MapReduce Online

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein
UC Berkeley

Khaled Elmeleegy, Russell Sears
Yahoo! Research

Abstract

MapReduce is a popular framework for data-intensive distributed computing of batch jobs. To simplify fault tolerance, many implementations of MapReduce *materialize* the entire output of each map and reduce task before it can be consumed. In this paper, we propose a modified MapReduce architecture that allows data to be *pipelined* between operators. This extends the MapReduce programming model beyond batch processing, and can reduce completion times and improve system utilization for batch jobs as well. We present a modified version of the Hadoop MapReduce framework that supports *online aggregation*, which allows users to see "early returns" from a job as it is being computed. Our Hadoop Online Prototype (*HOP*) also supports *continuous queries*, which enable MapReduce programs to be written for applications such as event monitoring and stream processing. HOP retains the fault tolerance properties of Hadoop and can run unmodified user-defined MapReduce programs.

1 Introduction

MapReduce has emerged as a popular way to harness the power of large clusters of computers. MapReduce allows programmers to think in a *data-centric* fashion: they focus on applying transformations to sets of data records, and allow the details of distributed execution, network communication and fault tolerance to be handled by the MapReduce framework.

MapReduce is typically applied to large batch-oriented computations that are concerned primarily with time to job completion. The Google MapReduce framework [6] and open-source Hadoop system reinforce this usage model through a batch-processing implementation strategy: the entire output of each map and reduce task is *materialized* to a local file before it can be consumed by the next stage. Materialization allows for a simple and elegant checkpoint/restart fault tolerance mechanism that is critical in large deployments, which have a high probability of slowdowns or failures at worker nodes.

We propose a modified MapReduce architecture in which intermediate data is *pipelined* between operators, while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks. To validate this design, we developed the Hadoop Online Prototype (HOP), a pipelining version of Hadoop.[1]

Pipelining provides several important advantages to a MapReduce framework, but also raises new design challenges. We highlight the potential benefits first:

- Since reducers begin processing data as soon as it is produced by mappers, they can generate and refine an approximation of their final answer during the course of execution. This technique, known as *online aggregation* [12], can provide initial estimates of results several orders of magnitude faster than the final results. We describe how we adapted online aggregation to our pipelined MapReduce architecture in Section 4.

- Pipelining widens the domain of problems to which MapReduce can be applied. In Section 5, we show how HOP can be used to support *continuous queries*: MapReduce jobs that run continuously, accepting new data as it arrives and analyzing it immediately. This allows MapReduce to be used for applications such as event monitoring and stream processing.

- Pipelining delivers data to downstream operators more promptly, which can increase opportunities for parallelism, improve utilization, and reduce response time. A thorough performance study is a topic for future work; however, in Section 6 we present some initial performance results which demonstrate that pipelining can reduce job completion times by up to 25% in some scenarios.

[1] The source code for HOP can be downloaded from `http://code.google.com/p/hop/`

Pipelining raises several design challenges. First, Google's attractively simple MapReduce fault tolerance mechanism is predicated on the materialization of intermediate state. In Section 3.3, we show that this can coexist with pipelining, by allowing producers to periodically ship data to consumers in parallel with their materialization. A second challenge arises from the greedy communication implicit in pipelines, which is at odds with batch-oriented optimizations supported by "combiners": map-side code that reduces network utilization by performing pre-aggregation before communication. We discuss how the HOP design addresses this issue in Section 3.1. Finally, pipelining requires that producers and consumers are co-scheduled intelligently; we discuss our initial work on this issue in Section 3.4.

1.1 Structure of the Paper

In order to ground our discussion, we present an overview of the Hadoop MapReduce architecture in Section 2. We then develop the design of HOP's pipelining scheme in Section 3, keeping the focus on traditional batch processing tasks. In Section 4 we show how HOP can support online aggregation for long-running jobs and illustrate the potential benefits of that interface for MapReduce tasks. In Section 5 we describe our support for continuous MapReduce jobs over data streams and demonstrate an example of near-real-time cluster monitoring. We present initial performance results in Section 6. Related and future work are covered in Sections 7 and 8.

2 Background

In this section, we review the MapReduce programming model and describe the salient features of Hadoop, a popular open-source implementation of MapReduce.

2.1 Programming Model

To use MapReduce, the programmer expresses their desired computation as a series of *jobs*. The input to a job is an input specification that will yield key-value pairs. Each job consists of two stages: first, a user-defined *map* function is applied to each input record to produce a list of intermediate key-value pairs. Second, a user-defined *reduce* function is called once for each distinct key in the map output and passed the list of intermediate values associated with that key. The MapReduce framework automatically parallelizes the execution of these functions and ensures fault tolerance.

Optionally, the user can supply a *combiner* function [6]. Combiners are similar to reduce functions, except that they are not passed *all* the values for a given key: instead, a combiner emits an output value that summarizes the input values it was passed. Combiners are typically used to perform map-side "pre-aggregation," which reduces the amount of network traffic required between the map and reduce steps.

```
public interface Mapper<K1, V1, K2, V2> {
  void map(K1 key, V1 value,
           OutputCollector<K2, V2> output);

  void close();
}
```

Figure 1: Map function interface.

2.2 Hadoop Architecture

Hadoop is composed of *Hadoop MapReduce*, an implementation of MapReduce designed for large clusters, and the *Hadoop Distributed File System* (HDFS), a file system optimized for batch-oriented workloads such as MapReduce. In most Hadoop jobs, HDFS is used to store both the input to the map step and the output of the reduce step. Note that HDFS is *not* used to store intermediate results (e.g., the output of the map step): these are kept on each node's local file system.

A Hadoop installation consists of a single master node and many worker nodes. The master, called the *JobTracker*, is responsible for accepting jobs from clients, dividing those jobs into *tasks*, and assigning those tasks to be executed by worker nodes. Each worker runs a *TaskTracker* process that manages the execution of the tasks currently assigned to that node. Each TaskTracker has a fixed number of slots for executing tasks (two maps and two reduces by default).

2.3 Map Task Execution

Each map task is assigned a portion of the input file called a *split*. By default, a split contains a single HDFS block (64MB by default), so the total number of file blocks determines the number of map tasks.

The execution of a map task is divided into two phases.

1. The *map* phase reads the task's split from HDFS, parses it into records (key/value pairs), and applies the map function to each record.

2. After the map function has been applied to each input record, the *commit* phase registers the final output with the TaskTracker, which then informs the JobTracker that the task has finished executing.

Figure 1 contains the interface that must be implemented by user-defined map functions. After the *map* function has been applied to each record in the split, the *close* method is invoked.

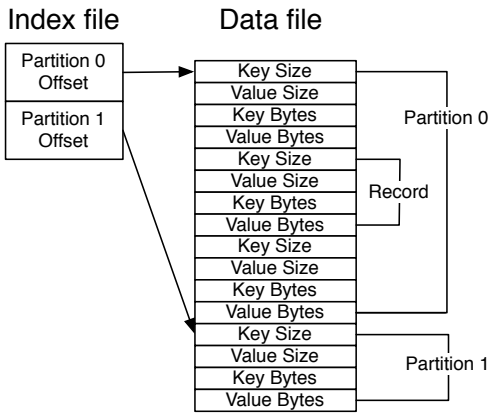


Figure 2: Map task index and data file format (2 partition/reduce case).

