The Case for Being Lazy: How to Leverage Lazy Evaluation in MapReduce

Kristi Morton, Magdalena Balazinska, and Dan Grossman
Department of Computer Science & Engineering, University of Washington
kmorton,magda,djg@cs.washington.edu

Christopher Olston
Yahoo! Research
olston@yahoo-inc.com

ABSTRACT

In this paper, we study the benefits and overheads of lazy MapReduce processing, where the input data is partitioned and only the smallest subset of these partitions are processed to meet a user's need at any time. We also develop guidelines for successfully applying the lazy MapReduce computation technique to reduce processing times of analysis tasks.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Parallel databases, Query processing*

General Terms

Design, Performance

Keywords

MapReduce, eScience

1. INTRODUCTION

As their dataset sizes explode [2], scientists are exploring new tools for analyzing their data. MapReduce [7], a flexible and highly-scalable parallel data-processing system, is a popular choice today. Many scientists are starting to apply MapReduce to their analysis tasks [18, 17, 26, 30].

Because of the volume of data, a single MapReduce job can easily take from minutes to hours to run. For example, a data clustering algorithm executed on an 18GB dataset took 1.6 hours to run on an 8-node cluster [17]. Additionally, scientists often need to perform an entire workflow of transformations in order to process their data [1]: clean the data, extract and transform features, match results against a reference database, etc. Each transformation maps onto one or more MapReduce jobs. Processing times are thus a *bottleneck to scientific discovery* as scientists must wait significant amounts of time whenever they want to answer a question using their data. Such high latencies are especially frustrating when a scientist needs only a subset of the analysis result. For example, given the output of an astronomy simulation in the form of a series of snapshots [19, 27], an astronomer is typically only interested in a subset of these snapshots. It would be wasteful to run the 1.6-hour data clustering algorithm on hundreds of snapshots in a simulation if the user will access only a few of the snapshots.

Our vision is to optimize scientific workflows by *carefully deciding when to process different subsets of the data at any step in a workflow.* In this paper, we study the problem at the level of a single MapReduce job. Our observation is that given a dataset and a transformation, a scientist often has the choice to apply that transformation either to the entire dataset or only to a subset of that dataset. For example, an astronomer can choose to run the clustering algorithm ahead of time on all simulation snapshots or he can choose to cluster individual snapshots as the need arises. In general, deciding what subsets of the input data to a MapReduce computation to process is not trivial for the following reason: If a user will access some small subset of the analysis result, it is faster to compute only that subset rather than the entire result. However, if the user processes a subset of the data and later realizes that he needs additional results, there is a penalty to obtaining such results compared with upfront processing of the entire dataset. For example, if the astronomer will end up studying the evolution of celestial structures over most simulation snapshots, it may be faster to cluster all snapshots of the simulation in a single MapReduce job rather than running a separate job for each snapshot.

We call the delayed computation of MapReduce results *lazy MapReduce processing.* For a given MapReduce job, the key idea is to partition the input data into small fragments and process only the smallest set of fragments to satisfy a user's needs at any given time. To use this technique effectively, a user must (1) decide whether lazy processing is both possible and cost-effective for her analysis task and dataset and, if so, (2) at what granularity to apply this technique.

We present preliminary work that addresses both issues. First, we study the potential gains and overheads of lazy MapReduce processing (Section 4). Second, we develop guidelines to help a user decide whether lazy MapReduce processing can be cost effective and at what data granularity to apply this technique (Section 5). We begin with an overview of MapReduce in Section 2 and discuss an illustrative example of lazy MapReduce processing in Section 3.

2. BACKGROUND

MapReduce [7] is Google's parallel data-processing sys-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ScienceCloud'11, June 8, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0699-7/11/06 ...$10.00.

tem. It facilitates large-scale data analytics by providing a simple programming interface: the user writes only a serial map and a serial reduce function to analyze data and the MapReduce implementation takes care of efficiently executing these functions in parallel across nodes in a cluster. Hadoop [12] is an open-source version of MapReduce.

Several high-level languages exist for MapReduce including Pig Latin [23] and HiveQL [14]. Such languages enable users to express their analysis tasks in the form of simple scripts or declarative queries. These queries are compiled into directed acyclic graphs (DAGs) of MapReduce jobs. In this paper, however, we focus on single MapReduce jobs.

