Query Progress Indicators in PostgreSQL

CMSC 724 Project Report

Mike Lam
Department of Computer Science
University of Maryland, College Park
lam@cs.umd.edu

Aaron Schulman
Department of Computer Science
University of Maryland, College Park
schulman@cs.umd.edu

ABSTRACT

Progress indicators are a well-known and widely-implemented tool in computing and provide valuable feedback to end users about a task's completion percentage, as well as the time remaining. Building accurate progress indicators for database queries is difficult. There have been some recent efforts aimed at approximating the progress of queries. We implemented and compared two such methods to determine which method works best in various scenarios. We discovered that Luo works better when blocking operations dominate the execution time, while Chaudhuri works better when input nodes such as scans dominate the execution time. In this paper, we describe the two methods, present the results of our comparison, and propose several ideas for combining the two algorithms for better progress estimation.

1. INTRODUCTION

The goal of our research project was to answer the following question: How accurately can we measure the progress of an SQL query? The project is motivated by a complaint from former University of Maryland graduate student, Rob Sherwood. Sherwood's work [10] required processing-intensive SQL queries. Under the pressure of a paper submission date, Sherwood was unable to monitor the progress of his queries to see if they would be completed before the submission date, or even before the end of the millennium (some of his queries were theoretically that large). Sherwood observed that any sort of progress indicator would be a great help in ensuring the timely progress of his research.

Progress indicators are a well-known concept in computing, both in the form of progress bars and simple labels identifying the percentage completed or time remaining. They are used to track the progress of file operations, data processing routines, searches, and many other processes of various magnitudes. Researchers in the HCI community have studied progress indicators [9] and concluded that users prefer to know the progress of an operation.

A progress indicator can help a database user in many ways. Intuitively, a progress estimate can aid the user by giving them insight into the viability of their task. If a task is taking much longer than the user expected, that knowledge can allow them to cancel the task and restart it with improved parameters, thus saving time. In addition, the knowledge of how much time is remaining can help by allowing the user to be ready and start the next task promptly. Even a rough estimate of a task's duration can enable an automatic scheduler to build efficient job queues. Finally, the user is saved the frustration of not knowing how long they will need to wait, and they have assurance that the system is still processing and has not stalled.

However, building accurate progress indicators for database queries turns out to be a harder problem than one might originally expect. The problem reduces to same one that query optimizers face: obtaining an accurate estimate of the number of fixed-cost operations required by a specific query. There is a large body of research devoted to improving cost estimates specifically for query optimizers, but the suggested improvements have been largely internal to the database system. Lately, several authors have made efforts to improve progress estimates.

Following a survey of the query progress indicator literature, we identified two solutions to this problem:

1. Estimate the progress of all blocking operations in a query (Luo) [5, 6, 7].
2. Estimate the progress of all scan operators in a query (Chaudhuri) [4, 3].

To date, the methods have not been compared on the same database system using the same benchmarks with the same measures of accuracy. We set out to put these algorithms to the test in the same controlled environment in order to more rigorously compare their accuracy.

Our hypothesis was that the methods provide benefit for a mutually exclusive set of query types and inputs. We expect Luo to be accurate on queries where the processing time is spent doing blocking operations. Chaudhuri should be accurate on queries that involve more scanning of input data.

To test our hypothesis we:

- Reimplemented the two progress indicators in PostgreSQL (§4).
- Measured their performance on several queries, including some from a standard benchmark (§5).
- Compared the results using the same measures of accuracy (§5.6).

2. BACKGROUND

2.1 Problem Statement

A SQL progress indicator must perform the following tasks: 1) it must track the percentage of a query completed, and 2) it must estimate remaining execution time. An accurate progress indicator should have the following qualities: 1) it should be continuously revised to reflect rate and environment changes, 2) it should have acceptable pacing such that the progress updates are smooth but not too CPU-intensive, and 3) it should add minimal overhead to the query processing.

A trivial approach might measure progress as the number of steps completed of the number of total steps in query plan, but this would be too coarse for many queries, providing too little feedback during a lengthy step. An alternate approach might use the query optimizer's cost estimate as the upper bound on execution time and measure progress as elapsed time divided by total time. However, this would be inaccurate because of error in the cost estimation and system load variance.

2.2 Measuring query progress with pipelined segments

The Luo et al. approach [5] partitions a query plan into pipelined segments (where blocking operators mark the beginning of a downstream segment). It measures progress as the actual output of all segments in the query, divided by the estimated output. It seeds the predictions with the optimizer's estimates for each segment's output size, then as the query progresses, it continuously refines the estimated output size using linear interpolation driven by the input progress.

There are two cases for estimating the input size of a segment. Either the segment is an "upper-level segment," which means that the size of the input depends on the results of upstream operators, or the segment is a "base segment," which means that the input comes directly from an on-disk relation. In the former case, the optimizer's size estimates are used until the segment begins to execute, at which point the exact size as returned by the upstream operator can be used. In the latter case, the input size can be determined from the relation's estimated cardinality (from the cost estimator) multiplied by the average tuple size, which are continuously refined and updated during the input scan.

Luo's initial paper states that *time* is the unit of measurement most likely to be useful to end users, although for the purpose of internal calculations it is helpful to describe costs in terms of another metric. The authors use a purposefully vague U unit and mention that I/O cycles and CPU cycles are both reasonable units. For their purposes, they define one U to be one page of bytes processed. Using the input size estimates, they calculate an overall estimate (in U) for the entire query.

At all times, the sizes and U measurements for upstream operators are known exactly, and the estimates for downstream operators depend on the results for the current segment, so the authors focus on refining estimates related to the current segment. To do this, the authors introduce the concept of *dominant inputs*, which are generally the largest input relations.

