ParaTimer: A Progress Indicator for MapReduce DAGs*

*University of Washington Technical Report UW TR: #UW-CSE-09-12-02

Kristi Morton, Magdalena Balazinska, Dan Grossman

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

ABSTRACT

Accurate progress estimation for parallel queries is a challenging problem that has received only limited attention. The challenges are especially great when users are interested in estimates of *time remaining* rather than a count of processed records. Previous work has focused only on time-remaining estimates for single-site queries and a very limited class of parallel queries, which exclude joins, data skew, node failures, and other important challenges that arise in practice. In this paper, we present ParaTimer, a *comprehensive* time-remaining indicator for parallel queries. ParaTimer builds on previous techniques and makes two key contributions. First, it estimates the progress of parallel queries that include joins, which requires a radically different approach than in prior work. Second, it handles a variety of real systems challenges such as failures and data skew. To handle unexpected changes in query execution times due to runtime condition changes, ParaTimer provides users not only with one but with a set of time-remaining estimates, each one corresponding to a different carefully selected scenario.

Several parallel data processing systems exist. In this paper, we target environments where declarative Pig Latin queries are translated into MapReduce DAGs. We implement our estimator in the Pig system and demonstrate its performance on experiments running on a real, small-scale cluster.

1. INTRODUCTION

Whether in industry or in the sciences, users today need to store, archive, and most importantly analyze increasingly large datasets. For example, the upcoming Large Synoptic Survey Telescope [17] is predicted to generate on the order of 30 TB of data every day.

Parallel database management systems [1, 11, 14, 26, 27] and other parallel data processing platforms [6, 8, 12, 15] are designed to process such massive-scale datasets: they enable users to submit declarative queries over the data and they execute these queries in clusters of shared-nothing servers. Although parallelism speeds-up query execution, query times in these shared-nothing platforms can still exhibit large intra-query and inter-query variance.

In such an environment, accurate, time-remaining progress estimation for queries can be helpful both for users and also for the system. Indeed, the latter can use time-remaining information to improve resource allocation [28], enable query debugging, or tune the cluster configuration (such as in response to unexpected query runtimes).

Accurate progress estimation for parallel queries is a challenging problem because, in addition to the challenges shared with single-site progress estimators [3, 2, 19, 18, 21, 22], parallel environments introduce distribution, concurrency, failures, data skew, and other issues that must be taken into account. This difficult problem, however, has received only limited attention. Our preliminary prior work [23] provided accurate estimates, but only for a very limited class of parallel queries, which exclude joins. We also previously assumed uniform data distribution and the total absence of node failures, two assumptions that are unreasonable in practice.

To address these limitations, we have developed ParaTimer, a *comprehensive* time-remaining indicator for parallel queries. ParaTimer builds on previous techniques and makes two key contributions. First, ParaTimer estimates the progress of parallel queries that include joins, which requires a radically different approach than in our prior work. Second, it includes techniques for handling a variety of real system challenges including failures and data skew. To handle unexpected changes in query execution times such as those due to failures, ParaTimer provides users not only with one but with *a set of* time-remaining estimates that provide *useful bounds* on the expected query execution times. We call ParaTimer comprehensive because it provides this set of bounds instead of a single best guess as the other estimators do.

Many parallel processing systems exist. We developed ParaTimer for Pig queries [24] running in a Hadoop cluster [12], an environment that is a popular open-source parallel data-processing engine under active development. 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, a MapReduce-style scheduler requires intermediate result materialization, schedules small pieces of work at a time, and restarts small query fragments when failures occur (rather than restarting entire queries). All three properties affect query progress and its estimates.

ParaTimer is designed to be accurate while remaining simple and addressing the above Pig/Hadoop-specific challenges. At a high level, ParaTimer works as follows. For basic progress estimation, ParaTimer builds on our prior system Parallax[1] [23]. Parallax estimates time-remaining by breaking queries into pipelines and, for each pipeline, estimating the amount of work to be done and the speed at which that work will be performed. To get processing speeds, Parallax relies on earlier debug runs of the same query on input data samples generated by the user. ParaTimer extends Parallax along two important directions.

First, Parallax handles only queries that comprise a sequence of MapReduce jobs. ParaTimer, on the other hand, adds support for joins that can translate into MapReduce trees or, more generally, MapReduce DAGs. To estimate the progress of joins, ParaTimer includes a method to identify critical paths in the query plan and estimates progress along that path, effectively ignoring other paths.

Second, ParaTimer provides support for a variety of practical challenges related to parallel query processing. Most notable, ParaTimer handles failures and data skew. For data skew that can be predicted and planned for, ParaTimer takes it into account upfront. For failures and data skew that cannot be completely pre-planned, ParaTimer takes a radically new strategy. Instead of showing users a single "best guess" progress estimate, ParaTimer outputs a set of estimates that bound the expected query execution time within given possible variations in runtime conditions. An interesting side-effect of this approach is that when a query time goes outside ParaTimer's initial bounds, a user knows that there is a problem with either his query or the cluster. ParaTimer's output can thus help detect problems with queries or cluster setup.

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[2]. We thus evaluate ParaTimer's performance through experiments on a small eight-node cluster (set to a maximum degree of parallelism of 32 split into 16 maps and 16 reduces). We compare ParaTimer's performance against Parallax [23], three other state-of-the-art single-node progress indicators from the literature [3, 19], and Pig's current progress indicator [25]. We show that ParaTimer is more accurate than all these alternatives on a variety of types of queries and system configurations. For all queries we evaluated, ParaTimer's average accuracy is within 5% of an ideal indicator.

The rest of this paper is organized as follows. The next section provides background on MapReduce, Hadoop, and our prior work. Section 3 presents ParaTimer's approach to handling joins (and in general, MapReduce DAGs), failures, and data skew. Section 4 presents empirical results. Section 5 discusses related work. Section 6 concludes.

2. BACKGROUND

In this section, we present an overview of MapReduce [6], Pig [24], the naive progress indicator that currently ships with Pig, and our recent work on the Parallax progress indicator for Pig [23].

[1]Name changed for double-blind reviewing

[2]http://wiki.apache.org/hadoop/PoweredBy

2.1 MapReduce

MapReduce [6] (with its open-source variant Hadoop [12]) 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 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.

2.2 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 [24]. In Pig, queries are written in Pig Latin, a language combining 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 example from Section 2.1.

