2012 IEEE 28th International Conference on Data Engineering

GSLPI: a Cost-based Query Progress Indicator

Jiexing Li #1, Rimma V. Nehme *2, Jeffrey Naughton #3

#Computer Sciences, University of Wisconsin – Madison
[1]jxli@cs.wisc.edu
[3]naughton@cs.wisc.edu

*Microsoft Jim Gray Systems Lab
[2]rimman@microsoft.com

***Abstract*—Progress indicators for SQL queries were first published in 2004 with the simultaneous and independent proposals from Chaudhuri et al. and Luo et al. In this paper, we implement both progress indicators in the same commercial RDBMS to investigate their performance. We summarize common cases in which they are both accurate and cases in which they fail to provide reliable estimates. Although there are differences in their performance, much more striking is the similarity in the errors they make due to a common simplifying uniform future speed assumption. While the developers of these progress indicators were aware that this assumption could cause errors, they neither explored how large the errors might be nor did they investigate the feasibility of removing the assumption. To rectify this we propose a new query progress indicator, similar to these early progress indicators but without the uniform speed assumption. Experiments show that on the TPC-H benchmark, on queries for which the original progress indicators have errors up to 30X the query running time, the new progress indicator is accurate to within 10 percent. We also discuss the sources of the errors that still remain and shed some light on what would need to be done to eliminate them.**

I. Introduction

Many modern software systems provide progress indicators (PIs) for long-running tasks (e.g., file downloads and software installations). Typically, a progress indicator estimates how much of the task has been completed and when the task will finish. Figure 1 shows an example of a progress indicator that we are trying to develop for database queries. It continuously updates the elapsed time, estimated remaining time and percentage of completion for a given query.

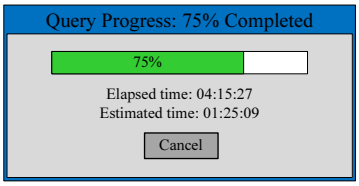


Fig. 1. Estimated remaining time

Most commercial database vendors provide tools for monitoring queries (e.g., DB2 [1], Teradata [2], Microsoft SQL Server [3], and Oracle [4]). However, none of the existing tools in commercial DBMSs provides accurate progress estimates at a fine granularity. The first progress indicates, the MSRPI [5] and the WiscPI [6][1], were first published by two different research institutes independently in the same conference. The variants [7], [8], [9], [10] of the MSRPI and the WiscPI, were proposed later to explore issues such as broadening the class of queries handled, or investigating the interaction between concurrent queries, or reducing cardinality estimation errors. Despite the passage of time, no work has addressed the quality of the original progress indicators on the problems for which they were proposed.

Accordingly, in this work we implemented both the MSRPI and the WiscPI in the same software and hardware framework and studied their performance using the TPC-H benchmark. Depending on different query characteristics, we found that there are cases where the MSRPI and the WiscPI are expected to be accurate, and cases in which they fail to provide reliable estimates. For the cases in which they are inaccurate, although there are differences in their performance, the most striking thing we found is that the estimation errors mostly arise from a common simplifying uniform future speed assumption: *at any point in a query's execution, the time to process a unit of work is uniform throughout the remainder of its execution*, where a unit of work is one GetNext() call for the MSRPI and one byte processed for the WiscPI, respectively. While the inventors of the MSRPI and the WiscPI were aware that this assumption could cause inaccuracies, presumably they adopted it to simplify the problem to produce approximate estimates. They did not explore the impact of this assumption on the accuracy of progress indicators. As it is shown by our experiments, this assumption leads to highly inaccurate progress estimates. For some TPC-H queries, the remaining time estimates provided by these two progress indicators are 30 times longer than the actual remaining time. Since their variants inherit this uniform speed assumption, we expect similar performance from them.

Inspired by this observation, we designed and implemented a new *cost-based* progress indicator, called GSLPI, to rectify the problem. The basic idea of GSLPI is to decompose an execution plan into a set of *speed-independent pipelines* delimited by blocking/semi-blocking operators. Then for each pipeline, we estimate its speed of processing the remaining work by utilizing its *wall-clock pipeline cost*. Our experimental results indicate that our approach produces more accurate progress estimates than just making uniform speed assumptions. For some of the TPC-H queries, our progress indicator reduces the estimation errors by more than an order of magnitude.

[1]We name the progress indicators after the institutions where they were developed.

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.74

Finally, we summarize the challenges that we encountered during the development of GSLPI. Similar to the MSRPI and the WiscPI, our progress indicator also suffers from the well known difficulties of cardinality estimation inherited from the query optimizer. Some of the challenges are still open questions, e.g., skewed data layout, speed fluctuations inside pipelines, etc. We hope that our work can bring these issues to the attention of other researchers and serve as a foundation of building more accurate progress indicators.

The rest of the paper is organized as follows. Section II describes the details of the MSRPI and the WiscPI. Section III presents our experimental evaluation of these two progress indicators. Section IV elaborates on our new cost-based progress indicator. Section V discusses how to refine cardinality and cost estimates by using runtime information. Section VI experimentally confirms the effectiveness of our techniques and discusses some remaining challenges that need to be resolved. Section VII briefly reviews the previous work that is related to ours. Finally, Section VIII concludes the paper with directions for future work.

II. Preliminaries

In this section, we give a brief overview of the MSRPI and the WiscPI for database queries. Their techniques serve as basic knowledge to understand the discussion in Section III and our new progress indicator in Section IV.

To estimate the progress of a given query, both the MSRPI and the WiscPI use an *execution plan*, which is a tree of physical operators chosen by the query optimizer. The physical operators include the most commonly used operators in a DBMS such as Table Scan, Index Scan, Index Seek, Filter, Hash Join, Merge Join, Nested Loops (NL) Join, Index Nested Loops (INL) Join, Group-by (Hash-based), Sort and Compute Scalar. An operator is referred to as a *blocking operator* if it does not produce any output before it has processed all tuples in as least one of its inputs. An execution plan is divided into a set of *pipelines* delimited by blocking operators (e.g., Hash Join, Group-by and Sort). A pipeline consists of a set of concurrently running operators. The goal of partitioning an execution plan into multiple pipelines is to gain more insight into the intermediate progress of a query execution.

Every pipeline has a set of *driver nodes*. They are the set of all leaf nodes of the pipeline, except those that are in the inner subtree of a Nested Loops/Index Nested Loops join. Once a pipeline has processed all the tuples in its driver nodes, it finishes execution. In general, for certain pipelines to start executing, one or more other pipelines have to complete. An execution plan can be viewed as a partial order of pipelines, denoted as P_1, .. P_p according to the order in which they are scheduled, where p is the number of pipelines in the plan. Figure 2 shows an example execution plan that contains three pipelines (the driver nodes of the pipelines are shaded). Note that this is not the only way for dividing plans into pipelines. In fact, the MSRPI and the WiscPI deploy two slightly different but essentially equivalent ways for division. The MSRPI prefers to put a blocking operator together with its descendants which provide input for it, while the WiscPI binds a blocking operator with its upper operator which consumes its output. In Figure 2, we follow the definition used in the MSRPI. For the WiscPI, its three pipelines are P_1 = {Table Scan A, Filter}, P_2 = {Table Scan B}, and P_3 = {Hash Join, Sort}. Both definitions of pipeline are equally adoptable for our solution in Section IV.

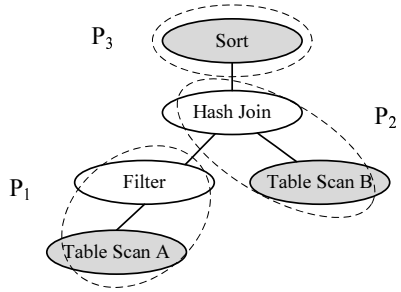


Fig. 2. An execution plan with 3 pipelines

The MSRPI calculates a query's progress based on the number of GetNext() calls. The total work done by a query is the total number of GetNext() calls issued by all operators in its execution plan. The MSRPI models a query's completion percentage as the fraction of the total number of GetNext() calls that have finished. More formally, suppose the execution plan has n operators. Let N_i $(1 \leq i \leq n)$ be the total number of tuples output by operator Op_i (which indicates the number of GetNext() calls made by that operator) throughout the execution of the query, and let K_i be the number of tuples processed by operator Op_i so far. The fraction of a query executed is $percent = \frac{\sum_{i=1}^{n} K_i}{\sum_{i=1}^{n} N_i}$. This GetNext() model assumes that the total time required to execute the query is amortized across multiple GetNext() calls, and therefore the percentage of GetNext() calls done thus far is a good indicator of the time taken by the query.

