KAMD: A Progress Estimator for MapReduce Pipelines

Kristi Morton, Abe Friesen
University of Washington
Seattle, WA
{kmorton,afriesen}@cs.washington.edu

ABSTRACT

Limited user-feedback exists in cluster computing environments such as MapReduce. Accurate, time-oriented progress indicators could provide much utility to users in this domain, where job execution times can have high variance due to the amount of data being processed, the amount of parallelism available, and the types of operators (often user-defined) that perform the processing. This feedback would help users make informed decisions, such as whether a job should be terminated and restarted at a later time when the cluster has more resources available. However, none of the techniques used by existing tools or available in the literature provide a non-trivial progress indicator for queries running in a distributed environment. In this paper, we apply recently developed techniques for estimating the progress of single-site SQL queries to parallel environments. In particular, we target environments where queries consist of MapReduce job pipelines. We also present techniques that improve the accuracy and usefulness of progress estimators operating in this environment. We implemented our estimators in the Pig system and demonstrate its performance on experiments with real data (search logs) and with a real cluster.

1. INTRODUCTION

Very limited user-feedback exists in cluster computing environments such as MapReduce. Additionally, the high variance in job execution times due to dynamic parallelism and large datasets results in very unpredictable query times. It would greatly benefit users if running MapReduce [3] jobs could accurately report their progress. This feedback could be useful in a number of ways. If the user sees that the query will take much longer than expected he may be able to conclude that there is a mistake in the query without waiting for the results. Additionally, this information could allow the user to make better use of the time spent waiting on the query. Finally, this would enable the user to make informed decisions, such as whether to terminate and restart the job at a later time when more resources become available.

MapReduce [3] is a programming model for processing and generating large data sets. Users specify a *map* function that generates a set of key/value pairs, and a *reduce* function that merges or aggregates all values associated with the same key. A single combination of a map function and a reduce function is called a MapReduce job. MapReduce programs are automatically parallelized and executed on a large cluster of commodity machines. Data partitioning, scheduling, and inter-machine communication are all handled by the run-time system. This programming framework allows programmers to easily write parallel, distributed software.

In order to extend the MapReduce model beyond the simple one-input, two-stage data flow model and to remove the need to constantly rewrite basic operations, Olston et. al developed the Pig system [9]. Pig compiles queries written in Pig Latin, a language that combines the high-level declarative style of SQL with the low-level procedural programming model of MapReduce. Compiled Pig Latin queries are submitted to the MapReduce cluster as MapReduce jobs, which can be viewed as a series of pipelines. The power and flexibility of Pig Latin and the Pig System allow query optimization, ad-hoc data analysis of large datasets, and the ability to operate over plain input files that contain no schema information. For these reasons, Pig is quickly becoming a very popular system for data analysis. Furthermore, because Pig generally operates on very large datasets, progress information would greatly assist users and reduce frustration.

Current work on progress estimators in DBMSs focuses only on single-node SQL queries running in isolation. The techniques developed thus far include modeling the fraction of "work" completed or the time remaining based on the iterator model of query execution. This model relies on having accurate initial cardinality estimates of the input base tables. These estimates are based on database statistics (e.g. histograms) and execution feedback. Even in single-node DBMS systems, cardinality estimation is a nontrivial task and is subject to errors.

Our goal is to provide accurate estimates of MapReduce jobs in terms of the fraction of work complete and fraction of time remaining when good cardinality estimates are available. We apply the existing techniques from the literature [2, 1, 5, 4, 8, 7] to MapReduce jobs and present techniques that improve the accuracy and usefulness of progress estimators operating in a distributed environment. We implement these techniques in Hadoop, an open source MapReduce package, and Pig, an open source engine for compiling Pig Latin scripts to MapReduce jobs. These estimation techniques are not directly translatable to MapReduce for a number of reasons. First, no static database statistics are available to MapReduce jobs because the data and schemas are dynamically specified per job. Second, the execution behavior in this environment is much more dynamic and is subject to throttling due to the available parallelism. For example, Hadoop dynamically provisions the execution of subtasks based on system resource availability. As such, any progress estimator will have to deal with speculative execution, machine failures, and variations in data partitioning as any of these could dramatically alter the estimate. Additionally,

a time estimator will have to consider network congestion, machine load, data skew, and variations in processing power across machines, none of which have been considered in previous work. Third, while Pig Latin provides features similar to SQL, it is not equivalent to SQL and its operator set. Finally, while Pig provides a rich set of operators, users are still able to define their own functions to operate on the data within a map or reduce. Estimating progress within these functions will be very difficult.

In this paper, we implement and improve upon three progress estimation techniques from the SQL literature. Of these, two report progress as the fraction of work completed and the third reports the time remaining. We evaluate the accuracy of these estimators with and without our improvements by testing them on a Pig Latin script which performs queries over various sample sizes of the 42 MB 'excite' data set. This script and data set are provided in the standard Pig software distribution.

In Section 2 we discuss related query estimation work for SQL queries. Section 3 provides an overview of the problem and our model of a MapReduce query executing in a distributed environment. In Section 4 we present our solution, its merits, and its limitations. Section 5 analyzes the functionality, properties, and performance of our estimator. Finally, we conclude in Section 6.