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

In Pig Latin, this example could be written as:

```
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 outputting tuples of the form (time, searchlog-entry). The reduce phase would then count the searchlog entries for each distinct time value.

Because Pig scripts can contain multiple filters, aggregations, and other operations in various orders, in general a

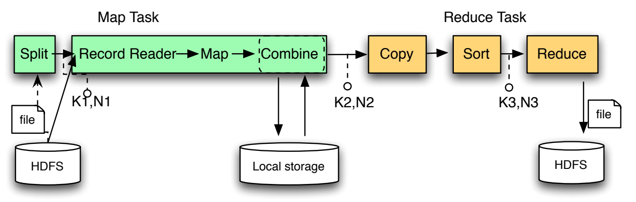


Figure 1: Detailed phases of a MapReduce Job. Each N_i indicates the cardinality of the data on the given link. K_i's indicate the number of tuples seen so far on that link. Both counters mark the beginning of a new Parallax pipeline (Section 2.5).

query will not execute as a single MapReduce job but rather as a directed acyclic graph (DAG) of jobs. For example, one of the two sample scripts (script1) distributed with the Pig system compiles into a sequence of five MapReduce jobs.

2.3 MapReduce Details

Each MapReduce job contains seven phases of execution, as Figure 1 illustrates. 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.

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 relevant data from the node where the map executed onto the local nodes where the reduces are running. Once all outputs have been copied, the sort phase of each reduce task 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.

2.4 Pig's Progress Indicator

The existing Pig/Hadoop query progress estimator provides limited accuracy (see Section 4). This estimator considers only 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.

2.5 Parallax Progress Estimator

Our prior work on the Parallax progress estimator [23] is significantly more accurate than Pig's original estimator, but Parallax is designed to be accurate only for very simple parallel queries. It adapts and extends related work on single-site SQL query progress estimation [3, 19] to parallel settings.

Like in single-site estimators, Parallax breaks queries into pipelines, which are groups of interconnected operators that execute simultaneously. From the seven phases of a MapReduce job, Parallax ignores two and constructs three pipelines from the remaining five: (1) the record reader, map runner, and combiner operations taken together, (2) the copy, and (3) the reducer. In our experiments, however, we found that the sort phase can impose a significant overhead and, hence, ParaTimer accounts for it as a fourth pipeline.

Given a sequence of pipelines, Parallax estimates their time remaining as the sum of time remaining for the currently executing and future pipelines. The time remaining for each 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. Parallax defines the remaining work as the number of *input tuples* that a pipeline must still process. If N is the number of tuples that a pipeline must process in total and K the number of tuples processed so far, the work remaining is simply $N - K$.

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 $\alpha_p(N_p - K_p)$. The time-remaining for a computation is the sum of the time-remainings for all the jobs and pipelines. Of course, N_p and α_p must be estimated for each future pipeline.

Estimating Execution Costs and Work Remaining

An important contribution and innovation of Parallax is its estimation of pipeline per-tuple processing costs (the α_p for each pipeline). Previous techniques ignore these costs [3, 2], assume constant processing costs [19], or combine measured processing cost with optimizer cost estimates to better weight different pipelines [18]. In contrast, Parallax estimates the per-tuple execution time of each pipeline by observing the current cost for pipelines that have already started and using information from earlier (e.g., debug) runs for pipelines that have not started. This approach is especially well-suited for query plans with user-defined functions. Debug runs can be done on small samples and are common in cluster-computing environments.

Additionally, Parallax dynamically reacts to changes in runtime conditions by applying a slowdown factor, s_p to current and future pipelines of the same type, though the effectiveness of this factor has not been previously evaluated.

For cardinality estimates, N_p, Parallax relies on standard techniques from the query optimization literature. For predefined operators such as joins, aggregates, or filters, cardinalities can be estimated using cost formulas. For user-defined functions and to refine pre-computed estimates, Parallax can leverage the same debug runs as above.

We adopt the same strategy in this paper. We do not study cardinality estimation and assume they are derived using one of the above techniques. We also use α processing costs computed from debug runs of the same query fragment.

Accounting for Dynamically Changing Parallelism

The second key contribution of Parallax is how it handles parallelism, i.e., multiple nodes simultaneously processing a map or a reduce. Parallelism affects computation progress by changing the speed with which a pipeline processes input data. The speedup is proportional to the number of partitions, which we call the pipeline width.

Given J, the set of all MapReduce jobs, and P_j, the set of all pipelines within job $j \in J$, the progress of a computation is thus given by the following formula, where N_{jp} and K_{jp} values are aggregated across all partitions of the same pipeline and $\text{Setup}_{\text{remaining}}$ is the overhead for the unscheduled map and reduce tasks.

$$T_{\text{remaining}} = \text{Setup}_{\text{remaining}} + \sum_{j \in J} \sum_{p \in P_j} \frac{s_{jp}\alpha_{jp}(N_{jp} - K_{jp})}{\text{pipeline\_width}_{jp}}$$

When estimating pipeline width, Parallax takes into account the cluster capacity and the (estimated) dataset sizes. In a MapReduce system, the number of map tasks depends on the size of the input data, not the capacity of the cluster. The number of reduce tasks is a configurable parameter. The cluster capacity determines how many map or reduce tasks can execute simultaneously. In particular, if the number of map (or reduce) 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, in a system with a 256 MB chunk size (a recommended value that we also use in our experiments) and enough capacity to execute 16 map tasks simultaneously, would be processed by a round of 16 map tasks followed by a round with only 4 map tasks. Parallax takes this slowdown into account by computing, at any time, the average pipeline width for the remainder of the job.

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

$$P_{\text{complete}} = \frac{T_{\text{remaining}}}{T_{\text{complete}} + T_{\text{remaining}}} \quad (1)$$

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

3. ParaTimer

