

# A Lightweight Online Framework For Query Progress Indicators

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

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

## Abstract

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

## 1 Introduction

Recently there has been increasing interest in the development of progress indicators for SQL queries [13, 12, 8, 5]. A progress indicator aims to estimate precisely the value of a function that is related to the progress towards completion of a running query. Availability of such indicators can be of great help both to database administrators and end users. Given the complexity of queries in decision support applications, it is common for queries to take hours or days to terminate. Such indicators can greatly aid a user’s understanding of the progress of a query towards completion and allow the user to plan accordingly (e.g., terminate the query and/or change the query parameters). Similarly, from the point of view of administrators, unsatisfactory progress of queries may point to bad plans, poor tuning or inadequate access paths.

There are several important parameters associated with this problem. To begin with, one needs to specify the desired measure of query progress. Such a measure defines

a model under which a progress indicator *measures* the amount of work done by the query, and *predicts* the total amount of work that will be done by it. In addition to providing accurate estimates of work, a progress indicator also needs to impose a low overhead on the execution of the query. Support for such functionality in current systems is fairly limited [1].

In this paper, we develop a lightweight online estimation framework that significantly benefits query progress indicators. Our framework can aid *any* measure of progress that relies on estimating intermediate cardinalities of operators in the query plan. By monitoring query execution at select points in the plan and collecting statistics from the tuples we’ve seen, we are able to make accurate cardinality estimates at nodes in the plan during query execution. Our contributions include online estimators for joins, join pipelines and aggregation (group-by) operations. Our estimates become increasingly more accurate, converging to the true values as the query executes and we are, under general distributional assumptions, able to analytically quantify their accuracy. We experimentally demonstrate that our framework, when implemented in a real database management system, imposes minimal overhead on query execution time, while enabling highly accurate estimation of query progress.

## 2 Related Work

Progress indicators have been studied in various contexts (e.g., HCI [17] and file downloads) but there exists limited work on this topic in a data management context.

Recent work on query progress indicators [13, 12, 8, 5] introduced several novel ideas. Chaudhuri et al., [8] introduced the idea of decomposing a query plan into a number of segments (pipelines) delimited by blocking operators. Focusing inside each segment, they use the total number of *getnext()* calls made over all operators during the run of the query as an indicator of query progress. Subsequent work [5] extended this approach proposing estimators that are worst case optimal, and showed that it is not possible to provide non trivial guarantees on progress estimation in

the general case. An independent but related approach was presented by Luo et al., [13]. This approach also makes use of segments for query partitioning. The model of work adopted is bytes processed at the input and output of segments. In subsequent work [12], they increase the coverage of progress indicators to a broader class of SQL queries, and introduce refinements that capture the work done within operators in a more fine-grained manner. This work was also extended to handling multiple queries [14]. Although we phrase our presentation in terms of the *getnext()* model, our techniques are equally applicable to this model as well.

Our work is also related to techniques for online and adaptive query processing which try to do online processing of statistics at runtime. This includes work on online aggregation [9], which introduced techniques to progressively approximate group-by results at runtime, and adaptive query processing techniques [11, 15, 2] which make use of statistics collection at various points in the query plan for reoptimization purposes. Our proposed estimation framework is also similar to sampling based techniques for join result estimation [6] and estimating number of distinct values [10, 4]. However, these techniques are not designed to operate at runtime during query execution.

### 3 Model and Assumptions

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

In accordance to previous work [13, 12, 5, 8] we assume query plans consisting of a tree of physical operators. The set of operators we consider are **scan**,  $\sigma$ ,  $\pi$ ,  $\bowtie$  **NL**,  $\bowtie$  **INL**,  $\bowtie$  **hash**,  $\bowtie$  **merge**, **sort**,  $\gamma$  (**group by**). A query plan consists of one or more *pipelines*. Pipelines are defined as maximal subtrees of concurrently executing operators. We refer to [8, 13] for a discussion of the various pipelines related to each blocking operator. In this paper, we adopt the *getnext* model (*gnm*) of progress introduced in [8]. Our technique is also applicable to the notion of query progress introduced by Luo et al. [13], which is similar to *gnm*.