The WiscPI estimates the remaining query execution time by deploying a model based on the number of bytes processed. It keeps track of the total number of bytes that have not been processed U_i $(1 \leq i \leq p)$ in the input and output for each pipeline P_i. The remaining work for a query is therefore the sum of U_i for all pipelines in its plan. Additionally, it records the number of bytes S_i that have been processed by each pipeline P_i in the past T seconds, where T is a pre-defined parameter. The total number of bytes processed in the past T seconds can be thought of as the estimated execution speed of the query. Thus the estimated remaining execution time is $RT = \frac{\sum_{i=1}^{p} U_i}{\sum_{i=1}^{p} S_i}$.

Though the MSRPI does not explicitly give a formula for remaining time estimation, it assumes that the percentage of GetNext() calls done thus far is a good indicator of the progress it has made toward completion. This corresponds to the idea that if $p\%$ of GetNext() calls have finished, then $p\%$ of execution time has elapsed. In other words, the remaining $(100 - p)\%$ of GetNext() calls take $(100 - p)\%$ of the execution time. We adopt this interpretation to convert percentage completion for the MSRPI to time remaining.

III. Evaluating MSRPI and WiscPI

In this section, we present our experimental evaluation of the MSRPI and the WiscPI on an isolated system where only the given query is running. We also summarize the cases in which they are accurate and the cases in which they fail to provide good estimates.

A. When Are They Accurate?

While the MSRPI and the WiscPI represent units of work for a SQL query differently for each progress estimate, they both assume that each unit of work in the unexecuted portions takes *the same amount of time to process*, regardless of which pipeline this unit of work belongs to. The speed is assumed to be uniform for the remainder of query and equal to the speed in the past T seconds. We implemented both the MSRPI and the WiscPI in Microsoft SQL Server 11 [11] – the latest version of SQL Server, and tested their performance for different queries. For the WiscPI, we set the the number of bytes processed by an operator to be the number of finished GetNext() calls times the average tuple width. The work of a pipeline is the sum of the number of bytes processed by the input and output operators of the pipeline. The datasets and the system setup we used in our experiments can be found in Section VI. We set T to be 10 seconds, which is the same as in [6].

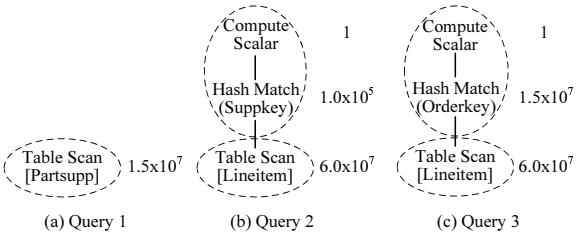


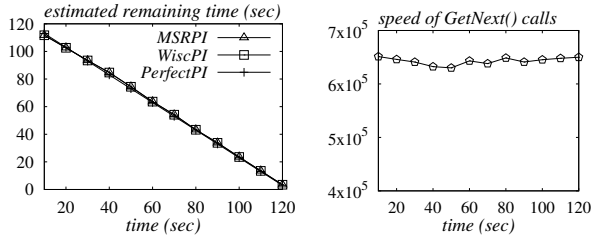
Fig. 3. Execution plans for tested queries

Query 1: *select * from partsupp*

The first query that we tested is Query 1 as illustrated above. Its execution plan is shown in Figure 3a. It consists of only one pipeline, which contains a single scan operator with the output cardinality shown next to it. Figure 4a plots the remaining time estimated by the MSRPI, the WiscPI and the PerfectPI. The PerfectPI is a fictitious ideal progress indicator that knows exactly how much time is left for a query. Figure 4b shows that the rate of GetNext() calls is approximately constant. We omit the graph depicting the speed in terms of the number of bytes, as it is also approximately constant. Since the speed is stable, both PIs can accurately estimate query progresses. From this example, we can generalize the idea to queries containing a single pipeline: *when a query consists of a single, uniform-speed pipeline, the MSRPI and the WiscPI should be able to accurately estimate the remaining query execution time*.

Query 2: *select count(distinct suppkey) from lineitem*

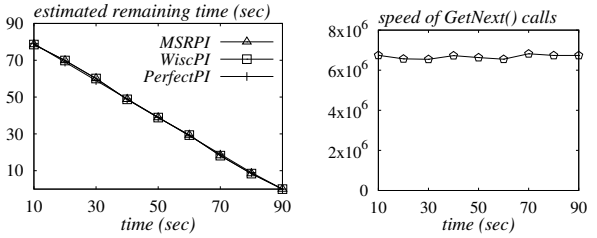
The second query we tested is Query 2, which consists of two pipelines (see Figure 3b). The first pipeline scans tuples in *lineitem* table. In the second pipeline, the Hash Match operator first computes a hash value for each *suppkey* value and inserts it into a hash table; subsequently, after all the *suppkey* values have been inserted, the Compute Scalar operator counts the number of distinct *suppkey* values in the hash table to produce the final answer.



(a) Progress estimation (b) Speed of GetNext() calls

Fig. 4. Test results for Query 1

The progress estimates produced and the speed of GetNext() calls are shown in Figure 5a and 5b, respectively. Although Query 2 contains two different pipelines that process tuples at varying speeds, the estimated remaining time is still accurate. Looking further into the query execution, we found that almost all the execution time has been spent on the first pipeline, and nearly all the GetNext() calls were issued within the same pipeline. We refer to such a pipeline as a *dominating pipeline*. The speed of GetNext() calls is stable throughout the query (see Figure 5b), since its non-dominating pipeline has almost no effect on the speed. Based on this observation, we can generalize to the following: *As long as PIs have an accurate speed estimate for the dominating pipeline, the remaining time estimate for the query will be accurate, even if they use inaccurate estimates for the other pipelines.*



(a) Progress estimation (b) Speed of GetNext() calls

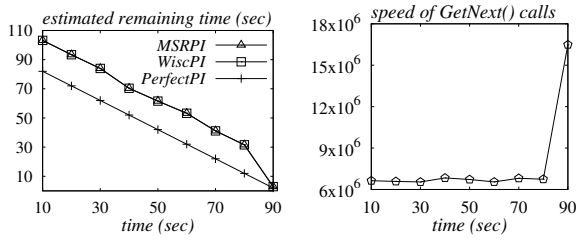
Fig. 5. Test results for Query 2

B. When Are They Not Accurate?

Unfortunately, there are many queries that do not fall into the cases described above. For example, for the queries in the TPC-H benchmark [12], we found that out of 22 queries only Q_1 contains a dominating pipeline, and only Q_6 contains a single pipeline. The rest of the queries typically contain between 3 and 9 different pipelines. In the following, we evaluate the performance of the progress indicators when a query has non-uniform speeds.

Query 3: *select count(distinct orderkey) from lineitem*

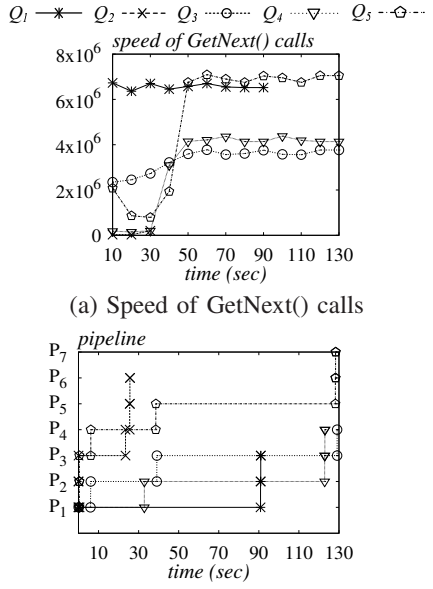
To understand the behavior of these PIs when a query contains multiple pipelines, we tested Query 3, which is similar to Query 2, as shown in Figure 3c . We increased the number of GetNext() calls in the Hash Match operator by changing the hash key to *orderkey*, which contains more distinct values.