2. RELATED WORK

There has been significant recent work on developing progress indicators for SQL queries executing within single-node DBMSs [1, 2, 4, 5, 6, 7, 8] Our approach extends these earlier efforts to parallel queries. In particular, we focus on estimating the progress of MapReduce pipelines.

Chaudhuri *et al.* [2] 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. The gnm's use of cardinality estimates for all intermediate nodes in the query plan can lead to highly inaccurate results. To address this challenge, Chaudhuri *et al.* introduced the *Driver Node Estimator (dne)*. The dne breaks a query plan into pipelines, which are maximal subtrees of concurrently executing operators. The progress of each pipeline is then 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. In follow-up work [1], 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 while the second, *safe*, provides an estimate that is worst case optimal. All of these techniques (gnm, dne, pmax, and safe), however, strive to produce a single value of query progress and cannot provide non-trivial guarantees in the general case [1]. In this paper, we show how to adapt the dne and gnm techniques to parallel queries. We extend them to output accurate time estimates in addition to percent work done.

Luo *et al.* [5, 4] proposed estimators similar to those of Chaudhuri *et al.* but they also estimate the remaining query execution times, in addition to percent complete. To convert the fraction of work done into the remaining processing time, their 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 [5] or it uses the output of the query optimizer as an estimate of query execution time for those pipelines that have not yet started [4]. Recent work also considers the impact of concurrent queries and their expected completion times to improve estimates [6]. In contrast to those techniques, our progress estimator converts progress to time with per-phase rates, functions well in distributed environments, and is resilient to system variability such as load and faults.

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 [7, 8] 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.

3. MAPREDUCE PROGRESS ESTIMATION

In this section we present an overview of MapReduce jobs and describe the challenges in applying the progress estimation techniques from the literature to a MapReduce framework.

3.1 Definitions

When a Pig Latin query is executed, Pig generates an operator-based query plan. Each operator processes a single key/value record at a time, much like the GetNext() iterator model used in traditional DBMSs. Pig splits the operators into multiple chained MapReduce jobs (as illustrated in Figure 1, taken from [9]) such that there can be multiple operators in each map or reduce. For example, every `GROUP` or `JOIN` operation forms a MapReduce boundary and the other operators are pipelined into map and reduce phases. Thus, the structure of the query plan is maintained while allowing Hadoop to take care of copying and materializing data, handling failures, and balancing load.

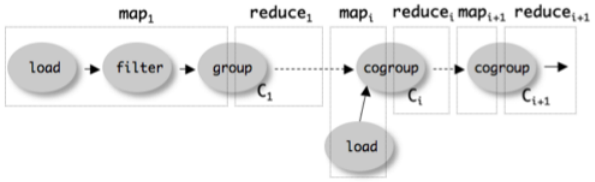

Figure 1: Compiled Pig Latin into MapReduce Jobs

Within a MapReduce job, there are seven phases of execution, as illustrated in Figure 2. These are the split, record reader, map runner, combine, copy, sort, and reducer phases. The split phase does very minimal work as it only gener-

ates byte offsets at which the data should be partitioned. We chose to ignore it in our estimator 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 records from the underlying data blob. These records are passed into the map runner and through the appropriate operators as they are created. As records are output from the map runner, they are passed to the combine phase which, if enabled, performs pre-aggregation to reduce the amount of data that will need to be transferred and then writes the records locally. If the combine phase is not enabled, the records are just written locally without any aggregation. Once the map task is complete, 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 to the node that the reduce task is executing on. After all of the map task outputs have been copied to the reduce node, the sort phase sorts the map output records. Once sorting is complete, the reducer phase reads each record and passes it through its own set of operators which generally perform some sort of aggregation. The output records from the reducer phase are written to disk as they are created. This is possible because the input to the reducer phase is already sorted.

Using the pipeline model presented in [2] and [5], an operator is defined as blocking if it does not produce any outputs until it has consumed at least one of its inputs completely. Each pipeline within a query plan consists of the tree of connected operators that have no blocking connections between them. The driver nodes of a pipeline are the leaf nodes of that pipeline. When Pig splits the operators into the sequence of MapReduce jobs, it chooses the split points such that each blocking operator always begins a map or a reduce. This does not mean, however, that each map or reduce begins with a blocking operator as Pig may choose additional split points.

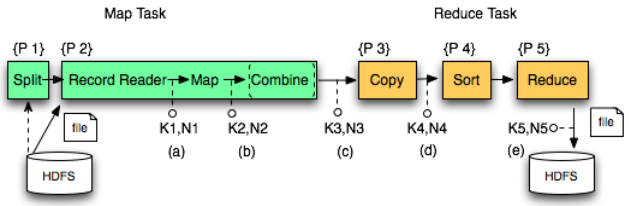

Figure 2: MapReduce DAG & Estimator Architecture

In choosing how to construct our pipelines, the specific MapReduce implementation details of Hadoop and Pig had to be taken into consideration. Between separate MapReduce jobs, Hadoop materializes the output of the last job to the distributed file system (DFS). Thus, each job is its own blocking pipeline. Within each job, we identified five distinct pipelines. The first pipeline contains only the split phase, which we ignore. The second pipeline consists of the entire map operation and thus contains the record reader, map runner, and combine phases. The third, fourth, and fifth pipelines contain only the copy, sort, and reducer phases, respectively, because each of these phases waits until the previous phase has completed before beginning. It is important to note that, although copy immediately operates on each input it receives, it is still a blocking operator because it does not receive any input from a map task until that task has completed. The different pipelines are illustrated in Figure 2.