We use as illustrative examples and in the evaluation real data transformation tasks from the data analysis workflows of the University of Washington's Astrophysics "NBody Shop" group [19]. The tasks that we consider consist of a single job each, running many map and reduce tasks in parallel (over 450 tasks in our experiments). The types of operations performed by these jobs include selections/filters on items of interest, joins, and the application of user-defined functions (UDFs) for more complex analysis.

3. ILLUSTRATIVE EXAMPLE

We present an illustrative example of lazy MapReduce processing, how it can help improve performance, and how it can hurt performance if applied indiscriminately.

In astronomy, cosmological simulations are used to study how structures form and evolve in the universe on scales ranging from a few million light-years to several billion light-years [19, 27]. Typically, the simulations start shortly after the Big Bang and run the full lifespan of the universe, roughly 14 billion years. Every few simulation timesteps, the simulator outputs a snapshot of the universe in the form of a set of particles, their position, and their properties (mass, velocity, etc.). The particles in each snapshot are then clustered to identify celestial structures such as galaxies. Given the raw particle data and cluster information, astronomers often want to ask various questions on the data and these questions can be expressed as Pig Latin scripts [19].

One such question (Query 3 from Loebman *et al.* [19]) is the following: "Return all particles of type T (gas, stars, or dark matter) within distance R of point P_1 whose property X is above a threshold D computed at timestep S." Given this question, the user can proceed in one of three ways:

- *Standard approach*: Run Query 3 as specified on the entire input dataset.
- *Eager approach*: Extract all particles of type T whose property X is above a threshold D at timestep S. Do not apply the predicate on distance to P_1 because particles near other points may also be of interest later.
- *Lazy approach*: Partition the input data into small spatial regions and run the above Query 3 only on the regions that contain P_1 and particles within R of P_1. Do not apply the predicate on distance to P_1 in case other points within the processed region become of interest.

The standard approach extended with caching is similar to the lazy approach except for two differences. First (assuming the input dataset is a file with no index), the lazy approach consumes a subset of the input data rather than the entire dataset to compute the result. Second, the lazy approach explicitly defines the granularity with which data is processed. Additionally, both lazy and eager output extra results that the user needs to filter out until they are needed. In this paper, we study only the lazy and eager techniques.

When to be eager vs lazy? Lazy processing can be applied to select-project-join queries[1] and queries containing user-defined functions (UDFs) that process tuples independently of one another. For queries with operations that consume the entire dataset to compute their results such as aggregations (*e.g.*, AVG, SUM, COUNT), the lazy approach cannot be applied. Query 3, for example, could benefit from lazy processing assuming that (1) the initial dataset is large, (2) the predicate on distance is costly, and (3) the scientist will end-up studying only a small number of points P_i. Indeed, in MapReduce, processing a dataset incrementally is slower than processing the entire dataset in one shot. Thus, if the user will end-up consuming the output from processing most of the input data or the overall MapReduce job is short, the lazy approach will only add overhead.

Goal and assumptions. We apply the lazy MapReduce computation as a way to accelerate an individual query. We do not consider the alternate optimization goal of optimizing the total resource utilization of the cluster.

We assume a basic MapReduce cluster where the input data is stored in a file-system such as the Google File System (GFS) [9] or equivalent Hadoop Distributed File System (HDFS) [4]. GFS and HDFS take each input data file and break it into large chunks on the order of 64 MB or larger. The files are not indexed.

4. LAZY MAPREDUCE PROCESSING

In this section, we evaluate the potential benefits and overhead of the lazy approach on a 52GB, uncompressed astrophysical dataset and query workload from the University of Washington's "N-body Shop" group. All experiments were run on an eight-node MapReduce cluster with Yahoo!'s Hadoop 0.20.104 release and Pig version 0.6.0. Each node contains eight, 2.66GHz Intel Xenon processors and 16GB of RAM. The cluster's HDFS block size was configured with 128MB and has 32 designated parallel slots for map tasks and 32 parallel slots for reduce tasks.

As mentioned in the previous section, with the lazy technique we partition the input dataset into smaller fragments and run the query only on the fragment of interest. In our experiments, we partition the dataset on subregions of space (x, y, z coordinates). We measure the time to run a query on either the entire dataset or on one partition of the dataset at a time. We experiment with partitions that range in size from $\frac{1}{2}$ of the data to $\frac{1}{32}$ of the data.

