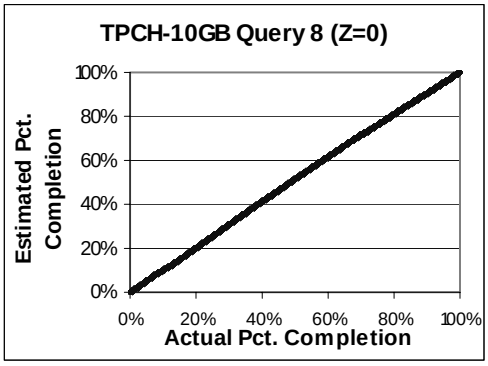

Figure 4. Scatter plot of actual vs. estimated percentage completed (TPC-H **Q8**), Uniform distribution

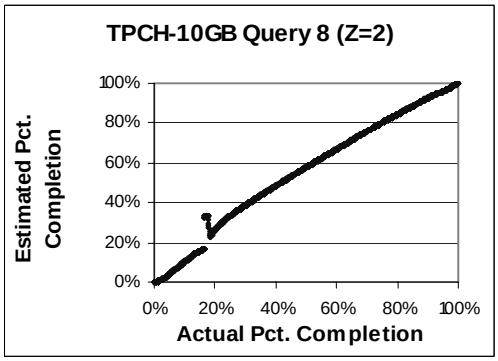

Figure 5. Scatter plot of actual vs. estimated percentage completed (TPC-H **Q8**), Skewed distribution

5.2.2 Validation of Driver Node Hypothesis

In this experiment, we demonstrate the importance of estimating overall progress based only on progress of driver nodes within a currently executing pipeline. We do this by comparing with an estimator that also uses the GetNext() model, but does not use *dne* (i.e., the driver node hypothesis) to estimate the cardinality of all nodes in the currently executing pipeline, but relies only on the optimizer estimated cardinalities.

We show the results for TPC-H query Q9 against the 10 GB database with Zipfian skewed data (Z=2). The results for our estimator and the estimator that uses only optimizer estimates (OPT) for currently executing pipelines is shown in Figures 6 and 7 respectively. The mean and max errors for our estimator is 2.9% and 8.3% respectively, whereas the errors for OPT are 23% and 47% respectively. The reason is that OPT, due to the inclusion of several join nodes (whose cardinality estimates are inaccurate), ends up with a significant overestimate of the actual work which gets refined only near the very end of query execution (when the estimated completion jumps from 48% to 89%).

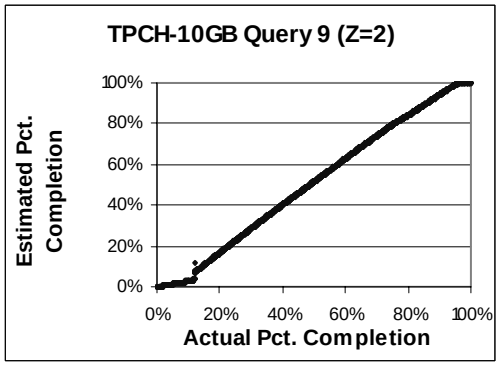

Figure 6. Scatter plot of actual vs. estimated percentage completed (TPC-H **Q9, Using Driver node hypothesis**)

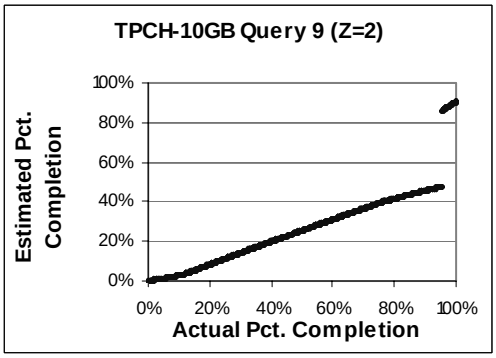

Figure 7. Scatter plot of actual vs. estimated percentage completed (TPC-H **Q9, Using only optimizer estimates**)

5.2.3 Queries on SALES Database

In this experiment, we evaluate our estimator on complex decision support style queries from a real database application. As in TPC-H, we see that the mean estimation errors are quite low (around 10%) and the max errors are around 20% (see Table 3). This

experiment shows that that the accuracy of our estimator does not degrade appreciably for this set of real world queries that contain non foreign-key joins, and significant grouping and aggregation.