The third argument to the *map* method specifies an *OutputCollector* instance, which accumulates the output records produced by the map function. The output of the map step is consumed by the reduce step, so the OutputCollector stores map output in a format that is easy for reduce tasks to consume. Intermediate keys are assigned to reducers by applying a partitioning function, so the OutputCollector applies that function to each key produced by the map function, and stores each record and partition number in an in-memory buffer. The OutputCollector spills this buffer to disk when it reaches capacity.

A spill of the in-memory buffer involves first sorting the records in the buffer by partition number and then by key. The buffer content is written to the local file system as an index file and a data file (Figure 2). The index file points to the offset of each partition in the data file. The data file contains only the records, which are sorted by the key within each partition segment.

During the *commit* phase, the final output of the map task is generated by merging all the spill files produced by this task into a single pair of data and index files. These files are registered with the TaskTracker before the task completes. The TaskTracker will read these files when servicing requests from reduce tasks.

2.4 Reduce Task Execution

The execution of a reduce task is divided into three phases.

1. The *shuffle* phase fetches the reduce task's input data. Each reduce task is assigned a partition of the key range produced by the map step, so the reduce task must fetch the content of this partition from every map task's output.

2. The *sort* phase groups records with the same key together.

```
public interface Reducer<K2, V2, K3, V3> {
  void reduce(K2 key, Iterator<V2> values,
              OutputCollector<K3, V3> output);

  void close();
}
```

Figure 3: Reduce function interface.

3. The *reduce* phase applies the user-defined reduce function to each key and corresponding list of values.

In the *shuffle* phase, a reduce task fetches data from each map task by issuing HTTP requests to a configurable number of TaskTrackers at once (5 by default). The JobTracker relays the location of every TaskTracker that hosts map output to every TaskTracker that is executing a reduce task. Note that a reduce task cannot fetch the output of a map task until the map has finished executing and committed its final output to disk.

After receiving its partition from all map outputs, the reduce task enters the *sort* phase. The map output for each partition is already sorted by the reduce key. The reduce task merges these runs together to produce a single run that is sorted by key. The task then enters the *reduce* phase, in which it invokes the user-defined reduce function for each distinct key in sorted order, passing it the associated list of values. The output of the reduce function is written to a temporary location on HDFS. After the reduce function has been applied to each key in the reduce task's partition, the task's HDFS output file is atomically renamed from its temporary location to its final location.

In this design, the output of both map and reduce tasks is written to disk before it can be consumed. This is particularly expensive for reduce tasks, because their output is written to HDFS. Output materialization simplifies fault tolerance, because it reduces the amount of state that must be restored to consistency after a node failure. If any task (either map or reduce) fails, the JobTracker simply schedules a new task to perform the same work as the failed task. Since a task never exports any data other than its final answer, no further recovery steps are needed.

3 Pipelined MapReduce

In this section we discuss our extensions to Hadoop to support pipelining between tasks (Section 3.1) and between jobs (Section 3.2). We describe how our design supports fault tolerance (Section 3.3), and discuss the interaction between pipelining and task scheduling (Section 3.4). Our focus here is on batch-processing workloads; we discuss online aggregation and continuous queries in Section 4 and Section 5. We defer performance results to Section 6.

3.1 Pipelining Within A Job

As described in Section 2.4, reduce tasks traditionally issue HTTP requests to *pull* their output from each TaskTracker. This means that map task execution is completely decoupled from reduce task execution. To support pipelining, we modified the map task to instead *push* data to reducers as it is produced. To give an intuition for how this works, we begin by describing a straightforward pipelining design, and then discuss the changes we had to make to achieve good performance.

3.1.1 Naïve Pipelining

In our naïve implementation, we modified Hadoop to send data directly from map to reduce tasks. When a client submits a new job to Hadoop, the JobTracker assigns the map and reduce tasks associated with the job to the available TaskTracker slots. For purposes of discussion, we assume that there are enough free slots to assign all the tasks for each job. We modified Hadoop so that each reduce task contacts every map task upon initiation of the job, and opens a TCP socket which will be used to pipeline the output of the map function. As each map output record is produced, the mapper determines which partition (reduce task) the record should be sent to, and immediately sends it via the appropriate socket.

A reduce task accepts the pipelined data it receives from each map task and stores it in an in-memory buffer, spilling sorted runs of the buffer to disk as needed. Once the reduce task learns that every map task has completed, it performs a final merge of all the sorted runs and applies the user-defined reduce function as normal.

3.1.2 Refinements

While the algorithm described above is straightforward, it suffers from several practical problems. First, it is possible that there will not be enough slots available to schedule every task in a new job. Opening a socket between every map and reduce task also requires a large number of TCP connections. A simple tweak to the naïve design solves both problems: if a reduce task has not yet been scheduled, any map tasks that produce records for that partition simply write them to disk. Once the reduce task is assigned a slot, it can then pull the records from the map task, as in regular Hadoop. To reduce the number of concurrent TCP connections, each reducer can be configured to pipeline data from a bounded number of mappers at once; the reducer will pull data from the remaining map tasks in the traditional Hadoop manner.

Our initial pipelining implementation suffered from a second problem: the map function was invoked by the same thread that wrote output records to the pipeline sockets. This meant that if a network I/O operation blocked (e.g., because the reducer was over-utilized), the mapper was prevented from doing useful work. Pipeline stalls should not prevent a map task from making progress—especially since, once a task has completed, it frees a TaskTracker slot to be used for other purposes. We solved this problem by running the map function in a separate thread that stores its output in an in-memory buffer, and then having another thread periodically send the contents of the buffer to the connected reducers.

3.1.3 Granularity of Map Output

Another problem with the naïve design is that it eagerly sends each record as soon as it is produced, which prevents the use of map-side combiners. Imagine a job where the reduce key has few distinct values (e.g., gender), and the reduce applies an aggregate function (e.g., count). As discussed in Section 2.1, combiners allow map-side "pre-aggregation": by applying a reduce-like function to each distinct key at the mapper, network traffic can often be substantially reduced. Eagerly pipelining each record as it is produced prevents the use of map-side combiners.

A related problem is that eager pipelining moves some of the sorting work from the mapper to the reducer. Recall that in the blocking architecture, map tasks generate sorted spill files: all the reduce task must do is merge together the pre-sorted map output for each partition. In the naïve pipelining design, map tasks send output records in the order in which they are generated, so the reducer must perform a full external sort. Because the number of map tasks typically far exceeds the number of reduces [6], moving more work to the reducer increased response time in our experiments.

We addressed these issues by modifying the in-memory buffer design described in Section 3.1.2. Instead of sending the buffer contents to reducers directly, we wait for the buffer to grow to a threshold size. The mapper then applies the combiner function, sorts the output by partition and reduce key, and writes the buffer to disk using the spill file format described in Section 2.3.