The workload we evaluate is the Pig Latin script in Figure 1, which is a variant of Query 3 and has a high-cost UDF function, `VirialTemp`. Figure 2 shows the results for running the above query in three different scenarios: first with the original UDF (per tuple processing cost of 477 μs to 507 μs), second with a less expensive version of the same UDF (cost of 232 μs to 293 μs), and finally without the UDF (cost of 140 μs to 335 μs). The figure shows the results of running the query in these three configurations and on different size data partitions. Each bar in the bar graph consists of dif-

[1]For joins, one input relation can be processed lazily while the other one must be processed in its entirety to yield correct results unless both datasets are partitioned on the join attribute.

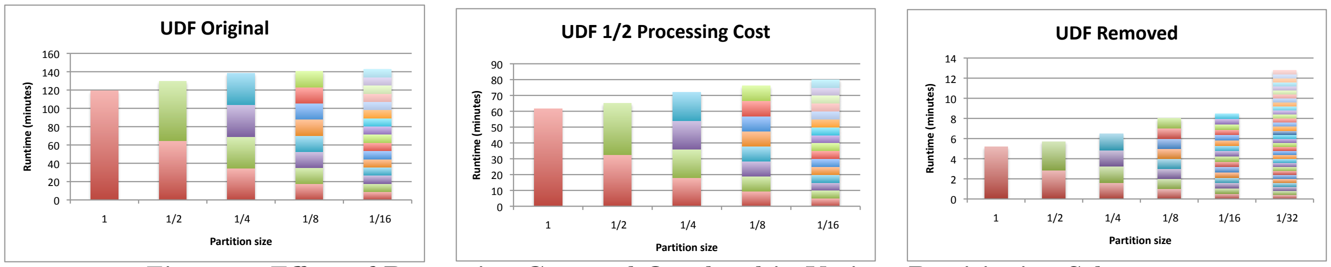

Figure 2: Effect of Processing Cost and Overhead in Various Partitioning Schemes

```
REGISTER udfs.jar;
gas43 = LOAD 'gas43full' USING BinStorage() AS (pid:long,
          mass:double,px:double,py:double,pz:double,vx:double,
          vy:double,vz:double, phi:double,rho:double,temp:double,
          hsmooth:double,metals:double);
-remove any records with null values
gas43 = FILTER gas43 BY pid is not null AND mass is not null
          AND px is not null AND py is not null AND pz is not null
          AND vx is not null AND vy is not null AND vz is not null
          AND phi is not null AND rho is not null
          AND temp is not null AND hsmooth is not null
          AND metals is not null;
-classification step:
-search a particular region of the space for particles
-whose temperature is above some threshold
regionA = FILTER gas43 BY temp > udfs.VirialTemp(Rvir,Mvir,pid)
            AND px >= -0.5 AND px < -0.25 AND py >= 0 AND
            py < 0.5 AND pz >= -0.5 AND pz < 0;
STORE regionA INTO 'gas43result' USING BinStorage();
```

Figure 1: Pig Latin Script

ferent color boxes, which represent the individual times for running the query on each partition.

The graphs show two main trends. First, we see, as expected, that as the number of partitions increases, so does the effect of partitioning overhead: the time to process the entire dataset increases. Lazy evaluation thus cannot be applied indiscriminately and the user needs to pick the partitioning strategy wisely. Also, the right-most graph in Figure 2 demonstrates that partitioning too small (*e.g.*, into 32nds) is not useful, as the overhead tends to outweigh any benefit. Finally, we see that the benefits of lazy evaluation are most pronounced when the per tuple processing costs are high. The lazy approach is effective because it only scans and applies the expensive UDF on a small amount of data when possible. The eager approach performs a significant amount of extra processing (see the first two graphs in Figure 2) compared to the lazy approach as it applies the expensive UDF operation to a large number of particles that may never be of interest to the user. However, if the user needs to study particles near other points P_i, the data is readily available.

5. LAZY PROCESSING GUIDELINES

The previous section presented the motivation and challenges of lazy MapReduce processing. We now provide a preliminary strategy for applying this technique.

5.1 Estimating the Runtime of a Job

In a cluster with N slots, when a MapReduce job runs, it can execute up to N map and N reduce tasks in parallel, with the reduce tasks following the map tasks. We call such groups of N tasks, *parallel processing units (PPUs)*. The default behavior of MapReduce is that the number of map tasks is automatically determined by the size of the input data, with one map task processing one block of data. Hence, given a MapReduce job, the map phase runs as a series of PPUs. In contrast, the number of reduce tasks is set by the user and the best practice is for the reduce phase to run as a single PPU.