3.2 Existing Pig/Hadoop Progress Estimates

The existing Pig/Hadoop query progress estimator provides poor estimation (see Section 5). 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 in from the assigned data partition. 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, reducer progress is computed as the percentage of bytes that have been read in so far. The progress of a MapReduce job is computed as the normalized unweighted sum of the percent complete of these three phases. The progress of a Pig Latin query is then just the normalized unweighted sum of the percent complete of all of the jobs in the query.

There are a number of problems with this approach. First, an unweighted sum of the progress of different jobs assumes that each job is performing the same amount of work. This, however, is not the case. In a traditional DBMS, operators at different points in the query plan can have widely different cardinalities, especially after filters or joins. This same variance holds for Pig queries as many of the operators are the same. Second, computing the progress of a MapReduce job as the unweighted sum of the progress of these three phases is also inaccurate. While these three phases tend to encapsulate a significant percentage of the total work done, the amount of work performed in the map runner and combine phases is not negligible.

3.3 The GetNext() Model

As described in [2], the GetNext() model uses the total number of GetNext() calls issued throughout the duration of the query's execution over all operators in the execution plan to represent the total work done by the query. Thus the GetNext() model query progress (*gnm*) would be ideally estimated as:

$$gnm = \frac{\sum_i K_i}{\sum_i N_i} \quad (1)$$

Where K_i is the number of tuples that have flowed out of an operator Op_i at any point in the query execution and N_i is the total number of GetNext() calls invoked on that operator across the entire query. To reduce the need to obtain N_i from each operator, the authors proposed the Driver Node Estimator (*dne*) which estimates N_i for all non-driver nodes in a pipeline (where Op_1 is the driver node of the pipeline) as:

$$N_i = \frac{N_1}{K_1} K_i \quad (2)$$

There are two main assumptions underlying the GetNext() model. The first assumption is that all of the un-

derlying CPU and I/O work for an operator can be encapsulated and amortized across all of the GetNext() calls for that operator. The second assumption is that the average amount of work performed by each call to GetNext() is approximately equal across operators. These assumptions are shown to be valid in many single-node SQL queries in [2], although additional consideration is required to handle spills of tuples to disk.

To apply the GetNext() model to Pig and Hadoop, we had to include estimates of the additional non-operator-based phases, such as copy. We chose to ignore the sort phase in our progress computation as our tests showed that the time spent in the sort phase is negligible, taking around 5 milliseconds. This is because map tasks sort their output before passing it to a reduce task (see 4). Figure 2 shows the locations in the MapReduce pipeline at which we retrieve the K values, i.e. (a) through (e). Thus, the phases that we use to estimate the progress of a MapReduce job are the record reader, map runner, combine, copy, and reducer phases. The GetNext() model inherently provides weights for the different phases and jobs. This allows us to just sum the K and N values across all jobs and phases to obtain a weighted progress estimator.

Data compression is commonly used in distributed systems because network bandwidth generally creates one of the main bottlenecks in the system. For this reason, Pig and Hadoop allow users to process compressed files and to compress intermediate results when transferring data between nodes. There is no correlation for compression within the GetNext() model so our queries were performed on uncompressed data with intermediate compression disabled.

3.4 Estimating Time-Remaining

The time-remaining estimator of [5] also uses cardinality estimates and K counts to compute the percentage of work remaining to perform. The main difference is that the progress of a pipeline is estimated as the sum of the input and output tuples processed divided by the sum of the total number of tuples that the input and output nodes will process. This results in the tuples output by intermediate phases being double-counted. [5] reasons that this will account for any materialization or buffering of the data that may be performed. In addition to this double-counting, the current processing rate (in bytes/second) is estimated from a ten second moving window. The time remaining is computed by multiplying this processing rate by the amount of work remaining to be performed. Unlike gnm and dne, which assume that work done per tuple is the same across all operators in the query, the time remaining estimator assumes that all future segments process tuples at the same speed. This is not a valid assumption in MapReduce or other parallel environments, as discussed in Section 5. Follow-up work in [4], however, uses the query optimizer's CPU and I/O cost estimates to improve the rate estimates. Unfortunately, Pig does not provide optimizer estimates; furthermore, multiple operators can be combined into a single map or reduce, and queries can contain user-defined functions with no estimates available a priori. Our use of alpha weights is based on the time-remaining approach but avoids the limitations of cost estimation, as discussed in (Section 4).

Our implementation of this estimator is true to the [5] paper, with the exception that we compute the processing rate in terms of tuples per second. Time remaining is computed as the product of the current throughput and the number of tuples that remain to be processed.

3.5 Cardinality Estimates

In traditional DBMSs, complex histograms and statistics are maintained about each relation and its component fields. These statistics are used to estimate the cardinalities of the operators in the query plan. In Pig and Hadoop, no statistics are available. Such statistics would not make much sense in Hadoop as users write their own map and reduce functions which can perform drastically different functions on the same data set. However, because Pig generates a query plan from a set of common operators and the same data set is often used for many queries, it might soon make sense to compute these statistics to aid query plan optimization. Statistic-driven cardinality estimation is a field of study orthogonal to our focus on progress estimation. To obtain our estimates for the different N_i values, we recorded each N_i from a previous run and used the actual values in our computations. Thus, our estimators demonstrate the ideal case where the N_i estimates are accurate.