In their second paper [6], Luo et al. extend their previous work in two general ways: 1) they improve the accuracy of previous estimates and 2) they add new functionality. They compare their work at a high level with that of Chaudhuri [3]. They say that Chaudhuri's technique establishes bounds on the work estimates rather than using linear interpolation, and Chaudhuri's technique does not provide time estimates.

Luo et al. also describe five "patterns" of segments and claim that their previous work is sufficient to accurately estimate the time remaining for the first three patterns but not the latter two: 1) multi-stage operators and 2) complex operators that require re-evaluation for each input tuple. The new contributions in this paper are supposed to address these two patterns.

In this second paper, the authors claim to improve on their previous work by doing the following:

- Redefining "segment" at a finer level of granularity. Segments are now split at joins so that multiple joins are no longer included in the same segment.
- Allowing for independent estimation of work unit processing speed in each segment (previously, work speed was assumed to be constant in future segments).
- Developing special handling for sort, union, intersection, and difference operators, as well as nested queries that have not been rewritten.

As in the first paper, the authors include an evaluation of their implementation in PostgreSQL.

In their third paper [7], Luo et al. extend their previous work to simultaneously incorporate time and cost estimates from multiple queries. In this paper, the authors claim that examining multiple queries can improve the progress estimates for all queries. This consists mainly of estimating how the completion of shorter queries will speed up the finishing times of longer queries.

Evaluation experiments, and the authors claim the multi-query techniques have "significant advantages" over single-query techniques.

2.3 Measuring query progress with driver operators

Chaudhuri et al. [4] point out that it is much easier to estimate the cardinalities of scans (which they call "driver" operators) than of other operators such as joins, and claim that they can improve "accuracy" (although it's not clear what they compare it to) by monitoring progress of these driver operators. The authors also propose a method of estimating the cardinalities of operators during query execution by establishing bounds for the output size.

They develop a concept that they call "pipelines" that is very similar to Luo's "segments." The general idea is: base the overall estimate mainly on the progress of driver operators, which can be known far more accurately than the other operators, using constantly re-evaluated upper and lower bounds to refine the progress estimates for the rest of the query.

Finally, they present an implementation of their algorithm inside Microsoft SQL Server, and evaluate it using the TPC-H benchmark using a single machine. They report a mean estimation error of 10%. For future work, the authors suggest improving handling of runtime load variations, reporting progress at a finer level of granularity, and adding actual time estimates.

In their second paper [3], Chaudhuri et al. claim to show that in the worse case, it is not possible to do "much better" than to simply say that the progress lies somewhere between 0% and 100%. However, they also offer some reassurance that the "good" cases are far more common than the worst cases. They examine several different estimators (including the one from their previous paper, which they call "dne") and describe several ideas for better results, including a "pmax" estimator and a "safe" estimator. They do not appear to have implemented these estimators.

2.4 Other techniques

Mishra and Koudas [8] base their work on that of Chaudhuri et al., although they claim it is equally applicable to that of Luo et al. They show several figures suggesting that their techniques improve that of Chaudhuri, but they don't provide much analysis.

Their techniques involve some operators, such as joins and aggregations, visiting and partitioning the entire input set in preprocessing. Mishra and Koudas can then use this information to more closely estimate the work remaining for the rest of the operator. When this preprocessing step is not used, they revert back to the original estimator proposed by Chaudhuri.

Finally, they state that in the future, they wish to explore "accuracy-overhead tradeoffs" in estimators as well as their effect on and potential application to adaptive query processing.

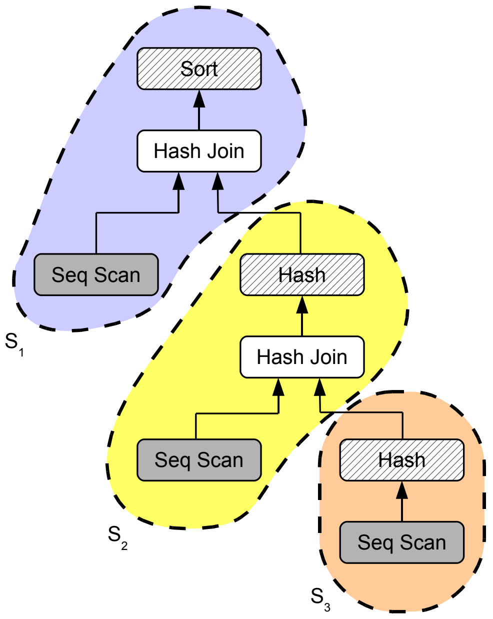


Figure 1: An example query plan. Luo and Chaud share the definition of segments (dotted and colored regions), but differ in their focus within a segment. Luo focuses on blocking operators (hatched), while Chaudhuri focuses on driver operators (shaded gray).

3. COMPARISON OF QUERY PROGRESS ESTIMATORS

We chose to implement and compare Luo [5, 6] and Chaudhuri's [4, 3] progress estimate algorithms. In a later section (§5) we evaluate the accuracy of these estimators.

The major difference between the Luo and Chaudhuri algorithms is their definition of what "drives" the progress of a query. Chaudhuri defines the "driver" of a query's progress as the output of leaf operators in a query pipeline (a pipeline is a set of query operators that can run concurrently). In practice, Luo measures a query's progress by measuring progress through the output of blocking operators. Luo also uses Chaudhuri's definition of a "driver" to help determine the progress of a blocking operator, since the output of leaf operators usually indicates the estimated output size of a segment. Figure 1 shows an example query plan with segments, drivers, and blocking operators highlighted.