In this section, we present ParaTimer: a progress indicator for parallel queries that take the form of directed acyclic graphs (DAGs) of MapReduce jobs. ParaTimer builds on Parallax but takes a radically different strategy for progress estimation. First, to support complex tree-shaped or DAG-shaped queries such as those which include joins, ParaTimer adopts a critical-path-based progress estimation technique: ParaTimer identifies and tracks only those map and reduce tasks on the query's critical path (Section 3.1). Interestingly, when the critical path includes many nodes executing in parallel, ParaTimer can monitor more of the nodes to improve progress-estimation accuracy or fewer of the nodes to reduce monitoring overhead. Additionally, ParaTimer is designed to work well under a variety of adverse scenarios including failures (Section 3.2) and data skew (Section 3.3). For this, ParaTimer introduces the idea of providing users with a *set* of estimated query runtimes assuming different execution scenarios (e.g., with and without failures or worst-case and best-case schedule). Because each execution scenario could be associated with a probability (i.e., probability of a single failure, probability of two failures, etc.), these multiple estimators can be seen as samples from the query-time probability distribution function.

3.1 Critical-Path-Based Progress Estimation

To handle complex-shaped query plans in the form of trees or DAGs, ParaTimer adopts the strategy of identifying and tracking the critical path in a query plan. For this, ParaTimer proceeds in four steps. First, it pre-computes the expected task schedule for a query (Section 3.1.1). Second, it extracts path fragments from this schedule (Section 3.1.2). Third, it identifies the critical path in terms of these path fragments (Section 3.1.3). Finally, it tracks progress on this critical path (Section 3.1.4).

3.1.1 Computing the Task Schedule

To identify the critical path, ParaTimer first mimics the scheduler algorithm to pre-compute the expected schedule for all tasks and thus all pipelines in the query.

In this paper, we assume a FIFO scheduler, the default in Hadoop. With a FIFO scheduler, jobs are launched one after the other in sequence. All the tasks of a given job are scheduled before any tasks of the next job get any resources. Hence, the only possibility for concurrent execution of multiple jobs is when a job has fewer tasks remaining to run than the cluster capacity, C. At that time, the remaining capacity is allocated to the next job (unless it must wait for the previous job to finish, as indicated by the DAG). Both map and reduce task scheduling follow this strategy. Reduces are further constrained by the map schedule. They can start copying data as soon as the first map task ends, but the last round of data copy as well as the sort and reduce pipelines must proceed in series with the maps from the same job.

Figure 2 shows an example query plan that includes a join and enables inter-MapReduce-job parallelism in addition to intra-job parallelism. Figure 3(a) shows a possible schedule for the resulting map and reduce tasks in a cluster with enough capacity for five concurrent map and five concurrent reduce tasks.[3] For clarity, the figure omits the copy and sort pipelines but shows the map and reduce pipelines. In this example, we assume that Job 1 has two map tasks and one reduce task, Job 2 has six map tasks and one reduce task,

[3]In Hadoop terminology, we say that the cluster has five map slots and five reduce slots.

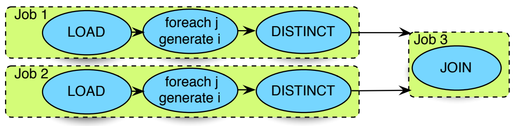


Figure 2: Example Pig Latin query plan with a join operator.

and Job 3 has one map task and one reduce task. As the figure shows, the map tasks for Jobs 1 and 2 can execute concurrently before the map tasks for Job 3 run. Reduce tasks execute after their respective map tasks.

Given a DAG of MapReduce jobs, ParaTimer thus computes a schedule, S, such as the one shown in Figure 3(a) but including also copy and sort pipelines.

While pre-computing the schedule using the given scheduler algorithm, ParaTimer uses Parallax to estimate the time that each pipeline will take to run. Given a schedule, ParaTimer breaks the query plan into path fragments as we describe next.

3.1.2 Breaking A Schedule into Path Fragments

Given a FIFO scheduler, a MapReduce task schedule has a regular structure because, typically, batches of tasks are scheduled at the same time. If all tasks in the batch process approximately the same amount of data and do so at approximately the same speed, they all end around the same time and a new batch of tasks can begin. For example, in Figure 3(a), $m11$ and $m12$ form one such batch. When this batch ends, another batch comprising tasks $m24$ and $m25$ begins. Tasks $m21$, $m22$, and $m23$ form yet another batch. We call each such batch a *round of tasks*. A round of tasks can be as small as one task. For example $r1$ forms its own round of tasks. A round of tasks can be no larger than the cluster capacity, C, which is five tasks in the example. More precisely:

DEFINITION 3.1. *Given a schedule S, a* task round*, T, is a set of tasks $t \in S$ that all begin within a time δ_1 of each other and end within a time δ_1 of each other.*

δ_1 defines how much skew is tolerable while still considering tasks to belong to the same round. This is a configurable parameter. We discuss skew further in Section 3.3. In MapReduce systems, task rounds are typically scheduled one after the other in sequence. More precisely, we say that two rounds are *consecutive* if the delay between the end of one round (the end of the last task in the round) and the beginning of the next round (start time of the first task in the new round) is no more than the setup overhead, δ_2, of the system (δ_1 and δ_2 are independent of each other).

Given the notion of consecutive path rounds, we define a path fragment as follows:

DEFINITION 3.2. *A* path fragment *is a set of tasks all of the same type (i.e., either maps or reduces) that execute in* consecutive rounds*. In a path fragment, all rounds have the same width (i.e., same number of parallel tasks) except the last round, which can be either full or not.*

Note that each task belongs to exactly one path fragment, i.e., path fragments partition the tasks. Given the above definition, the schedule in Figure 3(a) comprises the following six path fragments: $p1 = \{m11, m12, m24, m25\}$, $p2 = \{m21, m22, m23, m26\}$, $p3 = \{r1\}$, $p4 = \{r2\}$, $p5 = \{m3\}$, and $p6 = \{r3\}$.

It is worth noting that the map path fragments comprise only map pipelines. Reduce path fragments, however, comprise copy, sort, and reduce pipelines.

To understand how these path fragments represent parallel query execution, it is worth considering three job configurations:

Sequence of MapReduce Jobs. If a query comprises only a sequence of MapReduce jobs, the tasks for different jobs never overlap and we simply get one path fragment for each job's map tasks and a second one for each job's reduce tasks. The critical path is the sequence of all these path fragments and our algorithm implicitly becomes equivalent to Parallax.