(a) Progress estimation (b) Speed of GetNext calls

Fig. 6. Test results for Query 3

The test results for Query 3 can be seen in Figure 6. The first pipeline P_1 processes about 6.7×10^6 GetNext() calls in 10 seconds, while the second pipeline P_2 can finish 1.5×10^7 GetNext() calls in only 3 seconds. The main reason is that tuples processed by P_1 are brought from disk to main memory, while tuples processed by P_2 are already in memory. When the first pipeline is running, if the PIs use the speed of P_1 to estimate the remaining time of P_2 (that has not yet started), they get $(1.5 \times 10^7)/(6.7 \times 10^6) \approx 22.4$ seconds, which is about 20 seconds slower than the actual execution time. As a result, the remaining time estimated by the MSRPI and the WiscPI is about 20 seconds longer than the PerfectPI. If P_2 contained more tuples, or it could process tuples faster, the gap between the estimated and the true remaining time would increase. From this example, we can see that the MSRPI and the WiscPI make errors when the *speeds of pipelines are different and there is no dominating pipeline.*



(a) Speed of GetNext() calls

(b) Pipeline execution time

Fig. 7. Experiment results for TPC-H Q_1 to Q_5

TPC-H Example: To have a better understanding of the speed of a query, we tested the speed of GetNext() calls for all 22 TPC-H queries. Nearly all the queries exhibit different speeds in different execution periods, except for Q_1 and Q_6. As we have mentioned before, Q_1 contains a dominating pipeline and Q_6 contains a single pipeline. Figure 7a shows the speeds of the GetNext() calls for the first 5 TPC-H queries. As can be seen, four of them change their speeds dramatically during the execution. The execution times of the pipelines in each query are plotted in Figure 7b. For a given query, the length of the horizontal line segment on P_i $(1 \leq i \leq 7)$ indicates the execution time spent on processing pipeline P_i, and the vertical line segment that goes from P_i to P_{i+1} $(1 \leq i \leq 6)$ denotes the end of the execution of P_i and the beginning of the execution of P_{i+1}. If we compare the speed changing points with the pipeline switching points in Figure 7a and 7b, we can see that *when a query switches from one pipeline to another, its processing speed also changes*. During the execution of a pipeline, the speed of processing usually tends to be approximately constant.

The assumption made by the MSRPI and the WiscPI that the speed of the unexecuted portions of the query is equal to the speed in the past T seconds, contradicts the reality that many queries have dramatically different processing speeds for different pipelines. As a result, these PIs produce inaccurate progress estimates.

IV. Our Proposed PI: GSLPI

In this section, we present GSLPI[2]: a new cost-based progress indicator for SQL queries. The distinguishing characteristics of GSLPI include: (1) the decomposition of an execution plan into a set of speed-independent pipelines, (2) the utilization of the wall-clock pipeline cost to represent the cost of a pipeline, and (3) the estimation of the speed of each future pipeline based on its wall-clock pipeline cost.

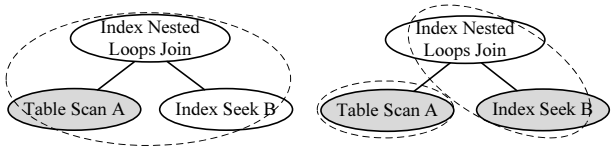
A. Speed-Independent Pipelines

From Figure 7, we can see that each pipeline processes tuples at its own speed, and this speed is usually stable during its execution. Motivated by this observation, we developed an approach for estimating the speed for each individual pipeline. In [5], [6], execution plans are divided into pipelines by blocking operators, and a pipeline may contain a Nested Loops/Index Nested Loops join and its inner and outer subtrees, as shown in Figure 8a.

In the case that the physical implementation of the join is an (Index) Block Nested Loops (BNL) join, it behaves like a semi-blocking operator. Note that a Hash Join is a blocking operator, since it must consume all tuples from the build relation before it can produce any output. Compared to Hash Join, (Index) BNL Join must first consume a certain number of tuples from its inner subtree before it can process its outer subtree and produce output. Since the inner/outer subtrees may include different relations and operators, they may process tuples at different speeds. Furthermore, when one of them is executing and another one is halted, the speed of the executing operator(s) is independent of the speed of other operator(s). If we group both inner and outer subtrees into a single pipeline, we may mix up two sets of operators with different speeds. Such a combination will cause difficulties in

[2]Named after Jim Gray Systems Lab, where the progress indicator was developed.

accurate estimations of how fast the pipeline is running. To separate operators processing tuples at different speeds from each other, we define a *speed-independent pipeline*.



(a) Pipelines for MsPI and WiscPI (b) Pipelines for GSLPI

Fig. 8. Pipelines for progress indicators

Definition 1: A **speed-independent pipeline** is a group of interconnected operators that execute concurrently and process tuples at a speed independent of the speeds of other operators (in other pipelines) in the execution plan.

Based on the above definition, we break an execution plan into a set of speed-independent pipelines and estimate their speeds. A breaking point in a plan is a blocking or semi-blocking operator. Since we introduce additional breaking points (semi-blocking operators), we may break a "traditional" pipeline into finer pieces. For simplicity of discussion, we refer to a speed-independent pipeline as a "pipeline" in the rest of the paper. The driver nodes of a pipeline are defined as the set of all leaf nodes in the pipeline. Figure 8b shows the new pipelines and driver nodes. We adopt the model of work in terms of the number of GetNext() calls involved, and define the work of a pipeline as the total number of GetNext() calls in the driver nodes of a pipeline. The speed of a pipeline is the amount of work that has been done for the pipeline in the past T seconds, where T is a user-set parameter. Next, we describe our approach to predict the speed for each unfinished pipeline using estimated CPU and I/O costs provided by the query optimizer. The method proposed here is equally applicable to the model where work is defined as the number of bytes processed.

B. Total and Wall-clock Costs

To process tuples, each operator in a pipeline needs to perform CPU and/or I/O tasks. The cost of these tasks represents the "cost of the pipeline", which is the actual amount of work that takes time to finish. In addition, we make a distinction between total and wall-clock pipeline costs, which is imperative for remaining time estimation.

Definition 2: The **total pipeline cost** is the total amount of CPU and I/O done by a pipeline from its beginning until the end. The **wall-clock pipeline cost** is the maximum amount of non-overlapping CPU and I/O done by a pipeline from its beginning until the end.

The difference between the total and the wall-clock pipeline costs is that the total pipeline cost represents the total amount of work that "must be done" by a pipeline during its execution, regardless of whether the execution is parallelized or not. The wall-clock pipeline cost, on the other hand, identifies the non-overlapping parts of work and finds the most expensive one.

For example, consider a plan that consists of only a Table Scan operator on *lineitem*. It has only one pipeline P_1, which is similar to the bottom pipeline depicted in Figure 3b. Let us assume that the cost of the operator obtained from the optimizer is *CPU_cost = 66* and *I/O_cost = 767*. Here, the total cost denoted *Tol(P_1)* is 833, while the wall-clock cost *Clk(P_1)* is equal to 767. Since CPU and I/O tasks can be performed in parallel by this operator, the execution time is dominated by the I/O cost, and the CPU cost has a trivial contribution to the overall execution time.

