Toward Progress Indicators on Steroids for Big Data Systems

Jiexing Li, Rimma V. Nehme, Jeffrey Naughton
University of Wisconsin – Madison, Microsoft Jim Gray Systems Lab
jxli@cs.wisc.edu, rimman@microsoft.com, naughton@cs.wisc.edu

1. INTRODUCTION

Recently we have witnessed an explosive growth in the complexity, diversity, number of deployments, and capabilities of big data processing systems (see Figure 1[1]). This growth shows no sign of slowing; if anything, it is accelerating, as more users bring more diverse data sets and more system builders strive to provide a richer variety of systems within which to process these data sets. To process a large amount of data, big data systems typically use massively parallel software running on tens, hundreds, or even thousands of servers. Most of these systems target commodity hardware based clusters, including MapReduce, Hyracks, Dryad, and Microsoft SQL Azure, just to name a few. They achieve scalable performance through exploiting data parallelism.

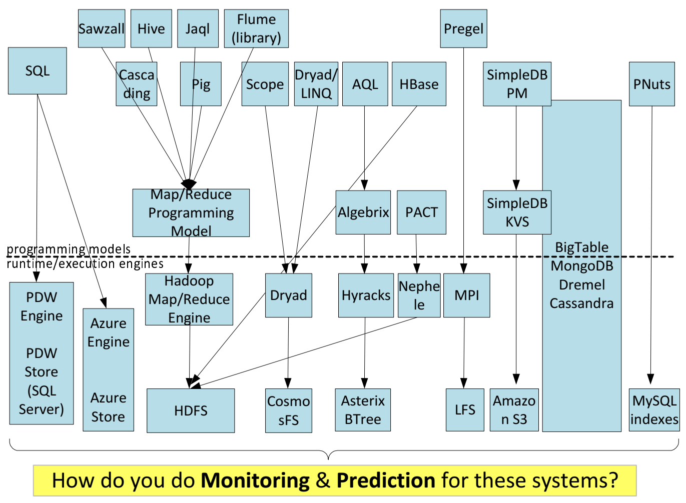


Figure 1: Parallel Data Processing Stacks.

Although parallelism can significantly reduce data processing time, issues such as hardware/software skew, resource contention, and failures are more likely to arise. All big data systems have to face this unwanted but inevitable fact. Due to all these issues, it gets harder to anticipate the future state of a system, and a one-time decision model used by schedulers, optimizers or resource managers will be vulnerable to state changes. For example, suppose that a scheduler initially assigns a task of a map-reduce job to a machine based on the current state of the system. Then another program overloads the disk and significantly slows down the task. If the scheduler does not monitor the system and revise its decision, the whole job will be delayed due to this problem. This unexpected problem can be avoided by monitoring the system's state continuously and rescheduling the task to a different, more lightly-loaded machine. With such a monitor and revise capability, "users" of the system (be they optimizers, schedulers, resource managers, or even humans submitting and waiting for jobs to complete) may be able to improve the usability and performance of the system. Of course, whenever a scheduler or optimizer or human uses current system observations to make a decision, it is implicitly using these observations to make a prediction about the future. But this is currently done in an ad-hoc way, with no systematic approach that makes use of all the information that is available and combines it in a principled way to make predictions. While the capabilities of these big data systems continue to grow impressively as measured by the variety, complexity, and quantity of computations they can provide, there has not been an accompanying growth in the ability of a system to monitor its state, to make useful predictions about future states, and to revise these predictions as time passes.

In some sense this situation is surprising. In a wide variety of complex endeavors, humans have built up mechanisms to make predictions, then to monitor and revise predictions as time passes. Examples can be found in weather forecasting, logistics, transportation, project management, and so forth. Airlines do not simply look at a snapshot of current takeoff and landing queues, make a schedule, and stick with it despite changes in weather. Weather forecasts are continuously revised and updated. Shipping schedules are modified in response to unforseen circumstances or changes in demand. Yet in complex data processing systems one finds many examples of "users" simply making a scheduling or execution plan based on some ad-hoc collection of information, then sticking with this plan no matter what happens subsequently.

But the community has not completely ignored this issue — there has been work in the data management community that does focus on prediction, monitoring, and refining predictions: *query progress indicators* [3, 7, 8, 9, 11, 13]. For readers who have not followed this line of work, these progress indicators begin with a prediction of the running time of the query, and while the query executes, they modify their predictions based on the perceived progress of the query. They may also utilize improved information like intermediate result cardinality estimates. Thus progress indicators fit the "predict, monitor, revise" paradigm.

[1]Figure 1 is based on Figure 3 in [2].

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2012, Asilomar, California, USA.

However, current progress indicator technology is still in its infancy. We suggest that to better serve different "users" of a big data processing system, we need to first change how we view and evaluate progress indicator technology. Then, to fulfill the role we envision for them in big data management systems, current progress indicators need to grow in every dimension. They need to expand the class of computations they can service, expand the kinds of predictions they can make, broaden the sources of inputs they monitor, and increase the accuracy of their predictions. We term such a greatly expanded progress indicator a "progress indicator on steroids."