The algorithms also differ in how they refine the estimated units of work that the query must perform. As a non-blocking operator (e.g. hash join) learns more about the size of its inputs, Luo uses the cost estimator to recompute the operator's output size. Blocking operators in Luo's algorithm refine their estimated output size using a linear interpolation that depends on the completed fraction of a blocking operator's input. Chaudhuri defines a lower and upper bound on input for a given operator, and refines these bounds as the query is processed.

Many progress estimators not only estimate the percent of the task that is completed, but also speculate when it will be completed, usually in the form of a "time remaining" indicator. Luo provides a method for estimating query completion time, but Chaudhuri does not.

To compare the two algorithms, we extended Chaudhuri, using a simple total average tuple processing rate to estimate the amount of time remaining for a query. We expected the work-based calculations of Luo to be more accurate than Chaudhuri's more naive estimations. However, we found that for many of the queries Chaudhuri's estimate of remaining time was far more accurate than Luo's (§5.6). We are curious to see if Luo's simple method for computing the time remaining could be improved by using Chaudhuri's query completeness estimates as input, instead of Luo's.

4. IMPLEMENTATION

The goal of this project was to compare several well-known progress estimation algorithms on the same platform. We chose to implement the Luo and Chaudhuri algorithms inside PostgreSQL [1] because it is the most popular and full-featured open source database available. At the beginning of the semester we did not have any experience with the PostgreSQL code base. It took several weeks to became acquainted with the general layout of the source code, including the query execution path and the methods that create a plan. This section contains some technical notes on our implementation, and a description of a few of the difficulties we encountered.

4.1 Luo's Algorithm

Our implementation of Luo's algorithm represents our best interpretation of the progress estimation algorithm in [5]. This required adding the following functionality to the blocking operators:

- tracking the estimated and actual numbers of output tuples,
- tracking the average size of an output tuple, and
- propagating updates of output estimates to predecessors that should recompute their estimated output (e.g. hash joins).

Our implementation differed from [5] in the following ways:

- Only hash joins can recompute their estimated output without linear interpolation.
- The estimated output of a blocking operator is the maximum of the cost re-estimation and the linear interpolation.
- Only the first predecessor hash join updates its output size.
- For aggregate operators, the output size from the cost estimator is not used. A similar fix was proposed in [6].
- The implementation does not work with subqueries. This was solved in [6], but we did not have time to implement this improvement.

4.2 Chaudhuri's Algorithm

Our implementation of Chaudhuri's algorithm represents our best interpretation of the progress estimator described in [4].

We implemented the pipelines and drivers as described in the paper by storing two flags for each operator in the query execution state tree: one to indicate whether a operator is the root of a pipeline and the other to indicate whether a operator is a driver operator. We are also using the "dne" estimator described in the paper by storing K (the number of processed tuples) and N (the number of total tuples) at each driver operator, and maintaining upper and lower bounds of potential output tuples at every operator.

As Chaudhuri describes, we estimated the overall query percentage as the sum of processed tuples divided by the sum of total tuples, considering all pipelines. For instance, if there are two pipelines, one with 50 of 100 tuples processed and the other with 0 of 100 tuples processed, the overall percentage complete is $\frac{50+0}{100+100} = \frac{50}{200} = 25\%$.

We did not implement the spill handling described in Chaudhuri's original paper, nor did we implement the "pmax" or "safe" estimators described in [3].

4.3 Notable implementation hurtles

4.3.1 PostgreSQL is designed to plan a query once

PostgreSQL performs cost estimation on possible query paths and later copies this information into the plan structures. To recompute costs (for Luo) we had to pick out the cost estimation function for hash joins from the rest of the planning code. We called this routine manually whenever we needed an updated cost estimate.

Also, PostgreSQL does not provide parent pointers in the plan tree. To update estimates for an upstream operator, we needed to be able to perform an upward traversal of the query plan tree. This required us to add parent pointers to the query plan. We added these pointers in the plan state structure because they are more easily accessible during execution.

4.3.2 Inaccurate cost estimates of filtered scans

The query optimizer arbitrarily computes the output size for sequential scans with a filter condition. To the best of our knowledge the optimizer always computes the output size of a filtered scan as 30% of the input relation's size. This simplification may be reasonable when computing a query plan based on the output size, but it leaves progress estimators with an inaccurate estimate of the progress of a query during a filtered sequential scan. Luo's query Q_2 (see appendix) contains a filtered sequential scan; this causes Luo's algorithm to perform poorly on this query.

5. EVALUATION

5.1 Setup

We evaluated the two progress estimate algorithms with the TPC-R benchmark [2]. This standard test suite includes over 1 GB of data in raw CSV format and over 20 queries. Although it is now considered obsolete (according to their website), Luo et al. used the benchmark data to evaluate their progress indicator in 2004, and we believe it is still a decent test for our purposes.

5.1.1 Queries

Along with queries 1, 4, 5, 6, 11, 14, 19, and 21 from the TPC-R benchmark, we also used queries Q_1, Q_2 and Q_3 from Luo's 2004 paper [5], on the TPC-R data set. Further, we included Q_1 and Q_2 from Luo's 2005 paper [6]. The actual queries are featured in the appendix document.

5.1.2 Hardware

- CPU: Quad Core Intel Core 2 at 2.5GHz, 3MB Cache
- Memory: 4GB
- OS: Red Hat Linux 4, kernel version 2.6.9 (32bit)
- Hard drive: 80GB SATA

5.2 Plots

Figure 2 features examples of our query progress and time remaining graphs. To aid in understanding the results presented in future sections, we introduce the pieces of the graph here.

