Toward a Progress Indicator for Database Queries

Gang Luo Jeffrey F. Naughton
University of Wisconsin-Madison
{gangluo, naughton}@cs.wisc.edu

Curt J. Ellmann Michael W. Watzke
NCR Advance Development Lab
{curt.ellmann, michael.watzke}@ncr.com

Abstract

Many modern software systems provide progress indicators for long-running tasks. These progress indicators make systems more user-friendly by helping the user quickly estimate how much of the task has been completed and when the task will finish. However, none of the existing commercial RDBMSs provides a non-trivial progress indicator for long-running queries. In this paper, we consider the problem of supporting such progress indicators. After discussing the goals and challenges inherent in this problem, we present a set of techniques sufficient for implementing a simple yet useful progress indicator for a large subset of RDBMS queries. We report an initial implementation of these techniques in PostgreSQL.

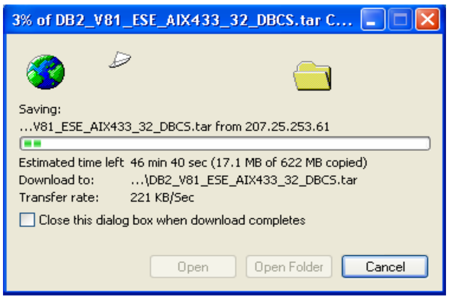
1. Introduction

Progress indicators are a widely used user-interface technique in modern software systems. For example, Figure 1 shows a progress indicator for file downloading. Typically, a progress indicator has the following two features:

(1) It keeps track of the percentage of the task that has been completed.

(2) It continuously estimates the remaining task execution time.

These two features make the software systems much more user-friendly: by knowing how long he/she needs to wait for a program to finish, the user can better utilize his/her time [16]. In fact, in many cases, even a rough estimate of the remaining task execution time can be beneficial to the user [4].


3% of DB2_V81_ESE_AIX433_32_DBCS.tar C...

Saving:
...V81_ESE_AIX433_32_DBCS.tar from 207.25.253.61

Estimated time left 46 min 40 sec (17.1 MB of 622 MB copied)
Download to: ...\DB2_V81_ESE_AIX433_32_DBCS.tar
Transfer rate: 221 KB/Sec

□ Close this dialog box when download completes

Figure 1. A typical file download interface.

Such progress indicators are useful whenever a user might have to wait for a task to complete. Unfortunately, some RDBMS queries definitely fall into this category, as queries can take a long time to run. Hence, progress indicators are desirable in RDBMSs [18, 15]. To the best of our knowledge, however, none of the existing commercial RDBMSs provides a non-trivial progress indicator, and we are unaware of any published techniques for supporting such a progress indicator.

Some RDBMSs provide trivial progress indicators for complex queries by breaking the query plan into steps, and then reporting at any time which steps have completed and which steps are still left to run (see, e.g., [8].) While such a progress indicator is clearly much better than nothing, for many purposes it will be too coarse – even a long-running query may only have a few steps, and such a progress indicator does not give the user any feedback while a (potentially very long) step is running.

Another way to provide a trivial progress indicator is to use the optimizer's estimate of query running time. Providing a trivial progress indicator based upon the optimizer's estimate of query running time is simple. If the optimizer estimates that a query will take t seconds, and the query has run for t' seconds, we estimate that the remaining time is $t - t'$ seconds. While such a trivial progress indicator is also better than nothing, it is likely to be highly inaccurate. This inaccuracy arises from two main causes:

(1) Optimizers' query cost estimates typically contain errors. Furthermore, accurately predicting actual query running times is more challenging than choosing good plans over bad ones, as estimates that correctly rank plans only need to be correct about relative costs, not actual costs. For this reason, using optimizers' estimates for progress indicators is even more problematic than using them for query optimization.

(2) Due to concurrently running queries and other jobs, the system load may vary significantly. For a specific query, even if the optimizer provides an estimate that is precise for an unloaded system, this estimate may differ substantially from the actual query execution time in a loaded system.

In this paper, we propose techniques for supporting progress indicators for RDBMS queries. We demonstrate the utility of these techniques by an implementation for select-project-join queries in PostgreSQL. While the resulting progress indicator can be refined, our experiments show that it is a useful progress indicator even in the presence of optimizer estimation errors and varying run-time system loads, and that it imposes a negligible (less than 1%) penalty on the running time of queries.

Our basic approach is to separate a complex query plan into pipelined segments, where the boundaries of the segments are defined by blocking operators. We measure query progress in terms of the percentage of input processed by each of these segments. We begin with the optimizer's estimates for cardinalities and sizes. However, as a query runs, we obtain more and more precise information about the inputs to the segments in its execution plan. Also, at all times, we monitor the

ACM SIGMOD 2004, June 13–18, 2004, Paris, France.
Copyright 2004 ACM 1-58113-859-8/04/06 …$5.00.

speed at which segments are processing their inputs (which is a function of the query plan and the system load at runtime.) We use this more precise information to continuously refine the estimated query execution time and thus to update the progress indicator.

The rest of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we describe the goals of progress indicators for RDBMSs. In Section 4, we present a set of techniques for implementing progress indicators in an RDBMS. In Section 5, we present results from an initial implementation of our techniques in PostgreSQL. We conclude in Section 6.

2. Related Work

There has been a lot of work (e.g., [4], [16]) in the HCI (Human Computer Interaction) community for progress indicators. However, none of this work has addressed database queries.

Online aggregation, proposed in [11], shares with this work the goal of providing continuous feedback during query execution. For long-running aggregate queries, online aggregation provides continuously refined approximate aggregate query results. For simple aggregate queries (e.g., a single-table scan), the online aggregation interface contains a progress indicator that indicates the percentage of the query that has been completed [11]. However, online aggregation provides no estimate of the remaining query execution time. Online aggregation requires special non-blocking query evaluation algorithms [11, 10]. In contrast, the progress indicators discussed in this paper are not limited to aggregation queries, and do not require non-blocking query evaluation algorithms.

In dynamic query optimization [2, 14, 5, 12, 9, 17], people have proposed refining the query cost estimate at one or more points to change the query plan dynamically. However, such refinement is not continuous. Also, no estimate of the remaining query execution time or the percentage completed is provided in dynamic query optimization.

[1, 3] propose building and maintaining histograms by analyzing query results (rather than examining the data sets). Then they use the (refined) histograms to estimate the costs of future queries more precisely. However, [1, 3] did not use the intermediate results of a query to estimate the remaining query execution time/cost as the query is being processed.

Most commercial database vendors (DB2, Oracle, SQL Server, Teradata, Tandem, etc.) provide database monitoring tools. These tools provide various information (e.g., elapsed time, current execution step, number of I/Os performed) for a running query and can alert the DBA if the running query exhibits excessive overhead [7, 15, 8, 18]. In certain simple cases, such information can be used to estimate the remaining query execution time [15]. However, in general, the information provided by existing database monitoring tools is not enough to estimate either the percentage of the query that has been completed or the remaining query execution time [18].

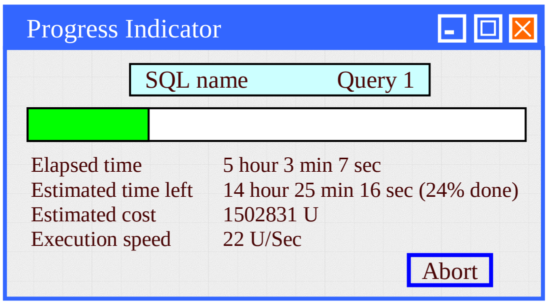
Some commercial RDBMSs provide query cost estimates measured in time (e.g., seconds) based on an unloaded RDBMS. As explained in the introduction, even if a query cost estimate is precise for an unloaded RDBMS, it can differ significantly from the actual query execution time in a loaded RDBMS.