Readers who are familiar with the progress indicator line of work may wonder: "These progress indicators have a difficult time giving accurate predictions even in artificially simple scenarios such as single queries running on dedicated hardware. Now, since this simple problem appears too difficult to solve, you are proposing that we extend this technology to much more complex domains?" Such readers are correct! But this brings us directly to the point of this paper.

Simply put, we believe that extending the goals of progress indicator technology to modern big data processing scenarios is a task that is well motivated, is rich with challenging and interesting research problems, and has a chance of success, provided that we evaluate and use the technology in a reasonable way. We believe that not only is the current progress indicator technology inadequate for the task, the goals we set for them and the way we evaluate their utility are also off the mark. It is our hope that if the community develops appropriate metrics and uses them to evaluate this technology, and steadily improves the technology itself, we will avoid the suboptimal usability and performance of big data processing systems that could result without this technology.

In the remainder of this paper, we discuss problems with using "classical" progress indicator metrics, enumerate some potential uses for this technology, and touch on some of the challenges that arise when we apply it to complex big data processing systems. Furthermore, we give a small case study that gives a taste of the kind of technical issues that can arise in this area by applying "debug runs," which were developed for map-reduce runtimes, to parallel database query runtime progress indicators.

2. EVALUATING PROGRESS INDICATOR TECHNOLOGY

Prior work has primarily focused on making progress estimation predictions as close to the actual running time as possible. The estimates are mainly used as feedback to users. We believe that to provide more helpful progress estimates for different components in the system, we need to change the "lens" through which we view the progress indicator technology. Next, we will give two motivating examples.

A simple progress score. Take for instance, the progress indicator shipped with the Pig system [14] running map-reduce jobs. It monitors a task's progress using a score between 0 and 1. To compute this score, the 7 phases of a map-reduce job are divided into 4 pipelines, 1 for the map task and 3 for the reduce task as follows: {Record reader, Map, Combine}, {Copy}, {Sort}, and {Reduce}. For a map task, since it only contains 1 pipeline, the score is defined as the percentage of bytes read from the input data. For a reduce task, since it consists of 3 pipelines, each of these phases accounts for 1/3 of the score. In each pipeline, the score is the fraction of the data that has been processed. The overall score of a map-reduce job is computed as the average of the scores of these pipelines. The progress of a Pig Latin query is then the average score of all jobs in the query. As mentioned by other researchers [13], the progress score provided by the Pig system is not accurate. It assumes that all pipelines in the same query contribute equally to the overall score (e.g., it is assumed that all pipelines perform the same amount of work or take the same amount of time to process), which is rarely the case in practice. Despite the fact that the estimate is not perfect, this simple progress score has been used by Hadoop for selecting stragglers (tasks making slow progress compared to others) and scheduling speculative executions of these stragglers on other machines, with the hope of finishing the computation faster. Google has noted that speculative execution can improve a job's response time by 44% [4]. Follow-up work proposed a new scheduler, LATE, which further improved Hadoop response times by a factor of 2 by using a variation of this score [17].

A state-of-the-art progress indicator. Later, ParaTimer [12] was proposed for Pig Latin queries to provide more accurate time-oriented progress estimates. These progress estimates can be used to handle stragglers in a different way. When data skew happens (e.g., some tasks need to process more data than others), tasks with too much input data will be considered as stragglers. Because these tasks run more slowly due to their input data and not the machine where they have been scheduled, speculative executions of these tasks on other machines will not reduce their execution times. An alternative approach is needed. SkewTune [6] handles this problem by utilizing ParaTimer. When a slot becomes available and there are no pending tasks, the task with the longest expected remaining time (determined by ParaTimer) is selected as a straggler. The unprocessed input data of this task is repartitioned, and then they are added to existing partitions to form new partitions, such that all new partitions complete at the same time. This detection-repartition cycle is repeated until all tasks complete. Their experimental results show that using this technique can significantly reduced job response time in the presence of skew.

For the first example with the simple progress score, although the score is only a rough approximation of the actual query execution time, the information it provides is sufficient for identifying stragglers. Thus, it is helpful for scheduling speculative executions. For the second example, identifying a straggler is not sufficient. To make all new partitions complete at the same time, SkewTune must also know more precise execution time of the tasks. In both cases, rather than using the progress estimates as user feedback, they are used by the system to identify the bottlenecks (e.g., the longest running tasks) and help reduce the response time.