If a file contains less data than a single map PPU can handle, the processing time for the map phase in an N-slot cluster will be the same as that of a full map PPU. Some slots will simply go unused. Hence, *we recommend to partition the input data into files, such that each input file gets processed by some number K of full map PPUs.*[2] This guideline assumes no other jobs are executing in the cluster.

To estimate the processing time of a MapReduce job on a file created as recommended above, we adapt equations from our prior work [22, 21].

$$T_{job\_eager} \approx T_{\text{start}} + KT_{\text{map\_ppu}} + T_{reduce\_ppu} \qquad (1)$$

where T_{start} is the overhead of starting a single MapReduce job over N slots. This overhead is significant and can reach tens of seconds even for medium-size clusters with up to 100 nodes [24]. $T_{\text{map\_ppu}}$ and $T_{\text{reduce\_ppu}}$ are the times to process one round of map tasks and one round of reduce tasks respectively. K is the number of map PPUs in the job. $T_{\text{map\_ppu}} \approx T_{\text{map\_start}} + N_{\text{recs}} * T_{\text{rec}}$, where $T_{\text{map\_start}}$ is the time to start a new round of map tasks, N_{recs} is the number of records in a single file-system chunk (or split) and T_{rec} is the per-record processing time for map tasks. Similarly, $T_{\text{reduce\_ppu}} \approx M_{\text{recs}} * U_{\text{rec}}$. Here, M_{recs} is the number of records assigned to a single reduce task and U_{rec} is the per-record processing time. There is no startup overhead for the reduce PPU as the first round of reduce tasks is always scheduled at the same time as the map tasks. For some MapReduce jobs, skew can occur [17], where some map or reduce tasks take much longer to process than others. In such cases, the time estimates of the skewed phase must be adjusted to take the skew into account [21].

To effectively use lazy MapReduce processing, a user must have an estimate of the above parameters. As in our prior work [22, 21], these values can be obtained from debug runs. Note that we do not try to perfectly model runtimes. We only need to model these parameters and the job runtime sufficiently well to discriminate between different processing strategies in the case where runtime differences vary significantly between approaches.

5.2 Lazy vs Eager Processing

Given a dataset partitioned into files, the question that the user wants to answer is whether to process all files at once as in Equation 1 or lazily process one file at a time as needed. To answer this question, we derive the expected processing time for the lazy approach.

[2]In Hadoop, the user can achieve this goal either by playing with the file sizes or by adjusting the “split size”, which determines how much data is assigned to each map task.

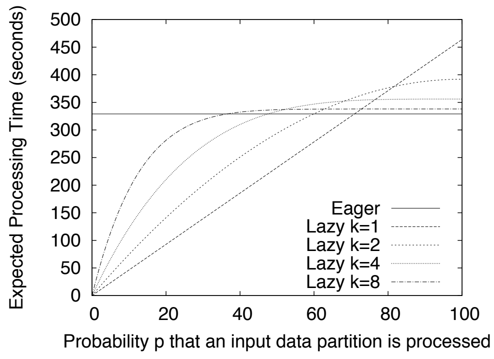

Figure 3: Eager and Lazy evaluation tradeoffs.

We assume that the dataset comprises K partitions such that each partition can be processed by one map PPU. We also assume that partitions are grouped into equal-sized files, each containing k partitions. There are thus $\frac{K}{k}$ files.

We assume that P_i is the probability that partition i from the set of K partitions will have to be processed. For simplicity, we assume the probability is the same for all partitions and equal to P. In that case, the expected processing time for the analysis task using the per-file lazy processing method is the following (other discrete probability distributions could be modeled by replacing P with P_i and expanding the following equation):

$$T_{job\_lazy} \approx \frac{K(1-(1-P)^k)(T_{\text{start}} + kT_{map\_ppu} + f(T_{reduce\_ppu}, k))}{k} \tag{2}$$

Where $(1-(1-P)^k)$ is the probability that a file with k partitions will be processed. As above, T_{reduce_ppu} is the runtime for the reduce phase when the entire input data is being processed. $f(T_{reduce\_ppu}, k)$ is a function that estimates the time for the reduce phase when only k out of K input partitions are processed. Frequently, $f(T_{reduce\_ppu}, k) = \frac{kT_{reduce\_ppu}}{K}$, but this may not always hold.