Parallel Map Tasks. In the absence of parallelism, a query is thus a series of path fragments, all of width equal to the cluster capacity (or less once fewer tasks remain). The effect of parallelism is to divide the concurrently executing tasks into multiple "thinner" path fragments because tasks from different jobs have different runtimes and violate the "time difference $< \delta_1$" rule. Hence, when two jobs execute concurrently, there are two path fragments operating simultaneously as in Figure 3(a). In our example, we know the cluster will first execute $m11, m12, m21, m22, m23$. Because map tasks belong to two different jobs and are thus likely to take different amounts of time, they are divided into two path fragments $p1 = \{m11, m12, m24, m25\}$ and $p2 = \{m21, m22, m23, m26\}$. Conversely, if Parallax estimated Job 1's map tasks to take longer than Job 2's map tasks, the fragments would be $\{m11, m12\}$ and $\{m21, m22, m23, m24, m25, m26\}$. Similarly, when N queries execute in parallel (for any $N \leq C$), there are N path fragments operating simultaneously.

Parallel Reduce Tasks. When parallel jobs comprise both map and reduce tasks, the number of path fragments further increases. Path fragments that involve map tasks are identified as described above. We now discuss path fragments in the reduce phases. We assume no data skew. We discuss data skew in Section 3.3.

There are three cases for reduce tasks:

- *Case 1*: Reduces run far apart from each other. This is the case in the example in Figure 3(a). Reduces run after their respective maps, but they are much shorter than the maps and thus create long gaps between themselves. In this scenario, ParaTimer places the reduces for different jobs in different path fragments.
- *Case 2*: Reduces overlap. Let's imagine that the reduce for Job 1 stretches all the way past the end of Job 2's map tasks. In this case, however, this reduce still remains in its own path fragment because Job 2's reduce can run right after Job 2's map tasks end in another available slot. Hence, the path fragments are the same as in Case 1.
- *Case 3*: Reduces run in sequence. Imagine case 2 but with more reduce tasks for Job 1, enough to fill the entire cluster capacity or more. In this last case, Job

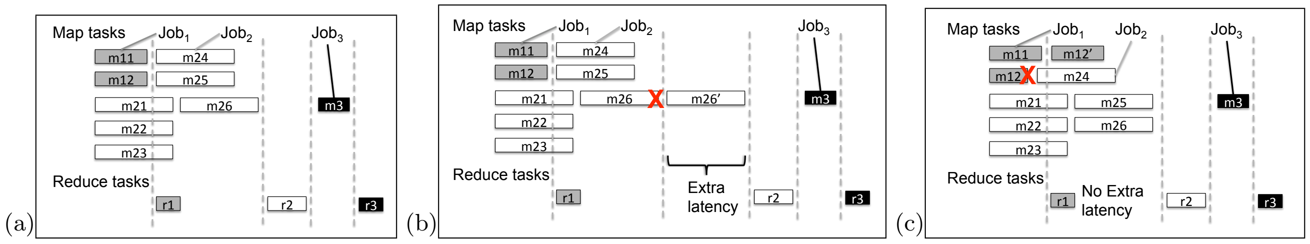


Figure 3: Possible execution schedule for jobs from Figure 2 on a cluster with 5 map and 5 reduce slots. (a) Execution without failure. (b) Worst-case failure in terms of latency. (c) Failure adds a path fragment but does not change latency

2 reduces will run directly after Job 1 reduces, forming either one path fragment (if Job 2 reduces were a multiple of cluster capability) or two (otherwise).

Once again, these reduce path fragments comprise the copy, sort, and reduce pipelines. Early copies are ignored for the purpose of path fragment identification: reduces are assumed to run entirely after the corresponding map path fragments end.

3.1.3 *Identifying the Critical Path Fragments*

Given a schedule and an assignment of tasks to path fragments, it is easy to derive a schedule in terms of path fragments where each path fragment is accompanied by a `start` time and a `duration`. The start time of a path fragment is simply the lowest start time of all tasks in the fragment. The duration of the path fragment is the sum of the durations of all the rounds (recall that by definition all tasks within a path fragment have approximately the same duration, given by Parallax).

Given a schedule expressed in terms of path fragments, ParaTimer identifies the fragments on the critical path using the following simple algorithm.

ParaTimer starts with the entire path-fragment schedule. As long as there exist overlapping-in-time path fragments in the schedule, perform the following substitutions:

- Case 1: If two overlapping path fragments start at the same time, keep only the one expected to take longer. In the example, $p1$ and $p2$ execute in parallel. Hence, the shorter $p1$ fragment can be ignored.
- Case 2: If two overlapping path fragments start at different times, keep the one that starts earlier. Remove the other one, but add back its extra time. In our example, $p2$ and $p3$ overlap. Because the overlap is total, $p3$'s time can be ignored. However, if $r1$ stretched past the end of $m26$, the extra time would be taken into account on the critical path.

The end-result is a schedule in the form of a series, and this is the critical path.

3.1.4 *Estimating Time Remaining at Runtime*

In the absence of changes in runtime conditions, path fragments and the critical path can be identified once prior to query execution. The path fragments on the critical path are then monitored at runtime and their time-remaining computed using Parallax. The time-remaining for the critical path is the sum of these per-path-fragment time-remainings. For path fragments that partly overlap, only their extra, non-overlapping time is added.

Instead of monitoring all tasks in a path fragment on the critical path, ParaTimer could monitor only a thread of tasks within the path fragment (or some subset of these threads), where a thread is a sequence of tasks from the beginning to the end of a path fragment. This opportunity enables ParaTimer to offer a flexible trade-off between overhead and potential progress estimation accuracy: wider path fragments can potentially smooth time-remaining estimates by smoothing away blips due to small inaccuracies and variations in task completion times (although we did not see significant differences in our experiments). Thinner path fragments, however, can reduce monitoring overhead. Furthermore, when tasks are grouped into path fragments, ParaTimer can easily change which tasks it tracks at runtime to better balance the monitoring load yet still track the critical path.

Alternatively, ParaTimer could also monitor *all* pipelines in an ongoing query (not just the critical path) and could recompute the schedule at each time tick. This choice represents the maximum overhead and maximum accuracy monitoring solution. In fact, when runtime conditions change, the schedule and critical path must be recomputed dynamically as we discuss next.

3.2 Handling Failures

The MapReduce approach has been designed for processing massive-scale datasets with queries running across hundreds or even thousands of nodes [6]. At that scale, failures are likely to occur. For this reason, MapReduce is designed to provide intra-query fault-tolerance. As a query executes, MapReduce materializes the output of each map and reduce task. If a task fails, the system simply restarts the failed task, possibly on a different node. The task reprocesses its materialized input data and materializes its output again from the beginning.

Failures can significantly affect progress estimation. As an example, Figure 3(b) and (c) shows two schedules for the query from Figure 3(a) for two different failure scenarios. Depending when the failure occurs, it may or may not affect the query time and it may affect it by a different amount.