1) Redistribution of Costs: When an execution plan is chosen and returned by a query optimizer, it typically includes both CPU and I/O costs for each operator. Intuitively, one might think that we can just sum up the CPU and I/O costs of all operators in a pipeline to get its total cost. We found, however, that sometimes this is not true in practice. For an operator Op contained in a pipeline P, it is possible that part of the "work" of Op may be done during the execution of other pipelines. Consider the query from Figure 3b where P_1 works as a producer providing tuples to be consumed by P_2. After a tuple is fetched from the base table by P_1, the tuple will be immediately inserted into the hash table by the Hash Match operator. When P_1 is running, P_2 does not generate any output tuples, since it needs to wait until *all* of the tuples output by P_1 have been inserted into the hash table. After P_1 is finished, P_2 reads tuples from the hash table and counts the number of distinct tuples. Though the work of building the hash table is done by an operator that belongs to P_2, it actually occurs during the execution of P_1 and affects the execution time of P_1. Thus, the corresponding cost should be added to P_1 and subtracted from P_2. We consider this kind of cost to be the *post-processing cost* of P_1 that provides the data and the *pre-processing cost* of P_2 that consumes the data.

To do this cost redistribution, for each piece of work we must determine *in which pipelines the work is done* and *how much of that work is done* in each of the pipelines. If an operator is in the middle of a pipeline, all the work done is within the life span of the pipeline. However, there are two operators that are exceptions: Hash Join and Group-By (Hash-based). If these operators are at the boundaries of a pipeline, part of their work is done in the producer pipeline(s), and part of their work is executed in the consumer pipeline(s). To redistribute the cost of a Hash Match or a Hash Join operator, we must first understand how much of the CPU and I/O is done by each pipeline. The better we can do the distribution, the more precise the time estimation can be attained by the progress indicator. For simplicity, we approximate each part of the work using a set of actions that take roughly the same amount of CPU and I/O.

For a Hash Match operator, the first part (done by the producer pipeline) is to build the hash table. The insertion of a tuple into a hash table contains the following actions: (1) reading the tuple, (2) computing the hash value, (3) finding the right bucket, (4) assigning an empty slot, and (5) inserting the tuple into the hash table. The second part (done by the consumer pipeline) consists of only one action: reading the tuples from the hash table. Let the number of input tuples be a, and the number of output tuples be b. The cost assigned

to the producer pipeline is: $5a/(5a+b) \times cost$, and the cost assigned to the consumer pipeline is $b/(5a+b) \times cost$ (the *cost* represents CPU or I/O cost).

Similarly, for a Hash Join operator, its first execution part (done by producer pipeline) consists of building the hash table, and the second part (done by consumer pipeline) is responsible for probing the hash table and outputting the result tuples. The probe by a tuple is modeled as reading the tuple, computing its hash value, finding the right bucket, finding the right tuple and doing the join. Suppose the build child contains a_1 tuples and the probe child has a_2 tuples, and b tuples are output as a result. The cost assigned to the producer pipeline is: $5a_1/(5a_1 + 5a_2 + b) \times cost$, and the cost assigned to the consumer pipeline is: $(5a_2 + b)/(5a_1 + 5a_2 + b) \times cost$.

Note that the need for cost redistribution is not due to our definition of pipeline. The Hash Match operator in Figure 3b is doing work for both P_1 and P_2, thus it is tricky to allocate it to one of the pipelines. In this paper, we assign it to P_2. One may use a different definition and assign it to P_1, or even assign it to both P_1 and P_2. But eventually, we will still end up with the problem of how much of the work of this operator is done by pipeline P_1 and P_2, respectively. As a result, we can adopt either of these three pipeline definitions as long as we can distribute the cost to the corresponding pipeline correctly.

2) Calculating the Costs: To calculate the wall-clock cost for a pipeline, we must know which parts of the work in the pipeline overlap. In this section, we present the details of our solution for a simple case, where only one processor and one disk are used for processing a query. However, the idea can be extended to a scenario where multiple processors and disks are used for processing. Since there is one processor and one disk available, the CPU cost and the I/O costs of each operator are expected to overlap. For a set of operators that execute concurrently in the same pipeline, their CPU and I/O costs also overlap. Thus, the wall-clock pipeline cost $Clk(P)$ of a pipeline P is as follows: $Clk(P)$ = max($\sum_{i=1}^{m} CPU(Op_i) + post\_CPU(P) - pre\_CPU(P)$, $\sum_{i=1}^{m} IO(Op_i) + post\_IO(P) - pre\_IO(P)$), where m is the number of operators in P and $post\_$ and $pre\_$ denote the post_processing and pre_processing CPU or I/O cost of a pipeline, respectively. Next, we extend the concepts of total and wall-clock costs of a pipeline to the entire query, and formally define them as follows:

Definition 3: The **total query cost** is the total amount of CPU and I/O done by a query, and the **wall-clock query cost** is the maximum amount of non-overlapping CPU and I/O.

Since there is no overlapping work between different pipelines, the wall-clock query cost is the sum of the wall-clock pipeline costs of all pipelines. To understand these different costs better, we calculated these costs for Query 2 and Query 3 (from Figure 3). The CPU and I/O costs obtained from SQL Server for Table Scan[Lineitem] are 66 and 767, for Hash Match(Suppkey) are 275 and 0, and for Hash Match(Orderkey) are 670 and 0, respectively. The cost of Compute Scalar is ignored since it is almost 0. The total and wall-clock costs for pipelines and queries are depicted in Table I.

The Hash Match operator in Query 3 needs to do much more CPU than the same operator in Query 2. As a result, the total query cost (1503) for Query 3 is obviously larger than the total query cost for Query 2 (1108). But from the table, we can observe that the actual execution time of Query 3 is only slightly longer than that of Query 2. The reason is that for both queries, most of the CPU work required to be done by Hash Match is done by pipeline P_1, and pipeline P_1 spends a lot of time doing I/O, which dominates the execution time compared to CPU. As a result, both P_1s in Query 2 and Query 3 take about 90 seconds to finish, and both P_2s finish fast, since they have only a little CPU to do (compared with P_1). For Query 3, we observe that while the number of GetNext() calls issued by P_2 is about 25% of that issued by P_1, the actual execution time is much lower than 25% of P_1's execution time. This is because tuples processed by P_2 are in memory and the average CPU or I/O needed for each tuple in P_2 is much lower than that in P_1.

Query Q	Pipeline P	Tol(P)	Clk(P)	Tol(Q)	Clk(Q)	Exe. Time
Query 2	P_1	1107.9	767	1108	767.1	90 (sec)
	P_2	0.1	0.1			
Query 3	P_1	1471	767	1503	799	93 (sec)
	P_2	32	32			

TABLE I
COMPARISON OF THE COSTS

In summary, having more GetNext() calls or bytes to process does not imply more CPU or I/O to do, thus it does not lead to longer execution time. The CPU and/or I/O is the actual "work" which needs to be done by a pipeline and takes time to finish; the non-overlapping part of the work (represented by wall-clock pipeline cost) determines the execution time.

C. Speed Estimation

To estimate the remaining time of a pipeline, we must know how fast it can process its work. Given a pipeline, the driver nodes provide sources of tuples to be processed by the remaining operators. We assume that the total amount of CPU and I/O is amortized across all tuples provided by the driver nodes. How fast tuples are being processed by the driver nodes reflects how fast the CPU and I/O requests are being processed by the entire pipeline. For the purpose of remaining time estimation, it is sufficient for us to consider the total number of tuples in the driver nodes as the total work, and the rate of processing these tuples as the speed.

Estimating the speed for an executing pipeline is straightforward, and the number of tuples processed by its driver nodes in the past T seconds is considered as its processing speed. For a pipeline that is pending, we can predict its speed based on its wall-clock pipeline cost and how fast the system can process CPU and I/O tasks. For a pipeline P_i, suppose its wall-clock cost is C_i, and the total number of tuples in its driver nodes is N_i, among which K_i have been processed. Let S_i be the speed that P_i can process its tuples in driver nodes. Without loss of generality, suppose that P_1 is the running pipeline and P_2 is the pipeline of which the speed that we want to predict. We assume that the system can process the same amount of CPU

or I/O for a query in every T seconds. Let S_1 be the speed of P_1, and S_2 be the speed of P_2 that needs to be predicted. We have $\frac{C_1}{N_1} \times S_1 = \frac{C_2}{N_2} \times S_2$. Then the speed of P_2 is:

$$S_2 = S_1 \times \frac{C_1 \times N_2}{N_1 \times C_2}.$$

The remaining time for an unfinished P_i can be estimated as $(N_i - K_i)/S_i$, and the remaining time for the entire query is the sum of the remaining time of all pipelines. In the formula above, the amount of CPU or I/O processed for this query in T seconds equals the amount of cost finished by the running pipeline in the past T seconds. In case the running pipeline has random speed fluctuation, we can take the average rate for processing the CPU or I/O by the running pipeline in the past to smooth the estimation.