Inspired by these examples, it seems that it may not always worth pushing hard on improving estimation accuracy. For some tasks, the current progress indicators are good enough to be useful. Instead, when evaluating progress indicator technology, it would be better to focus on the following questions: (1) Is the progress indicator good enough, so that it is more helpful than not for *specific* tasks? By analogy, query optimizer cost estimates do not have to be perfect to be useful. Similarly, progress indicator estimates do not have to be perfect to be useful for, say, scheduling. (2) Is the progress indicator accurate when nothing in its current universe of knowledge changes? Specifically, it would be silly to start a progress indicator, then add 100 "monster" queries, and say "*wow, that initial prediction was really terrible*!" As we mentioned before, when it comes to big data processing systems, future states are harder to predict. As "monster" queries start running on a system, as skew and failures happen, some tasks will slow down and become stragglers. Predicting the stragglers ahead of time may become too overwhelming for progress indicators. Instead, we should ask, how good is a progress indicator given the information available at the time? (3) Finally, does the progress indicator react to changes quickly? When the situation does change, how long does it take a progress indicator to get to a good prediction given the new information? A progress indicator that takes forever to adjust, but is very accurate when it does, may be less useful than one that adjusts immediately but is somewhat inaccurate. For the two examples we discussed, the sooner we can identify the stragglers, the sooner we can schedule speculative executions and repartitions.

3. PROGRESS INDICATORS ON STEROIDS

With this new view of progress indicators and how we evaluate this technology for big data systems, we discuss the need for a new type of progress indicators — the so called "progress indicators on steroids" that grow in every dimension to fulfill the role we envision for them.

First of all, we need to expand the set of components they can serve. In general, we believe that this technology can be beneficial to any computation that makes decisions based on a cluster state. The more the computation relies on the state prediction, the better decisions it can make by detecting the unexpected state changes. Here, we list many use cases for progress indicator technology, including but not limited to:

1. **User Interface**. Users would always like to get accurate feedback about query execution status, especially if queries tend to be long-running.
2. **Scheduling**. By using progress indicator technology to obtain the current state of a cluster and roughly predict its future state, a scheduler can likely do a better job at determining the order in which to run jobs than it could without such information.
3. **Resource Management**. Progress indicators can help in limiting resource contention in a distributed system, and help decide when to re-allocate work, so that system resources are efficiently utilized.
4. **Skew Handling and Stragglers**. By using progress indicator technology, skew can be detected, and the system can proactively response to skew in a way that fully utilizes the nodes in the cluster while still preserving the correct query semantics.
5. **Query Optimization**. Dynamic re-optimization is likely to be increasingly valuable in parallel data processing systems. Progress indicator technology can be useful in providing up-to-date query monitoring data, as well as possible future predictions about processing speed, resource availability and so forth, which can aid in re-optimization decisions.
6. **Performance Debugging**. Debugging queries and system performance is a very challenging task. Progress indicator technology can be useful here as well. For instance, the length of query runtime could be estimated prior to running, and then in the first few minutes of running and then used in identifying the possible reasons for why the process may have slowed down.

Second, we need to expand the kind of information progress indicators can provide by expanding their sources of inputs. So far, progress indicators are mainly used to predict the percent finished or the remaining execution time of a task. This is also the most obvious and direct usage of progress indicators. We believe that there is much more useful information we can derive from the progress indicator technology. For example, ranking a set of tasks by their completion percentage will reveal which ones are stragglers that may need more attention. Furthermore, we can have a deeper analysis of the collected statistics for these stragglers to automatically detect their causes. In addition, the average progress made by tasks running on a specific machine will indicate whether the machine is healthy, so that more tasks or speculative executions of stragglers can be assigned to these machines to make better progress. When a task finishes, its resources are released. Knowing the remaining execution time of the running tasks and the resources reserved for them will give us an idea about resource availability in the future. An intriguing possibility that has not yet been explored to date is the usage of changes in estimates over time as a first-class object for analysis (rather than the estimates themselves). For example, while Query 1 is running, Query 2 is started; the difference in the estimates for Query 1's running time before and after Query 2 was started may give useful, actionable information even when both estimates are inaccurate in the absolute sense.

So far we have emphasized the necessity of a new view and the importance of a new role for progress indicators. As always, we should strive to increase the accuracy of their prediction. For some use cases, however, such as identifying stragglers and healthy machines, rough progress estimates are good enough. For other use cases, such as predicting resource availability, more accurate progress estimates are required. Admittedly, this is a very challenging problem. Later, we will discuss some of the technical challenges that need to be tackled in order to produce accurate progress estimates.

4. HOW DO YOU BUILD IT?

We believe that progress indicator estimation is not a "cook book" process: while the general problem is always the same, the specific system being monitored and the task for which it is being monitored will dictate what the progress estimator should examine and how it should use this information.

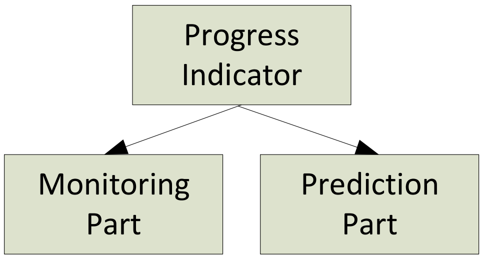


Figure 2: The two main components of a progress indicator.

