Toward A Progress Indicator for Parallel Queries*

* University of Washington Technical Report UW TR: #UW-CSE-09-07-01

Kristi Morton, Abe Friesen, Magdalena Balazinska, Dan Grossman

Computer Science and Engineering Department, University of Washington
Seattle, Washington, USA
{kmorton,afriesen,magda,djg}@cs.washington.edu

***Abstract*— In parallel query-processing environments, *accurate*, *time-oriented* progress indicators could provide much utility to users given that queries take a very long time to complete and both inter- and intra-query execution times can have high variance. In these systems, query times depend on the query plans and the amount of data being processed, but also on the amount of parallelism available, the types of operators (often user-defined) that perform the processing, and the overall system load. None of the techniques used by existing tools or available in the literature provide a non-trivial progress indicator for parallel queries. In this paper, we introduce Parallax, the first such indicator. Several parallel data processing systems exist. In this paper, we target environments where queries consist of a series of MapReduce jobs. Parallax builds on recently-developed techniques for estimating the progress of single-site SQL queries. It enhances and extends these techniques in non-trivial ways. We implemented our estimator in the Pig system and demonstrate its performance on experiments with the PigMix benchmark and other queries running in a real, small-scale cluster.**

I. Introduction

Over the last several years, our ability to generate and archive extremely large datasets has dramatically increased. Modern scientific applications – for example, those driven by widely-distributed sensor networks or simulation-based experiments – are producing data at an astronomical scale and rate [18], [31], [32]. Companies are increasingly storing and mining massive-scale datasets collected from their infrastructures and services (e.g., eBay, Google, Microsoft, Yahoo!).

To analyze these massive-scale datasets, users are turning to parallel database management systems [1], [12], [15], [30], [33] and other parallel data processing infrastructure [7], [10], [13], [16]. Although these systems significantly speed-up query processing, individual queries can still take minutes or even hours to run due to the sheer size of the input data.

When queries take a long time to complete, users need accurate feedback about query execution status [24]. In particular, they need a reasonably accurate estimate of how much time remains for their query to run. Such feedback can help users plan their time. It can also help users decide if they need to take action, such as canceling a query and restarting it later or on a system with different resources. Unfortunately, existing parallel systems provide only limited feedback about query progress. Most systems simply report statistics about query execution [5], [6], [8], [11], at best indicating which operators are currently running [8], [11], [13]. Such indicators, however, are too coarse-grained and inaccurate because different operators can take wildly different amounts of time.

In this paper, we address this limitation by introducing *Parallax, the first, non-trivial time-oriented progress indicator for parallel queries*. We developed our approach for Pig queries [27] running in a Hadoop cluster [13], an environment that is a popular open-source parallel data-processing engine under active development. As an initial step, we focused on Pig Latin queries that compile into a series of MapReduce [7] jobs. Hence, our current indicator does not handle joins. While the key ideas behind our technique are mostly not specific to the Pig/Hadoop setting, this environment poses several unique challenges that have informed our design and shaped our implementation. Most notable, user-defined functions (UDFs) are the norm, there is no query optimizer to estimate costs statically, and a MapReduce-style scheduler requires intermediate result materialization, schedules small pieces of work at a time, and adds a significant start-up cost to each newly scheduled query fragment.

Parallax is designed to be accurate while remaining simple and addressing the above Pig/Hadoop-specific challenges. At a high level, Parallax works as follows. First, Parallax uses statistics collected from previous runs of the same query to estimate input cardinalities and processing costs (in msec/tuple) for different fragments of the query plan. Such a run is typically a debug run on a user-generated representative data sample, a common occurrence in a cluster setting. Second, Parallax collects runtime statistics and combines them with the offline ones to produce its estimates. To deliver accurate estimates, Parallax takes into account (1) the number of tuples that remain to be processed by different fragments of the query plan, (2) the degree of parallelism of each fragment, and (3) any available data skew information. Parallax estimates the speed with which different fragments will process input tuples and uses these estimates to predict query times. Additionally, Parallax reacts to changes in runtime conditions (e.g., changes in degrees of parallelism or cluster load) and updates its estimates accordingly.

A. Building on Single-Node Query Estimation

Parallax builds on prior art in single-node SQL query indicators [2], [4], [19], [20], [21], [22], [23]. Similar to

single-node indicators, Parallax monitors the number of tuples processed at different points in the query plan and keeps track of the expected number of tuples that remain to be processed. Parallax also monitors the current processing speeds at these different points in the query plan. Extending this work to the parallel Pig/Hadoop setting requires changing the treatment of (1) query fragments that have not begun executing (because there is no query optimizer to estimate their cost) (2) parallelism (estimating how a cluster of machines will parallelize query fragments and how data skew will affect processing times), and (3) dynamically varying distributed-system conditions (unpredictable slowdowns and speedups).

All progress indicators are vulnerable to errors in cardinality estimation and previous techniques propose various mechanisms to mitigate this effect [4], [20], [22], [23]. In this paper, we do not focus on this problem. Rather, given some cardinality estimates, we focus on the problem of estimating query times in a parallel setting. We show that, even with perfect cardinality estimates, query time prediction in a cluster is a challenge. Errors in cardinality estimates, of course, magnify query time estimation errors.

B. Our Setting and Overall Results

Today, parallel systems are being deployed at all scales and each scale raises new challenges. In this paper, we focus on smaller-scale systems with tens of servers because many consumers of parallel data management engines today run at this scale even if they run in larger data centers. We thus evaluate Parallax's performance through experiments on a small eight-node cluster (set to a maximum degree of parallelism of 32). We compare its performance against three state-of-the-art single-node progress indicators from the literature [4], [20] and Pig's current progress indicator. We show that Parallax is more accurate than all these alternatives on a variety of types of queries and system configurations. For a large class of queries, Parallax's average accuracy is within 5% of an ideal indicator (in the absence of cardinality estimation errors).

C. Summary of Contributions and Paper Organization

In summary, the key contribution of this paper is to develop and *fully implement* techniques for estimating completion times for parallel queries running in a small-scale cluster with degrees of parallelism in the tens of processes. The queries that we study are Pig Latin queries that compile into a series of MapReduce jobs. Parallax does not currently handle all possible scenarios such as failures or speculative execution. It is, however, an important step toward providing accurate feedback to users in parallel systems.

II. Background

In this section, we present an overview of MapReduce [7], Pig [27], and Pig's current progress indicator.

A. MapReduce

MapReduce [7] (with its open-source variant Hadoop [13]) is a programming model for processing and generating large data sets. The input data takes the form of a file that contains key/value pairs. For example, a company may have a dataset containing pairs with a sequence number and a search log entry. Users specify a *map* function, which is similar to a relational groupby operator, that iterates over this input file and generates, for each key/value pair, a set of intermediate key/value pairs. For example, a map function could filter away uninteresting search log entries and group the remaining ones by time. For this, the map function must parse the value field associated with each key to extract any required attributes. Users also specify a *reduce* function that, similar to a relational aggregate operator, merges or aggregates all values associated with the same key. For example, the reduce function could count the number of log entries for each time period.

MapReduce jobs are automatically parallelized and executed on a cluster of commodity machines: the map stage is partitioned into multiple *map tasks* and the reduce stage is partitioned into multiple *reduce tasks*. Each map task reads and processes a distinct chunk of the partitioned and distributed input data. The degree of parallelism depends on the input data size. The output of the map stage is hash partitioned across a configurable number of reduce tasks. Data between the map and reduce stages is always materialized. As discussed below, a higher-level query may require multiple MapReduce jobs, each of which has map tasks followed by reduce tasks. Data between consecutive jobs is also always materialized.

