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-
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, 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, 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 and particles within R of P. Do not apply the predicate on distance to P 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 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. 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 1/4 of the data to 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 difer-
Figure 2: Effect of Processing Cost and Overhead in Various Partitioning Schemes

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 expected, that as the number of partitions increases, so does 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. 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].

\[
T_{job,eager} \approx T_{start} + KT_{map,ppu} + T_{reduce,ppu}
\]

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]. $T_{map,ppu}$ and $T_{reduce,ppu}$ 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. $T_{map,ppu} \approx T_{map,start} + N_{recs} * T_{rec}$, where $T_{map,start}$ 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, $T_{reduce,ppu} \approx M_{recs} * U_{rec}$. 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 may 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.

\[3\]
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):

$$T_{job, lazy} \approx \frac{K(1 - (1 - P)^k)(T_{start} + kT_{map, ppu} + f(T_{reduce, ppu}, k))}{k} \quad (2)$$

Where $(1 - (1 - P)^k)$ is the probability that a file with $k$ partitions will be processed. $T_{reduce, ppu}$ 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 $T_{start} = 9$ 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 $T_{job, eager}$ and $T_{job, lazy}$ 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., $T_{map, start}$, $N_{recs}$, 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.


<table>
<thead>
<tr>
<th>Experiment</th>
<th>Approach</th>
<th>Est. Time</th>
<th>Actual Time</th>
<th>% diff</th>
<th>K</th>
<th>k</th>
<th>T_est</th>
<th>T_mapPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDF Removed</td>
<td>Eager</td>
<td>5min 9secs</td>
<td>5min 11secs</td>
<td>-0.6%</td>
<td>15</td>
<td>n/a</td>
<td>9secs</td>
<td>20secs</td>
</tr>
<tr>
<td>UDF Removed</td>
<td>Process one partition with size $\frac{1}{4}$</td>
<td>2min 57secs</td>
<td>2min 51secs</td>
<td>-3.1%</td>
<td>16</td>
<td>8</td>
<td>9secs</td>
<td>21secs</td>
</tr>
<tr>
<td>UDF Removed</td>
<td>Process two partitions with size $\frac{1}{4}$ each</td>
<td>5min 45secs</td>
<td>5min 42secs</td>
<td>0.9%</td>
<td>16</td>
<td>8</td>
<td>9secs</td>
<td>21secs</td>
</tr>
</tbody>
</table>

7. CONCLUSIONS

We studied the problem of eager v.s. lazy processing of scientific analysis tasks using MapReduce. We showed that 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 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 and the anonymous reviewers for their comments. 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