Both types of graphs have a top and bottom portion. The top portion shows either query progress or time remaining, and the bottom part indicates when certain types of operators are executing.

The black line on the top graph shows the estimated query progress or time remaining over the query execution time. The superimposed grey line represents the "correct" estimate: essentially, a linear relationship with a positive slope for query completed and a negative slope for time remaining. For query completed graphs the correct line starts at 0 of the execution time and ends at 1. For time remaining this correct line starts at the time remaining for the actual execution of the query, and ends at 0.

Most of the predictions fall within a reasonable range, but there are predictions that are outliers. To avoid introducing inconsistent axis scales, we indicated outliers in the graphs with red Xs at the top of the graph (see Figure 2b for an example of this).

The lines on the bottom graph represent timelines, and there is one for each type of operator detected during execution. If there are data points on a particular line at a given time, that means that the algorithm generated a progress estimate while executing that kind of operator at the given time.

Unfortunately, the bottom parts of the graphs do not represent a perfect picture of the distribution of processing time during query execution, but only a rough guess. For example, since Luo emits estimates for fewer operator types than Chaudhuri, the bottom parts of the Luo graphs are significantly less complete than the bottom parts of the Chaudhuri graphs. Also, even Chaudhuri does not emit estimates during the processing of blocking operators that do not regularly emit tuples. Such operators will sometimes appear as gaps in the bottom graph if they take a significant amount of time to complete.

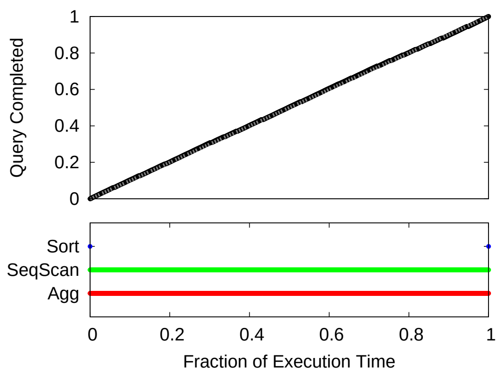
For this paper, we have only included those plots directly referenced in our discussion. We have included the full set of plots in the appendix document.

5.3 Overhead of Query Progress Estimators

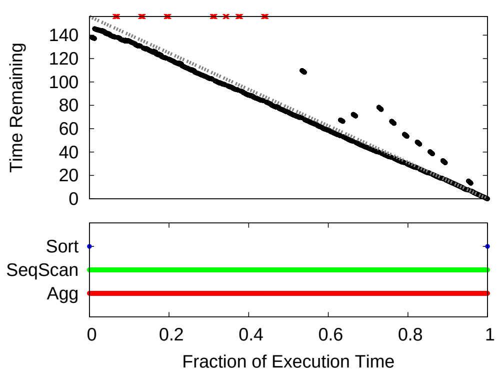
Both of the query progress estimation algorithms add overhead to the query execution time. Table 1 shows the query execution times without progress estimation as well as the relative time with the Luo and Chaudhuri progress estimators.

The rate of updating the user about the progress of a query is closely related to the overhead for these estimators. Put simply, more updates cause more overhead. To find the upper overhead bound, we called *print* to emit a progress estimate for each tuple that was processed (this incurs the cost of a system call for every tuple). To find the lower overhead bound, we ran the algorithms without any user output (*noprint*).

Ideally, the overhead of the progress estimators would be minimal, and the relative time should be close to 1. For all of the queries, the *print* version of Luo and Chaudhuri takes less than 2 and 3 times the default



(a) Query Progress



(b) Time Remaining

Figure 2: TPC-R 1 with Chaudhuri's algorithm. An example of the estimated "Query Progress" and "Time Remaining" graphs we use to evaluate the accuracy of the estimation algorithms; notice the outliers (denoted as red Xs) in the time remaining graph.

Query	Unmodified Time (s)	Chaud (noprint)	Luo (noprint)	Chaud (print)	Luo (print)
1	87	1.126	1.115	1.793	1.494
4	3	0.667	0.667	2.000	1.333
5	23	1.000	0.697	2.608	0.690
6	5	1.000	1.000	1.200	1.000
11	4	1.000	0.75	2.500	0.750
14	5	1.000	1.200	1.600	1.200
19	7	1.000	1.000	1.285	1.142
21	50	1.220	1.000	2.120	1.020
Luo Q_1	186	1.096	1.284	1.339	1.360
Luo Q_2	208	1.173	1.216	1.514	1.192
Luo Q_3	16	1.437	1.250	2.375	1.562
Luo 2005 Q_1	61	1.525	1.082	2.524	1.634
Luo 2005 Q_2	42	1.238	1.238	1.619	1.167

Table 1: Unmodified and relative query execution times for Luo and Chaudhuri. Highlighted rows indicate large differences between the execution times.

execution time, respectfully. For the *noprint* version, both algorithms take less than 1.6 times the default execution time.

This is not horrible, but not very compelling, either. If a query takes twice as long to complete with progress estimates, is it really worth it? However, our implementation was not optimized for execution speed, and we believe this contributed to the poor performance of these algorithms.

There were a few significant differences between the relative running times of the *noprint* version of the estimators. However, the highlighted results in Table 1 indicate that Chaudhuri's *print* version is often 2x slower than Luo's *print* version.