B. Pig

To extend the MapReduce framework beyond the simple one-input, two-stage data flow model and to provide a declarative interface to MapReduce, Olston et. al developed the Pig system [27]. In Pig, queries are written in Pig Latin, a language that combines the high-level declarative style of SQL with the low-level procedural programming model of MapReduce. Pig compiles these queries into ensembles of MapReduce jobs and submits them to a MapReduce cluster.

For example, consider the following SQL query, which corresponds to the above example:

```
SELECT S.time, count(*) as total
FROM  SearchLogs S
WHERE Clean(s.query)
GROUP BY S.time
```

This example translates into the following Pig Latin script:

```
raw      = LOAD 'SearchLogs.txt'
           AS (seqnum,user,time,query);
filtered = FILTER raw BY Clean(query);
groups   = GROUP filtered BY time;
output   = FOREACH groups GENERATE $0 as time, count($1) as total
STORE output INTO 'Result.txt' USING PigStorage();
```

This Pig script would compile into a single MapReduce job with the map phase performing the user-defined filter and group by operations and the reduce phase computing the count for each group.

Because Pig scripts can contain multiple filters, aggregations, and other operations in various orders, in general a query will not execute as a single MapReduce job. For example, one of the two sample scripts (script1) distributed with the Pig

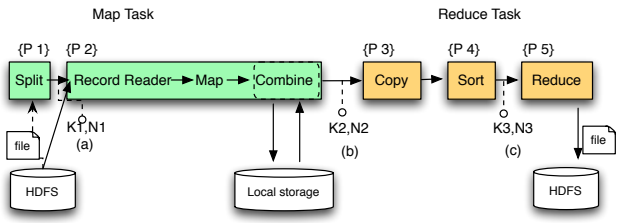


Fig. 1. Detailed phases of a MapReduce Job. Each N_i indicates the cardinality of the data on the given link. K_i's indicate the number of tuples seen so far on that link. Both counters mark the beginning of a new pipeline (Section III).

system compiles into a sequence of five MapReduce jobs (we test this script in Section V).

C. MapReduce Details

Within a MapReduce job, there are seven phases of execution, as illustrated in Figure 1. These are the split, record reader, map runner, combine, copy, sort, and reducer phases. The split phase does minimal work as it only generates byte offsets at which the data should be partitioned. For the purpose of progress computation, this phase can be ignored due to the negligible amount of work that it performs. The next three phases (record reader, map runner, and combine) are components of the map and the last three (the copy, sort, and reducer phases) are part of the reduce. An important point to note is that the Pig operator code only executes within the map runner and reducer phases; the other phases never change.

The record reader phase iterates through its assigned data partition and generates key/value pairs from the input data. These records are passed into the map runner and processed by the appropriate operators running within the map function. As records are output from the map runner, they are passed to the combine phase which, if enabled, sorts and pre-aggregates the data and writes the records locally. If the combine phase is not enabled, the records are sorted and written locally without any aggregation.

Once a map task completes, a message is sent to waiting reduce tasks informing them of the location of the map task's output. The copy phase of the reduce task then copies the data from the node where the map executed onto the local node where the reduce is running. Once all outputs have been copied, the sort phase merges all the files and passes the data to the reducer phase, which executes the appropriate Pig operators. The output records from the reducer phase are written to disk as they are created.

D. Pig's Progress Indicator

The existing Pig/Hadoop query progress estimator provides limited accuracy (see Section V). This estimator only considers the record reader, copy, and reducer phases for its computation. The record reader phase progress is computed as the percentage of bytes read from the assigned data partition. The copy phase progress is computed as the number of map output files that have been completely copied divided by the total number of files that need to be copied. Finally, the reducer progress is computed as the percentage of bytes that have been read so far. The progress of a MapReduce job is computed as the average of the percent complete of these three phases. The progress of a Pig Latin query is then just the average of the percent complete of all of the jobs in the query.

The Pig progress indicator is representative of other indicators that report progress at the granularity of completed and executing operators. This approach yields limited accuracy because it assumes that all operators (within and across jobs) perform the same amount of work. This, however, is rarely the case since operators at different points in the query plan can have widely different input cardinalities and can spend a different amount of time processing each input tuple. This approach also ignores how the degree of parallelism will vary between operators.

III. The Parallax Progress Estimator

In this section, we present the Parallax progress estimator for queries that translate into a sequence of MapReduce jobs. Parallax generates an estimate of the *amount of time* remaining for a query. Although designed and implemented in the context of MapReduce, most components of the indicator are more generally applicable. Throughout this section, we thus first focus on the key principles before showing the details for the specific, MapReduce setting.

The key techniques behind Parallax are the following:

1) As in prior work, Parallax breaks a query into pipelines, which are groups of interconnected operators that execute simultaneously. It then estimates time remaining for a query by summing the expected times remaining across all pipelines (Section III-A).
2) For each pipeline, time remaining requires an estimate of the amount of work that the pipeline needs to do and the speed at which the pipeline will perform that work. Parallax estimates both measures using offline and online statistics (Section III-B).
3) Because load conditions can change between different query executions and even while a query is in progress, Parallax monitors and adjust its estimates based on how conditions change (Section III-C).
4) In a distributed system, query setup costs are not negligible. Parallax takes these costs into account for more accurate estimates (Section III-D).
5) Parallax accounts for the degree of parallelism of query fragments, which can differ between consecutive fragments (Section III-E).
6) Finally, Parallax accounts for data skew and changes in the degree of parallelism during the execution of a single operator (Section III-F).

All of these techniques enable Parallax to produce accurate estimates in a broad range of scenarios as we discuss in Section V.

A. Estimating Time-Remaining

As in prior work [4], [20], Parallax estimates the time remaining for a query by breaking the query into fragments,

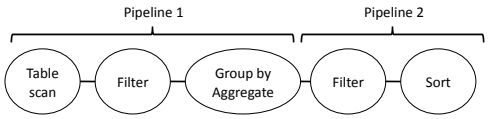


Fig. 2. Example of an execution plan for a query comprising two pipelines.

called pipelines. Given a sequence of pipelines, Parallax estimates the time remaining for the query as the sum of time remainings for individual pipelines (we only consider queries where pipelines execute in sequence).

A pipeline, is a group of interconnected operators that can execute simultaneously as illustrated in Figure 2. The key idea is that, at any given time during query execution, each pipeline is in one of three states: completed, executing, or blocked. The execution time for completed pipelines is known. The execution time for the other pipelines must be estimated.

Intuitively, the time remaining for a pipeline is the product of the amount of work that the pipeline must still perform and the speed at which that work will be done [20].

We define work to be done as the number of *input tuples* that a pipeline must still process. If N is the cardinality of the pipeline's input and K the number of tuples processed so far, the work left to be done is simply $N - K$. This definition is different from that used in prior work [4], [20] and we discuss the reason for our definition shortly.

Let's consider for a moment that we are given N_p, K_p, and an estimated processing cost α_p, expressed in msec/tuple, for a pipeline p. The time-remaining for the pipeline is then $\alpha_p(N_p - K_p)$. The time-remaining for a query is the sum of the time-remainings for the pipelines:

$$T_{\text{remaining}} = \sum_{p \in P} \alpha_p (N_p - K_p) \tag{1}$$

where P is the set of all pipelines in the query.