Next, we arranged for the TaskTracker at each node to handle pipelining data to reduce tasks. Map tasks register spill files with the TaskTracker via RPCs. If the reducers are able to keep up with the production of map outputs and the network is not a bottleneck, a spill file will be sent to a reducer soon after it has been produced (in which case, the spill file is likely still resident in the map machine's kernel buffer cache). However, if a reducer begins to fall behind, the number of unsent spill files will grow.

When a map task generates a new spill file, it first queries the TaskTracker for the number of unsent spill files. If this number grows beyond a certain threshold (two unsent spill files in our experiments), the map task does not immediately register the new spill file with the

TaskTracker. Instead, the mapper will accumulate multiple spill files. Once the queue of unsent spill files falls below the threshold, the map task merges and combines the accumulated spill files into a single file, and then resumes registering its output with the TaskTracker. This simple flow control mechanism has the effect of *adaptively* moving load from the reducer to the mapper or vice versa, depending on which node is the current bottleneck.

A similar mechanism is also used to control how aggressively the combiner function is applied. The map task records the ratio between the input and output data sizes whenever it invokes the combiner function. If the combiner is effective at reducing data volumes, the map task accumulates more spill files (and applies the combiner function to all of them) before registering that output with the TaskTracker for pipelining.[2]

The connection between pipelining and adaptive query processing techniques has been observed elsewhere (e.g., [2]). The adaptive scheme outlined above is relatively simple, but we believe that adapting to feedback along pipelines has the potential to significantly improve the utilization of MapReduce clusters.

3.2 Pipelining Between Jobs

Many practical computations cannot be expressed as a single MapReduce job, and the outputs of higher-level languages like Pig [20] typically involve multiple jobs. In the traditional Hadoop architecture, the output of each job is written to HDFS in the reduce step and then immediately read back from HDFS by the map step of the next job. Furthermore, the JobTracker cannot schedule a consumer job until the producer job has completed, because scheduling a map task requires knowing the HDFS block locations of the map's input split.

In our modified version of Hadoop, the reduce tasks of one job can optionally pipeline their output directly to the map tasks of the next job, sidestepping the need for expensive fault-tolerant storage in HDFS for what amounts to a temporary file. Unfortunately, the computation of the reduce function from the previous job and the map function of the next job cannot be overlapped: the final result of the reduce step cannot be produced until all map tasks have completed, which prevents effective pipelining. However, in the next sections we describe how online aggregation and continuous query pipelines can publish "snapshot" outputs that can indeed pipeline between jobs.

[2]Our current prototype uses a simple heuristic: if the combiner reduces data volume by $\frac{1}{k}$ on average, we wait until k spill files have accumulated before registering them with the TaskTracker. A better heuristic would also account for the computational cost of applying the combiner function.

3.3 Fault Tolerance

Our pipelined Hadoop implementation is robust to the failure of both map and reduce tasks. To recover from map task failures, we added bookkeeping to the reduce task to record which map task produced each pipelined spill file. To simplify fault tolerance, the reducer treats the output of a pipelined map task as "tentative" until the JobTracker informs the reducer that the map task has committed successfully. The reducer can merge together spill files generated by the same uncommitted mapper, but will not combine those spill files with the output of other map tasks until it has been notified that the map task has committed. Thus, if a map task fails, each reduce task can ignore any tentative spill files produced by the failed map attempt. The JobTracker will take care of scheduling a new map task attempt, as in stock Hadoop.

If a reduce task fails and a new copy of the task is started, the new reduce instance must be sent all the input data that was sent to the failed reduce attempt. If map tasks operated in a purely pipelined fashion and discarded their output after sending it to a reducer, this would be difficult. Therefore, map tasks retain their output data on the local disk for the complete job duration. This allows the map's output to be reproduced if any reduce tasks fail. For batch jobs, the key advantage of our architecture is that reducers are not blocked waiting for the complete output of the task to be written to disk.

Our technique for recovering from map task failure is straightforward, but places a minor limit on the reducer's ability to merge spill files. To avoid this, we envision introducing a "checkpoint" concept: as a map task runs, it will periodically notify the JobTracker that it has reached offset x in its input split. The JobTracker will notify any connected reducers; map task output that was produced before offset x can then be merged by reducers with other map task output as normal. To avoid duplicate results, if the map task fails, the new map task attempt resumes reading its input at offset x. This technique would also reduce the amount of redundant work done after a map task failure or during speculative execution of "backup" tasks [6].

3.4 Task Scheduling

The Hadoop JobTracker had to be retrofitted to support pipelining between jobs. In regular Hadoop, job are submitted one at a time; a job that consumes the output of one or more other jobs cannot be submitted until the producer jobs have completed. To address this, we modified the Hadoop job submission interface to accept a list of jobs, where each job in the list depends on the job before it. The client interface traverses this list, annotating each job with the identifier of the job that it depends on. The

JobTracker looks for this annotation and co-schedules jobs with their dependencies, giving slot preference to "upstream" jobs over the "downstream" jobs they feed. As we note in Section 8, there are many interesting options for scheduling pipelines or even DAGs of such jobs that we plan to investigate in future.

4 Online Aggregation

Although MapReduce was originally designed as a batch-oriented system, it is often used for interactive data analysis: a user submits a job to extract information from a data set, and then waits to view the results before proceeding with the next step in the data analysis process. This trend has accelerated with the development of high-level query languages that are executed as MapReduce jobs, such as Hive [27], Pig [20], and Sawzall [23].

Traditional MapReduce implementations provide a poor interface for interactive data analysis, because they do not emit any output until the job has been executed to completion. In many cases, an interactive user would prefer a "quick and dirty" approximation over a correct answer that takes much longer to compute. In the database literature, online aggregation has been proposed to address this problem [12], but the batch-oriented nature of traditional MapReduce implementations makes these techniques difficult to apply. In this section, we show how we extended our pipelined Hadoop implementation to support online aggregation within a single job (Section 4.1) and between multiple jobs (Section 4.2). In Section 4.3, we evaluate online aggregation on two different data sets, and show that it can yield an accurate approximate answer long before the job has finished executing.

4.1 Single-Job Online Aggregation

In HOP, the data records produced by map tasks are sent to reduce tasks shortly after each record is generated. However, to produce the final output of the job, the reduce function cannot be invoked until the entire output of every map task has been produced. We can support online aggregation by simply applying the reduce function to the data that a reduce task has received so far. We call the output of such an intermediate reduce operation a *snapshot*.

Users would like to know how accurate a snapshot is: that is, how closely a snapshot resembles the final output of the job. Accuracy estimation is a hard problem even for simple SQL queries [15], and particularly hard for jobs where the map and reduce functions are opaque user-defined code. Hence, we report job *progress*, not accuracy: we leave it to the user (or their MapReduce code) to correlate progress to a formal notion of accuracy. We give a simple progress metric below.

Snapshots are computed periodically, as new data arrives at each reducer. The user specifies how often snapshots should be computed, using the progress metric as the unit of measure. For example, a user can request that a snapshot be computed when 25%, 50%, and 75% of the input has been seen. The user may also specify whether to include data from tentative (unfinished) map tasks. This option does not affect the fault tolerance design described in Section 3.3. In the current prototype, each snapshot is stored in a directory on HDFS. The name of the directory includes the progress value associated with the snapshot. Each reduce task runs independently, and at a different rate. Once a reduce task has made sufficient progress, it writes a snapshot to a temporary directory on HDFS, and then atomically renames it to the appropriate location.