4. KAMD PROGRESS ESTIMATOR

In this section we present the KAMD progress estimator for MapReduce pipelines. KAMD generates an estimate of the amount of time remaining. In doing so, it uses an estimate of the time required to process a single record for each phase of each job and an estimate of the number of records that remain to be processed.

4.1 Alpha Weights

To compute the time remaining in a query, KAMD uses an estimate of the time that it takes for each phase of each job of the query to process a single tuple. We call this value the alpha weight of the phase. In KAMD, alpha weights are estimated by performing the same query on a small subset of the data, which we justify in Section 5.1. Determining the sample size and how to generate it is a difficult problem and there is a lot of research in this area. However, sampling is not a focus of this paper so we generally assumed that random sampling was sufficient. Section 5.1 discusses the accuracy of our sampling methods.

In addition to obtaining alpha weights from a sample run, the alpha values for an actively executing phase are estimated from the elapsed duration of the phase and the number of records that it has processed so far. To ensure stability, these measurements are incorporated into the KAMD estimate only after three seconds have elapsed for a given phase. These online alpha weight estimates are used to compute the slowdown factor as discussed in Section 4.3.

Estimating the duration of each phase is a nuanced problem. In our initial implementation, we estimated the duration of a phase as the total amount of time between when it received its first record to process and when it output its final record. However, this method leads to extremely inaccurate and variable alpha values due to the high amount of coupling between phases, especially in a pipeline. For example, the map runner phase operates on each record that the record reader produces before the record reader produces another tuple (as the relational DBMS iterator model would as well). These two phases exist concurrently, but are only active in alternating cycles of reading each record

and then mapping each record. Estimating active duration for each phase scales in a very complex way as the number of records to process increases and simply estimating alpha values for each phase becomes highly inaccurate. Thus, duration has to be measured as only the amount of time that is spent operating on a record by a phase. In addition to this bookkeeping, when a phase has to wait for another phase to complete this waiting time must be tracked separately (see Section 4.5).

An important consideration for measuring duration involves attempting to measure sub-millisecond phases with timers that only provide single-millisecond resolution. While this may not seem like a major issue, if one million records are processed in a phase and each record takes half of a millisecond, the total duration of the phase is still 500 seconds, which is over 8 minutes. Measuring this duration can be achieved by measuring the time to process multiple tuples. This is difficult when phases are nested (as with record reader and map runner), however, the solution is to subtract the duration of the nested phase from the duration of the outer phase. This must be done incrementally so that the estimator is able to compute alpha values online.

4.2 Estimating Time Remaining

KAMD loads the sampled alpha weights when the full query starts. It then computes the time remaining for each phase as the number of records remaining for that phase to process multiplied by the alpha weight of that phase (α_{ji}, where j is the job and i is the phase) and then sums these times across all jobs and phases. KAMD estimates the remaining work of a phase with the gnm. This will allow KAMD to use dne when cardinality estimates are put into place.

In gnm, dne, and the time-based estimator of [5], progress is mainly computed from the number of tuples processed by the different operators. However, each of these estimators also consider the cost of data materialization. Gnm and dne increment N for each tuple spilled to disk and increment K for each spilled tuple that is read from disk. Alternatively, the time-based estimator uses a double-counting mechanism that is intended to account for data buffering or materialization between operators. To minimize the cost of machine failure, Hadoop materializes all intermediate data to prevent work from having to be redone if a failure were to occur. Additionally, due to the inherently distributed nature of MapReduce, data must be transferred between different machines within the cluster. To account for the amount of work done, our estimator instruments (records K_{ji} and $duration_{ji}$) the record reader, map runner, combine, copy, and reducer phases.

Map and reduce tasks execute in parallel on separate machines within a cluster. KAMD receives instrumentation data that is an aggregate across all of the tasks of a job. This allows KAMD to easily compute the average alpha weight per phase per task by simply dividing the aggregate duration by the aggregate K value. See Section 4.4 for a more in-depth discussion.

4.3 Slowdown Factor

KAMD computes a slowdown factor for each phase,

$$s_{ji} = \alpha^e_{ji}/\alpha^s_{ji}$$

where α^e_{ji} is the online measurement of α_{ji} and α^s_{ji} is the value computed from the dataset sample. The online alpha measurement can only be generated for currently executing or completed phases. However, the same phase across multiple jobs tends to perform similar amounts of I/O and CPU work. Thus, KAMD propagates the most recently estimated slowdown factor for a phase forward to the same phase of jobs that have not yet started.

The slowdown factor provides a mechanism that can be used to counteract systemic calculation errors in the sampled alpha data. These errors can occur if the sample query is performed on a different computer or if there is different load on the system during sampling than when running the actual query.

The slowdown factor does not account for situations where the sampled alphas across phases are inconsistently inaccurate. In this case, the slowdown factor can actually increase estimation error. Unfortunately, it is very difficult to differentiate whether a sampled alpha is inaccurate due to poor sampling, load on the system, or due to actual variations in processing time across jobs. However, if sampling inconsistencies have a normal distribution then they will, on average, cancel each other out and KAMD will provide fairly accurate estimates (see Figure 5).