As we presented it so far, Parallax is identical to previously introduced progress indicators for single-site SQL queries [20], except for the minor change in the definition of the amount of work for a pipeline. Surajit *et al.* [4] define work as the total number of *output* tuples that will be produced by each operator in the query plan. In the context of MapReduce jobs, we found this definition problematic because many pipelines comprise only a single blocking operator that outputs a small amount of data, and the output is bursty. Luo *et al.* [20] define work as the total number of bytes on a pipeline's input and output. We found that considering the output tuples was not necessary because any work related to manipulating output tuples can simply be folded into the cost of the pipeline.

Prior work considered pipelines instead of individual operators as a way to protect against wrong cardinality estimates. In our setting, the use of pipelines is even more critical because of the complexity in scheduling different operators within a pipeline (i.e., in series, in parallel, or some combination of the two). Pipelines mask all this complexity and help us achieve our goal of developing a relatively simple, yet accurate estimator. In the case of MapReduce, we identify three pipelines as illustrated in Figure 1. The first pipeline comprises the record reader, map runner, and combine phases. The second pipeline corresponds to the copy phase only. The sort should be its own pipeline, but we ignore it because it takes a negligible amount of time since it only needs to merge sorted data. The third and last pipeline corresponds to the reducer. Because in the MapReduce setting, pipeline boundaries line up with node boundaries, we adopt the following, more precise, equation:

$$T_{\text{remaining}} = \sum_{j \in J} \sum_{p \in P_j} \alpha_{jp} (N_{jp} - K_{jp}) \tag{2}$$

where J is the set of all MapReduce jobs and P_j is the set of all pipelines within job $j \in J$.

Similar to previously proposed estimators, Parallax is vulnerable to errors in cardinality estimates (i.e., wrong N values). This problem, however, is not specific to parallel query processing and we do not address it in this paper. Parallax, however, could counter these errors using techniques similar to previous progress indicators that refine cardinality estimates as quickly as possible during query execution [4], [20], [22], [23].

Finally, given $T_{\text{remaining}}$, Parallax also outputs the percent query completed, computed as a fraction of expected runtime:

$$P_{\text{complete}} = \frac{T_{\text{remaining}}}{T_{\text{complete}} + T_{\text{remaining}}} \tag{3}$$

where T_{complete} is the total query processing time so far. In the paper, we use P_{complete} rather than $T_{\text{remaining}}$ to evaluate estimators.

B. Estimating Execution Speed

An important question that we must consider is how best to obtain estimates for the α weights for either currently executing or upcoming pipelines.

For the current pipeline, α can be estimated by looking at the current execution speed. Prior work suggested using a fixed-size window. We adopted this approach but chose to use an exponentially weighted moving average (EWMA) to achieve both smoothness and responsiveness to changing conditions. To further ensure stability, Parallax waits for a short time-period δ before producing the first estimate of the current execution speed. δ is a configurable parameter. We use 3 secondss in our experiments.

For pipelines that are still blocked, no information is available about their execution speed. Parallax must thus use an offline-computed estimate. Previous systems have chosen to use the current speed as a predictor of future speed [20]. We find this approach, however, to lead to highly inaccurate estimates (See Section V). An alternate approach is to use the optimizer's estimates [19]. This latter technique, however, may yield inaccurate results if the constants used by the optimizer are not accurate, and is not possible in MapReduce setings where no optimizer is available.

Instead, Parallax uses α values computed from previous executions of the same query. Such workload-aware or statement-specific statistics are used by some commercial systems to improve query optimization [28]. We adopt this technique for the purpose of progress estimation. As in any sampling-based technique, the accuracy of this approach depends on how closely the previous dataset represents the current one and on the model used in the extrapolation of the values to the new dataset. We do not address the problem of selecting the sample dataset, although techniques for selecting good samples exist in the literature [3], [17], [25], [26]. Instead, we propose that the system monitors query execution and opportunistically collects statistics. Such an approach is practical in large-scale cluster settings, where users commonly test their queries on small, representative samples or run the same queries multiple times on newly appended data for a given log file. To extrapolate cardinalities (N) and processing costs α from earlier runs on data samples, we use a simple linear model. This approach works well for cardinality estimates because we do not consider joins. For processing costs, the accuracy depends on the actual cost curve of an operator. In the evaluation, we find that Parallax produces good results even when a pipeline has a more complex cost curve. We leave the extension of Parallax to more complex models for future work.

Most important, progress estimation greatly benefits from α estimates even when they are not accurate. We find that α values can vary by orders of magnitudes between pipelines in a query. Even approximate estimates thus help the system better weigh the processing speeds of these pipelines and produce more accurate results as we demonstrate in Section V.

In the absence of offline estimates, Parallax assumes that all pipelines will process the same amount of data and will process that data at the same speed, corresponding to the current speed. Hence, the first execution of a query has very limited progress information.

C. Accounting for Changing Conditions

Offline α estimates can be inaccurate not only due to differences between input data used in different executions but also differences in runtime conditions (e.g., changes in load levels, machine heterogeneity, etc.). Such differences can lead to systemic errors in the α computations. Parallax dynamically adjusts its estimates to take into account such systemic error using what we call the per-pipeline *slowdown factor* s_p.

For the currently executing pipeline p, Parallax computes the ratio:

$$s_p = \alpha_p^e / \alpha_p^s$$

where α_p^e is the online measurement of α_p and α_p^s is the value estimated from earlier executions. Parallax then propagates this slowdown factor to pipelines that have not yet start executing.

However, because different pipelines have different bottleneck resources (CPU, disk or network bandwidth), Parallax propagates s values only across pipelines of the same type. In a parallel DBMS, the optimizer could serve to identify which pipelines are CPU bound and which ones are either I/O or network bound. In the case of MapReduce, the approach is simpler. We propagate the slowdown factor only between the same phases that belong to different jobs. For example, if a pipeline containing a map runner is executing, Parallax propagates its slowdown factor only to later pipelines that contain the map runner.

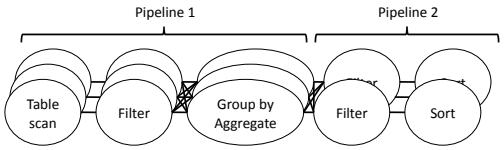


Fig. 3. Parallel query plan with different degree of parallelism for different pipelines.

The new time remaining equation thus becomes:

$$T_{\text{remaining}} = \sum_{j \in J} \sum_{p \in P} s'_{jp} \alpha_{jp} (N_{jp} - K_{jp}) \tag{4}$$

where s'_{jp} is equal to s_{jp} for pipelines that are already executing or, if pipeline jp had not yet started, it is the the slowdown factor of the most recently executed (or still executing) pipeline of the same type as jp.

D. Accounting for Setup Overhead

Finally, when a query executes in a distributed or parallel system, startup costs can be significant. Parallax models these costs as part of its progress estimation. The challenge lies only in implementation details that we omit. Here, we model these costs as a constant, $\text{SetupOverhead}_{\text{remaining}}$.

E. Accounting for Parallelism

The indicator described so far can accurately measure the progress of parallel queries (i.e., MapReduce sequences spread across multiple nodes), but where only one instance of each operator (e.g., one Map and one Reduce) executes at any time. This type of parallelism is called pipelined or inter-operator parallelism [9].

In addition to pipelined parallelism, parallel data management systems also use intra-operator parallelism, where individual operators are partitioned across many nodes. All operator partitions execute concurrently, each one processing a fragment of the input data

Intra-operator parallalelism affects query progress by changing the speed with which a pipeline can process intput data. In the case of uniform data distribution, the speedup is proportional to the number of partitions.