[13] proposed a method for estimating the optimizer compilation time of a query. [13] also proposed using the same method to monitor the progress of workload analysis tools. However, no method is proposed in [13] to monitor the progress of queries.

[15] proposed a method for monitoring the progress of long-running rollback operations. The idea is to monitor the number of update log records that have not been rolled back for a transaction. By calculating the speed that the update log records are being rolled back, we can estimate the remaining rollback time for this transaction. This method can be integrated into the progress indicators for RDBMSs so that these progress indicators can also monitor the progress of rollback operations.

3. Goals for Progress Indicators

Figure 2 shows an example of the sort of progress indicator we would like to support for database queries. This interface, which is continuously updated, displays the elapsed time, the estimated remaining query execution time, the estimated percentage of the query that has been completed, the estimated query cost, and the current query execution speed. Both the estimated query cost and the current query execution speed are measured in *U*'s, where *U* is an abstract quantity that represents one unit of work (we will return to the question of how to define *U* in Section 4.)



SQL name Query 1

Elapsed time 5 hour 3 min 7 sec
Estimated time left 14 hour 25 min 16 sec (24% done)
Estimated cost 1502831 U
Execution speed 22 U/Sec

Figure 2. A progress indicator for database queries.

Ideally, a progress indicator should satisfy the following goals:

(1) **Continuously revised estimates**: At any time, for all the information provided to the user, the progress indicator should give an estimate based on all the information available about the query and the system at that time. This estimate should be continuously refined, due to both changes in the estimates of intermediate result sizes and changes in the rate at which the query is progressing.

(2) **Acceptable pacing**: The progress indicator should be updated frequently enough that the user sees a smooth display. However, the update rate should not be so frequent as to overburden either the user interface or the user.

(3) **Minimal overhead**: The progress indicator should have a small effect on the efficiency of query execution.

4. Implementation Techniques

In this section, we present our techniques for implementing progress indicators in an RDBMS. We consider select-project-join queries, and assume that the available join algorithms are hash join, nested loops join, and sort-merge join, and that base relations can be accessed by either table-scans or index-scans.

Our main idea is as follows:

(1) We collect statistics at selected points of a query plan. As a query is being processed, we will have more and more precise information about intermediate results (e.g.,

cardinality, size) and the run-time system (e.g., amount of available memory). We use the improved information to continuously refine the estimated query cost.

(2) We continuously monitor the query execution speed (i.e., how many *U*'s are processed per second). At any time, the remaining query execution time is estimated to be the ratio of the estimated remaining query cost to the observed current query execution speed.

From time to time, the progress indicator presents the latest estimates to the user.

In Section 4.1, we describe how we choose the work unit *U* and how it is converted to time. In Section 4.2, we define the concept of segments that is crucial to our query cost estimation. Then we show how to get the cardinalities and sizes of segment inputs in Section 4.3. In Section 4.4, we present the statistics collection techniques. In Section 4.5, we describe how to continuously refine the query cost estimate. In Section 4.6, we discuss the techniques used in monitoring the query execution speed.

4.1 Choosing *U* and Converting to Time

As mentioned in Section 3, both the estimated query cost and the current query execution speed are measured by the abstract unit *U*. Each *U* represents one unit of work. We are purposely being rather vague and general in this statement, as many viable alternatives exist for *U*. The important requirements for *U* are that one can readily estimate how many *U*'s a query will take to execute, and that one can readily convert from *U*'s to estimated time, since ultimately time is likely to be the unit most meaningful to users. Reasonable candidates for *U* include I/Os, CPU cycles, or even a combination of the two, perhaps using some weighting factor.

Our progress indicator works by continuously refining both its estimate of how many *U*'s the segments in a query will take to execute (segments are defined in Section 4.2) and its estimate of the conversion factor from *U* to time. The estimated number of *U* required to process a segment changes as the system gathers more statistics about intermediate results as the query runs. The refinements in the estimates of the conversion factor from *U* to time result from observations of how quickly the system is processing *U*. (If *U* were chosen to be CPU cycles, this translates to the admittedly strange sounding question "how fast is the system processing CPU cycles?"; in this case, this question would really mean "how many CPU cycles per second are being devoted to this query?")

In this paper, for simplicity, we define *U* in terms of bytes processed, with the intuition that this is easy to measure and serves as a rough proxy for CPU and I/O. That is, the cost of a query *Q* is the total size of the (input and intermediate result) tuples that are to be read and written by *Q*. Similarly, at any time, we represent the amount of work that has been done on *Q* using the total bytes that have been processed so far for *Q*.

We set *U* to be one page of bytes, and assume initially (before the query starts running) that executing the query will require a number of *U* equal to the optimizer's estimate of the number of I/Os for the query. Before giving its first estimate of running time, the progress indicator "watches" some amount of processing to see how quickly the system is consuming *U*; we discuss this in more detail in Section 4.6.

As the query runs, the estimated time to process one *U* will change to reflect the observed processing rate in the system. The time to process one *U* could range from the time for one physical I/O (if the system is disk-bound) to the time to process one buffer-pool resident page of data (if the data accessed by the query is completely cached in memory) or anywhere in between. In fact, in a heavily loaded system, the time to process a *U* could even exceed the time to perform a physical I/O.

This simple definition of *U* limits the precision of our estimates; however, in our experiments, described in Section 5 below, this definition worked well in our tests, both for I/O and CPU intensive queries. We leave it as an interesting area for future work to explore how to improve the estimates without imposing undue overhead by refined definitions of *U*.

4.2 Definition of Segments

In order to support progress indicators, we divide a query plan into one or more segments so that we can focus on the individual segments rather than the entire query plan. (Dividing a query plan into parts has been proposed before, for resource management and parallel processing purposes [6].) Each segment contains one or more consecutive operators that can be executed as a pipeline. A pipeline continues within a segment and breaks at the end of a segment. In practice, blocking operators (e.g., hash-table build operators, sort operators, intermediate result materialization operators) serve as natural separation points of different segments [14].

Each segment can be viewed as a tree. The root of the tree is the output of the segment. The leaves of the tree are the inputs of the segment. The inputs of a segment either come from base relations or from the outputs of lower-level segments.

Figure 3 shows a query plan that contains five segments:

(1) Segment S_1 computes $\pi(\sigma(A))$ and hashes the results into multiple partitions P_A.

(2) Segment S_2 computes $\sigma(B)$ and hashes the results into multiple partitions P_B.

(3) Segment S_3 computes a hash join using P_A and P_B and sorts the results into multiple sorted runs $R_{AB.}$

(4) Segment S_4 computes $\sigma(C)$ and sorts the results into multiple sorted runs R_C.

(5) Segment S_5 computes a sort-merge join using R_{AB} and R_C and generates the final query result after projection.

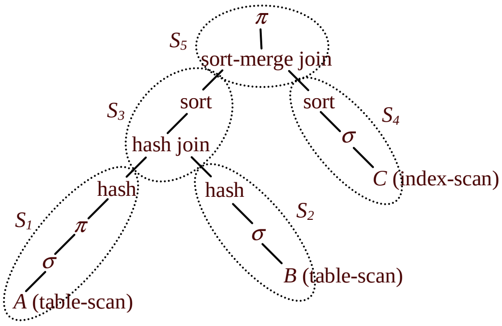


Figure 3. A query plan example.