Table 1 verifies that the above equations for eager and lazy processing produce reasonable estimates for processing time, with errors $< 3.1\%$. In this experiment, the original data fit into 15 map PPUs, which resulted in two sets of 8 map PPUs when the data was split in two.

Figure 3 illustrates the estimated processing times for the configuration from Table 1, where $T_{\text{start}} = 9$ s, $T_{map\_ppu} = 20$ s, $T_{reduce\_ppu} = 0$ s, $K = 16$, and varying both k and p. As the figure shows, for low probability values the lazy approach with $k = 1$ wins. The eager approach wins for high p values. In this configuration, it is never beneficial to use lazy evaluation with $k > 1$.

To decide whether to use lazy processing or not, a user should thus *estimate the parameters in the above equations and compare the expected runtimes for* $T_{\text{job\_eager}}$ *and* $T_{\text{job\_lazy}}$ *for different values of* k.

5.3 Online Data Partitioning

In the above equations, we assumed that the input data was already pre-partitioned into files. If this is not the case, the lazy processing time must be extended with the time to partition the data, which is equivalent to running a fast MapReduce job eagerly on the entire dataset.

5.4 Summary of Findings and Guidelines

To leverage lazy MapReduce processing, we recommend:

GUIDELINE 5.1. *From debug runs and domain knowledge, estimate* T_{start}, T_{map_ppu} *(*i.e., $T_{\text{map\_start}}$, N_{recs}, *and* T_{rec}*),* T_{reduce_ppu}, $f(k, T_{reduce\_ppu})$*, and* p *for the given system, dataset, and analysis.*

GUIDELINE 5.2. *Using the estimated parameters, compare the expected time given by* T_{job_eager} *(Equation 1) and* T_{job_lazy} *(Equation 2) for different values of* k*. Select the execution strategy that yields the lower expected runtime.*

GUIDELINE 5.3. *Partition the input data into files that are the size of* k *map PPUs.*

In general, the results from Section 4 indicate that the lazy approach offers greater gains for queries with high per-tuple processing costs, either from expensive relational operations or UDFs. If a workflow consists of multiple stages (*i.e.*multiple Pig jobs) executing in a pipeline, all it takes is one slow, blocking stage to affect the performance of the workflow. Hence, multi-stage workflows can potentially benefit more from lazy processing than shorter ones.

6. RELATED WORK

There is significant work in the area of scientific workflow management systems [8, 10, 28, 25, 29]. Existing systems, however, treat individual computation steps as black-boxes. In contrast, the goal of our line of work is to optimize such workflows by processing subset of data at each step eagerly and other subsets lazily in order to minimize total runtime.

Recent work on data intensive scalable computing systems based on MapReduce or its competitor, Dryad [15], has developed new techniques for incremental processing in face of appends to the input data [13, 20]. Our goal goes one step further: we seek to compute only a subset of the requested data eagerly, computing the rest lazily, whether the input data is updated or not.

The details of lazy MapReduce computations are related to incremental view maintenance [11]. The goal of this past work, however, was to efficiently maintain a materialized view (*i.e.*, workflow output) synchronized with continuously updated input data. In contrast, we focus on what parts to update at all when new data arrives and what parts to leave out-dated until a user requests the corresponding output. Even deferred view maintenance techniques [6, 31] refresh an entire materialized view rather than subsets of that view.

Our approach is most closely related to physical database tuning [5]: Choosing what data to compute eagerly is similar to materialized view selection [3]. Our problem space is different, though: physical tuning strives to minimize the total runtime for queries in a workload in face of disk-space constraints and updates [3]. In contrast, our goal is to minimize the expected running time for a query without such constraints but knowing that there is only some probability that each part of the query output will be needed.

Finally, there has been a plethora of new techniques that optimize various aspects of MapReduce (*e.g.*, [16, 20]). To the best of our knowledge, we are the first to study the specific eager/lazy processing trade-off presented in this paper.