As a simplification and because this is always the case for MapReduce jobs, Parallax is designed for systems where all operators inside the same pipeline execute with the same degree of parallelism as shown in Figure 3. Operators in different pipelines can have different degrees of parallelism. In this case, the time remaining computation becomes:

$$T_{\text{remaining}} = \text{SetupOverhead}_{\text{remaining}} + \sum_{j \in J} \sum_{p \in P} \frac{s'_{jp} \alpha_{jp} (N_{jp} - K_{jp})}{\text{pipeline\_width}_{jp}} \quad (5)$$

Where N_{jp} and K_{jp} values are aggregated across all partitions of the same pipeline.

If the query optimizer or scheduler fixes the degree of parallelism for each pipeline, the `pipeline_width` becomes an input to Parallax. This is not the case for MapReduce as we discuss next.

F. Accounting for Data Skew and Changes in Degree of Parallelism

The time-remaining computation above assumes a uniform data distribution: it assumes that all partitions of a pipeline end at the same time. Additionally, it assumes that the number of partitions does not vary during the execution of a pipeline. Unfortunately, both assumptions rarely hold in practice in a MapReduce system. We now discuss the causes for changes in pipeline width and causes for data skew.

In a MapReduce system, the number of map tasks depends on the size of the input data, not the capacity of the cluster. For example, given a 256 MB chunk size (a recommended value that we also use in our experiments), a 1 GB file is processed by 4 map tasks while a 16 GB file is processed by 64 map tasks. The system schedules simultaneously as many tasks as are available slots in the cluster.[1] Once some tasks complete and resources become available, the system starts scheduling tasks from the next round. For example, if 16 slots are available for map tasks in the cluster, the 16 GB file would be processed by four rounds of 16 map tasks each. If the number of map tasks is not a multiple of cluster capacity, the number of tasks can decrease at the end of execution of a pipeline, causing the pipeline width to decrease, and the pipeline to slow down. For example, a 5 GB file would be processed by a first round of 16 map tasks followed by a round with only 4 map tasks.

Similarly, the number of reduce tasks can be set by the user, also independent of the cluster capacity. In both cases, Parallax takes this slowdown into account by computing the average pipeline width for the duration of each pipeline. If a pipeline is already executing, Parallax computes the average pipeline width until the end of execution. For example, the average pipeline width for the 5 GB file would be $\frac{16+4}{2} = 10$.

Once a pipeline is down to its last round of tasks, another factor starts dominating the progress estimation: skew between individual tasks. Up to this point, when some tasks were finishing, others were starting and the pipeline width remained unchanged. At the end, however, no new tasks can fill the pipeline. In that case, skew between individual data partitions can cause some partitions to end later than others, distorting the query progress. Such skew can be due to uneven data distribution, where some partitions need to process more data than others, which is possible in the case of reduce tasks. It can also be due to high-variance in per-tuple processing times (possible for either task). For this latter challenge, Parallax can do nothing since it does not know the detailed processing time distribution for upcoming tuples. For the former challenge, however, Parallax estimates the completion time for a pipeline using a simple critical path computation: it considers that the time of the pipeline will be equal to the time of its slowest map or reduce task.[2]

Interestingly, in the case of MapReduce when map tasks end, reduces start copying their input files in the background. Parallax does not explicitly account for this overlap, yet it produces accurate estimates. The reason is that this process either happens without overlap as planned or happens slowly in the background and the estimator has time to incrementally adjust.

IV. Implementing other Indicators

For comparison purposes, in addition to Parallax, we implemented in Pig three other state-of-the-art query progress indicators: gnm [4], dne [4], and one of Luo *et al.*'s indicators [20]. We refer to the latter as the Luo indicator. In this section, we describe our implementation of these *single-node SQL query* indicators in a MapReduce setting. We compare their performance to Parallax in the following section.

a) General implementation remarks: We instrumented Pig to collect progress information from all four estimators once every second. We implemented this as a background thread that computes the progress for all the estimators and logs the progress estimate to a file. We found this collection overhead to be negligible, especially relative to the running times of the queries that we studied in Section V.

To implement all progress indicators, we leveraged and extended Hadoop's infrastructure for collecting performance counters. Our extended version collects information on bytes and tuples processed by each phase and also on the runtime for each phase.

Because we do not study cardinality estimation problems in this paper, for all indicators, we set the N values to the correct values obtained from an earlier execution. We compare the relative performance of the indicators given correct cardinalities.

b) Gnm indicator: Chaudhuri *et al.* [4] propose to estimate the percentage complete of a query by using the *GetNext() model (gnm)* of work. This model defines the progress of a query as the fraction of tuples output so far by all operators in the query plan, where the total number of tuples is determined from cardinality estimates.

$$gnm = \frac{\sum_{i \in I} K_i}{\sum_i N_i}$$

[1] A Hadoop configuration specifies the maximum number of map and reduce tasks that can execute on a single physical machine. That number times the number of physical machines gives the total number of slots for each type of task.

[2] Speculative execution [7] further complicates this problem but we disabled this feature for this paper

where I is the set of all operators in the query plan.

Translated into the MapReduce setting, gnm relies on cardinality information collected at the granularity of the phase level. We thus implemented this indicator by collecting cardinality information at the *output* of all significant phases, including: record reader, map reader, combine, copy, and reduce. Ignoring any of the phases reduced the accuracy of the indicator.

c) Dne indicator: The gnm's use of cardinality estimates at all intermediate nodes in the query plan can lead to highly inaccurate results in case of cardinality estimation errors. To address this challenge, Chaudhuri *et al.* introduced the *Driver Node Estimator (dne)*. The dne breaks a query plan into pipelines. The progress of each pipeline is derived from the progress of its *input operators*, called *driver nodes*, for which input cardinalities are known accurately when the pipeline starts. As the query progresses, cardinality estimates of all pipelines are refined, resulting in increasingly more accurate progress estimates. Since we used perfect cardinalities to test this estimator, we did not implement this latter refinement. In the case of MapReduce, the driver nodes of the pipelines are respectively the record reader, copy, and reducer phases. Given the N_1 and K_1 values for the driver node of a pipeline, dne estimates the N_i for all non-driver nodes in a pipeline as:

$$N_i = \frac{N_1}{K_1} K_i$$

It then uses the same equation as gnm to compute the overall query progress.

d) Luo's indicator: Luo *et al.* [20], [19] proposed an estimator similar to those of Chaudhuri *et al.* but that also estimates the remaining query execution time, in addition to percent complete.

For percent complete, Luo's indicator uses cardinality estimates and tuple counts similar to dne. The main difference is that Luo's indicator counts bytes rather than tuples and counts them at the input and output of each pipeline. This approach results in the tuples output by intermediate phases being double-counted, which may account for any materialization of data buffering between pipelines [20]. We implemented this indicator by tracking the K and N values for the inputs and outputs of all three of the key MapReduce pipelines in each job.

To convert the fraction of work done into the remaining processing time, Luo's approach observes the current speed with which a pipeline processes its input data. It then either assumes that all following pipelines will process their data at the same speed [20] or it uses the output of the query optimizer as an estimate of query execution time for those pipelines that have not yet started [19]. In the absence of a query optimizer (and because Pig Latin scripts are typically heavy in UDFs), we only implemented the former approach. We estimated the current processing rate from a 60 second moving window, which we found to provide much smoother estimates than the suggested 10 second window [20].

V. Evaluation

In this section, we evaluate the Parallax estimator and compare it to other estimators from the literature. We experiment with simple Pig Latin queries and the PigMix [29] benchmark on synthetic datasets up to 16GB in size with uniform and Zipfian distributions. We present the results of serial queries, the effects of random and systemic errors on our estimates, and the ability of our estimator to perform well for highly parallel queries and in the presence of data skew.