4.3 Cardinalities and Sizes of Segment Inputs

In this section, we show how to get the cardinalities and sizes of segment inputs. As mentioned in Section 4.2, there are two kinds of inputs to segments:

(1) **Upper-level segment inputs**: An upper-level input of a segment is the output of some other lower-level segment.

(2) **Base segment inputs**: A base input of a segment comes from a base relation.

For an upper-level input of a segment *S*, at the time *S* starts execution, all the segments that are below *S* must have finished. Hence, the output cardinalities and sizes of these lower-level segments are known exactly, since they are computed as the segments run (see Section 4.4 below).

In contrast, a base input of a segment is either a table-scan or an index-scan. At the beginning of a table-scan or index-scan, we have to use the optimizer's cardinality and size estimate for the cardinality and size of the input, even if this estimate is not precise. (We have no choice – we have not even seen any of the input data in question.) Suppose that the optimizer's cardinality estimate for the base segment input is N_e and the precise cardinality is N_p. There are two possible cases:

(a) $N_p \leq N_e$. During the table-scan or index-scan, we keep using N_e as the estimated segment input cardinality. After finishing the scan, we know the precise number N_p and use it as the precise segment input cardinality.

(b) $N_p > N_e$. During the table-scan or index-scan, we keep using N_e as the estimated segment input cardinality, until the actual number of tuples that have been read exceeds N_e. From then on until the finish of the scan, we use the actual number of tuples read so far as the estimated segment input cardinality.

During the table-scan or index-scan, we collect statistics about the average tuple size. The size of the base segment input is the product of its cardinality and its average tuple size.

If the estimated segment input cardinality and/or size changes, we need to (see Section 4.5 below) refine the estimates related to the current segment, and propagate these changes upward in the query plan tree.

4.4 Collecting Statistics

We collect statistics about cardinalities and average tuple sizes of the intermediate results, which can be computed on the fly inexpensively as the intermediate results are being generated. For any intermediate result, its size is the product of its cardinality and its average tuple size.

We collect statistics (output cardinality, average tuple size) at the output of each segment. The only exception is the last segment in the query plan, for which the output is the final query result that will be returned to the user. Therefore, no statistics are collected there.

Unlike [14], we do not collect statistics about the number of distinct values and histograms of the intermediate results. In our experiments, which are described in Section 5 below, our statistics collection techniques worked well. It is an interesting area for future work to explore whether either collecting more complex statistics or collecting statistics within segments can significantly improve the estimates without imposing undue overhead.

Unlike [14], we do not insert statistics collector operators into the query plan. Rather, we embed the statistics collection code in the operator code. For each operator, we augment its data structure so that the collected statistics can be held there. For each query plan, we use a flag to control whether statistics need to be collected. When the progress indicator feature is in use, the flag is turned on and we collect statistics in appropriate operators. If one does not wish to modify existing operator code, our approach to statistics collection can be modified to use that presented in [14].

4.5 Refining the Estimate of the Number of *U* Required by the Query

In this section, we describe techniques for refining the estimate of the number of *U* the query will require for its execution. The number of *U* required by a query is the sum of the number of *U* required by all the segments in the query plan. In the remainder of this section we refer to this number as the "cost of the query." Similarly, we call the number of *U* it takes to execute a segment the "cost of the segment." In Section 4.6 we will turn to the issue of estimating the conversion factor from *U* to time.

As mentioned in Section 3, we want to update the display on the progress indicators as smoothly as possible. Hence, we need to continuously refine the estimates of the segment costs. For segments that have finished execution, we know the exact costs. Therefore, we only need to focus on the cost of the segment that is currently being executed and the costs of the future segments that have not started execution.

In the following, we first show how to compute the cost of a segment. Then we give an overview of the refining procedure for query cost estimation. Finally, we describe the refining procedure for query cost estimation in detail.

Computing the Cost of a Segment

As mentioned in Section 4.2, each segment contains one or more steps that are executed in a pipeline. Recall that in our techniques we only monitor bytes processed at the boundaries of segments. This means we only need to consider the inputs to the segment and the final output.

Intuitively, a byte coming from a segment input is counted once as it is input into that segment. A byte produced by a segment is counted once as it is output by that segment (except when the segment output is the final output that is displayed to the user), and again as it is input by the next segment. If the intermediate result is indeed materialized to disk, this "double counting" corresponds to the cost of the byte being written to disk and then read back in. If at runtime this intermediate result actually ends up being buffered in memory, this double counting corresponds roughly to the cost of the byte being handled at the output of the lower segment and then again at the input of the next segment.

A special case arises if an operator at the leaves or root of a segment is a multi-stage operator (for example, a multi-stage partition operator for a hash join, or a multi-stage sort). For such operators, bytes handled by the operator will be counted once each time they are logically read or written.

The reader may wonder if computing costs only at segment boundaries is a good idea, since for deep pipelines this approach ignores a lot of computation within the pipeline. While it is true that our approach does not explicitly account for computation in the pipeline, this computation is implicitly considered because it impacts the speed with which a segment consumes its input. It is an interesting open question whether in general progress indicators can benefit from explicitly accounting for costs within pipelines.

Overview of the Refining Procedure for Query Cost Estimation

As the current segment is being processed, we continuously refine the estimates for its output cardinality, its average output tuple size, and the total *U* it will consume (we describe precisely how we do this later in this section.) We propagate the improved estimates for the current segment upwards in the query plan to the next segment. Then we refine the estimates of the output cardinality, average output tuple size, and *U* for the next segment. Recall that in our progress estimator, *U* is just the number of bytes processed by the segment. So the question arises: how can we compute the expected *U* for a future segment?

Fortunately, we can compute the expected *U* for a future segment by invoking the optimizer's cost estimation module with the improved estimates of output cardinality and output size for the current segment (and the existing estimates for any other inputs to the future segment, if it is a multiple-input segment – the estimates for these segment inputs are not being refined if they are not from the current segment.) Because the optimizer gives a number of I/Os in its estimate, we can convert this to bytes simply by multiplying the estimate by the page size.

We continue this propagation of estimates and recalculation of costs until we reach the top of the query plan. Then we use the exact costs of the past segments, the improved cost estimate of the current segment, and the improved cost estimates of the future segments, to refine the estimated query cost.

For example, consider the query plan example shown in Figure 3. Suppose the current segment is S_2. We continuously use the improved estimates related to segment S_2 to refine the estimates related to segments S_3 and S_5. The improved estimates related to segment S_2 will not influence the estimates related to segments S_4. Hence, for segments S_4, we use the original estimates provided by the optimizer. (The optimizer's estimates can be kept in the query plan using the annotated query plan technique in [14].)

From the above description, we can see that the key step of refining the query cost estimate is refining the estimates related to the current segment. Hence, we now turn to discuss how we refine the estimates of output cardinality, size, and *U* for the current segment.

Refining the Estimates Related to the Current Segment

Estimating the average output tuple size is easy: at any time, we use the average output tuple size computed so far as the estimated average tuple size of the final output. Since we are using bytes processed as *U*, the *U* required by the current segment is just the product of its estimated cardinalities and average tuple sizes of its inputs and the output. We have shown how to get the estimated cardinalities and average tuple sizes of the inputs in Section 4.3. Hence, in the following, we focus on estimating the output cardinality. We first introduce the concept of dominant inputs, which we use to enable an approximate indication of how far along the current segment is in processing its inputs.

For each segment, we define one or two dominant inputs. As mentioned in Section 4.2, each segment can be viewed as a tree. The leaves of the tree are the inputs of the segment. Among all the inputs of a segment, we choose a dominant input so that once all the tuples in the dominant input have been processed, the entire segment finishes execution. There is an exception: for a segment that contains a sort-merge join operator, we define two dominant inputs. In more detail,

(1) If a segment contains only one input, this input is defined as the dominant input.