4.4 Handling Parallelism

Since KAMD measures the aggregate K_{ji} and duration across all tasks, it simply calculates the average alpha weight for a single task. The total amount of remaining processing time for a phase then is the alpha weight multiplied by the number of records remaining to be processed. KAMD currently assumes that tasks that execute in parallel operate on identically-sized partitions of data. Hadoop attempts to partition the data as uniformly as possible among tasks.

Using this assumption, the remaining processing time for a phase just needs to be divided by the pipeline width to estimate the remaining running time of a phase. The degree of parallelism depends on the number of tasks and the number of nodes (known at run-time), and the pipeline width is calculated as the lesser of the two. An accurate estimate of the number of map tasks that will be created for a job can be obtained by dividing the amount of data to be processed at each stage by the system-configured blocksize of Hadoop. Finally, the number of reduce tasks for a job is specified in the Pig Latin script by the `PARALLEL` operator. Thus, the formula used by KAMD to estimate time remaining is as follows.

$$t.r. = \sum_j \sum_i \frac{s_{ji}\alpha^s_{ji}(N_{ji} - K_{ji})}{pipelineWidth_{ji}} \quad (3)$$

4.5 The Combine Phase

The map runner phase writes its output records into a combine buffer. Once the combine buffer reaches a prespecified limit (defaults to 80% full), the combine phase begins operating on that portion of the buffer while the map runner continues writing to the remainder of the buffer. To allow the map runner to write while the combine phase is executing, a separate thread is created to perform the combine. This thread first sorts the data in the buffer, performs the combine (if one is specified) and then spills the combined data to disk. This process leads to a race between the map runner phase and the sort, combine, spill (SCS)

thread. If the map runner fills the remainder of the buffer before the SCS thread finishes operating on the first portion of the buffer then the map runner must block and wait for the SCS thread to complete as there is no more space in the buffer. However, if the SCS thread completes first then the map runner does not have to wait and the SCS thread gets recreated the next time the buffer reaches the specified limit. When the map runner runs out of records to process, there are usually a number of records remaining in the buffer. These must be sorted, combined, and then spilled, like the rest of the records, however, this is now performed in serial with the map runner. We will refer to this serial SCS portion as the non-overlapped SCS. Finally, all of the spill files are merged and written to disk as a single file for each reduce task. Thus, to estimate the total remaining time in a map task we had to calculate the remaining wait time, the non-overlapped SCS time, and the merge and write time. Fortunately, we were able to ignore the merge time as the number of files to merge is generally quite small; these files are already sorted, and the merge typically completes in 15-20 milliseconds.

The wait time for a single buffer within a single map task is calculated by subtracting the time that it would take for the map read and runner phases to fill up the remainder of a single buffer from the time that it would take for the SCS thread to process the buffer. This wait time is then multiplied by the number of times that the buffer limit will be reached to estimate the total amount of wait time that will occur in the map task. The wait time for a single map task is then multiplied by the length of the map task pipeline (calculated as the total number of map tasks for a job divided by the number of nodes that can concurrently execute those map tasks) to obtain the total wait time for a job. The record reader, map runner, combine, and spill phases are all estimated using equation 3. However, the sort phase must be estimated slightly differently due to the non-linear cost of sorting. The time remaining for the sort phase is calculated as equation 3 multiplied by the log of the buffer limit that the SCS thread operates on. KAMD does not estimate the wait time perfectly because progress for the sort phase cannot be accurately computed. The sort operation is opaque, so its duration is not incorporated into the time remaining estimate until the combine phase begins reporting progress. Additionally, it is extremely difficult to estimate the number of records that remain to be sorted, combined, and spilled due to the high level of parallelism in MapReduce and the additional complications caused by the blocking nature of these phases. Instead, KAMD measures the amount of time that all map runners have spent waiting to write and subtracts that from the total estimated wait time.

The non-overlapped time is simpler to estimate.We divide the total number of records for a map task by the buffer limit and take the remainder to obtain an estimate of the number of records that will have to be processed in the non-overlapped SCS. This remainder estimate will always be the last set of records processed by a map task. Once this number is obtained, the previously discussed methods are used to compute time remaining. Progress of the non-overlapped SCS component can be estimated by calculating how many of the final set of records each map task has processed. This is not complicated when we assume that all map tasks process identical sets of data.

Proper estimation of the complicated interactions between all of these phases is necessary for an accurate estimator. This is evaluated in Section 5.3.

4.6 Job and Task Setup and Teardown Times

When a query is run on Pig and Hadoop, there are some system-dependent, constant-cost time penalties that occur. These penalties generally involve administration, setup, and teardown of the processes and threads needed to run MapReduce queries. The most significant three that we measured were: (a) the time from the instantiation of a job until the first work done by a map task, (b) the time from the end of a map task until the reduce task copy phase receives the message containing the map output location, and (c) the time from the end of the final reduce task until the start of the next job. KAMD accounts for (a) by recording the start time of each job and subtracting the amount of elapsed time from that value to estimate how much setup time is left for the first map task. Our system took approximately five seconds for this setup time. KAMD does not account for (b) and (c). The time taken for (b) is on the order of two to three seconds but typically occurs in parallel with other operations and thus has a less significant effect. (c) is also on the order of two to three seconds but is more difficult to instrument, which we take into consideration for future work.