Applications can consume snapshots by polling HDFS in a predictable location. An application knows that a given snapshot has been completed when every reduce task has written a file to the snapshot directory. Atomic rename is used to avoid applications mistakenly reading incomplete snapshot files.

Note that if there are not enough free slots to allow all the reduce tasks in a job to be scheduled, snapshots will not be available for reduce tasks that are still waiting to be executed. The user can detect this situation (e.g., by checking for the expected number of files in the HDFS snapshot directory), so there is no risk of incorrect data, but the usefulness of online aggregation will be reduced. In the current prototype, we manually configured the cluster to avoid this scenario. The system could also be enhanced to avoid this pitfall entirely by optionally waiting to execute an online aggregation job until there are enough reduce slots available.

4.1.1 Progress Metric

Hadoop provides support for monitoring the progress of task executions. As each map task executes, it is assigned a *progress score* in the range [0,1], based on how much of its input the map task has consumed. We reused this feature to determine how much progress is represented by the current input to a reduce task, and hence to decide when a new snapshot should be taken.

First, we modified the spill file format depicted in Figure 2 to include the map's current progress score. When a partition in a spill file is sent to a reducer, the spill file's progress score is also included. To compute the progress score for a snapshot, we take the average of the progress scores associated with each spill file used to produce the snapshot.

Note that it is possible that a map task might not have pipelined *any* output to a reduce task, either because the map task has not been scheduled yet (there are no free TaskTracker slots), the map tasks does not produce any

output for the given reduce task, or because the reduce task has been configured to only pipeline data from at most k map tasks concurrently. To account for this, we need to scale the progress metric to reflect the portion of the map tasks that a reduce task has pipelined data from: if a reducer is connected to $\frac{1}{n}$ of the total number of map tasks in the job, we divide the average progress score by n.

This progress metric could easily be made more sophisticated: for example, an improved metric might include the selectivity ($|output|/|input|$) of each map task, the statistical distribution of the map task's output, and the effectiveness of each map task's combine function, if any. Although we have found our simple progress metric to be sufficient for most experiments we describe below, this clearly represents an opportunity for future work.

4.2 Multi-Job Online Aggregation

Online aggregation is particularly useful when applied to a long-running analysis task composed of multiple MapReduce jobs. As described in Section 3.2, our version of Hadoop allows the output of a reduce task to be sent directly to map tasks. This feature can be used to support online aggregation for a sequence of jobs.

Suppose that j_1 and j_2 are two MapReduce jobs, and j_2 consumes the output of j_1. When j_1's reducers compute a snapshot to perform online aggregation, that snapshot is written to HDFS, and also sent directly to the map tasks of j_2. The map and reduce steps for j_2 are then computed as normal, to produce a snapshot of j_2's output. This process can then be continued to support online aggregation for an arbitrarily long sequence of jobs.

Unfortunately, inter-job online aggregation has some drawbacks. First, the output of a reduce function is not "monotonic": the output of a reduce function on the first 50% of the input data may not be obviously related to the output of the reduce function on the first 25%. Thus, as new snapshots are produced by j_1, j_2 must be recomputed from scratch using the new snapshot. As with inter-job pipelining (Section 3.2), this could be optimized for reduce functions that are declared to be distributive or algebraic aggregates [9].

To support fault tolerance for multi-job online aggregation, we consider three cases. Tasks that fail in j_1 recover as described in Section 3.3. If a task in j_2 fails, the system simply restarts the failed task. Since subsequent snapshots produced by j_1 are taken from a superset of the mapper output in j_1, the next snapshot received by the restarted reduce task in j_2 will have a higher progress score. To handle failures in j_1, tasks in j_2 cache the most recent snapshot received by j_1, and replace it when they receive a new snapshot with a higher progress metric. If tasks from both jobs fail, a new task in j_2 recovers the most recent snapshot from j_1 that was stored in HDFS and then wait for snapshots with a higher progress score.

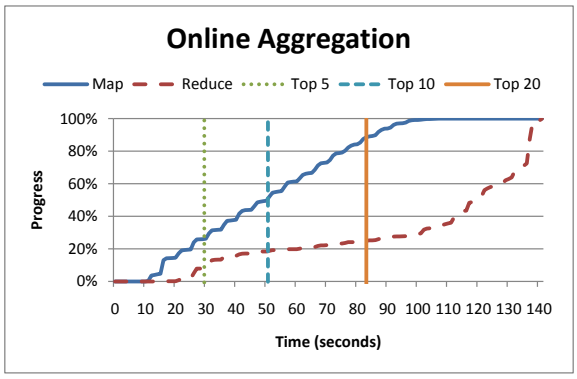


Figure 4: Top-100 query over 5.5GB of Wikipedia article text. The vertical lines describe the increasing accuracy of the approximate answers produced by online aggregation.

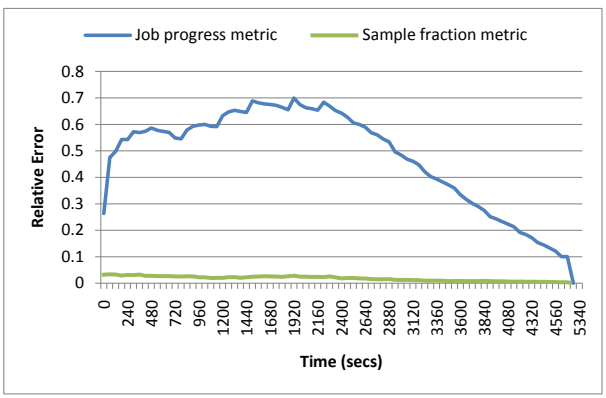
4.3 Evaluation

To evaluate the effectiveness of online aggregation, we performed two experiments on Amazon EC2 using different data sets and query workloads. In our first experiment, we wrote a "Top-K" query using two MapReduce jobs: the first job counts the frequency of each word and the second job selects the K most frequent words. We ran this workload on 5.5GB of Wikipedia article text stored in HDFS, using a 128MB block size. We used a 60-node EC2 cluster; each node was a "high-CPU medium" EC2 instance with 1.7GB of RAM and 2 virtual cores. A virtual core is the equivalent of a 2007-era 2.5Ghz Intel Xeon processor. A single EC2 node executed the Hadoop JobTracker and the HDFS NameNode, while the remaining nodes served as slaves for running the TaskTrackers and HDFS DataNodes.