(2) If a segment contains multiple inputs, this segment must contain at least one join operator. If this segment contains multiple join operators, we find the join operator at the lowest level of the segment. There are several possible cases for this join operator:

   (a) If it is a nested loops join operator, we define the dominant input to be the input of the segment that is a left descendant of the nested loops join operator (the outer relation [20]).

   (b) If it is a hash join operator, we define the dominant input to be the input of the segment that is a right descendant of the hash join operator (the probe relation).

   (c) If it is a sort-merge join operator, we define the dominant inputs to be the two inputs of the segment that are descendants of the sort-merge join operator.

As an example, consider the query plan example shown in Figure 3. We list the dominant inputs of the segments as follows:

(1) Segment S_1: *A*.
(2) Segment S_2: *B*.
(3) Segment S_3: P_B.
(4) Segment S_4: *C*.
(5) Segment S_5: R_{AB} and R_C.

Next we turn to discuss how to use the percentage of the dominant input that has been processed so far to refine the estimated output cardinality. We first discuss the case that the current segment contains one dominant input. Then we discuss the case that the current segment contains two dominant inputs.

At the time that the current segment starts execution, we give an initial estimate E_1 of its output cardinality. E_1 is computed using the input cardinalities of the current segment and the optimizer's cost estimation module. This estimate may of course be wrong; our goal is to detect this while the segment is running, and gradually replace it with an estimate that approaches the true output cardinality as the execution of the segment nears completion. We do this as follows.

Suppose that the dominant input cardinality of the current segment is *z*. Assume that so far, we have processed *x* of *z* and generated *y* output tuples. Then the percentage that the dominant input has been processed is $p=x/z$. If we assume that at any time, the number of output tuples that have been generated is proportional to the percentage that the dominant input has been processed, then we can estimate the final output cardinality of the current segment to be $E_2=y/p=yz/x$. In practice, this assumption may not be valid and we also want to consider the initial estimate E_1.

At any time, we use the following heuristic formula to estimate the final output cardinality *E* of the current segment: $E=p\times E_2+(1-p)\times E_1$. This heuristic formula intends to smooth fluctuations in the estimator and to let it gradually change from the initial estimate (when the current segment just starts execution, we know nothing about the actual segment output cardinality) to the actual segment output cardinality (when the current segment finishes execution, we know this quantity exactly).

Recall that a segment containing a sort-merge join operator has two dominant inputs. In this case, once we reach the end of either dominant input, the sort-merge join (and thus the segment) immediately finishes execution. Therefore, we need to use the dominant input that is being scanned relatively faster to decide the percentage *p* that the two dominant inputs have been processed [21].

We use an example to illustrate the procedure. Consider a sort-merge join operator with two input relations *A* and *B*. We assume that both *A* and *B* have already been sorted. Suppose that the cardinality of *A* is |*A*|, and the cardinality of *B* is |*B*|. Suppose that we have processed *x* tuples from *A* and *y* tuples from *B*. Let $q_A=x/|A|$ and $q_B=y/|B|$. Then we use the following formula to decide *p*: $p=max(q_A, q_B)$.

4.6 Monitoring Current and Predicting Future Query Execution Speed

Recall that our progress indicator depends on two things: the estimates of *U*, and the estimated conversion factor between *U* and time. The conversion of *U* to time should reflect what we are observing as the system is running. So, at all times, we keep track of the amount of work (measured in *U*'s) that has been done for query *Q* in the last *T* seconds, where *T* is a pre-defined number. The average speed that the work has been done for query *Q* in the last *T* seconds is used as the estimated current execution speed of query *Q*. To minimize the influence of temporary fluctuations, this *T* should not be too small. However, this *T* should also not be too large. Otherwise, the calculated execution speed will not closely reflect the actual current execution speed. In our implementation, we choose *T* to be 10. In our experiments, we found that this number is sufficient to provide a smooth estimate of the current query execution speed.

This approach to calculating the conversion from *U* to time is admittedly simplistic, and although it worked well in our experiments, there are cases in which it will be misleading.

One situation in which this approach is misleading is when the system load fluctuates substantially. At times of high load, the progress indicator will overestimate the execution time, since it will think that each *U* takes a relatively long time to process. At times of light load, it will underestimate the execution time for analogous reasons. There is not much that can be done about this – it is the same situation as the one that occurs during a file download, when varying available bandwidth causes the estimated download time to be inaccurate. One possible improvement to our approach would be to incorporate some history beyond *T* in order to "smooth" the estimates (e.g., perhaps computing a decaying average, so that while the most recent execution speed has the major impact, the overall execution speed also has an impact.)

The second situation in which our simple conversion from *U* to time could be misleading occurs when segments have radically different characteristics. In particular, a problem arises when one segment can be expected to process *U* much more quickly than another. For example, consider a two-segment plan, in which segment S_1 feeds segment S_2. If S_1 processes *U* more slowly than S_2 (perhaps S_1 is I/O-intensive whereas S_2 has a high buffer pool hit rate), then while S_1 runs it will overestimate the time it will take to run S_2. (Using our simple conversion approach, the progress indicator will eventually figure this out and improve its estimate - in this simple two-segment example, it will adjust once S_2 starts running.) This problem could be alleviated by a more complex conversion from *U* to time – ideally this conversion should take into account both the expected processing speed for the segments and the current system load. While space limitations precluded us from exploring such complex conversions between *U* and time in this paper, we think this is an interesting and promising area for future work.

5. Performance

In this section, we present results from a prototype implementation of progress indicators in PostgreSQL Version 7.3.4 [19]. We implemented all the techniques described in Section 4, except for those techniques that are used to handle sort-merge join. In all our tests, our prototyped progress indicators could be updated every ten seconds with less than 1% overhead, which we consider to have met the three goals mentioned in Section 3: continuously revised estimates, acceptable pacing, and minimal overhead.

5.1 Experiment Description

Our measurements were performed with the PostgreSQL client application and server running on a Dell Inspiron 4000 PC with one 600MHz processor, 512MB main memory, one 40GB IDE disk, and running the Microsoft Windows XP operating system. (We repeated some of the experiments on a computer with a 2.4GHz processor, 512MB main memory, and one 73GB SCSI disk. The results were similar, so we omit them here.)

The five relations used for the tests followed the schema of the standard TPC-R Benchmark relations [22]:

customer (custkey, name, address, nationkey, phone, acctbal, mktsegment),

orders (orderkey, custkey, orderstatus, totalprice, orderdate, ship-priority),

lineitem (orderkey, partkey, suppkey, linenumber, quantity, extendedprice, discount, tax, returnflag, linestatus).

The *customer_subset1* relation and the *customer_subset2* relation have the same schema as the *customer* relation while either of them contains only 3K tuples. In our tests, on average, each *customer* tuple matches ten *orders* tuples on the attribute *custkey*. Each *orders* tuple matches 4 *lineitem* tuples on the attribute *orderkey*.

Table 1. Test data set.

	number of tuples	total size
customer	0.15M	23MB
orders	1.5M	114MB
lineitem	6M	755MB
customer_subset1	3K	0.46MB
customer_subset2	3K	0.46MB

We evaluated the performance of progress indicators in the following way:

(1) Before we ran queries, we ran the PostgreSQL statistics collection program on all the five relations.

(2) We performed three kinds of tests:

   (a) **Unloaded system test**: We ran the whole query on an unloaded system.

   (b) **I/O interference test**: We started executing the query on an unloaded system. In the middle of query execution, we started a large file copy.

   (c) **CPU interference test**: We started executing the query on an unloaded system. In the middle of query execution, we started a CPU-intensive program.