4.7 Adding Compression Estimates

As mentioned in Section 3.3, data compression is commonly used in distributed systems. While our progress estimator does not properly handle compression and decompression, it should be straightforward to add this logic to our estimator. Percent-complete progress estimators for compression and decompression already exist and our estimator could incorporate existing techniques combined with an alpha weight to calculate the amount of time remaining in the operation. The ability to easily incorporate progress estimates that are not based on the GetNext() model is a direct result of working in the time domain.

5. EVALUATION

In this section, we evaluate the KAMD estimator, focusing primarily on how we compare to the other estimators from the literature. Additionally, we present the results of computing alpha (throughput) values for our estimator from different samples, the effects of random and systemic errors on our estimator, and the ability of our estimator to perform well for highly parallel queries.

5.1 Single-Node Alpha Tests

Experiment Since the KAMD estimator computes alpha values (expressed in milliseconds per tuple) online for each phase, the goal of the experiments in this section is to show how well alphas computed on smaller subsets of the full data set compare with alphas computed on the full data set. In this experiment we use a large data set constructed from concatenating five randomized copies of the original, 42 MB 'excite' data set. We run script1-hadoop.pig on the 5X data set once for each of the subset sample alphas used to estimate progress. For each run, the Pig script creates creates 5 MapReduce jobs. The results in Table 1 refer to running the experiment on a single-node cluster. Table 2 shows the results for running the same experiment locally on a single-node machine. The weighted averages of the alpha values

across all phases and jobs is compared against the weighted average of the 'perfect' sample collected from the entire 5X data set. This comparison is expressed as a percent difference. Please refer to the Appendix for the full table showing the comparison on a per phase level. We represent the alphas for each data set as a weighted average because it gives a better overall representation of throughput. The table in the Appendix shows that phases can have high variability in throughput especially for shorter phases like map write.

Table 1: Cluster Alpha Summary

Cluster	50K(1%)	100K(2%)	500K(10%)	1X(20%)	5X(100%)
Duration (s)	75	105	289	390	642
Alpha (Wt. avg)	0.0211	0.0188	0.0197	0.0178	0.0159
Alpha Error (%diff)	32.41%	18.54%	23.66%	11.97%	0.00%

Table 2: Local Alpha Summary

Local	50K(1%)	100K(2%)	500K(10%)	1X(20%)	5X(100%)
Duration (s)	73	92	248	363	766
Alpha (Wt. avg)	0.0179	0.0167	0.0170	0.0175	0.0198
Alpha Error (%diff)	9.49%	15.48%	13.92%	11.46%	0.00%

Overhead It is interesting to note that the weighted average alphas collected from the sample runs on the cluster are more pessimistic (i.e. lower throughput) than the 'perfect' run based on alphas sampled from the full data set. This is perhaps a result of the effect of the cluster overhead, which is a smaller component of the longer duration runs containing the larger sample data sets. Figure 3 demonstrates the effect of these pessimistic alphas on our estimator. Additionally, the percent difference calculations for the alpha error on the cluster are consistently higher than the calculations on the local machine, suggesting that there is more overhead on a cluster than on a local machine. This is not surprising as the cluster includes additional network overhead, and it is something that we will consider in future work to improve our estimator.

Performance Figures 3 and 4 show a relative comparison of the gnm, dne, Luo, original Pig, KAMD 'perfect' (in pink) and KAMD 2% sample alpha (in blue) estimators running script1-hadoop.pig queries on the 'excite' 5X data set on a single-node cluster. The 2% alpha sample was used in these figures (as opposed to the other 1%, 10%, or 20% samples) because it takes less than 16% of the full 5X run time to generate and has a reasonable weighted alpha compared to the 'perfect' alphas from the running the full 5X data set (see Table 1). KAMD with 2% does quite well, though its estimate is slightly pessimistic throughout due to the pessimistic alphas computed on the 2% sample. The error for both KAMD 2% and 'perfect' remains well under 10% across the duration of the run, unlike the other estimators which tend to overestimate their progress. Figure 4 presents the error, which was computed as in [2]:

$$error = \left| \frac{100 * (t_i - t_0)}{(t_n - t_0)} - f_i \right| \quad (4)$$

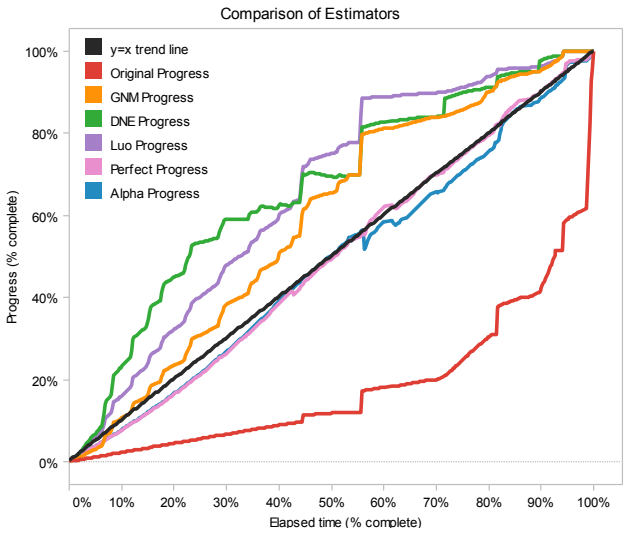

Figure 3: Excite 5X data set, single-node cluster

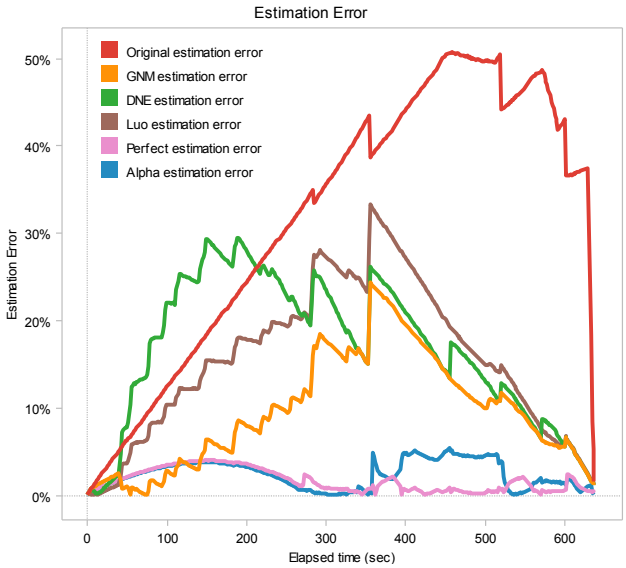

Figure 4: Estimation error on excite 5X data set, single-node cluster

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

Of all the estimators from the literature, gnm performs the best. Gnm gives more accurate estimates than dne because it uses more accurate cardinality estimates for the different pipeline nodes. It outperforms Luo's because Luo's assumes that current progress is indicative of future progress, and this is not a valid assumption in a MapReduce environment — processing rates are inconsistent across jobs and phases (see the Appendix). The original Pig estimator doesn't perform well because it instruments a limited number of phases

and oversimplifies the effects of pipelining. Finally, the discontinuity in each of the estimators near 55% completion (at 360 seconds elapsed time) is notable, and represents the completion of the map tasks and the start of the reduce phases within the first job of each run.

Skew The first of the five jobs that script1-hadoop.pig query generates has 2 maps and 2 reduces; the small result set produced by the first job limits the remaining jobs to only 1 map and 1 reduce each. This leads to some data skew across jobs, where the first job processes significantly more tuples than subsequent jobs. Gnm, dne, and Luo's estimator overestimate progress because they assume that the rate at which the first job processed tuples will apply to subsequent jobs, despite the differences in parallelism.

5.2 Single-Node Alpha Perturbation Tests

In the experiments in this section, we study the effect of inaccurate alphas by inducing two different types of errors: random and systemic. Random errors for each phase may occur due to the fact that we use a small subset of the data as a sample. Additionally, even the 'perfect' run has estimation error due to small variations between identical runs on the same cluster. To study these effects, we add random errors to the measured alphas for each phase. In separate runs we introduce errors in the range +-10% and in the range +-50%. 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 alpha for each phase, using multipliers of 2X and 0.5X. The goal of this experiment is to show the resilience of our estimator in the presence of error, namely the importance of the slowdown factor.

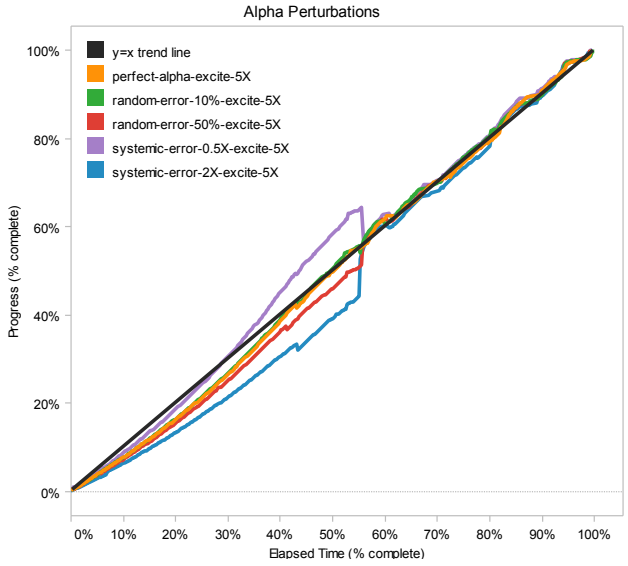

Figure 5: Systemic and random alpha error, excite 5X data set, single-node cluster

Figure 5 demonstrates that when errors are added our estimator continues to perform well. Figure 6 shows that our estimator is off by at most 12% during the worst-case 2X systemic error, where each alpha is off by 2X and the estimator mistakenly underestimates the progress during the

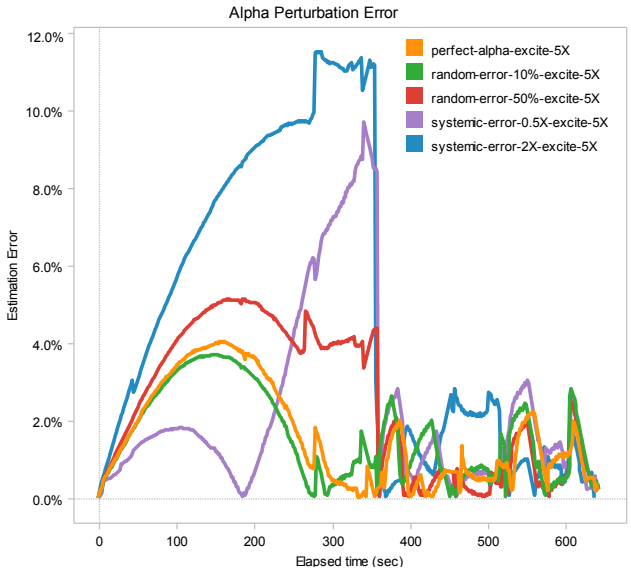

Figure 6: Estimation error on excite 5X data set, single-node cluster

map tasks of the first job. Additionally, in the systemic 0.5X case, our estimator is only off by at most 10%. When the alpha weights are off by significant but systemic amounts, the slowdown factor is able to preserve the accuracy of our estimator. Additionally, the random alpha errors seem to cancel each other out. In the future we will investigate having random errors that are uniformly positive or negative. This will prevent them from canceling but will also hinder the efficacy of the slowdown factor.