The challenge with handling failures is that the system, of course, does not know ahead of time what failures, if any, will occur. As a result, there is no way to predict the running time for a query accurately. The best answer that the system can provide about remaining query time is, "It depends."

To address this challenge, we take an approach that we call *comprehensive progress estimation*. Instead of outputting only one, best guess about the query time, ParaTimer outputs multiple guesses. Ideally, one would like to give the user a probability distribution function of the remaining query time. However, such a function would be difficult to estimate with accuracy. Instead, we take the approach of outputting a handful of select points on that curve.

3.2.1 *Comprehensive Progress Estimation*

For clarity of exposition, we refer to the standard ParaTimer approach described in previous sections as the *StdEstimator*. We now describe possible additional estimates that ParaTimer can output assuming that failures occur during query execution.

One important estimator in the presence of failures is what we call the *PessimisticFailureEstimator*. This estimator assumes a single task execution will fail but that the failure will have worst-case impact on overall query execution time. This estimator is useful because single-failures are likely to take place and the estimator provides an *upper bound* on the query time in case they arise. The upper bound is also useful because it *approximates* the time of an execution with a single failure that could *actually occur*. An example of upper bound that would be less useful would be to assume the entire query is re-executed upon a failure and to return as possible time-remaining the same value as StdEstimator plus the value of StdEstimator at time zero (basically the time-remaining plus the estimated total time without failure). In most cases, PessimisticFailureEstimator will return a much tighter upper bound.

Consider again the example in Figure 3. The StdEstimator would output the time remaining for the schedule shown in Figure 3(a), while PessimisticFailureEstimator would show the time for the schedule shown in Figure 3(b). Even though the failure is worst-case, the query time is extended by only a small fraction.

Three conditions make a failure a worst-case failure. First, the longest remaining task must be the one to fail. In the example, the map tasks of Job 2 are the longest tasks to run. Second, the task must fail right before finishing as this adds the greatest delay. Third, the task must have been scheduled in the last round of tasks for the given job and phase. Indeed, if one of tasks $m21$ through $m23$ failed, the query latency would not be affected.

PessimisticFailureEstimator assumes such a worse-case scenario. For simplicity, however, instead of examining the schedule carefully to determine the exact worst-case scenario that is possible, PessimisticFailureEstimator approximates that scenario by simply assuming the longest upcoming pipeline will fail right before finishing and will fail at a time when nothing else can run in parallel. As a result, PessimisticFailureEstimator produces the following time-remaining value for a query Q comprising a set of pipelines P partitioned into P_{done}, $P_{scheduled}$, and $P_{blocked}$:

$$\begin{aligned}&\text{PessimisticFailureEstimator}(Q) =\\ &= \text{StdEstimator}(Q) + max_{\forall p \in P_{\text{scheduled}} \cup P_{\text{blocked}}}(Parallax(p))\end{aligned}$$

In addition to PessimisticFailureEstimator, ParaTimer could output additional query time estimates. In particular, as the scale of a query grows and multiple failures become likely, ParaTimer could output estimates that allow for multiple failures. Going in the other direction, if users want tighter bounds than PessimisticFailureEstimator, ParaTimer could output time-remaining assuming failures that are not necessarily worst-case failures. ParaTimer's goal is to enable users to select from a battery of such additional query time bounds, depending on their system configuration and monitoring needs. However, we currently support and evaluate only the PessimisticFailureEstimator.

3.2.2 *Adjusting Estimates after Failures*

After a failure occurs, it is crucial to recompute all estimators. There is no sense in the StdEstimator reporting zero-failure execution time when we know a failure has occurred. Just as the StdEstimator should account for one past failure and no future failures, the PessimisticFailureEstimator should account for one past failure and another worst-case future failure. In the example, as soon as task $m26$ fails and $m26'$ starts, StdEstimator updates its schedule and recomputes time remaining. Similarly, PessimisticFailureEstimator leverages the new StdEstimator and assumes that $m26'$ will fail before finishing. Once $m26'$ ends, PessimisticFailureEstimator will start returning a time-remaining that assumes $r2$, the new longest remaining task, will fail.

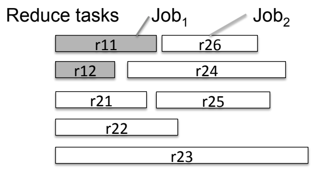
In general, a failure can affect all not-completed path fragments and the identity of the critical path, so it is necessary to recompute these entities from the revised schedule. For example, when a failure occurs, as illustrated in Figure 3(c), the failure can stagger the tasks inside a path fragment by more than value δ_1, which requires separating these tasks into two path fragments (e.g., $m11$, $m12'$ and $m24$ form two path fragments after the failure). As the figure shows, a failure can also cause some tasks to move to different path fragments (e.g., $m25'$), possibly splitting them in two (not shown in the figure). In other cases, such as when $m26$ fails, path fragments remain the same. To correctly handle all these cases, when a failure occurs, ParaTimer examines all currently scheduled tasks and runs the scheduler forward to get the correct new schedule, path fragments, and critical path.

3.3 Handling Data Skew

So far, we assumed uniform data distribution and approximately constant per-tuple processing times. Under these assumptions, all partitions of a pipeline process the same amount of data and end at approximately the same time. Frequently, however, data and processing times are not distributed in such a uniform fashion but instead are skewed. In this section, we address the problem of data skew, when imbalance comes from an uneven distribution of data to partitions.

In a MapReduce system, skew due to uneven data distribution can occur only in reduce tasks. It cannot arise for map tasks because each map task processes exactly one data chunk and all chunks (except possibly the last one) are of the same size. We thus focus on the case of data skew in reduce pipelines.

A possible schedule for a set of reduce tasks, where each task processes a different amount of data could be as follows:



When data skew occurs, we no longer have the nice, wide path fragments that we had before. Instead, each slot in the

Algorithm 1 Estimates in presence of data skew