However, we believe a simple and clean abstraction of progress indicator's main components will pave the way to a more systematic approach to this technology. One way to do it is by considering its two main components (see Figure 2):

1. **Monitoring**. Monitoring data tell you what is actually happening in the system. The monitoring component collects statistics and observation data, such as the number of queries running and waiting in the queue, the number of tuples that have been processed or are waiting to be processed, and resource usage, such as CPU, RAM, or network. This observation data may be collected from a variety of sources including the optimizer, query executor, automated profiler, scheduler, load profiler, or logger. Each observation provides one piece of the puzzle that makes up a total picture of the system.

2. **Prediction**. The prediction component is where analytics about progress takes place. Predictive analytics may perform some statistical analysis by extracting information from the observed data (from the monitoring component) and using it to predict future trends and behavior patterns.

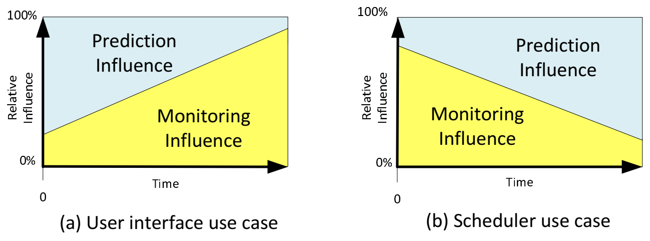


Figure 3: Relative Influence Diagrams in Progress Estimation.

Another part of the progress estimation involves the *relative weighting* of monitoring versus prediction. To be a bit more precise, progress estimators are always making predictions; the question we are addressing here is how heavily the prediction should be influenced by information external to the information gathered about the task while it is running (which we term "prediction"), and how much it should be influenced by information gathered during the execution of the task (which we term "monitoring").

Figure 3 is intended to show two possible scenarios of the relative influence of monitoring versus prediction on the progress estimation for 2 use cases: UI and scheduler. For example, in (a) depicting a UI use case, as the running query progresses, the impact of monitoring vs. prediction increases. This is perhaps intuitive; when the query is deep into its execution it makes sense that predictions about its future behavior rely more heavily on recently observed true cardinalities and processing costs than on pre-execution optimizer estimates.

As a contrasting example, suppose that in (b) we are considering a scheduling problem where a large number of relatively short tasks are being run, and the workload of computational tasks being presented to the system is relatively stable. In this case when predicting the performance of a given task, it may make sense to weight the *a priori* predictions of task performance based on the stable history of the system more heavily than predictions based upon recent fluctuations about the stable baseline monitored recently.

Our point here is not to give specific rules of thumb to decide which component (monitoring vs. prediction) should be weighted more heavily; rather, it is to highlight that there is an issue to be considered here, and there is a need for techniques to determine proper weightings.

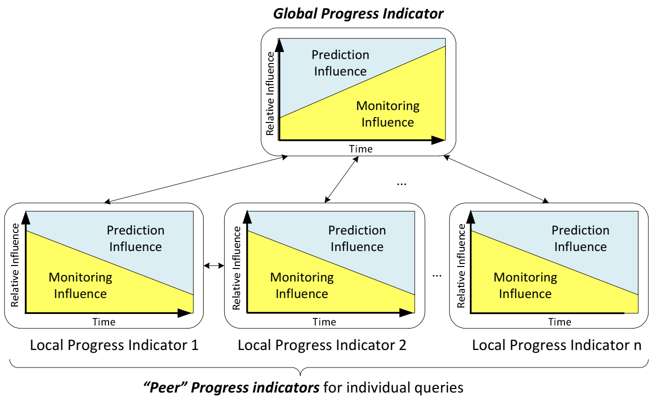

Global Progress Indicator
Relative Influence
Prediction Influence
Monitoring Influence
Time
...
Local Progress Indicator 1
Local Progress Indicator 2
Local Progress Indicator n
"Peer" Progress indicators for individual queries

Figure 4: The hierarchical progress indicator.