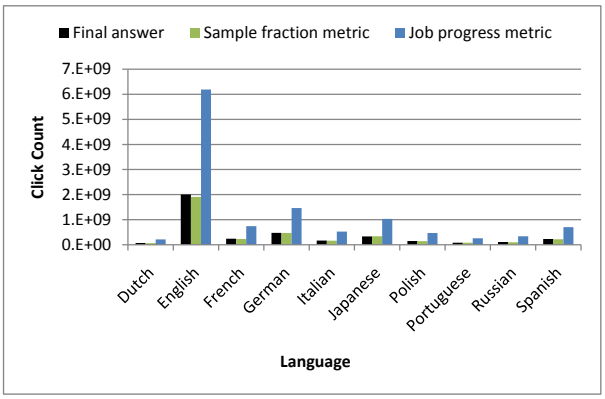
Figure 4 shows the results of inter-job online aggregation for a Top-100 query. Our accuracy metric for this experiment is post-hoc — we note the time at which the Top-K words in the snapshot are the Top-K words in the final result. Although the final result for this job did not appear until nearly the end, we did observe the Top-5, 10, and 20 values at the times indicated in the graph. The Wikipedia data set was biased toward these Top-K words (e.g., "the", "is", etc.), which remained in their correct position throughout the lifetime of the job.

4.3.1 Approximation Metrics

In our second experiment, we considered the effectiveness of the job progress metric described in Section 4.1.1. Unsurprisingly, this metric can be inaccurate when it is used to estimate the accuracy of the approximate answers produced by online aggregation. In this experiment, we com-



(a) Relative approximation error over time



(b) Example approximate answer

Figure 5: Comparison of two approximation metrics. Figure (a) shows the relative error for each approximation metric over the runtime of the job, averaged over all groups. Figure (b) compares an example approximate answer produced by each metric with the final answer, for each language and for a single hour.

pared the job progress metric with a simple user-defined metric that leverages knowledge of the query and data set. HOP allows such metrics, although developing such a custom metric imposes more burden on the programmer than using the generic progress-based metric.

We used a data set containing seven months of hourly page view statistics for more than 2.5 million Wikipedia articles [26]. This constituted 320GB of compressed data (1TB uncompressed), divided into 5066 compressed files. We stored the data set on HDFS and assigned a single map task to each file, which was decompressed before the map function was applied.

We wrote a MapReduce job to count the total number of page views for each language and each hour of the day. In other words, our query grouped by language and hour of day, and summed the number of page views that occurred in each group. To enable more accurate approximate answers, we modified the map function to include the fraction of a given hour that each record represents. The reduce function summed these fractions for a given hour, which equated to one for all records from a single map task. Since the total number of hours was known ahead of time, we could use the result of this sum over all map outputs to determine the total fraction of each hour that had been sampled. We call this user-defined metric the "sample fraction."

To compute approximate answers, each intermediate result was scaled up using two different metrics: the generic metric based on job progress and the sample fraction described above. Figure 5a reports the relative error of the two metrics, averaged over all groups. Figure 5b shows an example approximate answer for a single hour using both metrics (computed two minutes into the job runtime). This figure also contains the final answer for comparison. Both results indicate that the sample fraction metric provides a much more accurate approximate answer for this query than the progress-based metric.

Job progress is clearly the wrong metric to use for approximating the final answer of this query. The primary reason is that it is too coarse of a metric. Each intermediate result was computed from some fraction of each hour. However, the job progress assumes that this fraction is uniform across all hours, when in fact we could have received much more of one hour and much less of another. This assumption of uniformity in the job progress resulted in a significant approximation error. By contrast, the sample fraction scales the approximate answer for each group according to the actual fraction of data seen for that group, yielding much more accurate approximations.

5 Continuous Queries

MapReduce is often used to analyze streams of constantly-arriving data, such as URL access logs [6] and system console logs [30]. Because of traditional constraints on MapReduce, this is done in large batches that can only provide periodic views of activity. This introduces significant latency into a data analysis process that ideally should run in near-real time. It is also potentially inefficient: each new MapReduce job does not have access to the computational state of the last analysis run, so this state must be recomputed from scratch. The programmer can manually save the state of each job and then reload it for the next analysis operation, but this is labor-intensive.

Our pipelined version of Hadoop allows an alternative architecture: MapReduce jobs that run *continuously*, accepting new data as it becomes available and analyzing it immediately. This allows for near-real-time analysis of data streams, and thus allows the MapReduce programming model to be applied to domains such as environment

monitoring and real-time fraud detection.

In this section, we describe how HOP supports continuous MapReduce jobs, and how we used this feature to implement a rudimentary cluster monitoring tool.

5.1 Continuous MapReduce Jobs

A bare-bones implementation of continuous MapReduce jobs is easy to implement using pipelining. No changes are needed to implement continuous map tasks: map output is already delivered to the appropriate reduce task shortly after it is generated. We added an optional "flush" API that allows map functions to force their current output to reduce tasks. When a reduce task is unable to accept such data, the mapper framework stores it locally and sends it at a later time. With proper scheduling of reducers, this API allows a map task to ensure that an output record is promptly sent to the appropriate reducer.

To support continuous reduce tasks, the user-defined reduce function must be periodically invoked on the map output available at that reducer. Applications will have different requirements for how frequently the reduce function should be invoked; possible choices include periods based on wall-clock time, logical time (e.g., the value of a field in the map task output), and the number of input rows delivered to the reducer. The output of the reduce function can be written to HDFS, as in our implementation of online aggregation. However, other choices are possible; our prototype system monitoring application (described below) sends an alert via email if an anomalous situation is detected.

In our current implementation, the number of map and reduce tasks is fixed, and must be configured by the user. This is clearly problematic: manual configuration is error-prone, and many stream processing applications exhibit "bursty" traffic patterns, in which peak load far exceeds average load. In the future, we plan to add support for elastic scaleup/scaledown of map and reduce tasks in response to variations in load.

5.1.1 Fault Tolerance

In the checkpoint/restart fault-tolerance model used by Hadoop, mappers retain their output until the end of the job to facilitate fast recovery from reducer failures. In a continuous query context, this is infeasible, since mapper history is in principle unbounded. However, many continuous reduce functions (e.g., 30-second moving average) only require a suffix of the map output stream. This common case can be supported easily, by extending the JobTracker interface to capture a rolling notion of reducer consumption. Map-side spill files are maintained in a ring buffer with unique IDs for spill files over time. When a reducer commits an output to HDFS, it informs the JobTracker about the *run* of map output records it no longer needs, identifying the run by spill file IDs and offsets within those files. The JobTracker can then tell mappers to garbage collect the appropriate data.

In principle, complex reducers may depend on very long (or infinite) histories of map records to accurately reconstruct their internal state. In that case, deleting spill files from the map-side ring buffer will result in potentially inaccurate recovery after faults. Such scenarios can be handled by having reducers checkpoint internal state to HDFS, along with markers for the mapper offsets at which the internal state was checkpointed. The MapReduce framework can be extended with APIs to help with state serialization and offset management, but it still presents a programming burden on the user to correctly identify the sensitive internal state. That burden can be avoided by more heavyweight process-pair techniques for fault tolerance, but those are quite complex and use significant resources [24]. In our work to date we have focused on cases where reducers can be recovered from a reasonable-sized history at the mappers, favoring minor extensions to the simple fault-tolerance approach used in Hadoop.

5.2 Prototype Monitoring System

Our monitoring system is composed of *agents* that run on each monitored machine and record statistics of interest (e.g., load average, I/O operations per second, etc.). Each agent is implemented as a continuous map task: rather than reading from HDFS, the map task instead reads from various system-local data streams (e.g., `/proc`).

