Estimating the Progress of MapReduce Pipelines

Kristi Morton, Abram Friesen, Magdalena Balazinska, Dan Grossman
Computer Science and Engineering Department, University of Washington
Seattle, Washington, USA
{kmorton,afriesen,magda,djg}@cs.washington.edu

Abstract— **In parallel query-processing environments, *accurate*, *time-oriented* progress indicators could provide much utility given that inter- and intra-query execution times can have high variance. However, none of the techniques used by existing tools or available in the literature provide non-trivial progress estimation for parallel queries. In this paper, we introduce Parallax, the first such indicator. While several parallel data processing systems exist, the work in this paper targets environments where queries consist of a series of MapReduce jobs. Parallax builds on recently-developed techniques for estimating the progress of single-site SQL queries, but focuses on the challenges related to parallelism and variable execution speeds. We have implemented our estimator in the Pig system and demonstrate its performance through experiments with the PigMix benchmark and other queries running in a real, small-scale cluster.**

I. Introduction

Over the last several years, our ability to generate and archive extremely large datasets has dramatically increased. Modern scientific applications – for example, those driven by widely-distributed sensor networks or simulation-based experiments – are producing data at an astronomical scale and rate (*e.g.*, [11]). Similarly, companies are increasingly storing and mining massive-scale datasets collected from their infrastructures and services (*e.g.*, eBay, Google, Microsoft, Yahoo!).

To analyze these massive-scale datasets, users are turning to parallel database management systems (*e.g.*, [7], [22], [23]) and other parallel data processing infrastructures (*e.g.*, [4], [8], [10]). Although these systems significantly speed-up query processing, individual queries can still take minutes or even hours to run due to the sheer size of the input data.

When queries take a long time to complete, users need accurate feedback about query execution status, in particular how long their queries are expected to run [17]. Resource allocation algorithms can also benefit from access to such information. Unfortunately, existing parallel systems provide only limited feedback about query progress. Most systems simply report statistics about query execution [3], [5], [6], at best indicating which operators are currently running [5], [6], [8] and possibly mapping this information onto a percent-complete value [20]. Such indicators, however, are too coarse-grained and inaccurate because different operators can take wildly different amounts of time to run.

In this paper, we address this limitation by introducing *Parallax, the first, non-trivial time-oriented progress indicator for parallel queries*. We developed our approach for Pig queries [19] running in a Hadoop cluster [8], an environment that is a popular open-source parallel data-processing engine. As an initial step, we focused on Pig queries that compile into a *series* of MapReduce [4] jobs. Hence, our current indicator does not handle joins. Furthermore, in this paper, we do not consider failures, backup tasks [4], server heterogeneity, nor the presence of competing workloads. 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 notably, user-defined functions (UDFs) are the norm, all intermediate results are materialized, and each scheduled query fragment incurs a significant start-up cost.

Parallax is designed to be accurate while remaining simple and addressing the above Pig/Hadoop-specific challenges. At a high level, Parallax is based on the following key ideas. First, as in prior work on single-site query progress estimation [1], [13], Parallax breaks a query into pipelines, which are groups of interconnected operators that execute simultaneously. Parallax estimates time remaining for a query by summing the expected times remaining across all pipelines. Second, to compute time-remaining for a pipeline, Parallax estimates the number of tuples left to process by the pipeline, but also the *time-varying* degree of parallelism that the pipeline will exhibit. Finally, Parallax uses prior runs of the same query on a user-generated representative data sample (*e.g.*, a debug run), to estimate the relative processing speeds of different pipelines (including data materialization and network transfer steps).

Parallax is fully implemented in the Pig system. Experimental results on an 8-node cluster are promising. They show that, for a large class of queries, Parallax's average accuracy is within 10% and often even within 5% of an ideal indicator (in the absence of cardinality estimation errors).

II. Background and Related Work

MapReduce [4] (with its open-source variant Hadoop [8]) is a programming model and an implementation for processing massive-scale datasets. In MapReduce, a computation, or *job* is expressed as a sequence of two operators: map and reduce. MapReduce jobs are automatically parallelized and executed on a cluster of commodity machines: the map stage

Acknowledgements. We would like to thank Robert Morton, YongChul Kwon, Alan Gates, and Brian Cooper for feedback and advice. The Parallax project is partially supported by NSF CAREER award IIS-0845397, NSF CRI grant CNS-0454425, gifts from Yahoo! and Microsoft Research, and Balazinska's Microsoft Research Faculty Fellowship. K. Morton is supported in part by an AT&T Labs Fellowship.