A. Experimental Setup and Assumptions

All experiments in this section were run on an eight-node cluster configured with the Hadoop-17 release and Pig Latin trunk from February 12, 2009. Each node contains a 2.00GHz dual quad-core Intel Xeon CPU with 16GB of RAM. The cluster was configured to a maximum degree of parallelism of 16 map tasks and 16 reduce tasks.

In these experiments, gnm, dne, and Luo take perfect cardinality estimates, N, as input. Parallax takes N as input in addition to α values that were collected from prior runs. Parallax is demonstrated in two forms: *Perfect* Parallax, which uses N and α values from a prior run over the entire data set; and *1%* Parallax which uses α collected from a prior run over a 1% sampled subset (other sample sizes yielded similar results) and N values from a prior run over the full data set.

B. Distributed, Serial Query Experiments

We first evaluate how well the single-node SQL progress indicators from the literature (gnm, dne, and Luo), the current Pig estimator, and Parallax estimate the progress of MapReduce sequences in the absence of parallelism. For this we execute a Pig Latin script that compiles into five MapReduce jobs executed sequentially, with most of the work done in the first job. Although the jobs are executed in series, both the computation and the data for this job are still distributed amongst the nodes. The script, script1-hadoo.pig, comes from the standard Pig Latin distribution. It contains fourteen unique Pig Latin statements and five UDFs, and processes a search query log file from the 'Excite' search engine. We construct a much larger, 210MB data set from the original 'Excite' data set to see how well the estimators do on longer-running queries ($>$ 10 minutes). These experiments demonstrate the importance of adequately modeling the relative weight of different pipelines in a query.

1) Default Script1: Figure 4 shows the progress reported by each indicator throughout the execution of the script. The x-axis shows the actual percent time complete of the script. The y-axis shows the percent complete reported by each indicator. An ideal estimator is shown as a diagonal line. Furthermore, Table II(a) reports the average and maximum estimation error of each technique, computed as in [4]:

$$error = \left| \frac{100 * (t_i - t_0)}{(t_n - t_0)} - f_i \right| \quad (6)$$

where f_i is the percentage of overall run completion as reported by the estimator, t_i is the current time, t_n is the time

TABLE I

SCRIPT1 RELATIVE THROUGHPUT ACROSS JOBS AND PIPELINES

(a)

Job	Relative Throughput
Job 1	311
Job 2	83
Job 3	51
Job 4	3
Job 5	17

(b)

Pipeline	Relative Throughput
map	447
reduce	7,263
reduce copy	22,323

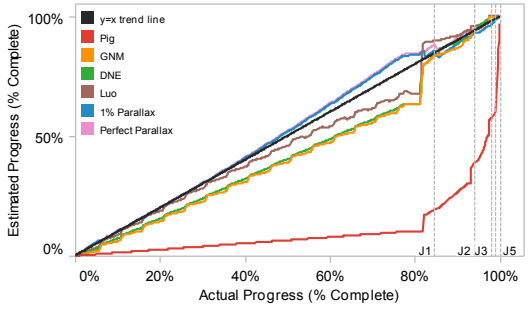


Fig. 4. Default script1-hadoop, 210MB, serial execution

when all the jobs complete, and $(t_i - t_0)/(t_n - t_0)$ represents the actual fraction of the jobs completed.

The script is dominated by the map pipeline of the first job, which takes 79% of the total running time. As shown in Figure 4, the uniform throughput of this pipeline allows Parallax and the estimators from the literature to perform well. Note that the 1% variant of Parallax performs as well as *Perfect* Parallax because of similar throughput measurements. The gnm and dne estimators are less accurate because of the assumption that the average amount of work performed by each call to GetNext() is approximately equal across operators and pipelines. This assumption is disproved by the results in Table I(a), which shows that the relative throughput for different pipelines varies greatly within each job (expressed as the MAX throughput divided by the MIN throughput). Additionally, Luo's estimator is less accurate than Parallax because of the assumption that the current throughput for a given pipeline is indicative of future throughputs. The results from Table I(b) reveal that the throughput varies significantly for a given pipeline across all jobs. Luo, gnm and dne are mostly pessimistic because the first job only processes 63% of the total tuples, but occupies 82% of the total running time. The original Pig estimator, unfortunately, does not give accurate estimates because it assumes that each pipeline and job perform the same amount of work.

Table II(a) shows that *Perfect* Parallax and *1%* Parallax yield accurate overall estimates with 2.2% and 2.0% average error respectively. Our estimates are more accurate than those from the literature because we don't assume constant throughput across jobs or pipelines. Instead, Parallax tracks throughput per pipeline per job.

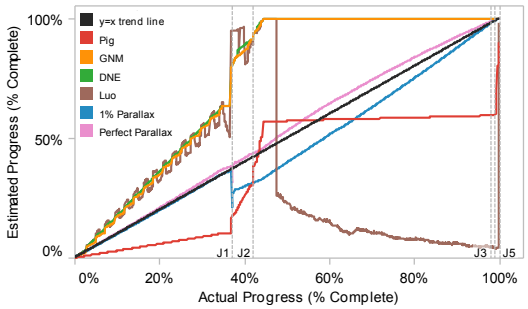


Fig. 5. Modified script1-hadoop, 210MB, serial execution

2) Altered Script1: To demonstrate the vulnerability of the estimators from the literature to throughput variance, we reduce the dominance of the map pipeline in the first job by increasing the latency of a later job. For this, we modify script1-hadoop by adding complexity to one of the UDFs used in the script. This change forces job 3 to exhibit a much longer latency for processing the same number of tuples, thus decreasing its throughput approximately by two orders of magnitude. The results from this experiment (see Figures 5 and Table II(a)) demonstrate that this small change causes all of the estimators from the literature to have less accurate estimates. Note that by the end of job 2, gnm, dne, and Luo estimate that the script is around 99% complete, not taking into account the low processing rate of the remaining 1% of work. Recall that our implementation of Luo's estimator uses a 60 second moving window to track changes in throughput. The discontinuity near 45% completion is the result of the moving window fully incorporating the extremely low throughput of job 3. The Luo estimator becomes pessimistic because it assumes this same processing rate applies to the remaining tuples in jobs 4 and 5, which actually have significantly higher throughput than job 3.

Perfect and *1%* Parallax produce almost perfect, identical estimates up until the end of the first job. The reason that *1%* Parallax diverges from the diagonal and trends pessimistic is due to the lower-throughput α values that were generated from the sample. However, the slowdown factor helps correct for this inaccuracy.