Each agent forwards statistics to an *aggregator* that is implemented as a continuous reduce task. The aggregator records how agent-local statistics evolve over time (e.g., by computing windowed-averages), and compares statistics between agents to detect anomalous behavior. Each aggregator monitors the agents that report to it, but might also report statistical summaries to another "upstream" aggregator. For example, the system might be configured to have an aggregator for each rack and then a second level of aggregators that compare statistics between racks to analyze datacenter-wide behavior.

5.3 Evaluation

To validate our prototype system monitoring tool, we constructed a scenario in which one member of a MapReduce cluster begins thrashing during the execution of a job. Our goal was to test how quickly our monitoring system would detect this behavior. The basic mechanism is similar to an alert system one of the authors implemented at an Internet search company.

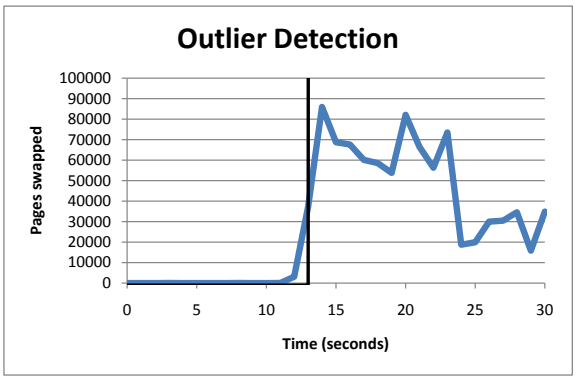


Figure 6: Number of pages swapped over time on the thrashing host, as reported by `vmstat`. The vertical line indicates the time at which the alert was sent by the monitoring system.

We used a simple load metric (a linear combination of CPU utilization, paging, and swap activity). The continuous reduce function maintains windows over samples of this metric: at regular intervals, it compares the 20 second moving average of the load metric for each host to the 120 second moving average of all the hosts in the cluster *except* that host. If the given host's load metric is more than two standard deviations above the global average, it is considered an outlier and a tentative alert is issued. To dampen false positives in "bursty" load scenarios, we do not issue an alert until we have received 10 tentative alerts within a time window.

We deployed this system on an EC2 cluster consisting of 7 "large" nodes (large nodes were chosen because EC2 allocates an entire physical host machine to them). We ran a wordcount job on the 5.5GB Wikipedia data set, using 5 map tasks and 2 reduce tasks (1 task per host). After the job had been running for about 10 seconds, we selected a node running a task and launched a program that induced thrashing.

We report detection latency in Figure 6. The vertical bar indicates the time at which the monitoring tool fired a (non-tentative) alert. The thrashing host was detected very rapidly—notably faster than the 5-second TaskTracker-JobTracker heartbeat cycle that is used to detect straggler tasks in stock Hadoop. We envision using these alerts to do early detection of stragglers within a MapReduce job: HOP could make scheduling decisions for a job by running a secondary continuous monitoring query. Compared to out-of-band monitoring tools, this economy of mechanism—reusing the MapReduce infrastructure for reflective monitoring—has benefits in software maintenance and system management.

6 Performance Evaluation

A thorough performance comparison between pipelining and blocking is beyond the scope of this paper. In this section, we instead demonstrate that pipelining can reduce job completion times in some configurations.

We report performance using both large (512MB) and small (32MB) HDFS block sizes using a single workload (a wordcount job over randomly-generated text). Since the words were generated using a uniform distribution, map-side combiners were ineffective for this workload. We performed all experiments using relatively small clusters of Amazon EC2 nodes. We also did not consider performance in an environment where multiple concurrent jobs are executing simultaneously.

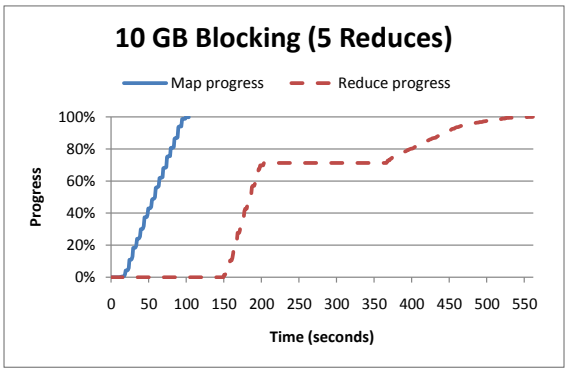
6.1 Background and Configuration

Before diving into the performance experiments, it is important to further describe the division of labor in a HOP job, which is broken into task phases. A map task consists of two work phases: *map* and *sort*. The majority of work is performed in the *map* phase, where the map function is applied to each record in the input and subsequently sent to an output buffer. Once the entire input has been processed, the map task enters the *sort* phase, where a final merge sort of all intermediate spill files is performed before registering the final output with the TaskTracker. The progress reported by a map task corresponds to the *map* phase only.

A reduce task in HOP is divided into three work phases: *shuffle*, *reduce*, and *commit*. In the *shuffle* phase, reduce tasks receive their portion of the output from each map. In HOP, the *shuffle* phase consumes 75% of the overall reduce task progress while the remaining 25% is allocated to the *reduce* and *commit* phase.[3] In the *shuffle* phase, reduce tasks periodically perform a merge sort on the already received map output. These intermediate merge sorts decrease the amount of sorting work performed at the end of the *shuffle* phase. After receiving its portion of data from all map tasks, the reduce task performs a final merge sort and enters the *reduce* phase.

By pushing work from map tasks to reduce tasks more aggressively, pipelining can enable better overlapping of map and reduce computation, especially when the node on which a reduce task is scheduled would otherwise be underutilized. However, when reduce tasks are already the bottleneck, pipelining offers fewer performance benefits, and may even hurt performance by placing additional load on the reduce nodes.

[3]The stock version of Hadoop divides the reduce progress evenly among the three phases. We deviated from this approach because we wanted to focus more on the progress during the *shuffle* phase.



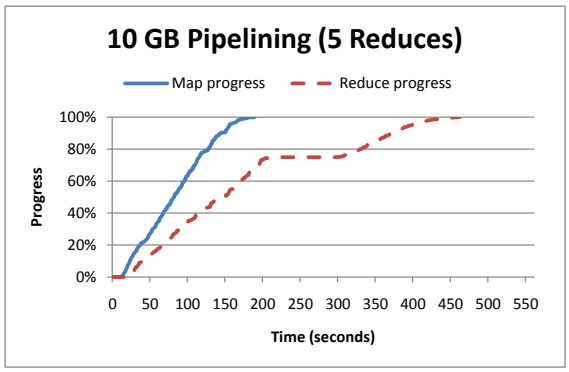
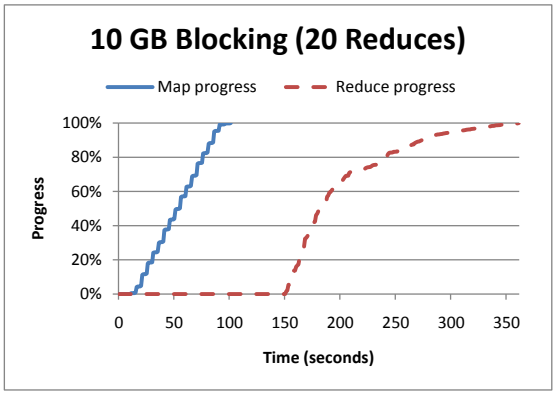


Figure 7: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 5 reduce tasks (512MB block size). The total job runtimes were 561 seconds for blocking and 462 seconds for pipelining.