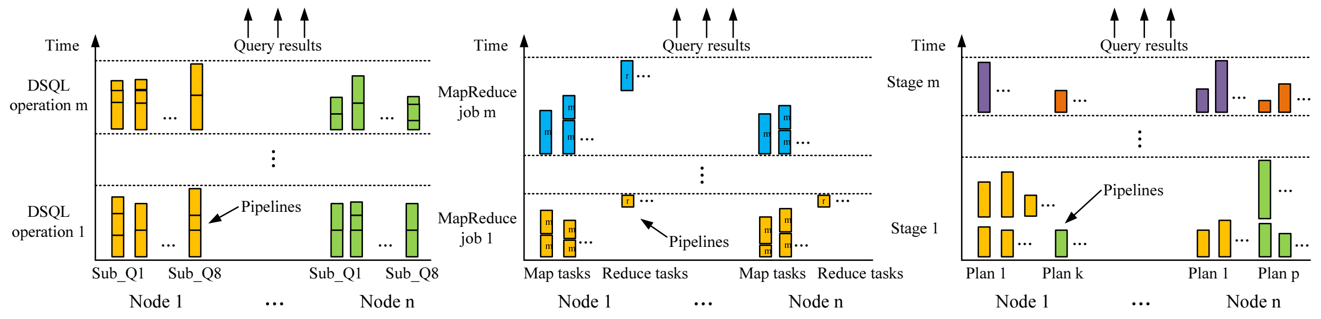
We close this section with some more thoughts on multi-query progress estimation. Most work on progress estimation has focused on individual-query progress estimation. However, in real-life queries don't run in isolation. For multiple query progress indicators, we envision a possible *hierarchical progress indicator framework* (see Figure 4), where each query may run its own *local* progress indicator with monitoring and prediction and we have an overall *global* progress indicator collecting and aggregating the information from the local progress indicators. Individual local progress indicators could also give feedback to other sibling progress indicators, especially if they are running on the same node. Note also that the relative influence of monitoring versus prediction may be different at each level in the hierarchical progress indicator; for example, leaf nodes may give more influence to monitoring, while the global (root) progress indicator may give more influence to prediction.

5. TECHNICAL CHALLENGES

A number of technical challenges must be addressed to augment and extend existing technology and make progress indicators for big data management systems a reality. In this section we highlight a few of these challenges.

1. **Pipeline Definition & Shape**. In the context of progress estimation, execution plans are partitioned into multiple pipelines to gain insight into intermediate progress. A challenge here is that as we transition to more diverse and complex systems, each (sub)system may have its own execution model, and thus as a result may have different pipeline definitions and shapes. For parallel DBMS, the number, shape, and execution speed of pipelines may be different on different nodes (see Figure 5(a)). For example, because each node processes different data partitions, the query plans for the subqueries running on Node 1 (in yellow) are very different from the query plans for the subqueries running on Node n (in green), even if they have identical query text. As a result, each node may have different pipelines. For Map-Reduce [1] systems (see Figure 5(b)), a pipeline consists of either a map or a reduce step, and the reduce tasks will not start before all map tasks in the same job have finished. For example, for the first map-reduce job (in yellow), the reduce tasks start after all map tasks are finished. Unlike the parallel DBMS, pipelines running on all nodes are identical. For DAG-based execution engines [5, 16] (Figure 5(c)), the variations in the number of pipelines, their shapes, execution speeds, and which nodes they execute on may be even more drastic. For example, Plan 1 (in yellow) is running across multiple nodes, and each node process different number of pipelines concurrently. Even for the the same plan running on the same node, the number of concurrent pipelines may vary for different phases. While for a map-reduce job, we have fixed number of slots per node. The shape of the pipelines will highly impact the degree of parallelism and contention of resources, and as a result, the query execution time. Progress indicator technology must be extended to handle such diversity.

2. **Work Definition**. A fundamental question in progress estimation is how to define a unit of work, and how to measure how fast the system is processing these units of work. Previous work has characterized work as the number of tuples [3] or the number of bytes [9] that must be processed. While these are relatively easy quantities to measure (although not always easy to predict!), they are in a sense too high level to be useful without some interpretation. For example, not every tuple or every byte to be processed takes the same amount of system resources. One could rectify this problem in two ways. The first is to explicitly capture the fact that different units of work take different amounts of resources to process. The second is to assert that the resources needed to process a



(a) Pipelines in shared-nothing parallel database (b) Pipelines in Map-Reduce (c) Pipelines in DAG-based execution

Figure 5: Pipelines in different runtime execution engines.

unit of work are constant, but that processing different bytes or tuples takes different amounts of work. Although the computations may be similar in both approaches, our intuition (perhaps gained from physics) is that work should correlate with resources expended, so we find the second approach more attractive.

3. **Speed Estimation**. Whatever definition of work is adopted, 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. Here again the choice of work unit affects the computation. Previous progress indicators have in effect adopted the tuple or byte as the unit of work, and assumed that each unit of work takes the same number of resources to process. However, this assumption is rarely true in reality, and as has been shown in [8] can lead to poor progress estimates. More research is needed on the definitions of speed and work and how they interact.

4. **Dynamicity**. Parallel data processing clusters tend to be dynamic. Nodes can join or leave a computation (perhaps due to failures and restarts), and concurrent, competing computations can start up or complete. A successful progress indicator will be one that detects these changes and reports them "as soon as possible."

5. **(Lack of) Statistics**. Virtually all progress indicator frameworks rely on statistics over data to make predictions. Making accurate progress estimations with no or limited statistics (as may be required in systems that process data stored in uninterpreted files rather than relational tables) is a substantial challenge, and one that is perhaps most likely to be solved by a combination of data analysis before execution and statistics collection/correction during execution.