(3) We tested five queries:

   (a) **Query Q_1**:

```
select * from lineitem;
```

(b) **Query Q_2**:

```
select c.custkey, c.acctbal, o.orderkey, o.totalprice,
        l.discount, l.extendedprice
from customer c, orders o, lineitem l
where c.custkey=o.custkey and o.orderkey=l.orderkey and
        absolute(l.partkey)>0;
```

(c) **Query Q_3**:

```
select c.custkey, c.acctbal, o1.orderkey, o1.totalprice,
        o2.totalprice
from customer c, orders o1, orders o2
where c.custkey=o1.custkey and o1.orderkey=o2.orderkey
        and c.nationkey<10;
```

(d) **Query Q_4**:

```
select c.custkey, c.acctbal, o.orderkey, o.totalprice,
        o.shippriority, l.discount, l.extendedprice
from customer c, orders o, lineitem l
where c.custkey=o.custkey and o.orderkey=l.orderkey and
        absolute(o.totalprice)>0 and absolute(l.partkey)>0;
```

(e) **Query Q_5**:

```
select *
from customer_subset1 c1, customer_subset2 c2
where c1.custkey<>c2.custkey;
```

For query Q_2, we report the test results for both the unloaded system test and the I/O interference test. However, for queries Q_1, Q_3 and Q_4, we only report the test results for the unloaded system test. This is because for the I/O interference test, the test results for queries Q_1, Q_3 and Q_4 are similar to those for query Q_2 and do not provide much extra information. For query Q_5, we report the test results for both the unloaded system test and the CPU interference test.

(4) Before we ran each test, we restarted the computer to ensure a cold buffer pool. (We repeated our experiments with a warm buffer pool. The results were similar, so we do not present them here.) In all tests, we stored the outputs from progress indicators into a file.

5.2 Test Results for Query *Q₁*

The purpose of the test with query Q_1 is to show that for simple queries, the optimizer's estimates can be fairly precise for an unloaded system.

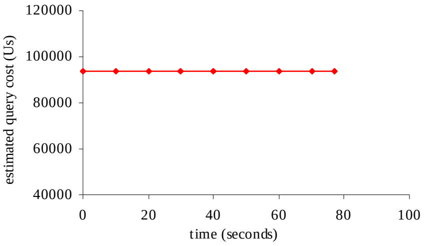


Figure 4. Query cost estimated over time (unloaded system test for Q_1).

Figure 4 shows the query cost estimated by the progress indicator over time. The curve that represents the query cost estimated by the progress indicator is almost a straight line. That is, during the entire query execution, the progress indicator estimates the cost of query Q_1 fairly precisely. This is because query Q_1 is a table-scan query on the *lineitem* relation. Due to statistics collection, PostgreSQL has accurate knowledge of the size of the *lineitem* relation.

Figure 5 shows the query execution speed monitored by the progress indicator over time. Query Q_1 is a table-scan query that only performs sequential I/O. Also, it is the only query that runs in the system. Hence, during the entire query execution period, the monitored query execution speed is quite stable.

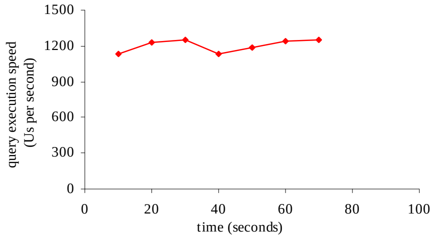


Figure 5. Query execution speed over time (unloaded system test for Q_1).

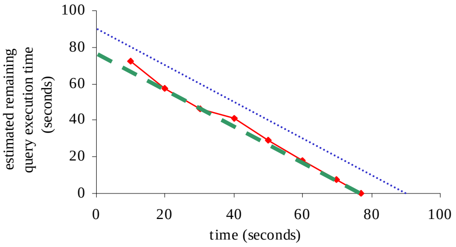


Figure 6. Remaining query execution time estimated over time (unloaded system test for Q_1).

Figure 6 shows the remaining query execution time estimated by the progress indicator over time. Besides the curve representing the remaining query execution time estimated by the progress indicator, there are two lines in Figure 6:

(1) The first line is a dashed line that shows the actual remaining query execution time over time. This dashed line almost coincides with the curve that represents the remaining query execution time estimated by the progress indicator. That is, during the entire query execution period, the remaining query execution time that is estimated by the progress indicator is fairly precise. This is because during the entire query execution period, (1) the progress indicator estimates the cost of query Q_1 fairly precisely, and (2) the query execution speed is quite stable.

(2) The second line is a dotted line that shows the optimizer's "estimate" of the remaining query execution time over time. We use the optimizer's estimate of the query running time to predict the remaining query execution time. (The optimizer's estimate of the query running time is computed as the optimizer's estimated number of I/Os for the query divided by the optimizer's estimate of the system's disk I/O speed.) We can see that compared to the dotted line, the curve that represents the remaining query execution time estimated by the progress indicator is "closer" to the dashed line. That is, the progress indicator is better than the optimizer in estimating the remaining query execution time. Also, the dotted line is not far from the dashed line. This is because for query Q_1, the optimizer's estimate of the number of I/Os is fairly precise while the optimizer's estimate of the disk I/O speed is a little bit different from the monitored query execution speed.

Figure 7 shows the progress indicator's estimate of the percentage of the query that has been completed over time. The completed percentage curve is fairly close to a straight line, as work is continuously being done at a rather steady speed.

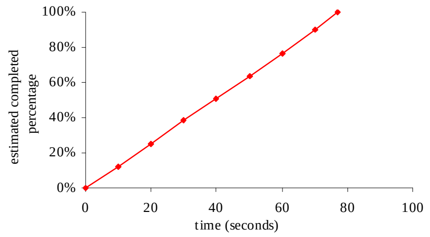


Figure 7. Completed percentage estimated over time (unloaded system test for Q_1).

5.3 Test Results for Query Q_2

The purpose of the tests with query Q_2 is to show how our progress indicator adjusts to the optimizer's estimation errors that result from complex query plans. Query Q_2 contains two joins:

(1) A join of the *customer* relation and the *orders* relation on the join attribute *custkey*.

(2) A join of the *orders* relation and the *lineitem* relation on the join attribute *orderkey*.

The query plan chosen by PostgreSQL is shown in Figure 8. In the rest of Section 5.3, we refer to the hybrid hash join between the *customer* relation and the *orders* relation as the first hybrid hash join. We refer to the hybrid hash join between the intermediate result that is generated by the first hybrid hash join and the *lineitem* relation as the second hybrid hash join.

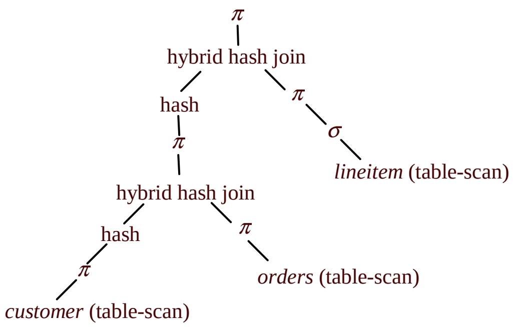


Figure 8. Query plan chosen by PostgreSQL.

We first discuss the test results from the unloaded system test in Section 5.3.1. Then we discuss the test results from the I/O interference test in Section 5.3.2.

5.3.1 *Unloaded System Test Results for Query Q_2*

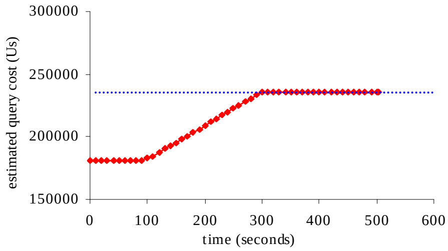


Figure 9. Query cost estimated over time (unloaded system test for Q_2).