```
Input: $R_{scheduled}$: Set of scheduled reduce tasks
Input: $R_{blocked}$: Set of blocked reduce tasks
Input: n: Expected number of rounds
Output: UpperBoundEstimate and LowerBoundEstimate
 1: // Compute time of r using Parallax
 2: $\forall r \in R_{scheduled}$ $\texttt{Time}_{\texttt{scheduled}}[r] = Parallax(r)$
 3: $\forall r \in R_{blocked}$ $\texttt{Time}_{\texttt{blocked}}[r] = Parallax(r)$
 4: Sort($\texttt{Time}_{\texttt{scheduled}}$) descending
 5: Sort($\texttt{Time}_{\texttt{blocked}}$) descending
 6: UpperBoundEstimate $= \texttt{Time}_{\texttt{scheduled}}[0] + \sum_{i=0}^{n-1} \texttt{Time}_{\texttt{blocked}}[\texttt{i}]$
 7: $RS = |R_{scheduled}|$
 8: $RB = |R_{blocked}|$
 9: LowerBoundEstimate $= \texttt{Time}_{\texttt{scheduled}}[RS-1] +$
10:         $\sum_{i=RB-n}^{RB-1} \texttt{Time}_{\texttt{blocked}}[\texttt{i}]$
```

cluster becomes its own path fragment.

If the MapReduce scheduler is deterministic, ParaTimer can pre-compute the expected task schedule for a query. It can then use it to reliably estimate the time on all path fragments and identify the critical path.

The challenge is when the scheduler is not completely deterministic. In particular, the challenge arises when ParaTimer does not know how tasks within a job will be scheduled exactly. As a consequence, ParaTimer cannot be certain of the query time because the schedule will affect that time. To address this challenge, we also adopt the *comprehensive estimation* approach. That is, ParaTimer outputs multiple estimates for the query. Each estimate gives the expected query time under a different scenario.

For data skew, different estimates could be useful. We propose to show users two estimates: an upper bound and a lower bound on the expected query time. For a set of reduce tasks, the approach works as follows:

Given a set of reduce tasks R and a cluster capacity C, expressed in terms of number of slots, if cardinality estimates point to data skew, ParaTimer considers that there are C parallel path fragments for both the copy and reduce pipelines. The expected number of rounds within each of these path fragments is given by: $n = \lceil \frac{R}{C} \rceil$. Before the tasks in R start executing, ParaTimer reports the time of chaining together either the n longest tasks (UpperBoundEstimate) or n shortest tasks (LowerBoundEstimate).

Once the tasks start executing, we take R to contain just the not-yet-completed tasks. We then partitition R into two disjoint sets $R_{scheduled}$ and $R_{blocked}$ where $R_{scheduled} \cap R_{blocked} = \emptyset$, $R_{scheduled} \cup R_{blocked} = R$, and $R_{scheduled}$ refers to tasks that have started. We update n to be $\lceil \frac{R_{blocked}}{C} \rceil$. We then report as an upper bound the time of chaining together the longest currently executing task followed by the $n-1$ longest unscheduled tasks and similar for the lower bound as shown in Algorithm 1.

When multiple jobs are chained together, time-remaining estimation errors accumulate and ParaTimer reports the sum of all upper bounds as the upper bound. It reports the sum of all lower bounds as the lower bound

Other upper and lower bounds are possible. In particular, one could examine the current schedule more carefully to make the bounds tighter. However, our current choices yield useful results as we show next.

4. EVALUATION

In this section, we evaluate the ParaTimer estimator through a set of microbenchmarks. In each experiment, we run a Pig Latin query in a real small-scale cluster. The input data is synthetic with sizes up to 8GB and either uniform or Zipfian data distribution.

We compare the performance of ParaTimer against that of Parallax [23], Pig's original progress estimator [25], and previous techniques for single-site progress estimation, in particular GNM [3], DNE [3], and Luo [19]. We reimplemented the GNM, DNE, and Luo estimators in Pig/Hadoop. We demonstrate that ParaTimer outperforms all these earlier proposals on parallel queries with joins. We also show ParaTimer's performance in the presence of failures and data skew and, for the latter, compare again against Parallax.

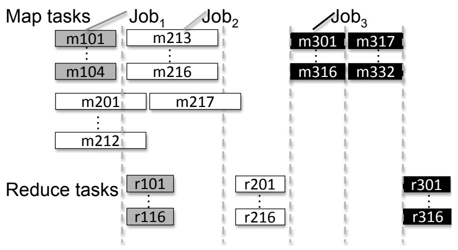


Figure 4: Parallel join experiment: Task schedule for Pig Latin query comprising a join operator and translating into three MapReduce jobs.

4.1 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 concurrent map tasks and 16 concurrent reduce tasks.

In all experiments, we use perfect cardinality estimates (N values) in order to isolate the other sources of errors in progress estimation. Both Parallax and ParaTimer are demonstrated in two forms: *Perfect*, which uses N and α values from a prior run over the entire data set; and *1%* 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.

4.2 Parallel Queries with Joins

In this section we investigate how well ParaTimer handles Pig Latin queries containing a join operator through two experiments with different critical path configurations. Our experiments include a foreign-key join of two uniformly-distributed data sets. All experiments in this section consist of three jobs: the first two perform a DISTINCT operation in parallel on two input data sets followed in sequence by a third job that performs an equi-join of their outputs (as in Figure 2).

The schedule of the first join experiment is depicted in Figure 4. This experiment runs for approximately 28 minutes. Job 1 processes 1 GB of data through four parallel maps and 16 reduces. Job 2 processes 4.2 GB of data through 17 map tasks and 16 reduces.

Figures 5 and 6 show the results for ParaTimer, Parallax, Pig's existing indicator, and the other single-site indi-

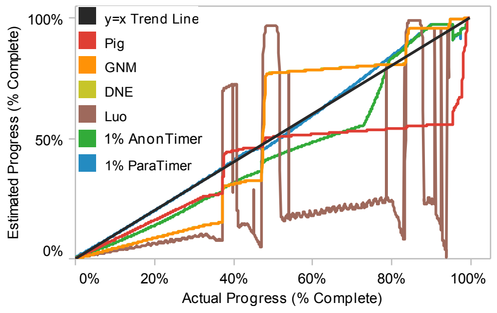


Figure 5: Parallel join experiment: Time-remaining estimates for parallel query with join. 4.2 GB and 1 GB data sets, eight-node cluster. Task schedule as in Figure 4

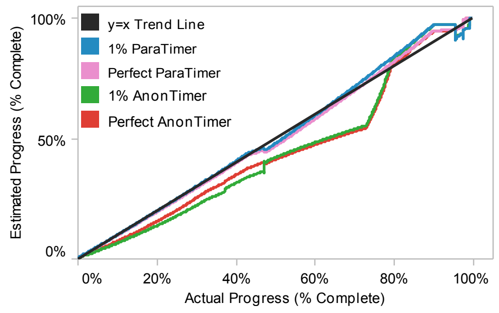


Figure 6: Parallel Join Experiment: Time-remaining estimates for parallel query with join. 4.2 GB and 1 GB data sets, eight-node cluster. Task schedule as in Figure 4

cators from the literature (GNM [3], DNE [3], and Luo [19]). In these figures, the x-axis shows the real percent-time remaining for the query and the y-axis shows the estimated percent-time remaining. Hence, the closer a curve is to the $x = y$ trend-line, the smaller the estimation error.

We report both the average and maximum across the instantaneous errors for all experiments in this section. The instantaneous error is computed as in [3]:

$$error = \left| \frac{100 * (t_i - t_0)}{(t_n - t_0)} - f_i \right| \quad (2)$$

where f_i is the reported percent-time done estimate, t_i is the current time, t_n is the time when the query completes, and $(t_i$ -$t_0)/(t_n$-$t_0)$ represents the actual percent-time done.

Overall, ParaTimer does very well with average error under 1.6% and maximum error under 7.1% The error is mostly concentrated at the end of the execution of the final round of map tasks in job 2. In this case optimistic estimates are reported but only for a brief amount of time. ParaTimer assumes that, in the absence of changes to external conditions, a pipeline will process data at constant speed. ParaTimer does not account for an extra blocking combine phase that is sometimes performed at the end of a map pipeline.[4]

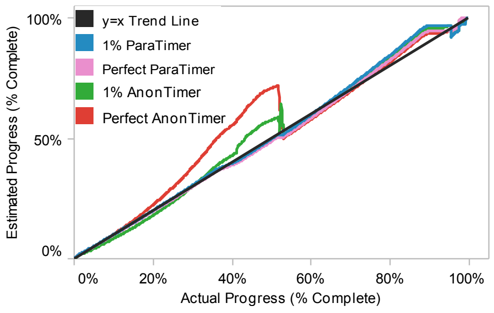


Figure 7: Parallel Join Experiment. Time-remaining estimates for parallel query with join. 4 GB and 1 GB data sets, eight-node cluster. Task schedule as in Figure 4 but without $m217$

A more refined model could improve these estimates, but would complicate the implementation.