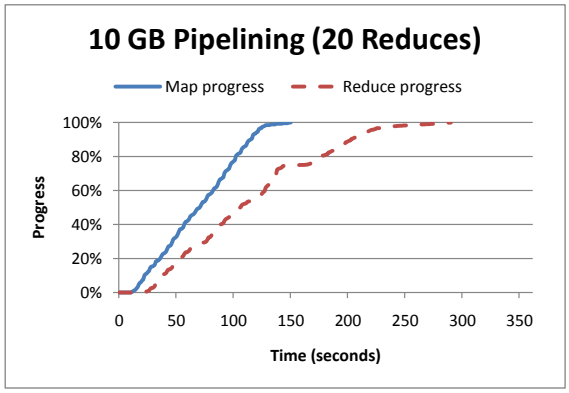


Figure 8: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 20 reduce tasks (512MB block size). The total job runtimes were 361 seconds for blocking and 290 seconds for pipelining.

The *sort* phase in the map task minimizes the merging work that reduce tasks must perform at the end of the *shuffle* phase. When pipelining is enabled, the *sort* phase is avoided since map tasks have already sent some fraction of the spill files to concurrently running reduce tasks. Therefore, pipelining increases the merging workload placed on the reducer. The adaptive pipelining scheme described in Section 3.1.3 attempts to ensure that reduce tasks are not overwhelmed with additional load.

We used two Amazon EC2 clusters depending on the size of the experiment: "small" jobs used 10 worker nodes, while "large" jobs used 20. Each node was an "extra large" EC2 instances with 15GB of memory and four virtual cores.

6.2 Small Job Results

Our first experiment focused on the performance of small jobs in an underutilized cluster. We ran a 10GB wordcount with a 512MB block size, yielding 20 map tasks. We used 10 worker nodes and configured each worker to execute at most two map and two reduce tasks simultaneously. We ran several experiments to compare the performance of blocking and pipelining using different numbers of reduce tasks.

Figure 7 reports the results with five reduce tasks. A plateau can be seen at 75% progress for both blocking and pipelining. At this point in the job, all reduce tasks have completed the *shuffle* phase; the plateau is caused by the time taken to perform a final merge of all map output before entering the *reduce* phase. Notice that the plateau for the pipelining case is shorter. With pipelining, reduce tasks receive map outputs earlier and can begin sorting earlier, thereby reducing the time required for the final merge.

Figure 8 reports the results with twenty reduce tasks. Using more reduce tasks decreases the amount of merging that any one reduce task must perform, which reduces the duration of the plateau at 75% progress. In the blocking case, the plateau is practically gone.

Note that in both experiments, the map phase finishes faster with blocking than with pipelining. This is because pipelining allows reduce tasks to begin executing more quickly; hence, the reduce tasks compete for resources with the map tasks, causing the map phase to take slightly

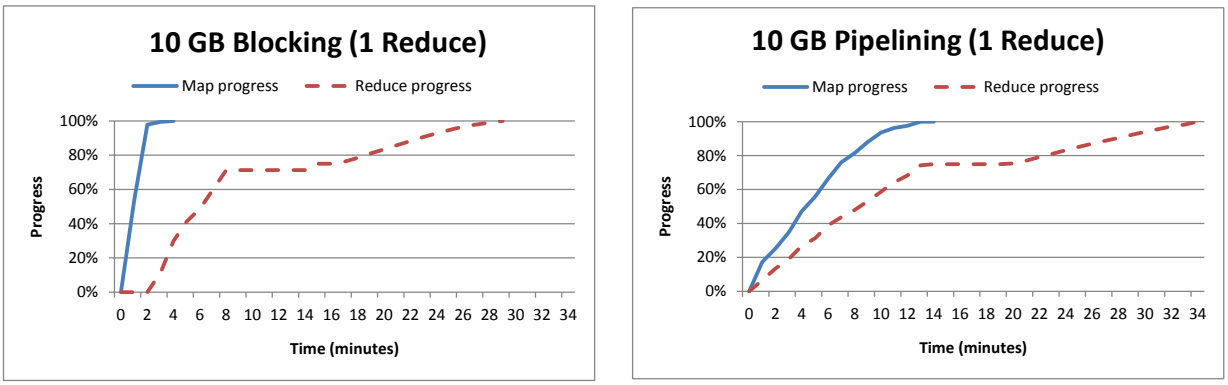


Figure 9: CDF of map and reduce task completion times for a 10GB wordcount job using 20 map tasks and 1 reduce task (512MB block size). The total job runtimes were 29 minutes for blocking and 34 minutes for pipelining.

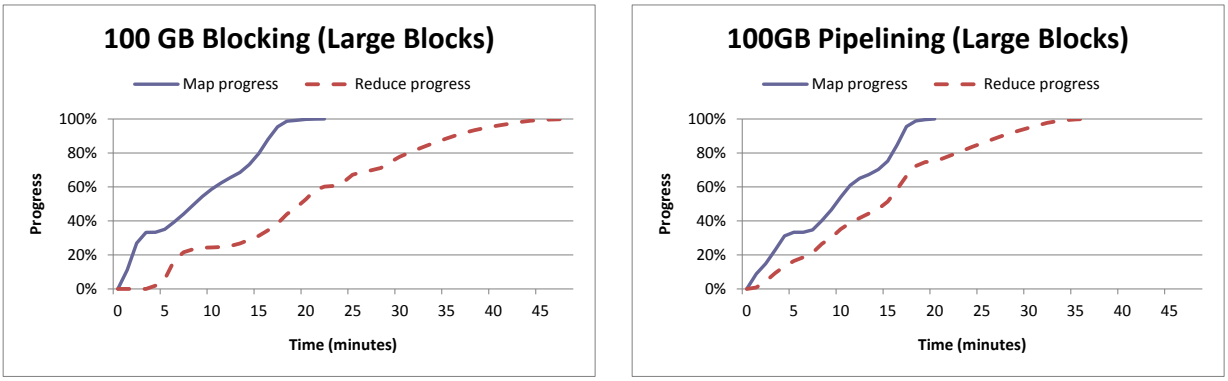


Figure 10: CDF of map and reduce task completion times for a 100GB wordcount job using 240 map tasks and 60 reduce tasks (512MB block size). The total job runtimes were 48 minutes for blocking and 36 minutes for pipelining.

longer. In this case, the increase in map duration is outweighed by the increase in cluster utilization, resulting in shorter job completion times: pipelining reduced completion time by 17.7% with 5 reducers and by 19.7% with 20 reducers.

Figure 9 describes an experiment in which we ran a 10GB wordcount job using a single reduce task. This caused job completion times to increase dramatically for both pipelining and blocking, because of the extreme load placed on the reduce node. Pipelining delayed job completion by ~17%, which suggests that our simple adaptive flow control scheme (Section 3.1.3) was unable to move load back to the map tasks aggressively enough.