The new speed estimate for an unfinished pipeline using its wall-clock pipeline cost can be much closer to its actual speed than one generated using the uniform speed assumption. For example, for Query 3 in Figure 6, when P_1 is running, the speed of P_2 is estimated as $6.7 \times 10^6 \times \frac{767 \times 1.5 \times 10^7}{6 \times 10^7 \times 32} \approx 4 \times 10^7$ GetNext() calls per 10 seconds. The estimated remaining time of P_2 is $\frac{1.5 \times 10^7}{4 \times 10^7} \times 10 \approx 3.75$ seconds, which is close to the actual value (about 3 seconds). By contrast, the MSRPI assumes that $S_2 = S_1 \approx 6.7 \times 10^6$, and gives an estimate of 22.4 seconds.

V. Utilizing Runtime Information

As is the case in query optimization, a key challenge for progress indicators is to accurately estimate cardinalities and costs of operators in the query plan. In the following, we describe how to collect execution feedback to continuously refine the cardinality and cost estimates.

A. Refining Cardinality Estimates

Both the MSRPI and the WiscPI collect information to refine cardinality estimates. The refinement in the MSRPI is based on refining upper/lower bounds of cardinality estimates [5], while the WiscPI relies on linear interpolation [6]. Since these two methods are compatible and each provides additional information for refining cardinalities, we adopt both upper/lower bounds and linear interpolation to refine cardinality estimates.

Before GSLPI estimates the remaining time of a query, the cardinality of each unfinished operator is first computed using linear interpolation. For an operator in a running pipeline, the percentage of the tuples in the driver nodes that have been processed is used to refine the output cardinality. Let E_d be the input cardinality of the driver node and K_d of them have been processed, then the percentage of the tuples that has been processed is $p = K_d/E_d$. For an operator (other than driver nodes) in the same pipeline, suppose its original estimate of cardinality is E_1 (before the pipeline has started executing) and it has processed K tuples. The new cardinality estimate based on linear interpolation is then $E_2 = K/p$. In case that the pipeline has only one driver node, a heuristic formula is used to estimate the final output: $E = p \times E_2 + (1-p) \times E_1$. In cases where a pipeline contains more than one driver node, the driver node which finishes processing relatively fast is chosen to calculate the value of p. Then the upper bound and the lower bound are calculated for each unfinished operator and are used to further refine their cardinality estimate. Intuitively, for each operator the estimated number of output tuples should never be less than the number of tuples seen so far (i.e., the lower bound). For most operators except joins, the estimated number of output tuples should never exceed the number of input tuples (i.e., the upper bound). For more details on upper/lower bounds, we refer the reader to [5].

Normally, the output cardinality of an operator does not affect the costs of other operators, unless it is a descendant of those operators. However, if the operator is in the outer subtree of a Nested Loops (NL) join or an Index Nested Loops (INL) join, it may affect both the CPU and I/O costs of the operators in the inner subtree. If the outer subtree produces more (fewer) tuples for the NL/INL join, the number of executions of the inner subtree may increase (decrease) as well (this is known as as *Rebinds* and *Rewinds* in SQL Server [13]). As a result, for a NL/INL join, the estimated output cardinality of the outer subtree should be propagated to the inner subtree to refine the CPU and I/O cost of the operators. Take the query plan in Figure 8 for example. If the number of tuples produced by Table Scan operator increases, the number of index seeks done by inner subtree also increases. Suppose the original and updated estimated output tuples for the outer subtree is E_1 and E_2, respectively. Let Ex_1 be the original estimated number of executions of the inner subtree. Then the updated number of executions of the inner subtree is $Ex_2 = E_2/E_1 \times Ex_1$. The updated cost (CPU or I/O) for each operator in the inner subtree is the cost for one execution times the updated number of executions.

B. Refining Cost Estimates

Since the wall-clock pipeline cost is critical to the accuracy of GSLPI, when the cardinality estimate of an operator changes, we also need to revise its cost estimate accordingly. The cost refinement is based on algebraic properties of the operator. For every operator, we use a function f to approximate its cost estimate with respect to its properties and cardinalities. If the cost of an operation increases linearly with the input cardinalities (e.g., Table Scan, Index Scan, Filter, etc.), the function is simply $f(N) = N$, where N is the input cardinality of the operator. Suppose C_1 is the cost (CPU or I/O) obtained from the optimizer for an operator when its input cardinality is N_1. When its input cardinality changes to N_2, its cost gets updated to $C_2 = C_1 \times \frac{N_2}{N_1}$. For costs that do not increase linearly with respect to input, a more complicated function is used.

Estimating I/O costs is even more error-prone than estimating CPU costs. This is due to the following two reasons: (1) the memory granted to the query may be different from the available memory assumption made by the optimizer, and (2) the execution of an operator may bring in the data needed by another operator that runs sometime later. To alleviate the

problem caused by I/O estimation error, we introduce two heuristic methods to eliminate the I/O cost of an operator, if it does not need to do any I/O (e.g., all the data can fit in memory). We first consider the maximum amount of memory required by a pipeline. For a non-blocking operator in a pipeline, a small fraction of memory is sufficient, since once a tuple is output by a non-blocking operator, it will be immediately propagated on to the next operator (if there is any). For a blocking operator, tuples are collected in the operator until all the input tuples are consumed, therefore, blocking operators are memory consuming and take up most of the memory. Thus, given a pipeline, the memory-consuming operators include (i) a Sort or a Group-by (Hash-based) operator providing data for the pipeline, (ii) a Hash Match or a NL/INL join operator inside a pipeline, and (iii) a blocking operator that takes in the output tuples of the pipeline. We ignore the amount of memory used by the non-blocking operators. Then the maximum amount of memory required by a pipeline is defined as the total amount of memory that is needed to hold the data for the three types of memory-consuming operators above.

If the memory available is more than the maximum amount of memory required by a pipeline, none of these memory consuming operators needs to do any I/O when their corresponding pipeline executes. Thus, we can safely remove this part of I/O cost from the wall-clock pipeline cost of the pipeline. Let P_i and P_j ($i < j$) be two different pipelines that contain a same subtree of operators, which generate the same intermediate results. In this case, SQL Server will detect the common subtrees and try to reuse the intermediate results. We check whether the intermediate results generated by P_i are still in memory when P_j runs. Let M_i be the maximum amount of memory required by pipeline P_i, and $Size_r$ be the size of the intermediate results. If $\sum_{k=i}^{j} M_k + Size_r$ is less than the available memory for this query, the data brought in by P_i should still be in memory when P_j runs. We subtract all the costs (both CPU and I/O) of the operators in the subtree from the wall-clock pipeline cost of P_j.

VI. Experimental Evaluation

This section presents experimental results showing the effectiveness of our proposed techniques. We first describe the experimental setup, and then evaluate the performance of our progress indicator and compare it to the MSRPI and the WiscPI.

A. Experimental Setup

We implemented all three progress indicators in Microsoft SQL Server 11. Our experiments use a TPC-H 10GB database with all tables stored on a single disk. We measured the performance of the progress indicators using all 22 queries in the TPC-H benchmark. Most of them contain more than 8 operators and 4 different pipelines. The experiments were run on a machine with an Intel Core 2 Duo CPU, with 8 gigabytes of memory. In the experiments, only the database server was executing, and we run each query one at a time. When a query is running, a progress indicator wakes up periodically (every 10 seconds, which is the same as the setting in [6]), collects the runtime information, and estimates the remaining execution time of the query. For all queries, GSLPI provides progress estimates with less than 1% overhead, and this is similar to the overhead introduced by the MSRPI or the WiscPI. With a smaller T, the PIs could adapt quicker to changes in speed, but it would not substantially improve the performance of either the MSRPI or the WiscPI with respect to the main issue (the uniform speed assumption) we address.