Parallax has good average error (6-8%), but has high maximum error (18-19%). Since Parallax assumes a serial schedule of jobs consisting of job 1 followed by jobs 2 and 3, it incorrectly assumes that each job will execute with access to full cluster resources and will run one after the other. Assuming serial execution leads to pessimistic estimates. Assuming access to full cluster capacity leads to optimistic estimates. In this configuration, the serial assumption weighs more heavily and the estimate is pessimistic.

Figure 5 demonstrates that, as expected, indicators from the literature that are designed for single-site systems cannot be directly applied to a parallel setting. All of them have average errors $> 11\%$ and maximum errors $> 28\%$

The next join experiment uses the same Pig Latin script as before, but this time job 2 processes a 4GB input data set, which creates 16 map tasks. The schedule of tasks for this experiment is similar to Figure 4, except $m217$ is omitted. However, the critical path has changed and is computed through the first job's map tasks and the second job's map tasks $m213$ through $m216$. Figure 7 shows the results. The experiment ran for approximately 25 minutes.

ParaTimer performs similarly well to the previous join experiment, with average errors under 1.6% and maximum errors under 7.2%. Parallax's average errors are in the 3-5% range and maximum errors are as high as 21%. Parallax's errors are due to the incorrect assumption that the second job's map tasks will be running at full cluster capacity. It expects the pipeline width to be close to 16 for the execution of these maps, when in fact the pipeline width starts as 12 and drops to 4 after the first job's maps complete. Because of this assumption, the estimates trend optimistic.

Overall, ParaTimer thus significantly outperforms Parallax, reducing maximum errors in the presence of joins from approximately 20% to approximately 7% in our experiments.

4.3 Failures

In this section we examine the robustness of ParaTimer through four single-task failure scenarios. We start with the query schedule from Figure 4 and test different configurations of failures on or off the critical path and either changing or not that critical path. The following table summarizes the

[4]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.

experimental configurations:

	Where failure occurs	
Changes critical path	Other path	Critical path
No	A	C
Yes	B	D

Given the schedule from Figure 4, to obtain case A, we fail map task $m104$ at 195 seconds into its execution (around 35% complete). The scheduler selects the next available map task (here: $m213$) and schedules it in place of the failed one resulting in a schedule analogous to that in Figure 3(c). Experiment C is similar to A except that we fail $m201$ around 59 seconds into its execution. $m201$ then gets rescheduled alongside $m217$. In both cases, the query time before and after failure remains the same as the latency for the extra tasks can be hidden by the execution of $m217$. Since both graphs look almost the same, due to space constraints, we show only results for one experiment.

Figure 8 shows the results for experiment C. For failures, we find it easier to reason about time-remaining rather than percent-time done. In Figure 9, we show results from experiment C again but, this time, in terms of time-remaining. In this figure, the black trend-line shows the actual time-remaining for the query. Curves above this line over-estimate time-remaining. Curves below the line under-estimate time-remaining.

As discussed in Section 3.2, ParaTimer produces multiple estimates in the case of failures: StdEstimate and PessimisticFailureEstimate. StdEstimate is represented as *Perfect* and *1%* ParaTimer in the figures. PessimisticFailureEstimate appears as PessimisticEstimate. Finally, we present an additional estimate, referred in this section as *FailureEstimate*, which provides an estimate in between StdEstimate and PessimisticEstimate. Before a failure occurs, FailureEstimate (like PessimisticFailureEstimate) is cautious and accounts for a failure of the longest-running task among all current and future pipelines. However, once a failure occurs, it assumes that no more failures will occur for the remainder of that job's pipeline. At this point, its time-remaining estimate is equivalent to StdEstimate but only until the end of that pipeline at which time, FailureEstimate assumes that a failure will occur again.

In the case of experiments A and C, since the query time is not affected by the failure, StdEstimate shows the correct time-remaining throughout query execution with an average error below 2% in both experiments. The PessimisticFailureEstimate over-estimates the time throughout most of the execution. At time 1500 seconds, once the long-running map tasks from Job 2 end, PessimisticFailureEstimate correctly updates itself by assuming only one of the remaining short tasks can fail. Finally, FailureEstimate correctly follows PessimisticFailureEstimate before failure and StdEstimate after the failure and until the end of the map tasks. It then follows PessimisticFailureEstimate again. The overestimation of the query-time by PessimisticFailureEstimate is 15% on average. It is a bit high at 30% right before the job2 map tasks end because of the large difference in execution times for the two types of tasks.

To obtain Case B, we fill-up the path fragment comprising tasks $\{m201, \dots, m212, m217\}$ to form a new path fragment with tasks $\{m201, \dots, m212, m217, \dots, m228\}$. With this setup, there is no more room to hide any restarted tasks. We then fail task $m104$ after 296 seconds (at 53% complete).

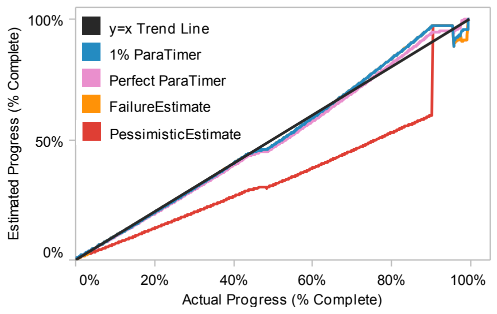


Figure 8: Failure case C, percent-done.

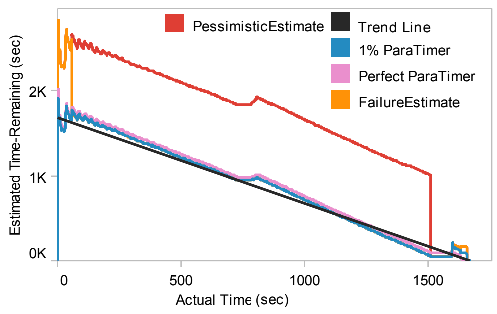


Figure 9: Failure case C, time-remaining

For case D, we use the same setup but fail task $m201$, which is on the critical path, at 676 seconds into its execution (at 93% complete). In both cases, the critical path changes and the time-remaining increases after the failure. Figure 10 shows the time-remaining curve for experiment D (experiment B has similar shape). As expected, before the failure happens, StdEstimate provides a lower-bound on query execution while PessimisticFailureEstimate and FailureEstimate are providing an upper-bound on query execution. This is exactly the desired behavior. The span between the two is small. The upper bound over-estimates query time by 8% while the lower-bound underestimates it by at most 10%. After the failure, all estimators adjust their predictions as expected.

Overall, the ParaTimer approach to query-time estimation in the presence of failures thus works very well for all these different failure configurations.

4.4 Data Skew

The goal of the experiments in this section is to measure how well ParaTimer handles data skew, which results from an imbalance in the distribution of the data processed per task or partition. Recall from Section 3.3 that such skew arises only in reduce pipelines.

We run two experiments. Each one comprises a Pig Latin script that performs a GROUP-BY operation through a single MapReduce job. Moreover, the script loads an 8 GB data set with a Zipfian distribution on the key used by the GROUP-BY operator, which results in data skew in the reduce pipeline.

For the first experiment, we manually configured the Pig Latin script to produce a single round of 16 reduce tasks.

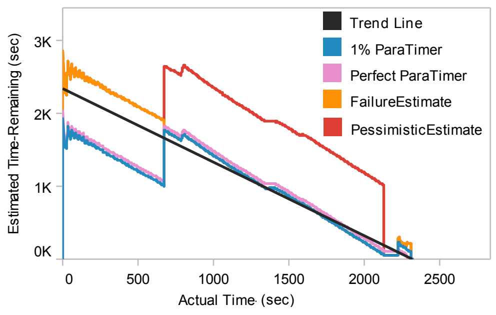


Figure 10: Failure case D, time-remaining

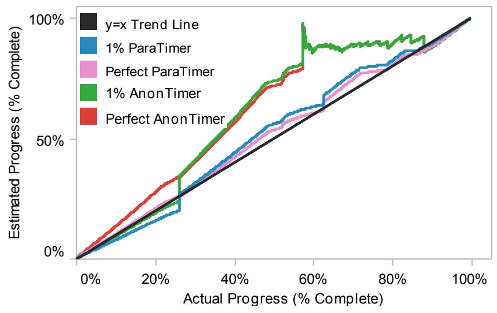


Figure 11: Simple data skew experiment. Percent time-remaining estimates in the presence of Zipfian skew 8GB data set (32 maps, 16 reduces), eight-node cluster. Skew occurs in the reduce phase