5.4 Accuracy of Luo's Algorithm

Now we evaluate the accuracy of Luo's query progress estimates. Based on the mean estimation error (Table 2), the estimator had less than 1% mean error on TPC-R 1, 4, 6, and Luo Q_1, Q_3 and 2005 Q_2. Luo performs well on queries that have a predictable number of expected work units, and a constant rate of consumed work units; all of the aforementioned queries have this

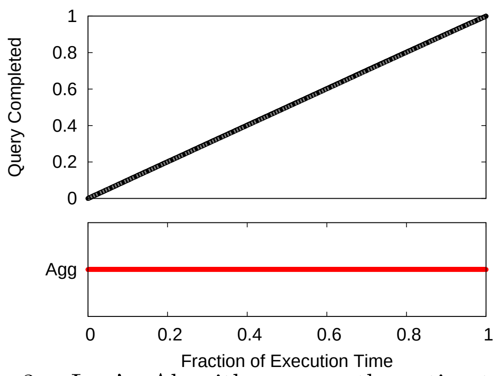


Figure 3: Luo's Algorithm correctly estimates the progress of the simple query TPC-R 6.

Query	Max	Mean	StdDev
1	0.01	0.00	0.00
4	0.13	0.05	0.05
5	0.91	0.34	0.30
6	0.00	0.00	0.00
11	0.43	0.27	0.13
14	0.82	0.32	0.25
19	0.85	0.43	0.25
21	0.98	0.24	0.24
Luo Q_1	0.01	0.00	0.00
Luo Q_2	0.48	0.22	0.15
Luo Q_3	0.23	0.07	0.07
Luo 2005 Q_1	0.26	0.14	0.07
Luo 2005 Q_2	0.00	0.00	0.00

(a) Luo

Query	Max	Mean	StdDev
1	0.01	0.00	0.00
4	0.14	0.05	0.04
5	0.17	0.13	0.04
6	0.03	0.02	0.01
11	0.40	0.22	0.12
14	0.31	0.10	0.08
19	0.18	0.08	0.05
21	0.00	0.00	0.00
Luo Q_1	0.04	0.02	0.01
Luo Q_2	0.27	0.13	0.07
Luo Q_3	0.27	0.12	0.06
Luo 2005 Q_1	0.60	0.28	0.17
Luo 2005 Q_2	0.59	0.25	0.16

(b) Chaudhuri

Table 2: Error for estimates of percent completed. Highlighted rows indicate that the estimator performed poorly (more than 20% error).

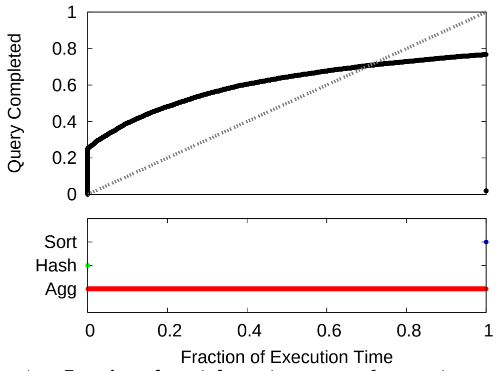


Figure 4: Luo's algorithm incorrectly estimates the progress of query TPC-R 21

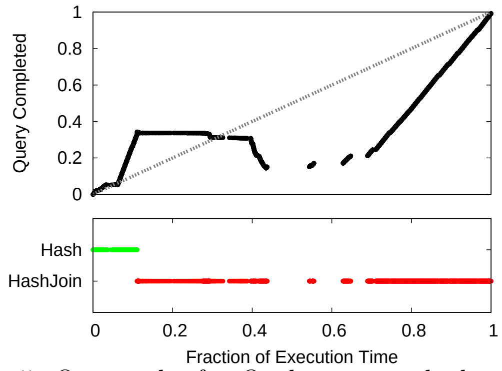


Figure 5: Our results for Q_2 do not match the results from Luo's implementation

quality. For example, with only one output operator, Luo perfectly estimates the percentage of query completed for TPC-R 6 (Figure 3). This is due to the query's simplicity; there is only one sequential scan that feeds into an a blocking aggregate operator. We obtain correct linear interpolation of the aggregate's observed progress by observing the sequential scan's progress. Also, as there is only one blocking operator, the rate of completed work units is constant. It should be noted that Luo can also perform well on more complicated queries, like Luo's Q_3.

The estimator does not perform well on TPC-R 1, 5, 11, 14, 21 and Luo Q_2. These queries can be grouped into several types: 11 and 21 have at least one subquery, 14 and 19 prematurely reach 100% completion, and Q_2 does poorly as a result of an inaccurate cost estimation of filtered scans (which contradicts the results from Luo's paper).

Both queries 11 and 21 are inaccurate (Figure 4) because the version of Luo's algorithm that we implemented [5] does not include the explicit support for subqueries. Luo later added this in [6].

For queries 14 and 19, the database devotes most of its time to computing aggregates. While the aggregates are executing, the estimator reports that the query is 100% complete. A careful examination revealed that the cause of this behavior is the extra processing that an aggregate operator performs even after it receives all of the incoming tuples.

Now we turn our analysis to results that conflict with [5]. In Luo's paper, the progress estimate for Luo Q_2 is near perfect while our implementation produces less than perfect results (Figure 5). There are two possible causes for this inconsistency: 1) our implementation is flawed or 2) the statistics in the Luo paper are the result of special optimizations not mentioned in the paper. Unfortunately, since the implementation used for the paper is closed-source, we cannot compare their implementation with ours.

We believe that the algorithm as presented in their paper can not perform as well as their graph suggests for the following reason: PostgreSQL's inaccurate estimate of filtered scans incorrectly estimates the output size of the root hash join (§4.3.2). As a result, the estimated work units are also incorrect. At time 0.1 the estimator reaches the point where the actual output size overtakes the initial estimate of the sequential scan's output size. From this point to time 0.7, the incorrect estimates of the sequential scan are being updated with every new tuple seen by the database. While this is happening the progress estimate holds at 0.3. At time 0.7 the errant sequential scan ends, and the new cost of the hash join stops being recomputed. Finally the rest of the hash join results are computed, and the query ends. These contradictory results make the case for releasing source code so others can reproduce published results.