Figure 9 shows the query cost estimated by the progress indicator over time, with the exact query cost indicated by the horizontal dotted line. At the beginning of query execution, the query cost estimated by the progress indicator remains as a constant that is far different from the exact query cost. Starting from 95 seconds, the query cost estimated by the progress indicator begins to increase and keeps approaching the exact query cost. This trend continues until 300 seconds, when the query cost estimated by the progress indicator reaches the exact query cost. From then on until the query completion time, the query cost estimated by the progress indicator remains as a second constant (the exact query cost).

We explain this behavior as follows:

(1) Due to statistics collection, PostgreSQL knows both the cardinalities and the sizes of the three relations used in query Q_2. Since all tuples in both the *customer* relation and the *orders* relation participate in the first hybrid hash join, PostgreSQL estimates the cost of the first hybrid hash join fairly precisely.

(2) The join between the *customer* relation and the *orders* relation is a key-foreign key join. Also, all tuples in both the *customer* relation and the *orders* relation participate in this join. Hence, PostgreSQL estimates both the cardinality and the size of the intermediate result that is generated by the first hybrid hash join fairly precisely.

(3) PostgreSQL does not give a good estimate of the selectivity of the select condition *absolute(l.partkey)>0* on the *lineitem* relation. Rather, for this select condition, PostgreSQL uses a default value 1/3 as an approximation to the real selectivity. This approximation is far from the real selectivity, which is 1 (since the absolute value of *l.partkey* is always positive). Hence, PostgreSQL gives a rather imprecise cardinality estimate for the intermediate result of the selection on the *lineitem* relation. As a result, PostgreSQL gives a rather imprecise cost estimate for the second hybrid hash join.

(4) Before 95 seconds, PostgreSQL is working on the first hybrid hash join. During this period, the progress indicator does not change the query cost estimate that is provided by PostgreSQL, even if it is far different from the exact query cost. This is because:

   (a) PostgreSQL estimates the cost of the first hybrid hash join fairly precisely.

   (b) During this period, the progress indicator does not change the cost estimate for the second hybrid hash join, as the progress indicator does not see anything during the execution that would cause it to change the estimates for the inputs to the second hybrid hash join.

(5) Between 95 seconds and 300 seconds, PostgreSQL is working on the selection on the *lineitem* relation. During this period, the progress indicator begins to detect that PostgreSQL was wrong in its cardinality estimate for the intermediate result of the selection on the *lineitem* relation. As a side effect, the progress indicator also continuously refines the cost estimate for the second hybrid hash join. As a result, the query cost estimated by the progress indicator keeps approaching the exact query cost.

(6) Between 300 seconds and query completion time, PostgreSQL is working on the second hybrid hash join. During this period, the cardinalities of both inputs to the second hybrid hash join are known exactly so the progress indicator can make accurate predictions.

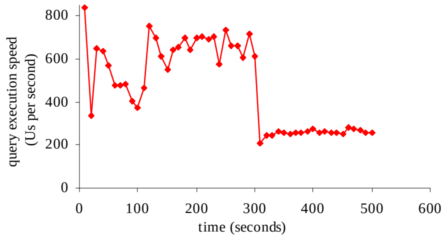


Figure 10. Query execution speed over time (unloaded system test for Q_2).

Figure 10 shows the query execution speed monitored by the progress indicator over time. During query execution, the monitored query execution speed fluctuates. This is mainly due to the following reasons:

(1) The system performance has some random fluctuations over time.

(2) The entire query execution is composed of multiple stages (e.g., sequentially scanning a relation, probing a hash table). Different stages have different performance characteristics. For example, during some stages we mainly perform sequential I/Os while during some other stages we mainly perform random I/Os. Also, during some stages we mainly perform I/O-intensive operations while during some other stages we mainly perform CPU-intensive operations.

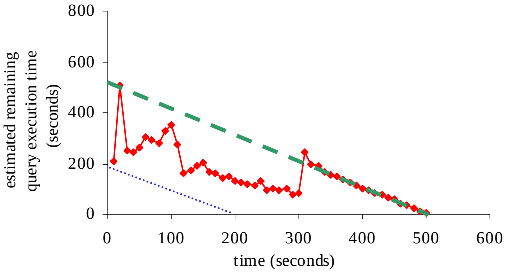


Figure 11. Remaining query execution time estimated over time (unloaded system test for Q_2).

Figure 11 shows the remaining query execution time estimated by the progress indicator over time. The closer to query completion time, the more precise the remaining query execution time estimated by the progress indicator. This is because the closer to query completion time, the more precise the query cost estimated by the progress indicator.

Like Figure 6, Figure 11 contains:

(1) a dashed line that shows the actual remaining query execution time over time.

(2) a dotted line that shows the optimizer's "estimate" of the remaining query execution time over time.

After 340 seconds, the dashed line almost coincides with the curve that represents the remaining query execution time estimated by the progress indicator. That is, after 340 seconds, the remaining query execution time that is estimated by the progress indicator is fairly precise.

Compared to the dotted line, the curve that represents the remaining query execution time estimated by the progress indicator is much "closer" to the dashed line. That is, the progress indicator is much better than the optimizer in estimating the remaining query execution time.

Figure 12 shows the progress indicator's estimate of the percentage of the query that has been completed over time. This percentage keeps increasing with time, as work is continuously being done.

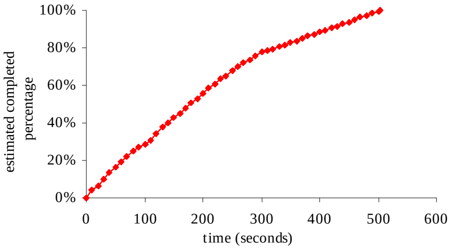


Figure 12. Completed percentage estimated over time (unloaded system test for Q_2).

As mentioned above, after 300 seconds, the progress indicator's estimate of the query cost remains as a constant. Also, as shown in Figure 10, after 300 seconds, the query execution speed does not change much. Hence, after 300 seconds, the slope of the completed percentage curve remains almost as a constant.

5.3.2 *I/O Interference Test Results for Query Q_2*

In the I/O interference test, we started executing the query on an unloaded system. At 190 seconds, we started a large file copy that ran until 885 seconds. While it was running, this file copy made the system heavily loaded and significantly decreased the execution speed of the query. Hence, the query execution time increased from 510 seconds to 1027 seconds.

In each figure of Section 5.3.2, we use two vertical dashed-dotted lines, one representing the start of the file copy, and another representing the end of the file copy.

Figure 13 shows the query cost estimated by the progress indicator over time, with the exact query cost indicated by the horizontal dotted line. We were initially perplexed by the shape of the curve in Figure 13. Why doesn't it match that of Figure 9? After all, a concurrently running job should not impact the estimate of the number of *U* a query will take, yet in Figure 13 the start of the file copy is clearly visible. After more reflection, we realized that this makes sense. At 190 seconds, when the file copy starts, the progress indicator is "learning" that the optimizer's estimates were wrong. It does so by watching the generation of intermediate results. When the file copy starts, the rate at which intermediate results are generated decreases, so the progress indicator begins "learning" more slowly, hence the decrease in the slope of the curve.