6. **Parallelism**. Pipeline parallelism is typically used for inter-operator parallelism in order to overlap the execution of several operators within a query. Data parallelism, on the other hand, is applicable for both inter- and intra-operator parallelism and requires a data partitioning so that different (sub) operators can concurrently process disjoint sets of data. Parallelism adds a level of complexity when estimating progress.

7. **Conveying Information in Input-Level Terminology**. A user typically submits a request (query) using a higher-level language (e.g., SQL, HiveQL, Pig, SCOPE, PACT). However, such separation of the programming model from the concrete execution strategy creates a challenge of translating an actual execution plan to what the user has actually submitted. Consider for example a HiveQL query: a HiveQL query gets translated into a sequence of map-reduce jobs. While we may predict the progress of an individual Map or Reduce pipeline rather accurately, translating this progress back into the overall progress of the original HiveQL query is a non-trivial task.

6. A SMALL CASE STUDY

Due to the complexity and diversity in different data processing systems (e.g., different pipeline definition/shape and different operators in the pipelines), technology developed for a specific system may not work for other systems at all. In this section we present a preliminary investigation of the issues we encountered when we attempted to apply a debug run-based progress indicator that was designed and well-suited for map-reduce jobs to a parallel database system. Briefly, before a computation begins, every progress indicator has to estimate how much work the computation will involve and how fast the work can be processed. One very natural and promising way to do this estimation was proposed in two recent papers on progress indicators for Map-Reduce workloads [12, 13]. The idea is to use a "debug run" — that is, to run the computation on a subset of the data, see how long that takes, then scale this quantity to predict the running time on the full data set. Our goal here is to attempt to apply this approach to query progress indicators for a parallel database system, specifically Microsoft's SQL Server Parallel Data Warehouse (PDW) [10, 15].

6.1 Cluster Setup

All our experiments were conducted with SQL Sever PDW. We used a version of PDW as of August 2012. PDW was run on a cluster of 18 nodes, including one control node, one "landing node," and 16 compute nodes. (The landing node is responsible for data loading only and does not participate in query execution.) All nodes were connected by a 1Gbit HP Procurve 2510G 48-port Ethernet switch. Each node had 2 Intel Xeon L5630 quad-core processors, 32GB of main memory, and 10 SAS 10K RPM 300GB disks. One of the disks was reserved for the operating system, while eight were used for storing data. Each node ran SQL Server 2008, which was limited to at most 24GB of memory.

We used a TPC-H 1TB database for our experiments. Each table was either hash partitioned or replicated across all compute nodes. When a table is hash partitioned, each compute node contains 8 horizontal data partitions, and they are stored on 8 different disks, respectively (so there are a total of 8*16 data partitions for a hash partitioned table). Table 1 summarizes the partition keys used for the TPC-H tables. Replicated tables were stored at every node on a single disk.

Table	Partition Key	Table	Partition Key
Customer	c_custkey	Part	p_partkey
Lineitem	l_orderkey	Partsupp	ps_partkey
Nation	(replicated)	Region	(replicated)
Orders	o_orderkey	Supplier	s_suppkey

Table 1: Partition key for TPC-H tables

6.2 Experimental Setup

Recall that we wish to evaluate the effectiveness of a debug run approach in estimating work and processing speed. To do so, we tried running queries on a 1% sample set first (1% sampling was used for Parallax and ParaTimer). Then the same query was run on the full (original) data set, and we estimated the progress of the query using a progress indicator based on the prior run.

We found that the debug run approach was effective for some queries. Figure 6 depicts the estimated remaining time for $Q1$ provided by a debug run-based progress indicator we implemented in PDW. $Q1$ consists of 7 different pipelines, and there are no joins between tables. Our progress indicator was configured to wake up every 5 seconds, using statistics collected to make a prediction about how long the query is going to take.

However, we found that there were two reasons why in general the debug run approach was problematic. We discuss these in the following two subsections.

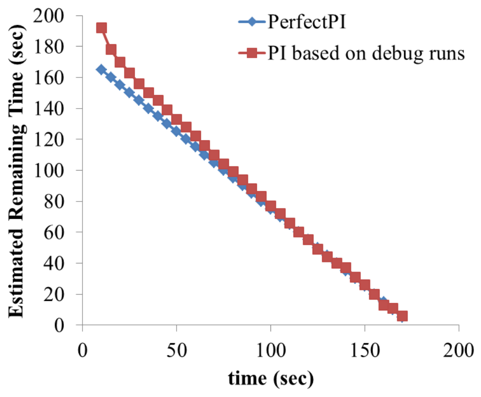


Figure 6: Progress estimation for $Q1$

6.3 Multi-Input Pipelines

The debug run-based approach was much more problematic when we moved to queries with workflows more general than the typical map-reduce linear pipeline. Specifically, with the multiple input workflows that result from joins, the sample search space grows as the product of the size of the input tables, and a 1% sample of each of the inputs is no longer effective. This is an instance of a well-known problem in sampling over relational query plans — the cross product of the input relations is so large that one must take unreasonably large samples of the input relations to even observe any joining tuples. Of course, one can try many different approaches to attack this problem, but they all involve taking some kind of “correlated” sample between the tables, which can either be expensive or inaccurate (due to correlations) or both.