Luo's time remaining estimates are not accurate for most of the queries. For example, in Figure 6 you can see even though the progress estimate is very accurate, the time to completion is consistently underestimated. We have not determined the cause for the abysmal accuracy of the time to completion.

5.5 Accuracy of Chaudhuri's Algorithm

The clearest strength of Chaudhuri's algorithm is manifested in query plans where the execution time is dominated by driver operators (e.g. sequential or index scans). The queries that clearly have this behavior are TPC-R 1, 4, 6, and 21, as well as Luo's Q_1. The mean errors for query progress percentages are 5% or less for all of these queries, and the query progress graphs (see Figure 2 for an example) show that the estimates are very close to the grey dashed line that defines a "perfect" estimate. In addition, the time remaining graphs show that the estimates are roughly correct for significant portions of the execution time, although they are less accurate than the query progress estimates.

The accuracy of Chaudhuri on TPC-R 21 (see Figure 7) is particularly remarkable, given the complexity of the query. The query includes a large variety of types of operators, many joins, a subquery, and a final sorted aggregation. Despite this heterogeneity, the driver operator estimate is close to the correct percentage for the majority of execution.

In the Chaudhuri results, Q_2 from Luo's paper exhibits another peculiar feature. The query progress graph for this query (see Figure 8) has an obvious "staircase" pattern of short bursts of activity followed by lengthy periods with no observed progress. This may be simply due to an oversight in implementation. We tried to ensure that every location that handled tuples in sequence were instrumented with progress estimate samples, but since these sections are spread throughout the PostgreSQL code we cannot guarantee that we handled them all.

Assuming that we correctly sample all processed tuples, however, there are two other explanations for the gaps in these results: 1) heavy disk I/O and 2) aggregate processing periods. The first possibility is that the system is paging memory or buffering I/O and there is actually no work being done by the query processing engine itself. To fix this sort of gap, you would need to predict the I/O behavior and interpolate results between checkpoints in the query execution. Chaudhuri has no such provision. The second possibility is that some operator is performing aggregate calculations (e.g. sorts or grouping) that do not process tuples in an easily discernible per-tuple manner. Chaudhuri does not provide any means of updating the progress estimate during such operations.

The overarching problem with Chaudhuri's algorithm is that it does not take into account the additional work that needs to be done in non-driver operators, such as hash joins, sorts, or aggregations. This results in progress estimates that remain at a certain percentage too long, or reach 100% prematurely and spend the rest of execution time unchanged. This can be seen most clearly in the results for TPC-R 5, 11, 14, 19, as well as for most of the queries from Luo's paper. See Figure 9 for an example of this behavior.

5.6 Comparing the Accuracy of Luo and Chaudhuri

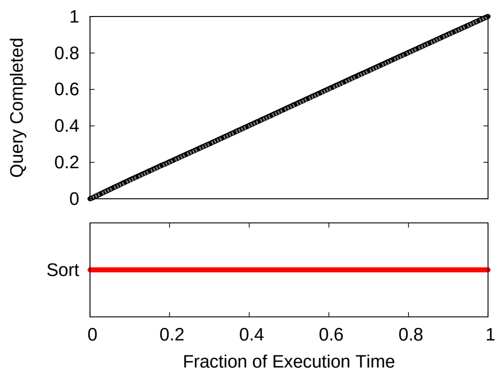
We used two methods to compare the estimation algorithms: 1) visual inspection of the query progress graphs, and 2) comparison of error statistics.

5.6.1 Visual Inspection

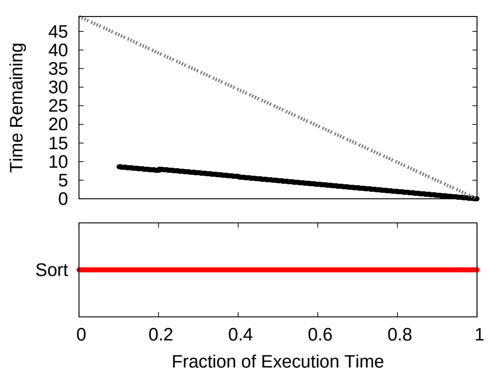
We performed a visual comparison by observing all of the graphs (such as those referenced in the previous section) and comparing how close the lines appear to be to the dotted lines that represent a "perfect" estimate. If the estimates fit the diagonal very closely, this is taken as an indication of a "good" estimate, and estimates that differ significantly from the diagonal are understood to be "bad." This kind of comparison is inexact, but it functions well enough as a qualitative evaluation of general trends in algorithmic performance. However, this comparison is still important because the progress estimators will eventually be implemented as progress bars observed by the user. Minor variations in the estimate will not make a discernible difference in the behavior of the progress bar.

Both algorithms do well for simple queries composed primarily of a single sequential scan (e.g. TPC-R 1 and 6, and Luo's Q_1). This is not surprising, since estimating the percentage completion for such queries is easy and the algorithms focus on the same operators. However, the algorithms also both perform well on more some complex queries like TPC-R 4 and Luo Q_3.