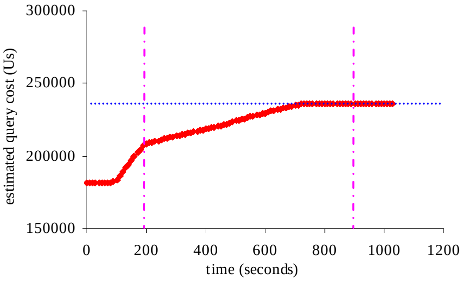


Figure 13. Query cost estimated over time (I/O interference test for Q_2).

Figure 14 shows the query execution speed monitored by the progress indicator over time. Before 190 seconds (before the file copy starts running), the shape of the curve in Figure 14 is similar to that in Figure 10. However, once the file copy starts, the query execution speed is decreased. This situation continues until 885 seconds, when the file copy finishes. After 885 seconds, the query execution speed again returns to that seen in the unloaded system test.

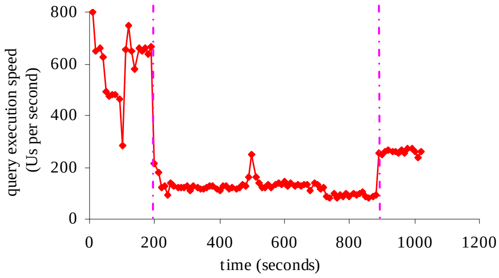


Figure 14. Query execution speed over time (I/O interference test for Q_2).

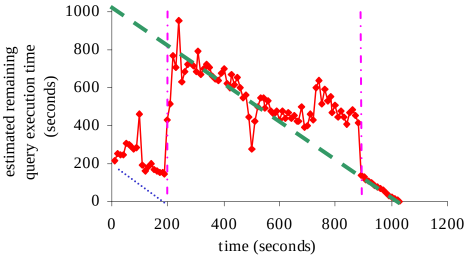


Figure 15. Remaining query execution time estimated over time (I/O interference test for Q_2).

Figure 15 shows the remaining query execution time estimated by the progress indicator over time. Before 190 seconds (i.e., before the file copy starts running), the shape of the curve in Figure 15 is similar to that in Figure 11. At 190 seconds, due to the start of the file copy, the remaining query execution time estimated by the progress indicator increases sharply. At 885 seconds, when the large file copy finishes, the remaining query execution time estimated by the progress indicator drops significantly.

Like Figure 6, Figure 15 contains:

(1) a dashed line that shows the actual remaining query execution time over time.

(2) a dotted line that shows the optimizer's "estimate" of the remaining query execution time over time.

The general trend shown in Figure 15 is similar to that shown in Figure 11:

(a) After 895 seconds, the dashed line almost coincides with the curve that represents the remaining query execution time estimated by the progress indicator. That is, after 895 seconds, the remaining query execution time that is estimated by the progress indicator is fairly precise.

(b) Compared to the dotted line, the curve that represents the remaining query execution time estimated by the progress indicator is much "closer" to the dashed line. That is, the progress indicator is much better than the optimizer in estimating the remaining query execution time.

Figure 16 shows the progress indicator's estimate of the percentage of the query that has been completed over time. In general, this percentage keeps increasing with time, as work is continuously being done. Again, the impact of the file copy is apparent between 190 seconds and 885 seconds.

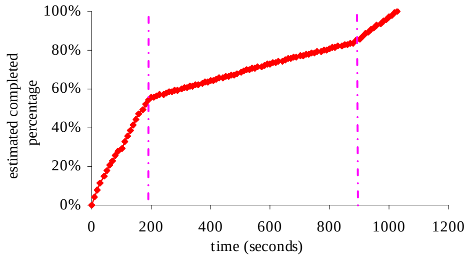


Figure 16. Completed percentage estimated over time (I/O interference test for Q_2).

5.4 Test Results for Query Q_3

The purpose of the test with query Q_3 is to show how the progress indicator handles the optimizer's estimation errors that occur due to correlation. In this test, we changed the data in the *orders* relation, so that on average, each *customer* tuple matches *r orders* tuples on the attribute *custkey*, where:

(a) *r=20* if the *nationkey* attribute value of the *customer* tuple is between 0 and 9.

(b) *r=0* if the *nationkey* attribute value of the *customer* tuple is between 10 and 19.

(c) *r=10* if the *nationkey* attribute value of the *customer* tuple is between 20 and 24.

Hence, in query Q_3, there is correlation in the *customer* relation between the *nationkey* attribute and the number of orders that the customer has made.

Query Q_3 contains two joins. The first join is between the *customer* relation and *o1*. The second join is between the join result generated by the first join and *o2*. Because of the correlations in the data, PostgreSQL's optimizer does not give a precise cardinality estimate for the first join.

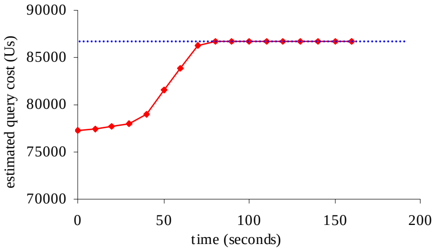


Figure 17. Query cost estimated over time (unloaded system test for Q_3).

Figure 17 shows the query cost estimated by the progress indicator over time, with the exact query cost indicated by the horizontal dotted line. At the beginning of query execution, the query cost estimated by the progress indicator, which comes from the optimizer, is far different from the exact query cost. Starting from 10 seconds, the progress indicator begins to detect that the optimizer was wrong, and its cost estimate begins to increase and approaches the exact query cost. This trend continues until 80 seconds, when the query cost estimated by the progress indicator reaches the exact query cost. From then on until the query completion time, the query cost estimated by the progress indicator remains as a constant (the exact query cost).

5.5 Test Results for Query Q_4

The purpose of the test with query Q_4 is to show how the progress indicator adjusts to the optimizer's estimation errors that grow with the number of joins in a query. Query Q_4 contains two joins. The first join is between the *customer* relation and the *orders* relation. The second join is between the result of the first join and the *lineitem* relation.

Query Q_4 contains two select conditions: *absolute(o.totalprice)>0* on the *orders* relation, and *absolute(l.partkey)>0* on the *lineitem* relation. Due to the same reason as explained in Section 5.3.1 for Figure 9, PostgreSQL's optimizer gives a rather imprecise selectivity estimate for both select conditions.

Due to errors in the selectivity estimate for the select condition *absolute(o.totalprice)>0*, PostgreSQL's cost estimate of the first join is imprecise. Due to errors in the selectivity estimate for the select condition *absolute(l.partkey)>0*, PostgreSQL's cost estimate of the second join is also rather imprecise.

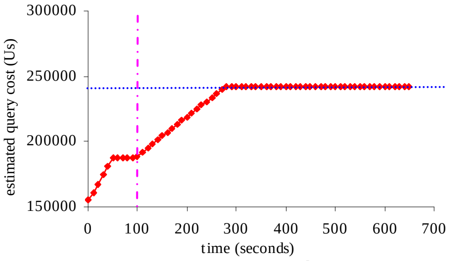


Figure 18. Query cost estimated over time (unloaded system test for Q_4).

Figure 18 shows the query cost estimated by the progress indicator over time, with the exact query cost indicated by the horizontal dotted line. The vertical dashed-dotted line represents the time when the first join finishes and the second join starts. We can see that the progress indicator makes adjusts to both optimizer estimation errors twice as the query is being processed: first, while the first join is running; second, during the second join.

5.6 Test Results for Query Q_5

In all the above tests, the four queries used (Q_1, Q_2, Q_3 and Q_4) were primarily I/O-intensive. In this section, we discuss the test results for query Q_5, which is CPU-intensive. We first discuss the test results from the unloaded system test in Section 5.6.1. Then we discuss the test results from the CPU interference test in Section 5.6.2.