3) Default Script1 α Perturbation Tests: We study the effect of inaccurate α values on Parallax's estimates by inducing two different types of errors: random and systemic. We compare the estimation accuracy for each perturbation run against an unperturbed run with *Perfect* α values. In the first experiment, we select errors of either +0.9% or -0.9% for each pipeline. In subsequent runs, we repeat with +/- 9% and +/- 99%error. We also consider systemic errors, which may occur when a cluster is heavily-loaded, for example. To study this, we introduce uniform error to the α for each pipeline, using multipliers of +1%, +9%, and +99%. The goal of this experiment is to show the resilience of our estimator in the presence of system variability, namely the importance of the slowdown factor. Parallax has the ability to recover from

TABLE II

(a) Script1 Error Comparison

	Pig % Error		gnm % Error		dne % Error		Luo % Error		**Perfect Parallax** % Error		**1% Parallax** % Error	
Experiment	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Default serial script1	39.9	71.5	7.9	17.6	7.2	17.6	9.33	29.7	2.2	7.3	2.0	6.9
Modified serial script1	14.4	39.1	21.9	55.6	22.5	55.6	32.9	95.5	1.7	5.9	3.1	15.8
Default parallel script1	38.4	74.9	31.5	68.8	31.5	68.8	34.1	77.3	2.2	6.8	2.8	8.1

(b) Script1 α Perturbation Error Comparison

	% Error	
Experiment	**AVG**	**MAX**
Perfect	0.9	3.6
Random (+-0.9%)	1.4	4.5
Random (+-9%)	1.6	4.9
Random (+-99%)	4.5	10.4
Systemic (1%)	1.4	5.2
Systemic (9%)	2.4	6.8
Systemic (99%)	4.5	10.4

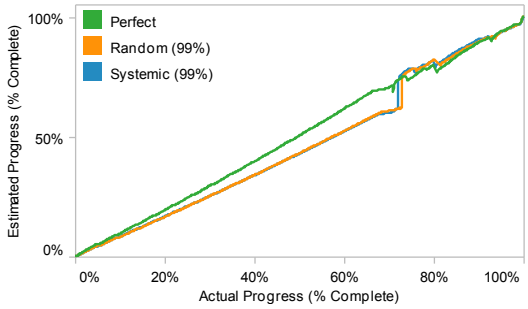


Fig. 6. Systemic and random alpha error, alpha perturbation experiment

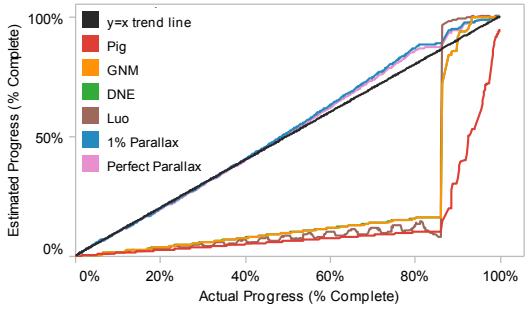


Fig. 7. Pig script1-hadoop, 210MB, parallel execution

systemic errors because the slowdown factor adjusts α values for future pipelines based on observed changes in conditions between the sample and current runs. Figure 6 and Table II(b) show that α values in the presence of random error on the order of +/-99% or systemic error on the order of +99% yield estimates with only 4.5% average error.

C. Parallelism Experiments

We now evaluate Parallax's performance in face of parallelism. For this, we present a series of experiments that stress different features of Parallax. We put all these features to the test together in the next section, where we show results from the PigMix benchmark.

1) Basic Parallelism Experiments: As a motivation for Parallax, we execute script1 again, but this time, we allow two of the pipelines (the copy and reduce pipelines of the first job) to execute with a degree of parallelism of 16. Figures 7 and Table II(a) demonstrate that the estimators from the literature do poorly in the presence of parallelism, which affects pipeline throughput in ways these estimators are not designed to handle. These estimators all yield pessimistic estimates because they assume that the processing rate of the map from the first job (which dominates over 85% of the overall running time) reflects the processing rate of future pipelines. In fact, the processing rate increases after the map from job 1 has completed due to the increased parallelism in the reduce pipelines. *1%* Parallax and *Perfect* Parallax remain well under 10% error for the entire duration of the script. The average estimation error for *1%* Parallax was 2.75% and 2.25% for *Perfect* Parallax.

We now study Parallax's performance in more detail through a series of five experiments that we label *P1* through *P5*. All experiments are run on our full 8-node cluster.

In experiment P1, we run a simple select query (i.e., a LOAD-FILTER-STORE Pig Latin script) with selectivity 0.5. This script translates into a single MapReduce job with only map tasks, the simplest parallel configuration possible. In this experiment, we increase the input data size from 256MB to 16GB. Given that our file chunk-size is set to 256 MB, this results in jobs containing 1, 2, 4, 8, 16, 32, and 64 map tasks respectively. Runs with up to 16 tasks require only one round of map tasks. The subsequent runs require two and four rounds of map tasks, respectively. The longest configuration took 68 minutes to run.

Figure 8 shows the results for the longest run, demonstrating that Parallax tracks the ideal progress indicator extremely well. The three dips correspond to the transition between rounds of map tasks. Indeed, even with a uniform data distribution, map tasks never end exactly at the same time. Due to scheduling overhead, new map tasks are not immediately activated. As some map tasks end, our algorithm adjusts the average pipeline width for the current round, temporarily reflecting the reduced concurrency.

However, even with these dips the average error remains below 2.2% for all configurations as summarized in Table III.

In experiment P2, we execute a LOAD-GROUPBY-STORE script that translates into a single MapReduce job, but this time with both Map and Reduce tasks. This is thus a 3-pipeline query. Again, uniform data distribution ensures that all map and reduce tasks in the same round end approximately at the same time. We vary the input data size from 1 to 32 map tasks. We set the degree of parallelism of the reduce pipelines to mirror those of the map pipeline. Figure 9 shows

TABLE III

EXPERIMENT P1: AVERAGE ESTIMATION ERROR ACROSS INCREASING NUMBER OF MAPS

Estimator	1		2		4		8		16		32		64	
	AVG	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**
1% Parallax	0.8%	0.5%	1.2%	0.6%	0.8%	0.6%	1.2%	0.8%	1.9%	1.2%	2.2%	1.7%	1.2%	1.5%
Perfect Parallax	0.8%	0.5%	1.2%	0.6%	0.8%	0.6%	1.2%	0.8%	1.8%	1.2%	2.2%	1.7%	1.2%	1.5%

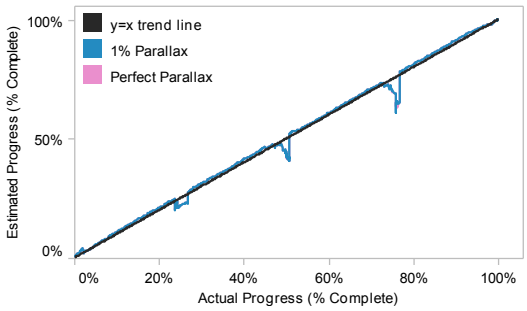


Fig. 8. Experiment P1: Progress estimation results during the execution of a query comprising four rounds of map tasks (64 maps total). Uniformly distributed 16GB dataset. Eight-node cluster

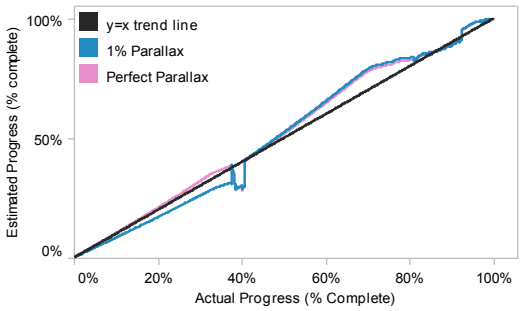


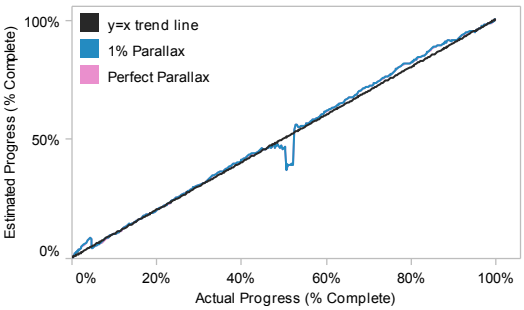
Fig. 9. Experiment P2: Progress estimation results during the execution of a query comprising two rounds of map and two rounds of reduce tasks (32 maps and 32 reduces). Uniformly distributed 8GB dataset, eight-node cluster

the result from the query on the largest input data. This query had two rounds of map tasks and two rounds of reduce tasks. Table IV shows the estimator error values for all configurations that we tested. Once again, the overall results are extremely encouraging, with average error values remaining below 4% in all configurations. Figure 9 shows the same dip as before when the first round of map tasks ends. The figure also shows that Parallax is optimistic during the second round of map tasks. This effect is due to our model. Parallax assumes that, in the absence of changes to external conditions, a pipeline will process data at constant speed. Parallax does not account for an extra blocking combine phase that is sometimes performed at the end of a map pipeline.[3] A more refined model could improve these estimates, but would complicate the implementation. Overall, however, Parallax tracks simple parallel queries extremely well.