Table 3. Estimation Errors SALES queries

	Estimation Error	
Query	**Mean**	**Max**
Q1	7.1%	17.3%
Q2	8.2%	16.9%
Q3	11.6%	18.2%
Q4	9.3%	21.4%
Q5	7.0%	18.1%

6. MONOTONICITY

As discussed in Section 2.2., *monotonicity* is a desirable property from a user's perspective. Consider a progress estimator that uses the GetNext() model of work (Section 2.3). Since the K_i values (observed cardinalities during execution) are monotonically increasing, the estimator will be monotonic provided any changes to the N_i values during query execution are monotonically decreasing. An estimator that has up front knowledge of the *exact* number of GetNext() calls that will be made by each operator (i.e., the N_i values) can guarantee monotonicity, since it would never need to change N_i. However, for any other technique that can only *estimate* the value of the N_i there is a trade-off between guaranteeing monotonicity and the *accuracy* of progress estimation.

One way to ensure monotonicity is to initially use a value for the estimated N_i that is much larger than the actual N_i, i.e., an *upper-bound*. The problem with such an approach is that accuracy can suffer, since the actual N_i may be much smaller. For example, consider a query plan which performs a hash join of relations R_1 and R_2 and then sorts the result of the join. Note that obtaining a tight upper-bound on the estimate of the Sort node cardinality can be problematic. If the join is a foreign-key join, then we know that an upper bound on the cardinality of the joined relation, and hence the Sort node, is the size of table with the foreign-key. However, for non foreign-key joins the upper-bound can be a considerable overestimate of the actual N_i for the Sort node, and thus the accuracy of the estimator may be poor until most of the query has completed executing. Therefore, the real challenge is to find *tight* upper-bounds so that accuracy of the estimator is not significantly compromised.

Given the difficulty of guaranteeing a tight upper bound for intermediate driver nodes, a trade-off between monotonicity and the accuracy of progress estimation appears unavoidable. Thus, an interesting issue is whether users prefer more accurate estimates or estimates that are guaranteed to be monotonic. A possible approach for addressing this issue is to present both the estimated progress as well as the progress based on the upper-bounds. Let the progress computed using upper bounds be p_1% and the corresponding one computed using estimates be p_2%. Then (p_1, p_2) as a pair of values would indicate to the user that the % done at any instant is not lower than p_1 and our current best estimate is the value p_2. Note that p_1 is monotonic, whereas p_2 may not be.

We observe that for a single pipeline query, the estimator *dne* (Section 3) is monotonic, since N_1 is known exactly and does not change during the execution of the pipeline (see Section 7 for runtime conditions that may cause monotonicity violations even for single pipeline queries). However, for the case of multi-pipeline queries, our estimator is not guaranteed to be monotonic. In particular, monotonicity violations can occur when a new pipeline starts executing, and we revise the optimizer estimates of N_i, with the estimate based on *dne* (as described in Section 4.2). For the queries against the TPC-H 10GB (Skew Z=2) data in our experiments, we computed progress estimates at regular intervals (approximately 4 times a second), and we measured: (a) the number of monotonicity violations, i.e., number of times in which a progress estimate was less than the previous estimate, (b) the average % by which the estimate decreased and (c) the maximum % by which the estimate decreased. We observed monotonicity violations in five queries (Q7, Q8, Q9, Q20, Q21). Moreover, except for Q8 and Q20, there was only 1 violation in the other three queries. The maximum decrease in estimated progress across all queries was 8.3% (for Q21) and the average decrease for each query respectively was 1.4%, 0.7%, 4.9%, 0.01%, and 8.3%. One reason for the relatively few and small monotonicity violations is that in these queries the N_i's are dominated by the leaf-level driver nodes (scans of *lineitem*, *orders* tables). Due to the filtering and aggregations performed in these queries, the *actual* N_i's of the upper-level nodes in the plan are usually much smaller. Thus, even in cases when the N_i's for the non-leaf driver nodes are initially under-estimated, the magnitude of the monotonicity violations are small.