As an example of this, Figure 7 shows the number of tuples surviving each query step (roughly speaking, a query step is a portion of a query plan delineated by data movement operations; please refer to [15] for more details about steps in PDW) of $Q4$ running on both the sample data and the full data. Note that the y axis is log scale. Figure 8 shows the ratio between them. As we can see, the

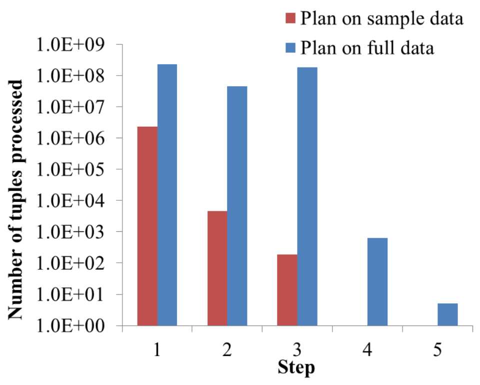


Figure 7: Number of tuples processed in each step of $Q4$

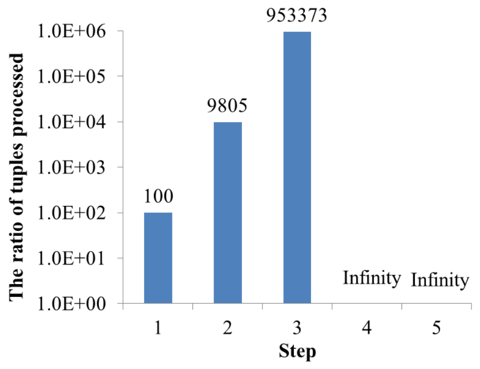


Figure 8: The tuple processed ratio between two plans of $Q4$

later steps in the debug run query process many fewer tuples than the corresponding steps in the original run, and after each step, the number of tuples processed is reduced by roughly another 99%. In fact, the debug run has no tuples to process in step 4 and step 5, which makes it impossible to estimate their processing speeds.

Even for steps that do yield a few tuples things are problematic. One issue is that for debug runs over a few tuples, startup overhead dominates and the debug run grossly overestimates the time to process a tuple and hence the future running time. $Q17$ is one of such example, as shown in Figure 9. The estimated tuple processing speed for a pipeline started after 360 seconds is highly inaccurate.

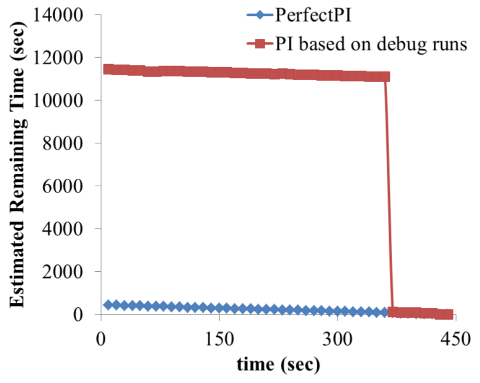


Figure 9: Progress estimation for $Q17$

6.4 Cost-based Optimization

Even if the large sample space problem were solved, we encountered another interesting problem when applying the debug run approach that worked so well for map-reduce to PDW. In a nutshell, the problem is that PDW, like virtually all relational database management systems, employs a cost-based optimizer to choose an execution plan. While it is useful to have an optimizer, in this case it works at cross purposes to our goals because the optimizer often chooses a different plan for the 1% sample data set than it does for the full data set (for the TPC-H queries, the execution plans generated for the full data set and the sample data set were only the same for 6 out of 22 queries). In such situations it is likely that the debug run will be a poor predictor of the execution characteristics of the full computation.

Query	1% sample with act. stat.		1% sample with fake stats.		1TB
	Time (sec)	Estimate (sec)	Time (sec)	Estimate (sec)	Time (sec)
Q1	3.0	47	5.5	223	198
Q2	6.7	539	4.5	330	91
Q3	6.5	3119	5.8	5780	1210
Q4	1.3	27	3.6	200	45
Q5	9.4	$1.1\text{x}10^9$	8.8	410766	77
Q6	0.8	24	0.9	23	33
Q7	11.4	714	6.2	514371	65
Q8	10.1	1041	7.9	958601	64
Q9	8.7	36238	8.5	$2.5\text{x}10^7$	181
Q10	6.0	10556	4.0	6450	71
Q11	7.3	577	4.2	304	109
Q12	0.9	30	3.6	198	50
Q13	17.4	1526	7.9	421	186
Q14	7.5	450	2.5	97	47
Q15	19.1	179	6.3	315	102
Q16	2.3	4317	5.7	5058	71
Q17	371.4	$1.26\text{x}10^6$	14.7	11527	432
Q18	14.1	11	6.0	405	122
Q19	2.0	77	2.0	61	58
Q20	21.0	$1.2\text{x}10^7$	5.9	380202	87
Q21	7.6	49677	5.5	199336	271
Q22	17.3	1369	5.0	143	77