2) Dynamic Parallelism Variations: As discussed in Section III-F, depending on input data size and configuration parameters, a query can execute with a number of map and/or reduce tasks that is not a multiple of the cluster capacity. As a result, the degree of parallelism of a pipeline can change during its execution. We explore the effects of this dynamic pipeline width variation through a series of three experiments.

In experiment P3, we execute again the query with only map tasks, but this time we set the input data size to require 17, 24, or 31 map tasks. In experiments P4 and P5, we execute a query with both map and reduce tasks. In experiment P4, the number of map tasks is a fixed multiple of the cluster capacity (i.e., 32 maps), while the number of reduces vary as 17, 24 and 31. Finally, in experiment P5, both the maps

[3]The combine phase processes the data one chunk at the time in parallel with the rest of the pipeline but may sometimes block that pipeline.



Estimator	17		24		31	
	AVG	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**
1% Parallax	4.6%	4.7%	3.0%	1.5%	2.2%	1.8%
Perfect Parallax	4.6%	3.1%	1.4%	4.7%	2.2%	1.9%

Fig. 10. Experiment P3: The figure shows a representative run on 7.8GB data set, 31 maps, eight-node cluster. The table shows the average estimation error across increasing dataset sizes (17, 24, and 31 maps).

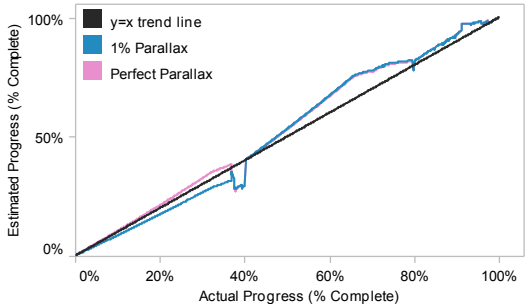
and reduces have variable degrees of parallelism during their execution (each has 17, 24, or 31 tasks depending on the run). Figures 10 through 12 show the results. Interestingly, Figures 10 and 11 show the exact same trends as above indicating that Parallax's average pipeline width computation handles this type of parallelism variation well. When all pipelines have variable degrees of parallelism during their execution (Figure 12), we see an interesting effect where Parallax produces optimistic progress estimates toward the end of the first round of map tasks. The transition between rounds causes Parallax to briefly generate optimistic estimates in this case because it observes more concurrency than expected, due to jobs lingering from the first round.

3) Parallelism Experiments with Data Skew: The goal of the experiment in this section is to measure how well Parallax

TABLE IV

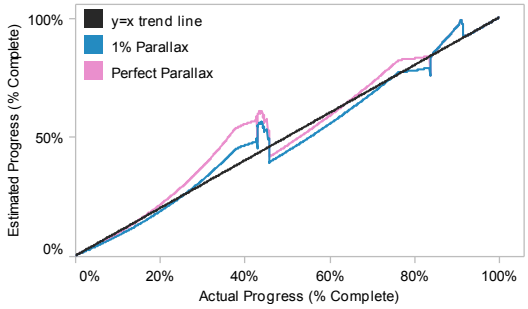
EXPERIMENT P2: AVERAGE ESTIMATION ERROR ACROSS INCREASING NUMBER OF MAPS AND REDUCES

Estimator	1		2		4		8		16		32	
	AVG	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**
1% Parallax	4.1%	2.3%	2.4%	1.6%	1.2%	0.7%	1.3%	0.9%	1.1%	0.8%	4.3%	2.7%
Perfect Parallax	3.0%	2.8%	2.8%	2.7%	3.7%	3.2%	2.9%	2.8%	3.5%	2.9%	3.0%	2.8%



Estimator	17		24		31	
	AVG	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**
1% Parallax	3.1%	2.0%	3.6%	2.2%	4.3%	3.0%
Perfect Parallax	1.9%	1.9%	1.9%	2.4%	3.6%	3.1%

Fig. 11. Experiment P4: The figure shows a representative run on 7.8GB data set, 31 maps, eight-node cluster. The table shows the average estimation error across increasing reduce parallelism (17, 24, and 31), 8GB data set (32 maps)



Estimator	17		24		31	
	AVG	**STDDEV**	**AVG**	**STDDEV**	**AVG**	**STDDEV**
1% Parallax	3.1%	2.8%	2.4%	1.2%	2.8%	1.8%
Perfect Parallax	6.3%	6.2%	2.3%	1.9%	2.7%	2.3%

Fig. 12. Experiment P5: The figure shows a representative run on 4.2GB data set, 17 maps, eight-node cluster. The table shows the avg estimation error across increasing dataset size (4.2GB -17 maps, 6GB- 24 maps, and 7.8GB - 31 maps) and reduce parallelism (17, 24, and 31)

handles data skew. In contrast to variations in the degree of parallelism, in the presence of data skew the number of tasks may well be a multiple of the cluster capacity but some tasks take much longer to process their input data than others. Parallax currently handles cases where such skew results from an imbalance in the number of input tuples processed (and not time to process each tuple). Parallax takes such imbalance into consideration if it can be predicted from sample runs.

For experiment P6, we use the same Pig Latin query as in experiment P2 and P4 above, but we alter the data makeup. We generated 8 GB of data with a Zipfian distribution on the

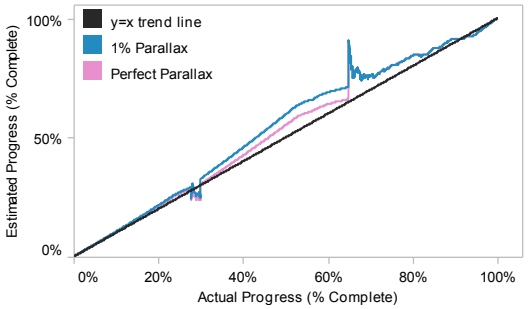


Fig. 13. Experiment P6. Zipfian skew 8GB data set, eight-node cluster

TABLE V

ESTIMATION ERRORS PIGMIX LATENCY BENCHMARKS, 15GB DATA SET

	Perfect Parallax % Error		1% Parallax % Error		dne % Error		gnm % Error		Luo % Error		Pig % Error	
Query	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
L1	2.2	8.8	1.9	9.0	2.9	8.0	3.1	8.5	7.9	12.8	14.9	31.2
L4	4.3	8.6	3.7	7.7	5.3	15.1	5.8	15.6	5.4	16.2	13.3	26.6
L6	3.0	8.6	2.4	7.3	8.5	19.3	9.1	19.5	6.4	19.7	13.2	26.2
L7	3.0	6.8	1.2	5.5	7.8	23.6	8.0	23.6	8.8	23.7	13.3	27.5
L8	4.1	8.4	3.7	7.7	5.1	10.3	5.1	10.3	7.1	11.1	15.2	30.8
L9	4.8	17.3	6.6	18.7	15.9	34.3	15.9	34.7	21.6	41.9	15.8	29.5
L10	4.3	11.9	5.6	17.1	16.5	39.5	16.5	39.5	20.5	46.3	12.5	23.4

key used by the GROUP-BY operator, which results in data skew in the reduce tasks. In Figure 13 we see that Parallax handles this skew quite well, with an average error of 2.5% for *Perfect* Parallax and 4.5% for *1%* Parallax.