7. RUNTIME CONDITIONS

The model of work done by a query (see Section 2) makes the simplifying assumption that the actual work done by a call to GetNext() is the same across all operators in the plan, i.e., GetNext() from all operators are weighted equally. In general, this assumption does not hold, e.g., due to an expensive operator like a UDF in a Filter node, or because one Table Scan reads from a fast disk whereas another Table Scan reads from a slow disk. A possible way to extend the basic model of work to account for different cost of GetNext() of different operators is to model the work as a weighted sum of the number of GetNext() calls done by operators in the plan. The weighting factor C_j associated with operator *j* is a relative measure of the work done by a GetNext() on that operator. Of course, this introduces an additional parameter (besides driver node cardinality) that needs to be estimated and refined. A possible solution is to start with uniform relative rates (i.e. $C_j = 1$ for all j) or use cost estimates made by the query optimizer, and then adjust the C_j values based on execution feedback. Modeling and computing per-tuple work for every pipeline could be an important factor in general, and developing techniques to address this issue is part of our ongoing work. In the rest of this section we discuss an important special case of a runtime condition, spills of tuples to disk due to insufficient memory, and show how our estimator can be adapted to handle spills without introducing an additional weighting factor, by treating spill processing as a "runtime pipeline".

Handling Spills: Spills of tuples to disk, which can occur as a result of insufficient memory can result in more work that is not accounted for by our model of work since it occurs within an operator. Consider a join between two relations A and B, where the optimizer picks a hybrid hash join operator. Hybrid hash proceeds by building a hash table of A in memory. During the scan of

relation A, if the memory budget of the hash join is exhausted, then certain buckets will be spilled to disk. When the table B is used to probe the hash partitions, the tuples of B that hash to the buckets that are not memory resident are also written to disk. Bucket spilling is a runtime effect and hence it can be difficult to accurately estimate in advance the number of tuples that will be spilled to disk.

We observe that we can model the query execution as comprising two parts, one that processes the original relations and another that processes the spilled partitions. In other words we can think of the original query as follows. Q = (A join B) ∪ (A' join B') where A' and B' denote the corresponding parts of relations A and B that have been spilled ($0 \le |A'| \le |A|$, $0 \le |B'| \le |B|$). The driver nodes for query Q would include scans of A, B, A' and B'. Thus the total work for Q would be |A|+|B|+|A'|+|B'|. The main problem is that |A'|, |B'| cannot be predicted at optimization time.

The main idea behind our solution to the spill problem is as follows. Whenever a tuple is spilled to disk (either from relation A or B) the denominator value (which denotes the total work) is incremented by one (i.e., another GetNext() call). We are in essence adding more work to be done later and the denominator value should reflect the estimated cardinality of the pipeline. Now, consider the point during execution when the first phase of hash processing is over and none of the spilled partitions have been processed. The modified estimator would have incremented the denominator counter for each tuple that had been spilled and would estimate the progress as (|A|+|B|)/ (|A|+|B|+|A'|+|B'|) which is correct as it accounts for the remaining tuples to be processed. When the spilled partitions are re-read the corresponding counts would be counted in the numerator and only when all the partitions have been processed will the estimator report the progress as 100%. This correction to the estimator works because of the symmetry of spills, i.e., exactly the tuples that have been written to disk will be processed later. It is also easy to see that this modification to the original algorithm would work for multiple recursion levels in a hash join pipeline. Finally, we note that spills could occur in other operators like hash-based Group-By or the merge phase in a Sort-Merge join if there are too many duplicates of a particular value. Thus, in general, a query can be considered as Q ∪ Q' where Q' accounts for the work done by the current query in handling data that is spilled.