B. Effectiveness of Speed-independent Pipeline

To justify the necessity of using semi-blocking operators for dividing pipelines, we tested Query 4, a simple and easy to understand example that well illustrate our point. Its execution plan is shown in Figure 9, which contains a Nested Loops join operator. The remaining time predictions made by the MSRPI, the WiscPI and GSLPI are ploted in Figure 10.

Query 4: *select * from nation loop join customer on nationkey = 1*

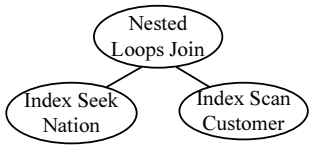


Fig. 9. The execution plan of Query 4

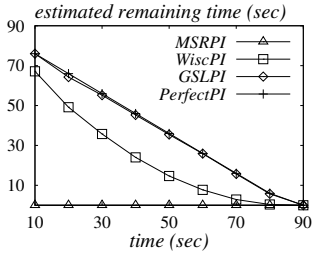


Fig. 10. Progress estimation for Query 4

For the MSRPI and the WiscPI, all operators belong to the same pipeline, and the driver node is the Index Seek Nation operator (refer to Section II for its definition). The MSRPI makes a driver node hypothesis, which says that the overall query progress can be estimated by the progress of only the driver node(s) of the pipeline. Unfortunately, this is not true for pipelines containing semi-blocking operators, where the execution of the inner subtree is independent of the execution of the outer subtree. For example, when the Index Seek operator in Query 4 finishes processing all its tuples at the very beginning, the MSRPI assumes that the entire pipeline is finished, and its estimated remaining time is 0 after that. But in fact, the Nested Loop operator and its inner subtree are still running and takes about 90 seconds to finish. The WiscPI makes a similar assumption about the driver node(s) (called dominant input(s) in the original paper). Although it uses a heuristic formula to smooth fluctuations, which prevents it from jumping to 0 directly, its estimates are still not quite accurate. Our GSLPI splits the query plan into

two speed-independent pipelines, which truly represent two sets of operators that run independently, thus produces accurate predictions. For TPCH queries, we also observe similar cases in which the inner subtree starts running independently after the outer subtree consumes a certain amount of tuples and halts. For these cases, GSLPI exhibits better performance.

C. Effectiveness of Wall-clock Pipeline Cost

In this section, we examine the accuracy of our progress indicator. We first show the overall performance of GSLPI for TPC-H queries, then provide an analysis for our progress indicator and compare its remaining time estimates with the other progress indicators.

1) Overall Performance: To evaluate the performance of our progress indicator, we employ the evaluation metric called estimation error used in [5], [14]. Assume that the query starts at t_0 and ends at t_n. Then let t_i ($t_0 \leq t_i \leq t_n$) be the time when the progress estimation is taken, and RT_i be the estimated remaining time. Together with RT_i, a progress indicator also returns a value f_i derived from RT_i to indicate the percentage of the time completed: $f_i = 100(t_i - t_0)/(RT_i + t_i - t_0)$. The estimation error at time t_i is defined as:

$$e_i = |\frac{100(t_i - t_0)}{(t_n - t_0)} - f_i|,$$

where $100(t_i - t_0)/(t_n - t_0)$ represents the actual percent-time done. For each query, we calculate its average and maximum estimation error. The results are shown in Table II. As we can see from the table, the average error is typically small (only 3 of them are above 5%), and the maximum error is usually below 10%. The larger errors in the estimates made by our progress indicator (e.g., Q_3, Q_{12}, Q_{20} and Q_{21}) are due to the cardinality estimation errors inherited from the query optimizer (see the verification in Section VI-E).

Query	Mean	Max	Query	Mean	Max
Q_1	0.5%	0.6%	Q_{12}	10.5%	24.7%
Q_2	0.9%	1.9%	Q_{13}	1.2%	3.5%
Q_3	4.0%	10.4%	Q_{14}	1.0%	2.2%
Q_4	1.5%	8.5%	Q_{15}	0.1%	0.5%
Q_5	2.4%	10.3%	Q_{16}	1.6%	4.0%
Q_6	0.3%	0.7%	Q_{17}	0.5%	0.9%
Q_7	3.6%	9.4%	Q_{18}	0.3%	1.6%
Q_8	3.2%	8.2%	Q_{19}	0.2%	0.5%
Q_9	1.3%	6.3%	Q_{20}	16.8%	41.1%
Q_{10}	1.7%	7.7%	Q_{21}	5.7%	16.0%
Q_{11}	1.1%	3.2%	Q_{22}	1.1%	3.6%

TABLE II
ESTIMATION ERRORS IN TPC-H QUERIES

2) Utility of Wall-clock Pipeline Cost: In this experiment we use TPC-H query Q_1 to show the necessity of redistributing the CPU and I/O costs to the pipelines where the actual work is being done. We tested and compared our GSLPI against a modified version that does not perform redistribution. The non-redistributing version of our progress indicator simply sums up the CPU and I/O costs of all operators inside a pipeline respectively, and chooses the larger value as the cost of the pipeline. We use GSLPI$_{dis}$ to denote our original GSLPI, which redistributes the cost to get the wall-clock pipeline costs, and GSLPI$_{nod}$ to denote the variant. The comparison of these two progress indicators for Q_1 is depicted in Figure 11.

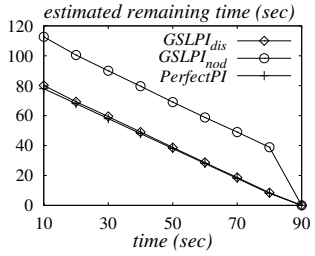


Fig. 11. Progress estimation for Q_1

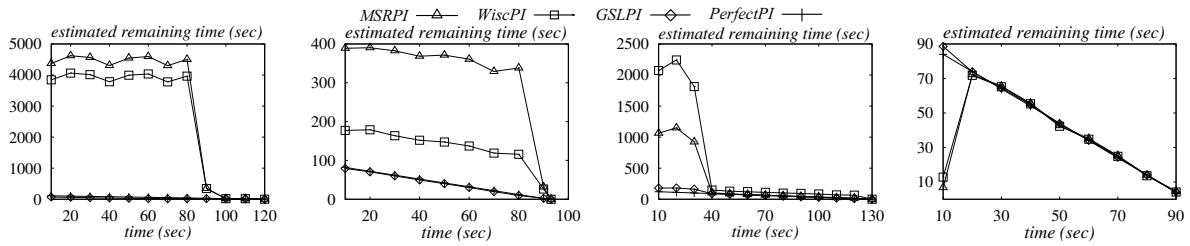
The graph above illustrates that GSLPI$_{dis}$ is almost identical to the PerfectPI, while the estimates of GSLPI$_{nod}$ are at least 40% longer than the actual remaining time. The reason is when the first pipeline in Q_1 scans a table, it also does most of the work (building the hash table) for the Hash Match operator in the second pipeline. Since I/O takes a longer time to finish, the execution time for building the hash table (by doing CPU) is excluded from the estimates. Without redistribution, GSLPI$_{nod}$ assumes that this part of work is done in the second pipeline, and it takes about 32 seconds. This assumption leads to the error gap between GSLPI$_{nod}$ and the PerfectPI in the graph.

The improved WiscPI [8] suggested an idea of using the CPU and I/O costs of the input and output operators in the pipeline to scale its speed. It also does not consider the redistribution of the costs among the pipelines as well. As a result, in the best case the improved WiscPI can provide estimates similar to that in GSLPI$_{nod}$. Since Hash Match and Group-By operators are common in query plans (e.g., 21 out of 22 TPC-H execution plans chosen by the SQL Server optimizer contain this kind of operator), the idea of cost redistribution must be deployed if better progress estimates are desired. By introducing the wall-clock pipeline cost, we are able to address this problem and get more accurate estimates.

3) Comparison of Progress Estimates: In this section, we show that using wall-clock pipeline costs to scale the speeds of the pipelines leads to better estimates than making the uniform speed assumption. We tested all the TPC-H queries and compared the estimates provided by the MSRPI, the WiscPI, GSLPI and the PerfectPI. From our experiments, we observed improvements for 20 queries (for the remaining 2 queries, all the progress indicators produce accurate results).