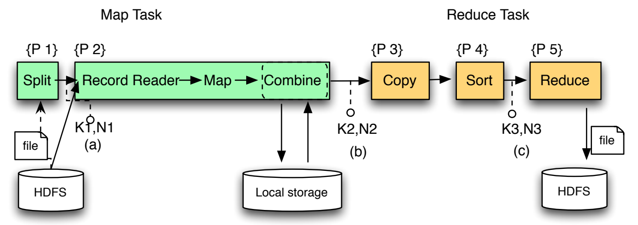


Fig. 1. The operations of a MapReduce Job. Each N_i indicates the cardinality of the data on the given link. K_i's indicate the number of tuples seen so far on that link. Each counter marks the beginning of a new pipeline (Section III).

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 as well as the final result are always materialized.

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 [19]. In Pig, queries are written in Pig Latin, a language that combines the high-level declarative style of SQL with the low-level procedural programming model of MapReduce. Pig compiles these queries into *ensembles* of MapReduce jobs and submits them to a MapReduce cluster.

Pig/Hadoop's existing progress estimator [20] has low accuracy (see Section IV) because it assumes all operators process data at the same speed. Several relational DBMSs, including parallel DBMSs, provide similarly coarse-grained progress indicators. In fact, most systems simply maintain and display a variety of statistics about (ongoing) query execution [3], [5], [6] possibly also indicating which operators are running [5], [6].

There has been significant recent work on developing progress indicators for SQL queries executing within single-node DBMSs [1], [2], [12], [13], [15], [16], possibly with concurrent workloads [14]. Our approach extends these earlier efforts to parallel queries.

Work on online aggregation [9] also strives to provide continuous query execution feedback to users. This 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.

III. The Parallax Progress Estimator

While our end goal is robust progress estimation for all Pig Latin scripts, in this paper, we consider only scripts that translate into MapReduce *sequences*.

a) Estimating Time-Remaining: As in prior work for single-site SQL query progress estimation [1], [13], Parallax breaks queries into pipelines, which are groups of interconnected operators that execute simultaneously. For a given MapReduce job, we identify five pipelines as illustrated in Figure 1: (1) the split operation, (2)the record reader, map runner, and combiner operations, (3) the copy, (4) the sort, and (5) the reducer. Parallax ignores the split and sort pipelines because they take a negligible amount of time (the sort operation only merges data previously sorted by the combiner). Parallax thus assumes three pipelines per job.

Given a sequence of pipelines, Parallax estimates their time remaining as the sum of time remaining for the currently executing and future pipelines. In both cases, the time remaining for a pipeline is the product of the amount of work that the pipeline must still perform and the speed at which that work will be done. We define 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, we must estimate N_p and α_p for each future pipeline.

b) Estimating Execution Speeds and Work Remaining: An important contribution and innovation of Parallax is its estimation of pipeline processing speeds. Previous techniques either ignore these speeds [1], [2], assume constant speeds [13], or combine measured speed with optimizer cost estimates to better weight different pipelines [12]. In contrast, to estimate the execution speed of each pipeline, Parallax observes the current speed for pipelines that already started and uses information from earlier debug runs for upcoming pipelines. This approach is especially well-suited for query plans with user-defined functions. Debug runs can be done on small samples of the dataset 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. More details about the slowdown factor can be found on our project website [18].

For cardinality estimates, N_p, Parallax relies on standard techniques from the query optimization literature. That is, for pre-defined 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. In this paper, however, we do not address the cardinality estimation problem and assume perfect cardinalities to study the other factors that affect progress estimates.

c) Accounting for Dynamically Changing Parallelism: The second key contribution of Parallax is its novel handling of query parallelism, which has not been addressed by other estimators.

When a query executes at large scale, Map and Reduce functions are parallelized across many nodes. 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 first 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 execution of a pipeline.

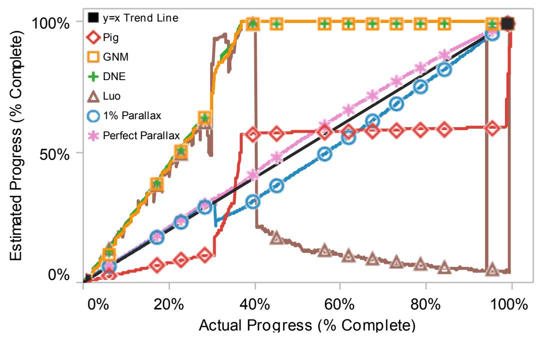
IV. Evaluation

We evaluate the Parallax estimator and compare it to other estimators from the literature. All experiments were run on an eight-node cluster configured with the Hadoop-17 release and Pig Latin trunk from 02/12/2009. Each node contains a 2.00GHz dual quad-core Intel Xeon CPU with 16 GB of RAM. The cluster was configured to a maximum degree of parallelism of 16 map tasks and 16 reduce tasks.