The following experiment on TPC-H Q18 highlights in the importance of handling spills. From Figure 8 we see that the progress estimator remains stuck on 44% for a relatively long time (more than 15% of total query execution time). This is because during this interval the query is writing and reading the spilled partitions, and the estimator does not capture this effect. On the other hand when we enable spill handling as discussed above (see Figure 9) the estimator is more accurate.

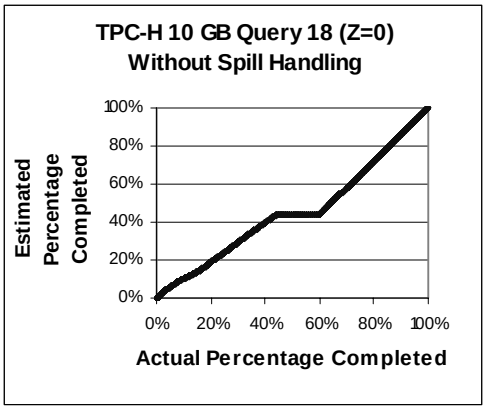

Figure 8. Scatter plot of actual vs. estimated percentage completed (TPC-H **Q18, No Spill handling**)

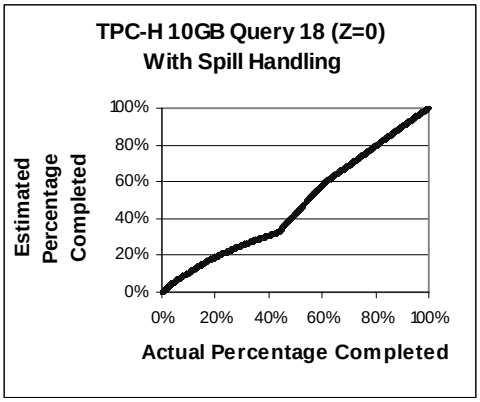

Figure 9. Scatter plot of actual vs. estimated percentage completed (TPC-H **Q18, With Spill handling**)

8. RELATED WORK

There are two broad areas that are related to our work. First is the area of estimating cardinality of query expressions. Selectivity estimation e.g., [9] plays a key role in enabling query optimizers to pick a suitable query execution plan. Our work leverages the query optimizer to provide an initial estimate of cardinality of nodes in an execution plan.

The second broad area that relates to this paper is the use of information gathered during query execution. One body of work e.g., [2,4] uses feedback of observed cardinalities at runtime to potentially re-optimize the same query, pick among competing query plans or to improve decisions on resource allocation for it. In contrast, we use observed cardinality of operators in the execution tree to improve estimate of total work that needs to be done, while leaving the query execution plan unchanged. We note that in principle, the techniques in [4] that collect statistics such as cardinalities/histograms etc. of intermediate query results can be adapted in our context for obtaining better estimates of N_i's by re-invoking the query optimizer's cardinality estimation module at runtime with more accurate statistics. While this does require non-trivial extensions to today's query processing engines, it represents an interesting avenue of future work for progress estimation. Another use of runtime feedback is to refine statistics e.g., [1,11] that can be used for selectivity estimation for

subsequent queries. In contrast, we limit the use of observed cardinality estimates to improving estimates for the same query.

A possible direction for using feedback from query execution is applying techniques for online aggregation [6,7] for estimating how many tuples would be output by a pipeline. Online aggregation techniques typically need to make assumptions about randomness of order of input tuples in order to give confidence intervals, and they exploit non-traditional join algorithms, e.g., ripple join. In contrast, we use a complementary (and more conservative) approach of refining lower and upper bounds based on observed cardinalities at runtime, and our runtime refinement is designed to work with traditional operators in today's database systems.