In that case, ParaTimer can predict the schedule with certainty: all reduce tasks will be scheduled concurrently. It can thus reliably identify and follow the critical path. It produces a "best guess" estimate from this offline, precomputed critical path. For the second experiment, we double the number of reduce tasks. In this scenario, ParaTimer may not know exactly how tasks will be scheduled and must thus output an upper- and lower-bound estimate. The results for the first experiment are in Figure 11 and for the second experiment in Figures 12. The first experiment ran in 49 minutes and the second in 45 minutes.

Figure 11 shows that, as expected, ParaTimer's "best guess" estimate is accurate for the simple case of data skew with a single round of reduce tasks and thus a predictable schedule. Average errors are 1.4% for *Perfect* ParaTimer and 3.2% for *1%* ParaTimer. The maximum errors for both were under 7.3%.

ParaTimer additionally produces accurate estimates for a more complex data skew scenario with less predictable schedules. Here "best guess" was within 5% average error for *1%* ParaTimer and within 3% for *Perfect* ParaTimer. As expected, Figure 12, shows the "best guess" estimates between the upper and lower bound curves. Furthermore, the bounds provide reasonable estimates: lower bound underestimates the query time by 12% while the upper bound overestimates it by at most 9%.

In both data skew experiments, Parallax produces significantly less accurate estimates. For both experiments, the average error was within 11% with very high maximum errors in the 30-40% range. Parallax's accuracy suffers because it assumes that each reduce partition processes a uniform amount of data. Since it does not take this skew into account, it produces overly-optimistic estimates for both experiments.

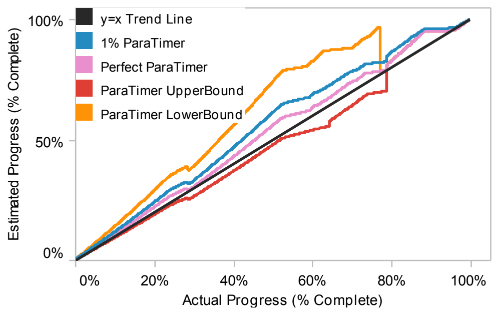


Figure 12: Complex data skew experiment. Percent time-remaining estimates in the presence of Zipfian skew 8GB data set (32 maps, 32 reduces), eight-node cluster

5. 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 [4, 5, 7, 10] (e.g., elapsed time, number of tuples output so far). Some systems [7, 10] 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. Pig/Hadoop's existing progress estimator [25] takes a similar approach. It shows a percent-remaining estimate but has low accuracy (Figure 5) because it assumes all operators process data at the same speed. Our approach strives to estimate time remaining with significantly more accuracy.

There has been significant recent work on developing progress indicators for SQL queries executing within single-node DBMSs [3, 2, 18, 19, 21, 22], possibly with concurrent workloads [20]. In contrast, ParaTimer focuses on the challenges specific to parallel queries: distribution across multiple nodes, concurrent execution, failures, and data skew.

Chaudhuri *et al.* [3] maintain upper and lower bounds on operator cardinalities to refine their estimates at runtime. These bounds are not analogous to ParaTimer's bounds. Chaudhuri *et al.* use bounds only to correct their single best-guess estimate of query progress when original cardinality estimates are incorrect or to produce approximate estimates with provable guarantees in the presence of join skew [2]. In contrast, ParaTimer focuses on producing *multiple useful* guesses on query times. Further, ParaTimer's guesses are also not necessarily absolute upper and lower bounds but rather additional estimates for different possible conditions.

In follow-on work, Chaudhuri *et al.* [2] study the problem of join skew in single-node estimators, where different input tuples contribute to very different numbers of output tuples. In contrast, we focus on data skew across partitions of an operator and do not consider join skew.

In preliminary prior work, we developed Parallax [23], the first non-trivial time-based progress estimator for parallel queries. However, Parallax only works for very simple queries in mostly static runtime conditions. In contrast, ParaTimer's approach works for parallel queries with joins and in the presence of data skew and failures.

Query progress is related to the cardinality estimation problem. There exists significant work in the cardinality estimation area including recent techniques [21, 22] that continuously refine cardinality estimates using online feedback from query execution. These techniques can help improve the accuracy of progress indicators. They are orthogonal to our approach since we do not address the cardinality estimation problem in this paper.

Query optimizers have a model of query cost and compute that cost when selecting query plans. These costs, however, are designed for selecting plans rather than computing the most accurate time-remaining estimates. As such, optimizer's estimates can be inaccurate time-remaining indicators [9, 19]. Ganapathi *et al.* [9] use machine learning to predict query times before execution. In contrast, we focus on providing continuously updated time-remaining estimates during query execution taking runtime conditions such as failures into account.

Work on online aggregation [13, 16] 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.

Finally, query schedulers can use estimates of query completion times to improve resource allocation. Existing techniques for time-remaining estimates in this domain [28], however, currently use only heuristics based on Hadoop's progress counters, which leads to similar limitations as in Pig's current estimator.

6. CONCLUSION

We presented ParaTimer, a system for estimating the time-remaining for parallel queries consisting of multiple MapReduce jobs running on a cluster. We leveraged our earlier work that determines operator speed via runtime measurements and statistics from earlier runs on data samples. Unlike this prior work, we support queries where multiple MapReduce jobs operate in parallel (as occurs with join queries), where nodes fail at run-time, and where data skew exists. The essential techniques involve identifying the critical path for the entire query and producing multiple time estimates for different assumptions about future dynamic conditions. We have implemented our approach in the Pig/Hadoop system and demonstrated that for a range of queries and dynamic conditions it produces quality time estimates that are more accurate than existing alternatives.

7. ACKNOWLEDGEMENTS

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

8. 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, V. Narassaya, and R. Ramamurthy. Estimating progress of execution for SQL queries. In *Proc. of the SIGMOD Conf.*, Jun 2004.

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

[5] 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.

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

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

[8] 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.

[9] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc. of the 25th ICDE Conf.*, pages 592–603, 2009.

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

[11] Greenplum database. `http://www.greenplum.com/`.

[12] Hadoop. `http://hadoop.apache.org/`.

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] C. Mishra and N. Koudas. A lightweight online framework for query progress indicators. In *Proc. of the 23rd ICDE Conf.*, 2007.

[22] C. Mishra and M. Volkovs. ConEx: A system for monitoring queries (demonstration). In *Proc. of the SIGMOD Conf.*, Jun 2007.

[23] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of mapreduce pipelines. In *Proc. of the 26th ICDE Conf. (To appear)*, 2010.

[24] 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.

[25] Pig Progress Indicator. `http://hadoop.apache.org/pig/`.

[26] A. Pruscino. Oracle RAC: Architecture and performance. In *Proc. of the SIGMOD Conf.*, page 635, 2003.

[27] Vertica, inc. `http://www.vertica.com/`.

[28] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in

heterogeneous environments. *Proc. of the 8th OSDI Symp.*, 2008.