Neither of the algorithms are accurate on TPC-R 11, 14, or Luo's Q_2. For TPC-R 14, they both are inaccurate because of a long-running aggregate operation; however, Luo sufferers more because of this. On TPC-R 11, both Luo and Chaudhuri overestimate the query progress. Chaudhuri is inaccurate on query Luo Q_2

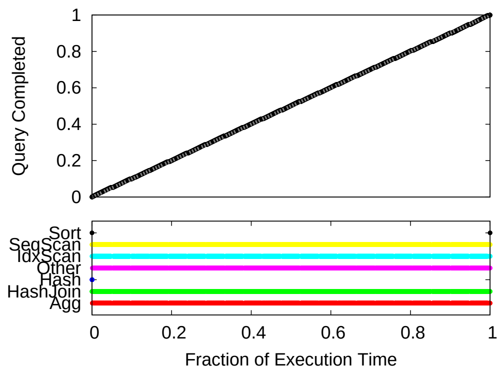


(a) Query Progress

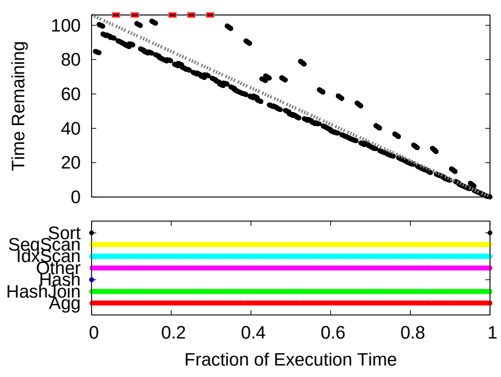


(b) Time Remaining

Figure 6: Luo's estimates of remaining time are inaccurate (Query: Luo 2005 Q_2).



(a) Query Progress



(b) Time Remaining

Figure 7: Chaudhuri's algorithm performs very well for TPC-R 21 despite its complexity.

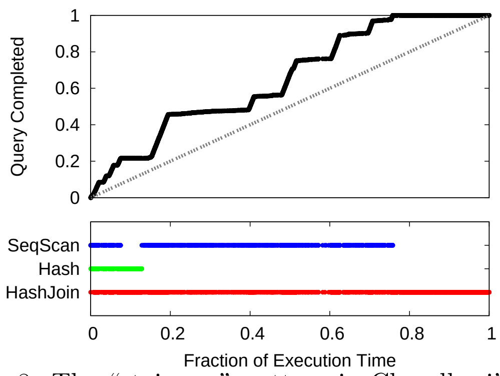


Figure 8: The "staircase" pattern in Chaudhuri's estimates for Luo's Q_2 is an interesting anomaly.

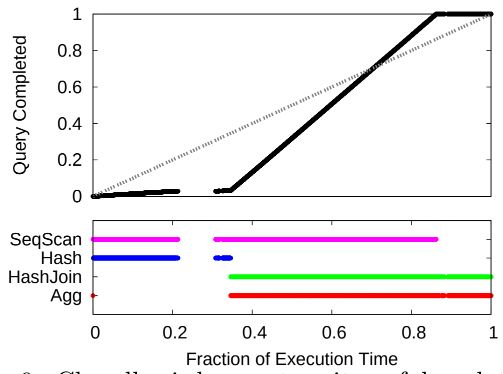


Figure 9: Chaudhuri does not emit useful updates for blocking aggregate operators like hashes (Query: TPC-R Q14).

because the database exhibits variable rate of incoming sequential scan tuples. Luo behaves poorly on this query because of inaccurate cost estimation (§4.3.2).

For queries with a deep operator tree (e.g. TPC-R 5 and 21), the graphs for Luo show a wider variation of behaviors (curves, lines, gaps, and patterns), and they are generally further from the ideal than the corresponding Chaudhuri lines. The Chaudhuri query percentage graphs follow a linear relationship; however, they still show significant deviation from the ideal for some of the queries. For TPC-R 19, both algorithms have inaccurate regions, but Chaudhuri performs better since the query is dominated by a sequential scan.

Luo 2005 Q_1 and Q_2 reveal a weakness in Chaudhuri's approach. For these queries the database spends a considerable amount of time working on operators that do not fit Chaudhuri's definition of "drivers." The most interesting case is the query Luo 2005 Q_1 (Figure 10). When we ran this query with the Chaudhuri algorithm the results were abysmal, because the execution time is dominated by a nested loop join. With the Luo implementation, this nested loop join appears to end much quicker (indicated by the blocking hash ancestor) and does not dominate the execution time. Based on these plots and the query execution time in Table 1, it appears that the overhead of the progress estimation affects the accuracy of the progress estimates for Chaudhuri in this case.

For the time remaining graphs, Chaudhuri's algorithm again works better in general than Luo's. This is probably a simple extension from the better performance on query progress; intuitively, the algorithm that is better able to estimate the query percentage will be better able to estimate the time remaining.

5.6.2 Error Statistics

We also wanted to perform a more quantitative comparison, so we decided to compare statistics relating to the error of the progress estimates. We define the errors as follows:

$$\epsilon_{qpct} = |E_{qpct} - A_{time}| \tag{1}$$

$$\epsilon_{time} = |E_{time} - A_{time}| \tag{2}$$

Where E_{qpct} is the estimated query percentage complete, E_{time} is the estimated time remaining (in seconds), and where A_{qpct} is the actual query percentage complete and A_{time} is the actual time remaining (in seconds).

Since A_{qpct} and A_{time} can only be determined after the query execution has finished, we did not calculate these statistics online. Instead, we logged all the progress estimates to disk and performed a post-processing step to calculate errors. We used regular sampling to extract 1000 ϵ_{qpct} and ϵ_{time} values evenly across the execution time, and we report the maximum, mean average, and standard deviation.