D. PigMix Benchmarks

This section studies the standard set of Pig Latin benchmarks called PigMix [29]. We only consider a subset of these benchmarks (referred to as *latency*) which test a wide-variety of Pig Latin queries and generate one or more MapReduce jobs. We exclude queries that do not translate into a sequence of jobs such as those containing joins. Furthermore, a 15GB data set is used, and depending on the attribute, contains both Zipfian and uniformly-distributed keys. These results are summarized in Table V.

Parallax outperforms all estimators with the best average and maximum errors. Across all queries, average estimation error is below 5% for *Perfect* Parallax and below 6.6% for *1%* Parallax. Maximum errors remain below 10% for the first five queries and below 20% for the last two queries. The dne and gnm estimators perform similarly though not as well on average, and have significantly worse maximum errors

reaching up to 40%.

VI. Related Work

We discussed the main single-node SQL progress indicators and how Parallax builds on them in earlier sections. Here, we present additional related work.

Several relational DBMSs, including parallel DBMSs, provide coarse-grained progress indicators for running queries. Most systems simply maintain and display a variety of statistics about (ongoing) query execution [5], [6], [8], [11] (e.g., elapsed time, number of tuples output so far). Some systems [8], [11] further break a query plan into steps (e.g., operators), show which of the steps are currently executing, and how evenly the processing is distributed across processors. Our approach strives to provide significantly more accurate time-remaining estimates.

In follow-up work to gnm and dne [2], Chaudhuri *et al.* extended their approach with two additional estimators. The first of these, *pmax*, provides increased accuracy in the case of input data skews. These skews refer to the difference in the per-tuple processing times for a single operator (rather than imbalance between operator partitions as we study in this paper). The second, *safe*, provides an estimate that is worst case optimal in the presence of such skew. Our work is orthogonal to the safe and pmax estimators since we did not consider this type of skew. Parallax could be extended with these techniques.

Recent work also considers the impact of concurrent queries and their expected completion times to improve estimates [21]. This extension is orthogonal to our work. and we could incorporate it to improve the accuracy of our indicator in the presence of concurrent queries. Currently, Parallax simply reacts to observed changes in dynamic conditions.

Query progress is related to the cardinality estimation problem. Indeed, given accurate predictions of intermediate result sizes, the GetNext() model can directly be used to compute query completion as a percentage. There exists significant work in the cardinality estimation area including recent techniques [22], [23] that continuously refine cardinality estimates using online feedback from query execution. These techniques can help improve the accuracy of progress indicators; however, they are orthogonal to our approach since we do not address the cardinality estimation problem in this paper.

Work on online aggregation [14], [17] also strives to provide continuous feedback to users during query execution. The feedback, however, takes the form of confidence bounds on result accuracy rather than estimated completion times. Additionally, these techniques use special operators to avoid any blocking in the query plans.

VII. Conclusion and Future Work

We presented Parallax, the first, non-trivial time-based progress indicator for parallel queries. Parallax is developed for Pig Latin queries that compile into a series of MapReduce jobs. To produce accurate estimates, Parallax combines runtime measurements with statistics collected from earlier executions of the same query on a data sample. Parallax handles queries with operators that process tuples at different speeds (including UDFs). It handles an environment where processing speeds vary, where the degree of parallelism changes between operators, and where data skew exists. Parallax is *fully* implemented in Pig and we evaluated it on queries from the PigMix benchmark, finding it more accurate than existing alternatives.

VIII. acknowledgements

The Parallax project is partially supported by NSF CAREER award IIS-0845397, NSF CRI grant CNS-0454425, gifts from Microsoft Research, and Balazinska's Microsoft Research New Faculty Fellowship. Kristi Morton is supported in part by an AT&T Labs Fellowship.

References

[1] C. Ballinger. Born to be parallel: Why parallel origins give Teradata database an enduring performance edge. http://www.teradata.com/t/page/87083/index.html.

[2] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust progress estimators for SQL queries. In *Proc. of the SIGMOD Conf.*, Jun 2005.

[3] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. *SIGMOD Record*, 28(2):263–274, 1999.

[4] S. Chaudhuri, V. Narassaya, and R. Ramamurthy. Estimating progress of execution for SQL queries. In *Proc. of the SIGMOD Conf.*, Jun 2004.

[5] DB2. SQL/monitoring facility. http://www.sprdb2.com/SQLMFVSE.PDF, 2000.

[6] DB2. DB2 Basics: The whys and how-tos of DB2 UDB monitoring. http://www.ibm.com/developerworks/db2/library/techarticle/dm-0408hubel/index.html, 2004.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.

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

[9] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[10] D. J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov. Clustera: an integrated computation and data management system. In *Proc. of the 34th VLDB Conf.*, pages 28–41, 2008.

[11] Greenplum. Database performance monitor datasheet (Greenplum Database 3.2.1). http://www.greenplum.com/pdf/Greenplum-Performance-Monitor.pdf.

[12] Greenplum database. http://www.greenplum.com/.

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

[14] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. of the SIGMOD Conf.*, 1997.

[15] IBM zSeries SYSPLEX. http://publib.boulder.ibm.com/infocenter/\\dzichelp/v2r2/index.jsp?topic=/com.ibm.db2.doc.admin/xf6495.htm.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of the European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

[17] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. A disk-based join with probabilistic guarantees. In *Proc. of the SIGMOD Conf.*, pages 563–574, 2005.

[18] Large Synoptic Survey Telescope. http://www.lsst.org/.

[19] G. Luo, J. F. Naughton, C. J. Ellman, and M. Watzke. Increasing the accuracy and coverage of SQL progress indicators. In *Proc. of the 20th ICDE Conf.*, 2004.

[20] G. Luo, J. F. Naughton, C. J. Ellman, and M. Watzke. Toward a progress indicator for database queries. In *Proc. of the SIGMOD Conf.*, Jun 2004.

[21] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL progress indicators. In *Proc. of the 10th EDBT Conf.*, 2006.

[22] C. Mishra and N. Koudas. A lightweight online framework for query progress indicators. In *Proc. of the 23rd ICDE Conf.*, 2007.
[23] C. Mishra and M. Volkovs. ConEx: A system for monitoring queries (demonstration). In *Proc. of the SIGMOD Conf.*, Jun 2007.
[24] B. A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *CHI '85: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 11–17, 1985.
[25] F. Olken and D. Rotem. Random sampling from b+ trees. pages 269–277, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.
[26] F. Olken, D. Rotem, and P. Xu. Random sampling from hash files. *SIGMOD Record*, 19(2):375–386, 1990.
[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of the SIGMOD Conf.*, pages 1099–1110, 2008.
[28] Oracle. `http://www.oracle.com/database`.
[29] PigMix Benchmarks. `http://wiki.apache.org/pig/PigMix`.
[30] A. Pruscino. Oracle RAC: Architecture and performance. In *Proc. of the SIGMOD Conf.*, page 635, 2003.
[31] Sloan Digital Sky Survey. `http://cas.sdss.org`.
[32] M. Stonebraker, J. Becla, D. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for science data bases and SciDB. In *Fourth CIDR Conf. – Perspectives*, 2009.
[33] Vertica, inc. `http://www.vertica.com/`.