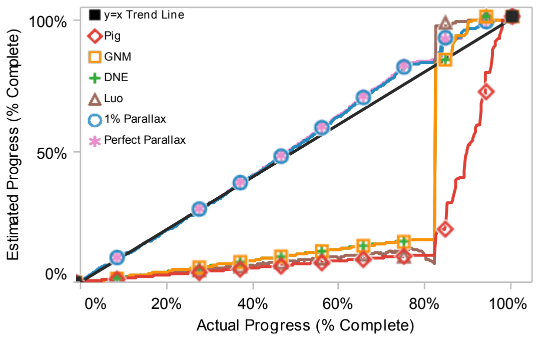
In all experiments, we show results for perfect cardinality estimates as we want to emphasize the impact of other sources of errors. Parallax is demonstrated in two forms: *Perfect* Parallax, which uses α values from a prior run over the entire dataset; and *1%* Parallax which uses α collected from a prior run over a 1% sampled subset (other sample sizes yielded similar results).

To demonstrate the importance of α weights and also as a motivation for Parallax, we first execute script1 from the Pig tutorial, on our small 8-node cluster and a 210MB dataset. Script1 contains fourteen unique Pig Latin statements, five UDFs, and translates into a sequence of five MapReduce jobs.

We first execute the script in series (*i.e.*, with exactly one task for each map and reduce pipeline in each job). To emphasize the effects of processing speed differences, we add a time-consuming UDF to the third job. The entire query takes 30 minutes to run. Figure 2(a) shows the results (the



(a)



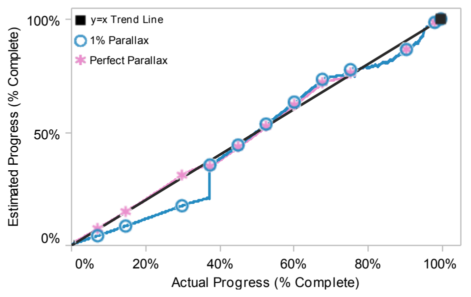
(b)

Fig. 2. Pig script1-hadoop, 210MB input data, (a) serial execution with extra user-defined function and (b) parallel execution of original script.

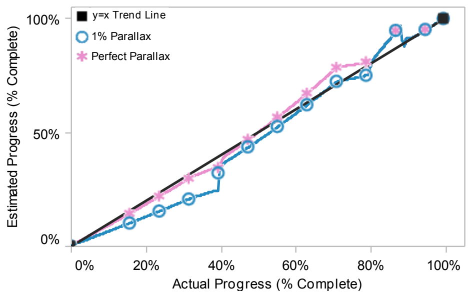
y=x trend line corresponds to a perfect progress estimator). As expected, the estimators from the literature (GNM [1], DNE [1], and Luo [13]), which either ignore processing speeds or assume constant speeds, do not perform well when different operators process data at different rates. In contrast, Parallax provides estimates within 4% of perfect. Additional α-weight experiments (including those on the original script) are available on our project website [18].

In a second experiment, we run the original (unmodified) script1 but, this time, we allow two of the pipelines (the copy and reduce pipelines of the first job) to execute with a degree of parallelism of 16 while the rest of the query executes in series due to the small data size. The query takes 13 minutes to run. Figure 2(b) shows that, as expected, estimators from the literature cannot directly be applied to parallel queries.

In the next experiment, we further study how Parallax handles simple, parallel queries. We execute a LOAD-GROUPBY-STORE script that translates into a single MapReduce job, with both Map and Reduce tasks. This is thus a 3-pipeline query. Uniform data distribution ensures that all map and reduce tasks in the same round end at approximately the same time. Figure 3(a) shows the result of running the query on an 8GB-size input dataset with 32 map tasks and 32 reduce tasks. Given that our cluster can execute 16 tasks in parallel, the script runs with two rounds of map tasks followed by two rounds of reduce tasks. Figure 3(b) shows the result for the case of dynamically changing degree of parallelism. In this experiment, the query runs on a 4.2GB dataset and executes with 17 maps and 17 reduces. Each experiment runs for approximately 28 minutes. The overall results are



(a)



(b)

Fig. 3. Progress estimation for a query comprising two rounds of map tasks followed by two rounds of reduce tasks. (a) All four round comprise 16 tasks each. Uniformly distributed 8GB dataset; (b) First round of maps and first round of reduces comprise 16 tasks each. Second rounds have only one task each. Uniformly distributed 4.2GB dataset.

extremely encouraging, with average error values remaining below 4.5% for all experiments. Figures 3(a) and (b) show that *1%* Parallax is pessimistic during the first round of map tasks. The pessimism is caused by a higher-cost α value computed on the sample run for the copy pipeline compared with the actual value measured during query execution. The α values were off by more than an order of magnitude, but thanks to the slowdown factor, Parallax recovers once the copy pipeline starts processing tuples, which happens right after the first round of maps completes. The corrected α values propagate to future pipelines. Overall, these results demonstrate that Parallax can accurately predict the runtime for parallel queries with dynamically variable parallelism (assuming no significant data skew and accurate cardinality estimates). Other configurations yield similar results [18].