The error statistics for ϵ_{qpct} are included in Table 2 (in the table, all percentages are expressed as a decimal fraction). Unfortunately, the errors for the time remaining metric are still so high that the statistics for ϵ_{time} were virtually meaningless. For this reason, we have not included them in this paper.

The error for both algorithms were similar on the queries TPC-R 1, 4, 6, 11, Luo Q_1 and Luo Q_3. In general, the average errors for the Chaudhuri estimates were lower than for the Luo estimates. The exceptions were Luo's 2005 Q_1 and Q_2. For these queries, Luo performed much better than Chaudhuri. In fact, Luo's estimates for the query Luo 2005 Q_2 are essentially perfect, and the mean error is nearly zero.

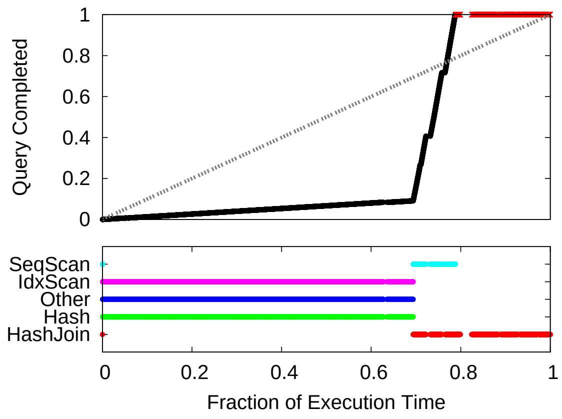
The error standard deviations generally reflected the mean error; high variances accompanied high average error. In most cases, the maximum error also reflects average error, although there are exceptions like Chaudhuri's estimates for TPC-R Q4 and Q14, where the max error is around three times the mean error.

6. FUTURE WORK: HYBRID PROGRESS ESTIMATORS

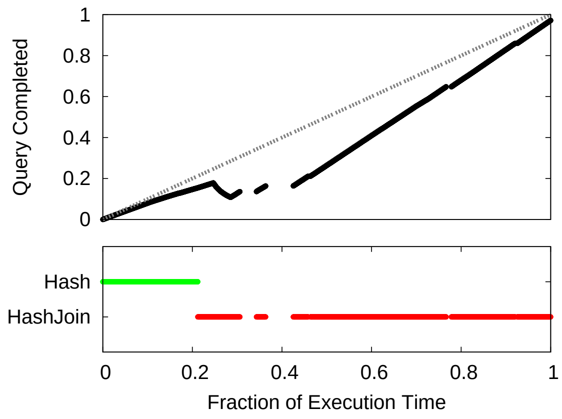
We have two ideas for improved hybrid progress indicators. Unfortunately, because of time constraints we did not implement or evaluate these algorithms.

Our first idea is to use Chaudhuri's algorithm as a base, layering Luo's algorithm on top for certain parts of a query tree. The Chaudhuri algorithm is incapable of effectively estimating the progress of blocking operators like hashing and sorting, while Luo does quite well with these kinds of operators. In addition, there are certain situations in which Chaudhuri adds an excessive amount of overhead. Thus, we propose using Chaudhuri's algorithm except when the driver of a pipeline is a blocking operator, at which point we would advocate switching to a Luo-based algorithm to estimate its progress.

Our second idea is to use Chaudhuri and Luo in parallel to capture information about progress both from inputs and outputs, since both Chaudhuri and Luo both use similar concepts for concurrent sections of a query ("segments" and "pipelines"). However, Luo works by observing progress through a section's output, while Chaudhuri works by looking at the progress through a section's input. By using Chaudhuri to estimate progress through the input of a segment and Luo to estimate progress through the output of the same segment, we believe we can create a weighted average of the two to obtain a more accurate overall progress estimate. We believe that by taking advantage of the similarities between the two algorithms, we can implement a com-



(a) Chaudhuri



(b) Luo

Figure 10: The overhead of Chaudhuri's algorithm causes it to inaccurately estimate the progress of Luo 2005 Q_1.

bined algorithm with minimal overhead.

7. CONCLUSION

We have examined, implemented, and compared two recent algorithms for estimating the progress of a query in a relational database system. We have evaluated these algorithms on a standard benchmark, and have described their differing results. It appears that Luo works better when blocking operations dominate the execution time, while Chaudhuri works better when driver operators dominate the execution time. We have also proposed ideas for better progress estimators by combining the current approaches into new hybrid algorithms.

8. REFERENCES

[1] PostgreSQL: The world's most advanced open source database. Accessed 10 April 2009. `http://www.postgresql.org`.

[2] Transaction Processing Performance Council. TPC-R. Accessed 9 April 2009. `http://www.tpc.org/tpcr/default.asp`.

[3] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust progress estimators for sql queries? In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 575–586, New York, NY, USA, 2005. ACM.

[4] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for sql queries. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 803–814, New York, NY, USA, 2004. ACM.

[5] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a progress indicator for database queries. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 791–802, New York, NY, USA, 2004. ACM.

[6] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Increasing the accuracy and coverage of sql progress indicators. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 853–864, Washington, DC, USA, 2005. IEEE Computer Society.

[7] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query sql progress indicators. *Advances in Database Technology - EDBT 2006*, 3896:921–941, March 2006.

[8] C. Mishra and N. Koudas. A lightweight online framework for query progress indicators. *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1292–1296, April 2007.

[9] B. A. Myers. The importance of percent-done progress indicators for computer-human interfaces. *SIGCHI Bull.*, 16(4):11–17, 1985.

[10] R. Sherwood, A. Bender, and N. Spring. Discarte: a disjunctive internet cartographer. *SIGCOMM Comput. Commun. Rev.*, 38(4):303–314, 2008.