Suppose that the execution of a query  $Q$  involves  $m$  operators. If for all operators  $O_i$ , we denote the total number of *getnext()* calls over the execution of the query as  $N_i$ , and the number of *getnext()* calls made thus far as  $K_i$ , then the *gnm* measure of progress is  $\sum_i^m K_i / \sum_i^m N_i$ . We denote the current number of *getnext()* calls made as  $C(Q) = \sum_i^m K_i$ , and the total number of *getnext()* calls over the lifetime of the query as  $T(Q) = \sum_i^m N_i$ .  $C(Q)$  can be computed easily, by observing tuples at operators. The primary challenge in progress estimation is to estimate  $T(Q)$  at runtime at a low overhead.

We make the following assumptions in our framework. We assume knowledge of the size of base tables, which is

usually available in the system catalogs. In order to provide confidence guarantees on our estimates, we require that table scans on base relations obtain on demand a (or have access to a precomputed) random sample of a specific size from disk. For ease of exposition, we assume randomly ordered tuple streams in the description that follows.

## 4 Our Approach

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

### 4.1 Joins

Due to space constraints, we do not present confidence guarantees behind our join estimation framework. Complete details may be found in the full version of this paper [16]. We derive accurate estimates of join result sizes for join pipelines based on samples. We discuss below the types of estimation in our framework.

#### 4.1.1 Binary Hash Joins

Consider a hash join between 2 relations  $R$  and  $S$  with  $|R|$  and  $|S|$  tuples respectively. Let  $R$  be the build input, and  $S$  be the probe input of the join. Assume we compute an equijoin on attributes  $A$  and  $B$  of  $R$  and  $S$ . Suppose that while partitioning  $R$ , we count the number of times each value in the join attribute is seen. This operation can be interleaved with the actual partitioning to keep overheads low (we provide implementation details in Section 5). In other words, we build a histogram that maintains a count  $N_i^R$  for each value  $i$  in  $R$ . Once the build input has been partitioned, the hash join reads in the probe input. Let  $D_t$  denote our size estimate for the join when  $t$  tuples of  $S$  have been read. If the  $(t + 1)$ th tuple has the value  $i$  on its join attribute, we can set:

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

This estimator has zero error in expectation and we can derive confidence bounds that shrink with increasing number of tuples read. Notice that this formula does not require us to build a histogram on the probe input. For each tuple of  $S$ , we probe the histogram built on  $R$  to get  $N_i^R$  and update our estimate as per the above formula. The computational overhead of this operation is negligible; the only overhead is storage for the histogram on  $R$ .

Another important advantage of using this estimator is that it converges to the exact cardinality of the join by the end of the *first pass* on the probe input. The hybrid hash join and the grace hash join algorithms partition the probe input on the first pass into buckets, and leave most (or all in the case of grace hash join) join processing to the second pass. In contrast the driver node estimator (*dne*) of [8] and the estimator of Luo et al., [13] do not guarantee convergence until the entire join has been processed.

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

### 4.1.2 Other Join Algorithms

**Sort-Merge Joins:** An estimator similar to the one proposed for hash joins can be used for sort-merge joins on unsorted inputs as well. The idea here is to build a histogram on the input that is sorted first, and then probe it while the other input is sorted. For further details please refer to the extended version [16]. If the relations are already sorted, it is not possible to push down estimation to a preprocessing phase. In such cases our techniques default to the usual *dne* estimate.

**Nested-Loops Joins:** In practice, nested-loops joins without indices are often optimized by building temporary indices on the inner input (in the absence of a permanent index), and by pre-sorting the outer input to improve locality in memory access. In the presence of such preprocessing phases, we can construct estimators similar to the incremental estimator for hash joins, and get early estimates.

If however, there is an index on the inner, or the above optimization is not used, the join does not involve a preprocessing phase. In such cases, we cannot push down the estimation to the preprocessing phase, and we default to the *dne* framework.

### 4.1.3 Join Pipelines

We now consider the case of multiple join operators on a pipeline. A common construction is a pipeline of hash joins, each taking the output of a lower hash join as its probe input. For ease of exposition we present our handling of both cases for pipelines of two hash joins (illustrated in Figure 1) (a) and (b), before we present the general case. Consider the join tree shown in Figures 1 (a) and 1(b). In one case, the probe column of the upper join comes from the probe input of the lower join ( $A.y = C.y$ ), while in the other case it comes from the build input of the lower join ( $A.y = B.y$ ). We now describe these cases in detail.