5.3 Parallelism and Skew

In this experiment, we run the same script1-hadoop.pig as before on the cluster, but we increase the number of maps and reduces as well as the number of nodes that are utilized. The number of concurrent map tasks was increased by decreasing the Hadoop DFS block size from 128MB to 8MB. The number of concurrent reduce tasks was increased by adding the `PARALLEL` Pig Latin keyword to the Pig script to explicitly use up to eight nodes in the cluster. This keyword was only added to queries in the script1-hadoop.pig file that contained operators responsible for generating reduces such as: `COGROUP`, `CROSS`, `DISTINCT`, `GROUP`, `JOIN` and `ORDER`. The added parallelism spawned 27 maps and 8 reduces in the first job, 8 maps and 8 reduces in jobs 2, 3, and 5, and 8 maps and 1 reduce in job 4.

Figure 7 shows how the KAMD 2% estimator and KAMD 'perfect' compare to dne, gnm, Luo's time-based estimator, and the original Pig estimator for this experiment. All but the original Pig estimator show very optimistic estimates of work completed. It is interesting that the KAMD estimator with the 2% sample (in blue) does the best of all estimators. Both the KAMD 'perfect' (in pink) and 2% estimators show optimistic estimates of completion; however, the 2% estimator is consistently less optimistic relative to 'perfect'. This is because the alpha values that were computed on the 2% sample were less optimistic (i.e. reported more time to

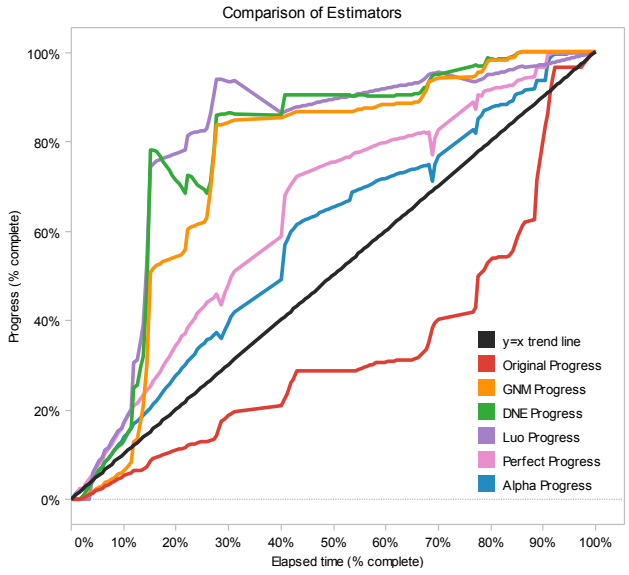

Figure 7: Excite 5X data set, 8-node cluster

process each tuple) than the alphas computed on the perfect estimator's 5X data set. The alphas for 2% are less optimistic in general on the cluster (which is evident in our results in Table 1) because their estimates for time per tuple contain a higher percentage of overhead than the larger samples.

Parallelism skew Since this experiment is a parallel version of the experiment in Section 5.1, the same data skew across jobs exists. Additionally, there are challenges with parallelism skew, in which the first job spawns over three times as many map tasks as subsequent jobs. This parallelism skew amplifies the existing data skew because the same percentage of records are processed but the first job has much greater speedup than subsequent jobs. The discontinuities in Figure 3 now occur much earlier, with the reduce phases in the first job starting at 25% completion as shown in Figure 7. The new discontinuity present at 15% completion represents the start of the combine phase (and its SCS thread), and demonstrates the limitations of progress estimation models that assume uniform time per unit of work. Unfortunately, our estimator is also affected by both parallelism skew and data skew, although to a much lesser degree. We intend to investigate using critical paths in our estimator to better handle all types of data skew.

6. CONCLUSION AND FUTURE WORK

Estimating progress is a complicated problem in both single-node and cluster environments. We applied techniques from the literature and found that they were fundamentally flawed in a cluster environment. For example, dne and gnm assume uniform throughput, and Luo's time-based estimator assumes that recent throughput is a perfect predictor of future throughput across all phases. We have demonstrated that the KAMD estimator provides the most accurate estimates in the MapReduce environment and is capable of handling highly-parallel queries. In future work, we intend to run our experiments on a wider variety of Pig Latin queries as well as on larger data sets such as the 54 GB astronomy data set from the University of Washington NuageDB group. We plan to examine the use of critical paths in our estimator to compensate for data skew. Additionally we will instrument and account for network and scheduling overhead in cluster environments. Finally, we will continue to explore robustness in handling random errors, including errors which are random but uniformly biased towards positive or negative shifts.

7. REFERENCES

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

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

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

[4] 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.

[5] 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.

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

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

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

[9] 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.