5.6.1 Unloaded System Test Results for Query Q_5

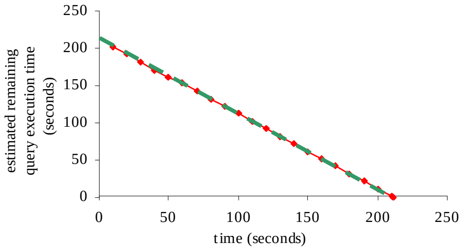


Figure 19. Remaining query execution time estimated over time (unloaded system test for Q_5).

Figure 19 shows the remaining query execution time estimated by the progress indicator over time, with the actual remaining query execution time indicated by the dashed line. We see that even for this CPU-bound query, calculating the progress of the query using bytes consumed gives good results. The query plan chosen was nested loops join, so in this case the progress indicator is really measuring progress through the "dominant input" (in this case the outer relation of the nested loops join.)

5.6.2 CPU Interference Test Results for Query Q_5

In this section, we discuss the test results from the CPU interference test for query Q_5. In the CPU interference test, we started executing the query on an unloaded system. At 120 seconds, we started a CPU-intensive program that kept running until the query finished execution. During its execution, this CPU-intensive program made the system heavily loaded and significantly decreased the execution speed of the query. Hence, the query execution time increased from 211 seconds to 463 seconds.

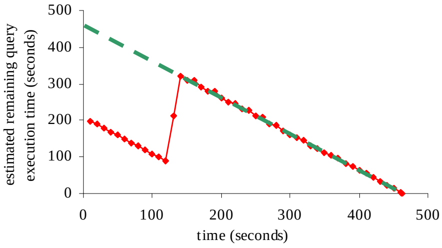


Figure 20. Remaining query execution time estimated over time (CPU interference test for Q_5).

Figure 20 shows the remaining query execution time estimated by the progress indicator over time, with the actual remaining query execution time indicated by the dashed line. At 120 seconds, when the CPU-intensive program starts execution, the progress indicator "notices" that the query execution has slowed down, and the remaining query execution time estimated by the progress indicator increases significantly. Starting from 140 seconds, the curve that represents the remaining query execution time estimated by the progress indicator almost coincides with the dashed line. That is, starting from 140 seconds, the remaining query execution time estimated by the progress indicator becomes fairly precise.

6. Conclusion

In this paper, we have proposed techniques for supporting progress indicators for RDBMS queries. Our main idea is that as a long-running query is being processed, we continuously refine the query cost estimate and monitor the current query execution speed. Then we continuously give the user an estimate of both the percentage of the query that has been completed and the remaining query execution time. Our experiments show that a progress indicator based upon our techniques is useful both for I/O-intensive and CPU-intensive queries, and that it adapts both to the optimizer's estimation errors and to varying runtime system loads.

There are several interesting directions that we are considering for future work:

(1) The estimation techniques described in this paper are fairly coarse. For example, while measuring bytes processed is a reasonable proxy for actually calculating CPU and I/O utilization, it is certainly not exact. It is a non-trivial task to make the estimates more precise by a more detailed consideration of these factors, and it would be interesting to see if the resulting increase in accuracy can be obtained at a reasonable overhead, and also if this increase in accuracy is significant from the user's perspective.

(2) The estimation techniques described in this paper do not make use of detailed statistics about intermediate results (for example, they consider only the sizes of the intermediate results, not histograms on their distributions.) Again, it would be interesting to see if the effort required to collect and use better statistics pays off in terms of user satisfaction.

(3) It would be interesting to extend our techniques in order to support wider classes of queries (one interesting such challenge is how to handle correlated subqueries) and to support systems with more options for query evaluation.

(4) As mentioned in Section 4, we do not explicitly count the CPU cost due to operators internal to segments. These costs are implicitly counted in that they slow the progress of the segment in consuming its input. It is an interesting open question whether and when progress indicators could be improved by "looking inside" the pipelined segments.

Progress indicators have other potential uses besides providing a user-friendly interface. For example, progress indicators can be useful for:

(1) **Load management**. Suppose that due to some reason (e.g., to speed up the execution of a certain query), the DBA decides to choose several queries from a pool of running queries and blocks their execution for a while. A progress indicator can help the DBA choose which queries to block.

(2) **Automatic administration**. The user may embed triggers in a progress indicator to facilitate automatic database administration. For example, the firing condition of such a trigger can be: "send an email to the user if after a whole day's execution, the query finishes less than 10% of the work." This trigger function can be achieved by keeping track of the history of the progress indicator.

(3) **Performance tuning**. By keeping track of the history of a progress indicator, we can see whether the originally estimated query cost is precise enough and where time goes during query execution. Such information can help us discover the performance bottleneck. Then we can decide whether we need to improve the query plan by keeping statistics up-to-date, using a higher level of optimization, or designing a better database schema.

It would be interesting to experiment with these and other uses of progress indicators in RDBMSs.

Acknowledgements

We would like to thank Yibin Pan for useful discussions. This work was supported by the NCR Corporation and also by NSF grants CDA-9623632 and ITR 0086002.

References

[1] A. Aboulnaga, S. Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. SIGMOD Conf. 1999: 181-192.

[2] G. Antoshenkov. Dynamic Query Optimization in Rdb/VMS. ICDE 1993: 538-547.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A Multidimensional Workload-Aware Histogram. SIGMOD Conf. 2001: 211-222.

[4] D.A. Berque, M.K. Goldberg. Monitoring an Algorithm's Execution. Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 15, pp. 153-163, 1992.

[5] R.L. Cole, G. Graefe. Optimization of Dynamic Query Evaluation Plans. SIGMOD Conf. 1994: 150-160.

[6] C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. PODS 1995: 255-265.

[7] DB2. SQL/Monitoring Facility. http://www.sprdb2.com /SQLMFVSE.PDF, 2000.

[8] M. Dempsey. Monitoring Active Queries with Teradata Manager 5.0. http://www.teradataforum.com /attachments/a030318c.doc, 2001.

[9] M.A. Derr. Adaptive Query Optimization in a Deductive Database System. CIKM 1993: 206-215.

[10] P.J. Haas, J.M. Hellerstein. Ripple Joins for Online Aggregation. SIGMOD Conf. 1999: 287-298.

[11] J.M. Hellerstein, P.J. Haas, and H. J. Wang. Online Aggregation. SIGMOD Conf. 1997: 171-182.

[12] Y.E. Ioannidis, R.T. Ng, and K. Shim et al. Parametric Query Optimization. VLDB Journal 6(2): 132-151, 1997.

[13] I.F. Ilyas, J. Rao, and G.M. Lohman et al. Estimating Compilation Time of a Query Optimizer. SIGMOD Conf. 2003: 373-384.

[14] N. Kabra, D.J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. SIGMOD Conf. 1998: 106-117.

[15] U. Larry. Monitoring Rollback Progress. http://www.interealm.com/technotes/larry/rollback_time.html, 2002.

[16] B.A. Myers. The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces. SIGCHI 1985: 11-17.

[17] K.W. Ng, Z. Wang, and R.R. Muntz et al. Dynamic Query Re-Optimization. SSDBM 1999: 264-273.

[18] Oracle. Communication with Oracle during long-running query. http://www.experts-exchange.com/Databases /Oracle/Q_20675711.html, 2003.

[19] PostgreSQL homepage, 2003. http://www.postgresql.org.

[20] R. Ramakrishnan, J.E. Gehrke. Database Management Systems, Third Edition. McGraw-Hill, 2002.

[21] M. Stillger, G.M. Lohman, and V. Markl et al. LEO - DB2's LEarning Optimizer. VLDB 2001: 19-28.

[22] TPC Homepage. TPC-R benchmark, www.tpc.org.