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

$$D_{t+1} = \frac{D_t t + N_y^A N_x^B |C|}{t + 1}$$

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

It is easy to see that if the upper and lower joins are on the same attribute (i.e  $y$  and  $x$  are the same), we can handle it in an identical fashion to Case 1 as described above. Such *push-down* estimation generalizes to any number of hash joins. The two cases described above provide the basic intuition on how to handle a pipeline containing a chain of multiple hash joins. Similar techniques can also be used for pipelines of other join algorithms. We do not formally describe the complete algorithm here due to space constraints. The reader is referred to the extended version [16] for further details.

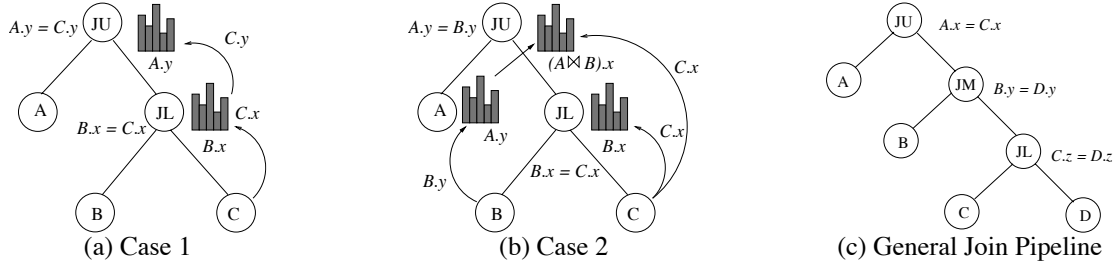


Figure 1. Join Pipelines

## 4.2 Aggregation

In database systems, aggregation is typically implemented by sorting or hashing. In both cases we can get perfect cardinality estimates of the output by counting the number of groups seen in the hashing/sorting phase. However, we would like to be able to estimate this cardinality even before the entire input has been seen. This is the distinct value estimation problem and it has been well studied in the database and statistics literature (see [4, 10] and references therein). Due to space constraints we only provide a brief overview of our solution. Further details may be found in the extended version [16]. We propose a new hybrid estimator that extends the Guaranteed Error Estimator introduced by Charikar et al. [4] with a Maximum Likelihood Estimation Framework [3]. Since we conduct distinct value estimation in an online fashion, our hybrid estimator is adaptive to the data distribution. We show that it incrementally converges to the correct number of groups, and can handle skew in the grouping column.

## 4.3 Selections

As in the case of nested-loops joins, selection operators do not have any preprocessing phases in which the entire input is seen before the actual selection takes place. Selection conditions are pushed as close to the leaves of the query plan as possible, and so it is not possible to further push down estimation. Since, by taking a sample, we ensure that part of the input is seen in random order, the *dne* estimator has zero error in expectation [5], and so we use the *dne* estimator to handle these cases.

## 4.4 Progress Estimation

We now have a set of techniques which allow us to refine cardinality estimates of operators in a query plan. Our techniques hinge on the fact that operators for joins and aggregations often have preprocessing phases where they see the entire input and partition it either by sorting or hashing. We revert to the *dne* estimate for operators that do not have

any preprocessing phase, such as certain forms of nested-loop joins and selections.

For a pipeline  $p$ , let  $C(p)$  be the number of *getNext()* calls made over all operators in it i.e.  $C(p) = \sum_{i \in p} K_i$ . Similarly let  $T(p)$  be our estimate of the number of *getNext()* calls that will be made over the run of the pipeline i.e.  $T(p) = \sum_{i \in p} N_i$ . Of these pipelines, some would have already finished executing. For these, we know  $C(p)$  and  $T(p)$  precisely. For pipelines, that are currently executing, we know  $C(p)$  precisely, while we estimate  $T(p)$  using our estimation techniques. For pipelines that are yet to begin,  $C(p) = 0$ , and we refine the optimizer estimates for  $T(p)$  using upper and lower bounds as in [8]. Summing these values over all pipelines gives us  $C(Q)$  and  $T(Q)$ , and we estimate the progress of the query as  $C(Q)/T(Q)$ .

## 5 Evaluation

In this section, we evaluate our estimation framework. We implemented our estimation framework in the PostgreSQL 8.0 database engine. We performed all experiments on a machine running Fedora Core 3 with a 2.80 GHz processor and 1 GB RAM. We utilized the TPC-H schema as our template and used the publicly available tool [7] to generate skewed data. We modified this tool in order to be able to vary the number of distinct values in a table column. Due to space constraints, we present just a few experiments on the effectiveness of our framework. A more detailed experimental evaluation can be found in the extended version of our paper [16].