The paper [10] describes the importance of progress bars for computer-human interfaces of long running programs. The paper also discusses an inherent problem in estimating progress in the following case: The input of one program is output of another; as can happen e.g., when using the Unix *pipe* command. The paper suggests ensuring that all programs in the pipeline consume input at the same rate, and then basing progress estimation only on the original data producer. The concept of driver nodes in an execution pipeline can be viewed as an application of the above idea in the context of SQL queries.

9. CONCLUSION and FUTURE WORK

Long running queries are common in decision support environments, but current systems provide inadequate feedback to the user about progress of such queries. As a first step, we look at the problem of providing a progress estimator for SQL queries. The proposed solution is simple and applicable to arbitrary SQL queries. Initial experiments on a prototype in Microsoft SQL Server are encouraging. We intend to extend our current framework to be more robust to runtime conditions.

Although the work in this paper focuses on returning a single percentage completion number for the entire query, the estimator has more granular information, e.g., progress information at a per operator level. It would be interesting to consider leveraging this more granular information in providing feedback to users.

As mentioned in the introduction, perhaps the most useful feedback on query execution progress is estimating the time remaining to completion in a system with concurrently executing queries. Our paper does not achieve this. Our hypothesis is that providing percentage completion for query execution is a partial step in this direction that is still useful. It would be important to validate this hypothesis based on actual user feedback.

10. ACKNOWLEDGMENTS

We are very grateful to Raghav Kaushik for his many thoughtful and insightful comments that have greatly influenced this paper. As a matter of fact, he has read the paper more times than the authors! We also thank the anonymous referees for their valuable feedback.

11. REFERENCES

[1] Aboulnaga, A., and Chaudhuri, S. Self-Tuning Histograms: Building Histograms Without Looking at Data. *Proceedings of ACM SIGMOD 1999*.

[2] Antoshenkov, G. Dynamic Query Optimization in Rdb/VMS. *Proceedings of IEEE ICDE 1993*.

[3] Chaudhuri, S. and Narasayya V. Program for TPC-D Data generation with skew. *ftp://ftp.resesearch.microsoft.com/users/viveknar/tpcdskew*

[4] Dewitt, D., and Kabra, N. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. *Proceedings of ACM SIGMOD 1998*.

[5] Graefe, G. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv. 25(2): 73-170 (1993)*.

[6] Haas, P.J., and Hellerstein J.M. Ripple Joins for Online Aggregation. *Proceedings of ACM SIGMOD, 1995*.

[7] Hellerstein, J.M, Haas, P.J., and Wang, H.J. Online Aggregation. *Proceedings of ACM SIGMOD 1997*.

[8] Ioannidis, Y., and Christodoulakis, S. On the propagation of errors in the size of join results. *ACM SIGMOD '91*.

[9] Ioannidis, Y., and Poosala, V. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. *Proceedings of ACM SIGMOD 1995*.

[10] Myers, B.A. The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces. *Proceedings of ACM SIGCHI 1985*.

[11] Stillger, M., Lohman, G., Markl, V, and Kandil, M. LEO: DB2's Learning Optimizer. *Proceedings of VLDB 2001*.

[12] TPC Benchmark H. Decision Support. http://www.tpc.org.

APPENDIX A

Claim: Consider a monotonically decreasing pipeline (i.e., having the logical property that no operator in the pipeline can increase its incoming cardinality). For a monotonically decreasing pipeline with *m* operators, the estimator *dne* is guaranteed to be accurate within a constant factor *m* of the ideal estimator *gnm*, i.e.

$$\frac{gnm}{m} \leq dne \leq m \,.\, gnm$$

Proof: We present two estimators E_1, E_2 that provides estimates that are within a constant factor *m* (number of operators in the pipeline) of *gnm* for the above class of queries. Intuitively, E_1 is a pessimistic estimator which always underestimates the progress whereas E_2 is an optimistic estimator that always overestimates the progress. We then show that *dne* must lie between E_1 and E_2.