7. CONCLUSIONS

We studied the problem of eager *v.s.* lazy processing of scientific analysis tasks using MapReduce. We showed that

Table 1: Estimated Processing Times from Equations vs Actual Processing Times from Experiment

Experiment	Approach	Est. Time	Actual Time	% diff	K	k	T_{start}	T_{MapPPU}
UDF Removed	Eager	5min 9secs	5min 11secs	-0.6%	15	n/a	9secs	20secs
UDF Removed	Process one partition with size $\frac{1}{2}$	2min 57secs	2min 51secs	3.1%	16	8	9secs	21secs
UDF Removed	Process two partitions with size $\frac{1}{2}$ each	5min 45secs	5min 42secs	0.9%	16	8	9secs	21secs

either lazy or eager processing can be preferable under different circumstances and we developed a set of guidelines to help users understand what granularity of lazy computation to use for their analysis tasks and when to use it rather than eagerly processing their entire dataset. In future work, we plan to build on this basic technique to optimize the computation of entire MapReduce workflows and also automate the lazy/eager processing decisions. This work is thus a preliminary but important step in helping the science community efficiently exploit MapReduce to analyze their data.

8. ACKNOWLEDGEMENTS

The astronomy simulation dataset was graciously supplied by T. Quinn and F. Governato of the UW Dept. of Astronomy. We thank J. P. Garner and Y. Kwon for their help with the data. We thank the anonymous reviewers for comments on drafts of this paper. This work is partially supported by NSF CAREER award IIS-0845397, a gift from Yahoo!, gifts from Microsoft Research, and Balazinska's Microsoft Research Faculty Fellowship.

9. REFERENCES

[1] LSST data management: DC3b processing flow. `http://dev.lsstcorp.org/trac/wiki/DC3bProcessingFlow`.

[2] The fourth paradigm: Data-intensive scientific discovery, 2009.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL Databases. In *Proc. of the 26th VLDB Conf.*, pages 496–505, 2000.

[4] D. Borthakur. The Hadoop distributed file system: Architecture and design. `http://lucene.apache.org/hadoop/hdfs_design.pdf`, 2007.

[5] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *Proc. of the 33rd VLDB Conf.*, pages 3–14, 2007.

[6] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. of the SIGMOD Conf.*, pages 469–480, 1996.

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

[8] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-science: An overview of workflow system features and capabilities. *Future Gen. Comput. Syst.*, 25(5):528–540, May 2009.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of the 19th SOSP*, pages 29–43, 2003.

[10] C. Goble and D. DeRoure. The impact of workflow tools on data-centric research. In T. Hey, S. Tansley, and K. Tolle, editors, *The 4th Paradigm: Data-Intensive Scientific Discovery.* Microsoft Research, 2009.

[11] A. Gupta and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. In *Materialized views: techniques, implementations, and applications*, pages 145–157. MIT Press, 1999.

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

[13] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proc. of SOCC Symp.*, pages 63–74, 2010.

[14] Hive. `http://hadoop.apache.org/hive/`.

[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 EuroSys Conf.*, pages 59–72, 2007.

[16] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proc. of SOCC Symp.*, pages 181–192, 2010.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of SOCC Symp.*, June 2010.

[18] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proc of 22nd SSDBM*, 2010.

[19] S. Loebman, D. Nunley, Y. Kwon, B. Howe, M. Balazinska, and J. P. Gardner. Analyzing massive astrophysical datasets: Can Pig/Hadoop or a relational DBMS help? In *Proc. of the IASDS Workshop*, 2009.

[20] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proc. of SOCC Symp.*, pages 51–62, 2010.

[21] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: A progress indicator for MapReduce DAGs. In *Proc. of the SIGMOD Conf.*, June 2010.

[22] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *Proc. of the 26th ICDE Conf.*, Mar. 2010.

[23] 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.

[24] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. of the SIGMOD Conf.*, pages 165–178, 2009.

[25] P. Romano. Automation of in-silico data analysis processes through workflow management systems. *Brief Bioinform*, 9(1):57–68, Jan. 2008.

[26] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[27] J. G. Stadel. *Cosmological N-body simulations and their analysis.* PhD thesis, University of Washington, 2001.

[28] T. Oinn et. al. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067âĂŞ–1100, 2006.

[29] Wassermann et. al. Sedna: a BPELbased environment for visual scientific workflow modelling. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for e-Science: Scientific Workflows for Grids*, pages 428–âĂŞ449. 2007.

[30] K. Wiley, A. Connolly, J. Gardner, S. Krughoff, M. Balazinska, B. Howe, Y. Kwon, and Y. Bu. Astronomy in the cloud: Using MapReduce for image coaddition. in submission.

[31] J. Zhou, P.-A. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proc. of the 33rd VLDB Conf.*, pages 231–242, 2007.