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

Let the *ratio error* ( $R$ ) of an estimator be the ratio of the estimated cardinality of the output of the operator to the final cardinality of the output. We present results for the ratio error on small domain sizes (5K elements) in Figure 2(a), and large domain sizes (125K elements) in Figure 2(b). In

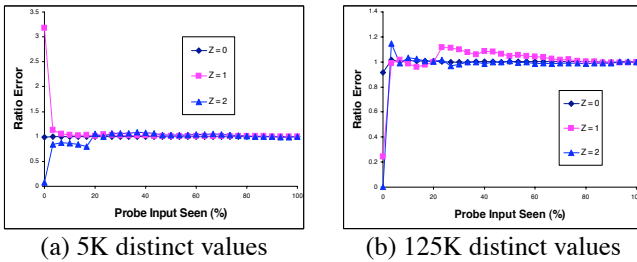


Figure 2. Binary hash joins: Varying Skew

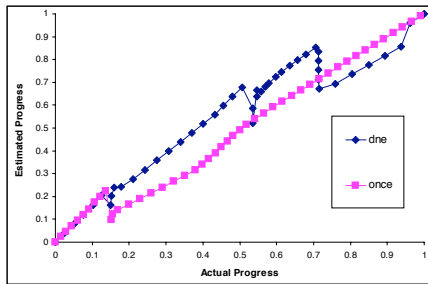


Figure 3. Comparison of progress estimators

these graphs, the value of  $Z$  corresponds to the skew parameter of the Zipfian distribution. As is evident in the figures, our estimators converge to an approximately correct ratio error estimate while having seen only a fraction of the probe input.

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

## 6 Conclusions

In this paper we have proposed an online framework for progress indication. We have proposed estimators to track

and progressively refine the cardinality estimates for various pipelines in a way that it is easily integrated to models of progress indication. This work raises various avenues for future work in this area. In particular, we wish to explore accuracy-overhead tradeoffs in constructing such estimators. In addition, we intend to explore problems in adaptive query processing that can benefit from query progress information

## References

- [1] DB2 Monitoring Facility. *DB2 Online Documentation*; [www.software.ibm.com](http://www.software.ibm.com), 2000.
- [2] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-Optimization. *SIGMOD*, 2005.
- [3] S. Boneh, A. Boneh, and R. Caron. On Estimating The Prediction Function and the Number of Unseen Species in Sampling With Replacement. *Journal Of the American Statistical Association*, 93(441):372, 1998.
- [4] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards Estimation Error Guarantees for Distinct Values. *SIGMOD*, 2000.
- [5] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When Can We Trust Progress Estimators for SQL Queries. *SIGMOD*, 2005.
- [6] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling Over Joins. *SIGMOD*, 1999.
- [7] S. Chaudhuri and V. Narasayya. Program for TPC-D Data generation with skew. [ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew](http://ftp.research.microsoft.com/users/viveknar/tpcdskew).
- [8] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating Progress of Execution for SQL Queries. *SIGMOD*, 2004.
- [9] P. Haas, J. Hellerstein, and H. Wang. Online Aggregation. *SIGMOD*, 1997.
- [10] P. Haas and L. Stokes. Estimating the Number of Classes in a Finite Population. *Journal of the American Statistical Association*, Vol 93. No 444, pages 1475–1487, 1998.
- [11] N. Kabra and D. DeWitt. Efficient Mid Query Reoptimization of Sub-Optimal Query Execution Plans. *SIGMOD*, 1998.
- [12] G. Luo, J. Naughton, C. Ellman, and M. Watzke. Increasing the Accuracy and Coverage of SQL Progress Indicators. *ICDE*, 2004.
- [13] G. Luo, J. Naughton, C. Ellman, and M. Watzke. Toward a Progress Indicator for Database Queries. *SIGMOD*, 2004.
- [14] G. Luo, J. Naughton, and P. Yu. Multi-query SQL Progress Indicators. *EDBT*, 2006.
- [15] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdžic. Robust Query Processing Through Progressive Optimization. *SIGMOD*, 2004.
- [16] C. Mishra and N. Koudas. A Lightweight Online Framework for Query Progress Indicators. <http://www.cs.toronto.edu/~cmishra/ProgressBars.pdf>.
- [17] B. A. Myers. The Importance of Percent Done Indicators For Computer Human Interfaces. *Proceedings of SIGCHI*, 1985.