Estimator E_1: $$E_1 = \frac{\sum_i K_i}{N_1 \cdot m}$$

Where N_1 is the number of tuples that flow out of operator Op_1 (the driver node of the pipeline), and *m* is the number of operators in the pipeline. E_1 assumes that none of the N_1 tuples flowing out of Op_1 would be discarded by the other operators (Op_2 ..Op_m).

Claim: For monotonically decreasing pipelines E_1 is a lower bound on *gnm* within a factor *m*.

Proof: $E_1 / gnm = \sum_i N_i / (N_1 * m)$. Since, the pipeline is guaranteed to be monotonically decreasing we know that $N_1 \leq \sum_i N_i \leq m . N_1$. In other words:

$1/m \leq E_1/gnm \leq 1$ or equivalently:

$$gnm.\frac{1}{m} \le E_1 \le gnm \quad \text{.....(1)}$$

Estimator E_2:

$$E_2 = \frac{\sum_{i=1}^{m} K_i}{N_1 + \sum_{i=2}^{m} K_i}$$

Thus, E_2 assumes that all of the N_1 tuples that flow out of Op_1 are discarded by subsequent operators.

Claim: For monotonically decreasing pipelines E_2 is an upper bound on *gnm* within a factor *m*.

Proof:

$$\frac{E_2}{gnm} = \frac{\sum_i N_i}{N_1 + \sum_{i=2}^{k} K_i}$$

We know that:

$$\left(N_1 + \sum_{i=2}^{m} K_i\right) \le \sum_i N_i \le m \cdot \left(N_1 + \sum_{i=2}^{m} K_i\right)$$

in other words we have $1 \le E_2/gnm \le m$, or equivalently,

$$gnm \le E_2 \le m \cdot gnm \quad \text{.....(2)}$$

Claim: Estimator *dne* always lies between estimators E_1 and E_2.

Proof: Let us compare *dne* to estimators E_1 and E_2 defined above.

$(dne - E_1) = \frac{K_1}{N_1} - \frac{\sum_i Ki}{m \cdot N_1}$. Cross multiplying,

$$(dne - E_1) = \frac{N_1.m.K_1 - N_1.(K_1 + K_2 + ...K_m)}{N_1.m.N_1}$$

Since the pipeline is monotonically decreasing, we have $m.K_1 \ge (K_1+K_2+\ldots K_m)$. Therefore, $(dne - E_1) \ge 0$ or $dne \ge E_1$.

From the definition of E_2, we know that:

$$E_2 = \frac{K_1 + \sum_{i=2}^{m} K_i}{N_1 + \sum_{i=2}^{m} K_i}$$

and since $\sum_{i=2}^{m} K_i \ge 0$, we know that $E_2 \ge dne$.

Thus, $E_1 \le dne \le E_2$. Using Equations (1) and (2) above:

$$\frac{gnm}{m} \le E_1 \le dne \le E_2 \le m.gnm$$

APPENDIX B

Operator specific refinements based on execution feedback to upper and lower bounds of a physical operator Op_i in an execution plan. K_i is the actual number of rows output from the operator thus far. UB_i (resp. LB_i,) is the upper (resp. lower) bound on the cardinalities of the rows that can be output from Op_i.

Table 1. Operator specific rules for refining upper and lower bounds on cardinality

Physical operator i	**Lower Bound (LB_i)**	**Upper Bound (UB_i)**
Filter	K_i	$(UB_{i-1} - K_{i-1}) + K_i$
Group By	*d* (#distinct values observed thus far)	$(UB_{i-1} - K_{i-1}) + d$
Sort	K_{i-1}	UB_{i-1}
NL Join (Foreign Key)	K_i	$(UB_{i-1} - K_{i-1}) + K_i$ i-1 refers to 'Outer relation
NL Join Not FK	K_i	$(UB_{i-1} - K_{i-1})$. $UB_{i-2} + K_i$ i-2 refers to Inner relation
Hash Join Not FK	K_i	$(UB_{i-1} - K_{i-1})$. S S is #rows of largest build partition
Table / Index Scan (table T)	|T| (#rows in table)	|T|