We illustrate the results for Q_{12} in Figure 12a. Q_{12} takes about 120 seconds to finish, and it spends around 90 seconds on P_1 and around 30 seconds on P_2. Although it has 4 different pipelines, only the first two pipelines are important in our discussion (the other two finish very fast). When P_1 is running, the estimates of both the MSRPI and the WiscPI are over 4000 seconds, which are far away from the PerfectPI as plotted at the bottom in the figure. GSLPI has an average estimation error of only 10.5%.

To identify the reasons for the errors made by the MSRPI and the WiscPI, we measured the speed of GetNext() calls.



(a) Query Q_{12} (b) Query Q_{14} (c) Query Q_{10} (d) Query Q_{19}

Fig. 12. Progress estimation for TPC-H queries

As illustrated in Figure 13, the GetNext() calls speed of P_1 is much slower than that of P_2. In fact, P_1 processes about 3.4×10^4 GetNext() calls every 10 seconds, while P_2 processes 5×10^6 GetNext() calls in the same amount of time. Since P_2 has 1.56×10^7 GetNext() calls, if it also processed tuples with the same speed as P_1 (the uniform speed assumption), it should take more than 4000 seconds to finish. However, in reality it finishes in about 30 seconds. Similarly, for the WiscPI, the number of bytes processed (computed as the number of GetNext() calls times the average tuple size) is also slow for P_1 and fast for P_2. As a result, it generates similar progress estimates. GSLPI, on the other hand, utilizes the wall-clock pipeline costs of P_1 and P_2 to estimate the speed of P_2 when P_1 is running. Since the wall-clock pipeline cost of P_2 is smaller than that of P_1 and the number of GetNext() calls in P_2 is much larger than that in P_1, the per tuple cost of P_2 becomes far smaller than P_1, which suggests that each tuple in P_2 takes much less time to process. By taking advantage of this information, GSLPI can obtain a more accurate speed estimate for P_2 and produce a more precise remaining time estimate.

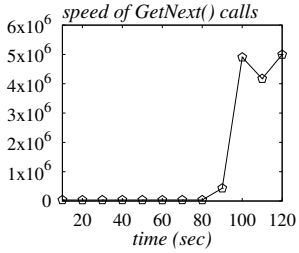


Fig. 13. Speed of GetNext() calls for Q_{12}

As shown in Figure 12b, we obtain similar results for TPC-H Q_{14} for the same reason. The performance of the MSRPI and the WiscPI is better compared to that for Q_{12}, but the estimated remaining time is still much longer than the actual remaining time. The GSLPI's results, however, are nearly identical to the PerfectPI's estimates.

If we compare the MSRPI against the WiscPI, we can see that the WiscPI is the winner for Q_{12} and Q_{14}. But according to our understanding, it is challenging to tell which one produces more accurate estimates in general. Figure 12c demonstrates a case where the MSRPI performs better than the WiscPI. For these two queries, a slow pipeline P_s starts first, and both progress indicators must estimate the speed of the fast pipeline P_f that runs later. When P_s is running, WiscPI takes the product of the speed of GetNext() calls and the average tuple size of P_s and uses it as the speed of P_f. If the average tuple size of P_s and P_f are the same, the estimates made by the MSRPI and the WiscPI will be the same. But in Q_{14}, the average tuple size of the slow pipeline happens to be larger than that of the fast pipeline, thus, the WiscPI ends up using a faster speed for P_f. As a result, the WiscPI provides more accurate progress estimates. The opposite case, where the slow pipeline has a smaller average tuple size, happens in Q_{10}, which makes the MSRPI perform better. Since more tuples or bytes does not necessarily imply longer execution time, using them for time estimation is inadequate.

The three example queries above demonstrate that when a slow pipeline runs first, both the MSRPI and the WiscPI overestimate the execution time for the queries. However, GSLPI is able to address this problem by using the wall-clock pipeline cost approach. Figure 12d shows a case when a fast pipeline starts first. In this case, both the MSRPI and the WiscPI underestimate the execution time, whereas GSLPI also handles this case successfully.

D. Effectiveness of I/O Elimination Heuristics

To show the effectiveness of the proposed I/O elimination heuristics, we tested another modified version of GSLPI$_{noh}$, where no elimination heuristics are used. In this section, we compare the performance of GSLPI$_{noh}$ with our original progress indicator, denoted as GSLPI$_{heu}$ here. Significant improvements can be observed for two queries, namely Q_{11} and Q_{18}. Figure 14 shows the result for Q_{18}. As can be seen, GSL$_{heu}$ progress indicator is almost identical to the PerfectPI, while GSL$_{noh}$ underestimates the execution time at the beginning. When we look into the execution plan, we find that there is one Hash Match operator, which behaves very different from what is suggested by the query optimizer. The I/O cost provided by the optimizer for this operator indicates that it should have some I/O work to do, but when the query is running, it obtains enough resources to store the entire hash table in the main memory. Thus, no I/O is actually performed. GSL$_{noh}$ does not consider this runtime information: when the first pipeline P_1 is running at the beginning, GSL$_{noh}$ assumes that P_1 is performing the I/O and gets a bigger wall-clock pipeline cost for P_1. Since no I/O is actually performed and tuples get processed quickly, GSL$_{noh}$ believes that the system

processes the CPU or the I/O tasks fast, and thus the query will finish in a short time. GSL$_{heu}$, on the other hand, collects the runtime information for memory. Since the granted memory is more than the required memory, GSL$_{heu}$ subtracts the I/O cost from the Hash Match operator's cost. The revised I/O cost depicts more accurately what happens in reality.

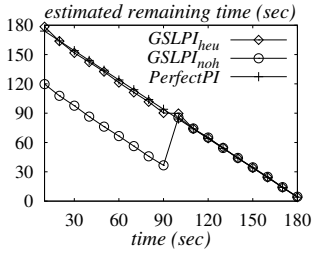


Fig. 14. Progress estimation for Q_{18}

E. Verification

For all the TPC-H queries that we tested, most of the estimation errors made by GSLPI are actually due to *cardinality estimation errors*. To verify this, we tested our progress indicator based on true cardinalities (obtained after one execution of the query). For the problematic queries shown in Table II, significant improvement occurs to Q_3 Q_{12}, Q_{20}, and Q_{21}. Figure 15 shows the estimated remaining time with the actual cardinalities (denoted GSLPI$_{act}$) and the cardinalities obtained from the optimizer at compile-time (denoted as our original progress indicator GSLPI$_{est}$). Since the cardinality estimates for 5 time-consuming operators (in 3 different pipelines) are significantly wrong, this leads to most of the remaining time estimates to become quite inaccurate as well. With the true cardinalities, GSLPI can successfully estimate the speed of each pipeline, thus the remaining time of the query.

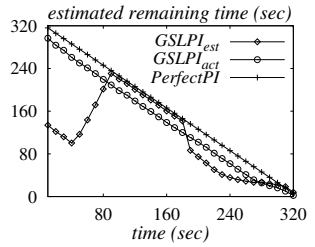


Fig. 15. Progress estimation for Q_{21}

F. Accurate Progress Estimations Challenges

In the above sections, we illustrate the improvement that we made over the MSRPI and the WiscPI. In the following, we present the challenges that we encountered, with a hope of inspiring the development of more accurate progress indicators.

A key challenge for GSLPI (as well as for any other progress indicators) is accurate cardinality estimation. This problem has been faced by query optimizers for a long time, and all the progress indicators proposed so far suffer from the cardinality estimation errors. As we mentioned above, cardinality estimation errors contribute to most of the errors made by GSLPI. In addition, if we can not provide a robust guarantee for the problem of cardinality estimation, it is impossible for us to develop a progress indicator with guarantees.

Even if we could obtain accurate cardinality estimates for a query, the layout of the data may also affect the estimation of the progress. For the queries that we consider, the processing speeds of most pipelines usually tend to be constant. If we have a skewed distribution of the data, the speed may vary during the execution. For example, if most of the tuples which satisfy the select conditions are clustered at the beginning of the table, the speed may increase after finish processing these tuples. A progress indicator which does not aware of the layout of the data is likely to produce inaccurate estimates.