Table 2: Initial estimated remaining time for TPC-H queries

To evaluate the impact of this effect, we tested the TPC-H queries evaluated in two ways. In the first, we kept the progress indicator deliberately oblivious to the differences in the plans between the debug and execute runs. That is, we simply viewed each execution of a step as a black box that processed some number of input tuples and generated some number of output tuples within a certain time interval. In the second, we opened up this black box and ensured that the debug run actually used the same plans as the full run. We did this by deliberately tricking the optimizer — we manually set the statistics during the debug run to match those for the full run, so the optimizer thought the full data set was present.

The results are shown in Table 2. As mentioned, among 22 queries, only 6 queries used the same plans for debug runs and actual runs. These queries are $Q2$, $Q6$, $Q10$, $Q11$, $Q19$, and $Q21$. The first column shows the execution time for the debug queries with the compute nodes treated as a black box (without paying attention to plan selection), while the second column shows the initial estimates for the full queries based on these debug queries. The third column shows the running times when we "tricked" the optimizer to use the same plan for the debug run and the full query run, while the fourth column shows the estimate for the full query based upon this debug run with the same plan. Finally, the fifth and last column shows the actual execution time of the queries over the original 1TB data.

Interesting, for those 6 queries that used same plans for debug runs and actual runs, the estimated remaining times are not the same in the second column and the fourth column, and for some of the queries, the difference is quite obvious, such as $Q10$ and $Q21$. Since the sample data does not contain many tuples, we may get high variance in the execution time of debug runs (e.g., the difference between 4 seconds and 6 seconds is already 50%). As a result, the estimated remaining time could be very different. In addition, the variance has different influence on the estimates, depending on in which step the difference happens.

We note that the debug runs that were tricked into using the same plans as the full query runs gave better, and sometimes substantially better, estimates than the ones that used different plans for the debug and full query run. This is an example of a general and interesting question in multiple layered systems: should we treat the lower level as a black box, or "open it up" to improve the prediction of the progress estimator?

In the context of relational database query evaluation, even ignoring the impact of the query optimizer, there are some additional reasons that may make a small debug run a poor predictor of a full computation. For example, for relational operators such as Sort, Hash Join, and Nested Loop Join, the per-tuple processing time changes as the number of input tuples changes, as these operators behave differently for different sized inputs. Moving beyond relational systems, similar issues will arise in other scenarios; for example, suppose we are running a computation on top of a distributed file system that tries to improve performance by deciding when to ship data and when to ship computations, perhaps as a function of the amount of data to be processed. One could likely see substantial differences between debug and full compute runs.

7. CONCLUSION

Very large data processing is increasingly becoming a necessity for modern applications. For state of the art parallel data processing systems, systematic monitoring and prediction of progress is a problem. In this paper, we propose the development of "progress indicators on steroids" for big data systems. The progress indicators we envision greatly expand and improve upon current progress indicator technology in every aspect. We believe the development of this technology represents both a tremendous opportunity for research and a much-needed component in big data systems. It is our hope that the community will begin addressing the challenges inherent in building this technology as a step toward enabling a new generation of data processing systems that are both user-friendly and efficient so as to more effectively serve the needs of modern applications and users.

8. REFERENCES

[1] Hadoop. http://hadoop.apache.org.

[2] A. Ailamaki, M. J. Carey, D. Kossman, S. Loughran, and V. Markl. Information management in the cloud. *Dagstuhl Reports*, 1(8):1–28, 2011.

[3] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating progress of long running SQL queries. In *SIGMOD Conference*, pages 803–814, 2004.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[5] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, Mar. 2007.

[6] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[7] Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SoCC*, pages 75–86, 2010.

[8] J. Li, R. V. Nehme, and J. F. Naughton. GSLPI: a cost-based query progress indicator. In *ICDE*, pages 678–689, 2012.

[9] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Toward a progress indicator for database queries. In *SIGMOD*, 2004.

[10] Microsoft Corporation. Microsoft SQL Server PDW. `http://www.microsoft.com/sqlserver/en/us/solutions-technologies/data-warehousing/pdw.aspx`.

[11] C. Mishra and N. Koudas. A lightweight online framework for query progress indicators. In *ICDE*, 2007.

[12] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for MapReduce DAGs. In *SIGMOD*, 2010.

[13] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *ICDE*, pages 681–684, 2010.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD ’08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[15] S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. DeWitt, and C. Galindo-Legaria. Query optimization in Microsoft SQL Server PDW. In *SIGMOD*, 2012.

[16] D. Warneke and O. Kao. Nephele: efficient parallel data processing in the cloud. In *MTAGS*, 2009.

[17] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI’08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.