We also evaluate Parallax on a subset of the *latency* queries from the PigMix [21] benchmark. The queries that we use translate into sequences of one to three MapReduce jobs (*i.e.*, we exclude queries that contain joins). For our cluster, these jobs comprise 60 to 119 maps and 16 reduces each and run for three to ten minutes. We use a 15GB dataset as input. Depending on the attribute, data distribution is either Zipfian or uniform. Table I summarizes the results.

Parallax outperforms the other estimators both in terms of average and maximum errors on all queries. Average estimation errors remain below 6.0% for *Perfect* Parallax and below 7.6% for *1%* Parallax. Maximum errors remain below 15% for all queries. Overall Parallax performs well relative to the other estimators, especially on the more complex queries such as L9 and L10, which consist of three jobs each. Parallax produces better estimates for these queries because it computes

TABLE I
ESTIMATION ERRORS PIGMIX LATENCY BENCHMARKS, 15GB DATASET

	Perfect Parallax % Error		1% Parallax % Error		dne % Error		gnm % Error		Luo % Error		Pig % Error	
Query	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
L1	2.9	7.1	2.5	6.4	3.1	9.0	3.1	9.0	6.7	11.6	14.0	30.2
L4	4.7	10.6	4.0	9.6	5.2	15.4	5.5	15.6	5.0	16.7	12.2	27.8
L6	4.6	10.7	3.9	9.7	7.1	18.9	7.6	18.9	6.1	19.7	10.6	23.4
L7	2.4	6.8	1.0	3.2	6.8	21.9	6.9	22.0	7.8	22.1	11.8	25.8
L8	6.0	13.0	5.6	12.1	7.8	14.3	7.8	14.3	8.8	14.8	12.0	26.3
L9	5.6	14.5	7.6	15.0	16.5	36.0	16.5	36.0	19.3	42.7	14.3	27.4
L10	3.2	7.1	5.6	13.4	15.9	37.7	15.9	37.7	19.1	45.2	12.8	21.9

a finer-grained estimate of processing speeds and concurrency.

V. CONCLUSION

We presented Parallax, the first, non-trivial time-based progress indicator for Pig Latin scripts that translate into a series of MapReduce jobs. Parallax handles varying processing speeds and degrees of parallelism during query execution. Parallax is fully implemented in Pig and outperforms existing alternatives on representative workloads. Additional information about Parallax including extra experimental results are available on the Nuage project website [18].

REFERENCES

[1] S. Chaudhuri, V. Narassaya, and R. Ramamurthy. Estimating progress of execution for SQL queries. In *Proc. of the SIGMOD Conf.*, Jun 2004.
[2] Chaudhuri et. al. When can we trust progress estimators for SQL queries. In *Proc. of the SIGMOD Conf.*, Jun 2005.
[3] DB2. SQL/monitoring facility. `http://www.sprdb2.com/SQLMFVSE.PDF`, 2000.
[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.
[5] M. Dempsey. Monitoring active queries with Teradata Manager 5.0. `http://www.teradataforum.com/attachments/a030318c.doc`, 2001.
[6] Greenplum. Database performance monitor datasheet (Greenplum Database 3.2.1). `http://www.greenplum.com/pdf/Greenplum-Performance-Monitor.pdf`.
[7] Greenplum database. `http://www.greenplum.com/`.
[8] Hadoop. `http://hadoop.apache.org/`.
[9] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. of the SIGMOD Conf.*, 1997.
[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of EuroSys*, pages 59–72, 2007.
[11] Large Synoptic Survey Telescope. `http://www.lsst.org/`.
[12] 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.
[13] 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.
[14] G. Luo, J. F. Naughton, and P. S. Yu. Multi-query SQL progress indicators. In *Proc. of the 10th EDBT Conf.*, 2006.
[15] C. Mishra and N. Koudas. A lightweight online framework for query progress indicators. In *Proc. of the 23rd ICDE Conf.*, 2007.
[16] C. Mishra and M. Volkovs. ConEx: A system for monitoring queries (demonstration). In *Proc. of the SIGMOD Conf.*, Jun 2007.
[17] B. A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *Proc. of CHI'85*, pages 11–17, 1985.
[18] Nuage project. `http://nuage.cs.washington.edu/`.
[19] 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.
[20] Pig Progress Indicator. `http://hadoop.apache.org/pig/`.
[21] PigMix Benchmarks. `http://wiki.apache.org/pig/PigMix`.
[22] Teradata. `http://www.teradata.com/`.
[23] Vertica, inc. `http://www.vertica.com/`.