Another challenge problem we have found is the Nested Loops join operator in Q_{20}. The inner subtree consists of only one Index Seek operator, and the speed of GetNext() calls for the pipeline in the outer subtree keep increasing quickly. One possible reason for the increasing speed is that the index seeks finished earlier brought in pages from disk needed by the Index Seeks operator executed later on. To provide accurate remaining time estimates for this Nested Loops join, we must be able to model or predict its speed for processing tuples. We do not have a satisfying solution for it so far, and more effort must be made for solving this problem.

In addition, the speed may be affected by available resources, interaction among different parts of the query, and queries that arrive at or leave the system, etc. While multi-query progress indicators are clearly the ultimate goal, it is our belief that identifying and resolving problems in this simplified setting is a useful step in moving toward addressing more complicated problems related to progress estimation.

VII. Related Work

Recently, there has been an increasing interest in the development of progress indicators for database queries. Previous work can be roughly classified into three categories: commercial progress indicators, research progress indicators, and techniques that can be useful for query progress estimation. In the following, we survey these three categories of work.

The first category includes progress indicators provided by commercial database vendors. Tools for monitoring queries are available in DB2 [1], Teradata [2], Greenplum [15], SQL Server [3], and Oracle [4]. Some progress indicators collect and return statistics (e.g., number of rows and pages processed, current execution operators, etc.) for a given running query. Some progress indicators [2], [15] decompose an execution plan into a number of steps, and indicate which steps are completed/running. Some progress indicators calculate the percent-complete for long running operators [4] or percent-complete for certain validation and recovery statements [3]. These progress indicators are simple and coarse-grained.

The second category includes progress indicators for database queries proposed by research groups. They aim at providing estimates at sufficiently fine granularity. Two pioneering progress indicators are introduced in [5] and [6], respectively. We refer to them as the MSRPI and the WiscPI in the context of this paper. Both the MSRPI and the WiscPI

adopt the idea of dividing a plan into a set of pipelines. The MSRPI calculates the percentage of GetNext() calls finished as an estimation of the current query progress. The follow-up work [7] proves that in the worst case, it is impossible for the MSRPI to provide robust guarantees for the problem of progress estimation. The WiscPI, on the other hand, estimates the remaining execution time by modeling the work of a query as the number of bytes processed at the input and output of pipelines. The two follow-up papers [8], [16] on the WiscPI aim to increase the coverage of the progress indicator to a wider set of SQL queries and extend the single-query progress estimation to enable progress estimation for multiple queries. In [9] and [10], the authors propose a lightweight progress indicator, and they focus on improving cardinality estimation accuracy for various operators in the query plan. Since refining cardinality estimates is not the focus of the paper, we do not incorporate them into our progress indicator. Recently, machine leaning techniques have been adopted for query performance prediction. Kernel canonical correlation analysis is used in [17] to find the relationship between query plan feature matrices (e.g., number of joins and the cardinality sum for the query) and performance feature matrices (e.g., execution time and number of disk I/Os). Regressions are used in [18] to predict the execution time of concurrently running queries. For these two approach, a training model must be built first, and then the model can be used for prediction. For an ad-hoc query, if its characteristics (e.g., number of joins and the operators used) are very different from queries in the training sample, the prediction made by these two approach will be inaccurate. Unlike these two performance predictor, other progress indicators work for any ad-hoc queries. Our work falls in this category. We address the problems faced by existing fine-granularity query progress indicators and provide solutions for improving progress estimation accuracy.

The third category consists of techniques that do not aim at developing progress indicators directly, but can be used by progress indicators instead. The cardinality estimation techniques [19], [20] can provide the basic information for many progress indicators. Both the MSRPI and the WiscPI take advantage of runtime statistical information to refine initial cardinality estimation by optimizers [21], [22]. The cost estimates [23] are exploited by our proposed query progress indicator. Since a challenging problem for progress estimation is to estimate the total work/cost of a query, any method that can be used to increase the cardinality/cost estimation accuracy, either before the query starts or during its execution, fall into the third category.

VIII. Conclusion

Previous progress indicators have made a uniform speed assumption for progress estimation. We present a deeper insight into a query's execution, which directly affects prediction accuracy. We also provides the first performance evaluation for the MSRPI and the WiscPI in the same hardware and software framework, and point out where they do and do not give good estimates. To address their limitations, we introduce a new cost-based progress indicator GSLPI, which utilizes wall-clock pipeline cost to produce higher quality progress estimates. The effectiveness of our techniques are verified with extensive experiments.

This work lays down a foundation for further development of progress indicators. One interesting direction would be to extend our progress indicator for parallel database systems where additional challenges exist (e.g., data skew, new operators, etc.). Another promising direction would be to provide a cost-based progress indicator for multiple concurrently running queries and utilize the information provided by progress indicators for better workload and resource management.

Acknowledgment

This research was supported by a grant from Microsoft Jim Gray Systems Lab, Madison, WI. We would like to thank everyone in the lab for valuable suggestions on this project.

References

[1] DB2, "IBM DB2 query monitor for z/OS," ftp://ftp.software.ibm.com/software/data/db2imstools/whitepapers/db2querymon-wp05.pdf, 2005.
[2] M. Dempsey, "Monitoring active queries with Teradata manager 5.0," http://www.teradataforum.com/ attachments/a030318c.doc, 2001.
[3] Microsoft, "Execution related dynamic management views and functions," http://msdn.microsoft.com/en-us/library/ ms188068.aspx, 2010.
[4] Oracle, "Oracle database data warehousing guide," http://download.oracle.com/docs/cd/B19306_01/server.102/b14223/bi.htm, 2005.
[5] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, "Estimating progress of execution for SQL queries," in *SIGMOD*, 2004, pp. 803–814.
[6] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke, "Toward a progress indicator for database queries," in *SIGMOD*, 2004.
[7] S. Chaudhuri, R. Kaushik, and R. Ramamurthy, "When can we trust progress estimators for SQL queries?" in *SIGMOD*, 2005.
[8] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke, "Increasing the accuracy and coverage of SQL progress indicators," in *ICDE*, 2005.
[9] C. Mishra and N. Koudas, "The design of a query monitoring system," *ACM Trans. Database Syst.*, vol. 34, pp. 1:1–1:51, 2009.
[10] C. Mishra and N. Koudas, "A lightweight online framework for query progress indicators," *ICDE*, 2007.
[11] Microsoft, "SQL server," http://www.microsoft.com/sqlserver.
[12] T. Homepage, "TPC-H benchmark," http://www.tpc.org.
[13] Microsoft, "Logical and physical operators reference," http://msdn.microsoft.com/en-us/library/ms191158.aspx.
[14] K. Morton, M. Balazinska, and D. Grossman, "Paratimer: a progress indicator for MapReduce DAGs," in *SIGMOD*, 2010.
[15] Greenplum, "Database performance monitor," http://www.greenplum.com/pdf/Greenplum-Performance-Monitor.pdf.
[16] G. Luo, J. F. Naughton, and P. S. Yu, "Multi-query SQL progress indicators," in *EDBT*, 2006.
[17] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *ICDE*, 2009, pp. 592–603.
[18] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, "Performance prediction for concurrent database workloads," in *SIGMOD*, 2011, pp. 337–348.
[19] Y. E. Ioannidis and V. Poosala, "Balancing histogram optimality and practicality for query result size estimation," *SIGMOD Rec.*, vol. 24, May 1995.
[20] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," *SIGMOD Rec.*, vol. 25, June 1996.
[21] N. Kabra and D. J. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," *SIGMOD Rec.*, vol. 27, June 1998.
[22] K. W. Ng, Z. Wang, R. R. Muntz, and S. Nittel, "Dynamic query re-optimization," in *SSDBM*, 1999.
[23] M. Jarke and J. Koch, "Query optimization in database systems," *ACM Comput. Surv.*, vol. 16, June 1984.