6.3 Large Job Results

Our second set of experiments focused on the performance of somewhat larger jobs. We increased the input size to 100GB (from 10GB) and the number of worker nodes to 20 (from 10). Each worker was configured to execute at most four map and three reduce tasks, which meant that at most 80 map and 60 reduce tasks could execute at once. We conducted two sets of experimental runs, each run comparing blocking to pipelining using either large (512MB) or small (32MB) block sizes. We were interested in blocking performance with small block sizes because blocking can effectively emulate pipelining if the block size is small enough.

Figure 10 reports the performance of a 100GB wordcount job with 512MB blocks, which resulted in 240 map tasks, scheduled in three waves of 80 tasks each. The 60 reduce tasks were coscheduled with the first wave of map tasks. In the blocking case, the reduce tasks began working as soon as they received the output of the first wave, which is why the reduce progress begins to climb around four minutes (well before the completion of all maps). Pipelining was able to achieve significantly better cluster utilization, and hence reduced job completion time by ~25%.

Figure 11 reports the performance of blocking and pipelining using 32MB blocks. While the performance of pipelining remained similar, the performance of blocking improved considerably, but still trailed somewhat behind pipelining. Using block sizes smaller than 32MB did

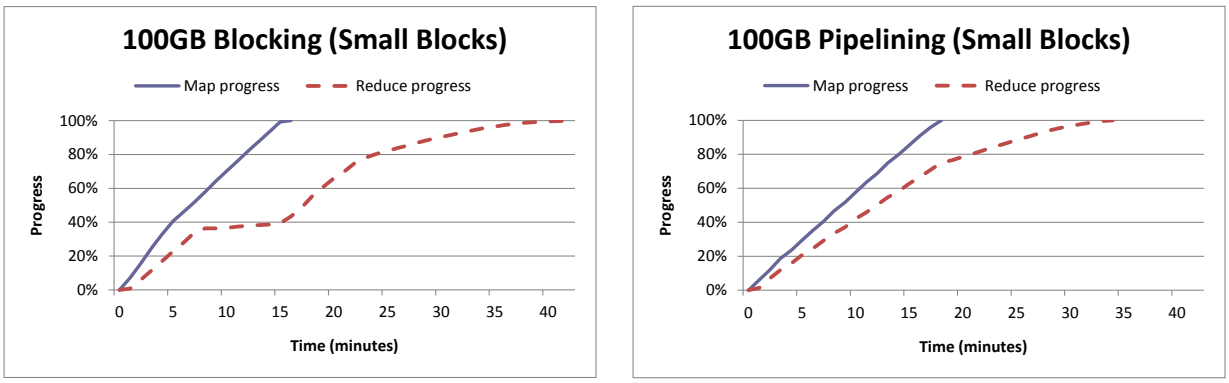


Figure 11: CDF of map and reduce task completion times for a 100GB wordcount job using 3120 map tasks and 60 reduce tasks (32MB block size). The total job runtimes were 42 minutes for blocking and 34 minutes for pipelining.

not yield a significant performance improvement in our experiments.

7 Related Work

The work in this paper relates to literature on parallel dataflow frameworks, online aggregation, and continuous query processing.

7.1 Parallel Dataflow

Dean and Ghemawat's paper on Google's MapReduce [6] has become a standard reference, and forms the basis of the open-source Hadoop implementation. As noted in Section 1, the Google MapReduce design targets very large clusters where the probability of worker failure or slowdown is high. This led to their elegant checkpoint/restart approach to fault tolerance, and their lack of pipelining. Our work extends the Google design to accommodate pipelining without significant modification to their core programming model or fault tolerance mechanisms.

Dryad [13] is a data-parallel programming model and runtime that is often compared to MapReduce, supporting a more general model of acyclic dataflow graphs. Like MapReduce, Dryad puts disk materialization steps between dataflow stages by default, breaking pipelines. The Dryad paper describes support for optionally "encapsulating" multiple asynchronous stages into a single process so they can pipeline, but this requires a more complicated programming interface. The Dryad paper explicitly mentions that the system is targeted at batch processing, and not at scenarios like continuous queries.

It has been noted that parallel database systems have long provided partitioned dataflow frameworks [21], and recent commercial databases have begun to offer MapReduce programming models on top of those frameworks [5, 10]. Most parallel database systems can provide pipelined execution akin to our work here, but they use a more tightly coupled iterator and *Exchange* model that keeps producers and consumers rate-matched via queues, spreading the work of each dataflow stage across all nodes in the cluster [8]. This provides less scheduling flexibility than MapReduce and typically offers no tolerance to mid-query worker faults. Yang et al. recently proposed a scheme to add support for mid-query fault tolerance to traditional parallel databases, using a middleware-based approach that shares some similarities with MapReduce [31].

Logothetis and Yocum describe a MapReduce interface over a continuous query system called *Mortar* that is similar in some ways to our work [16]. Like HOP, their mappers push data to reducers in a pipelined fashion. They focus on specific issues in efficient stream query processing, including minimization of work for aggregates in overlapping windows via special reducer APIs. They are not built on Hadoop, and explicitly sidestep issues in fault tolerance.

Hadoop Streaming is part of the Hadoop distribution, and allows map and reduce functions to be expressed as UNIX shell command lines. It does not stream data through map and reduce phases in a pipelined fashion.

7.2 Online Aggregation

Online aggregation was originally proposed in the context of simple single-table SQL queries involving "Group By" aggregations, a workload quite similar to MapReduce [12]. The focus of the initial work was on providing not only "early returns" to these SQL queries, but also statistically robust estimators and confidence interval metrics for the final result based on random sampling. These statistical matters do not generalize to arbitrary MapReduce jobs, though our framework can support those that have been developed. Subsequently, online aggregation was extended to handle join queries (via the *Ripple Join* method),

and the *CONTROL* project generalized the idea of online query processing to provide interactivity for data cleaning, data mining, and data visualization tasks [11]. That work was targeted at single-processor systems. Luo et al. developed a partitioned-parallel variant of Ripple Join, without statistical guarantees on approximate answers [17].

In recent years, this topic has seen renewed interest, starting with Jermaine et al.'s work on the *DBO* system [15]. That effort includes more disk-conscious online join algorithms, as well as techniques for maintaining randomly-shuffled files to remove any potential for statistical bias in scans [14]. Wu et al. describe a system for peer-to-peer online aggregation in a distributed hash table context [29]. The open programmability and fault-tolerance of MapReduce are not addressed significantly in prior work on online aggregation.

An alternative to online aggregation combines precomputation with sampling, storing fixed samples and summaries to provide small storage footprints and interactive performance [7]. An advantage of these techniques is that they are compatible with both pipelining and blocking models of MapReduce. The downside of these techniques is that they do not allow users to choose the query stopping points or time/accuracy trade-offs dynamically [11].

7.3 Continuous Queries

In the last decade there was a great deal of work in the database research community on the topic of continuous queries over data streams, including systems such as Borealis [1], STREAM [18], and Telegraph [4]. Of these, Borealis and Telegraph [24] studied fault tolerance and load balancing across machines. In the Borealis context this was done for pipelined dataflows, but without partitioned parallelism: each stage ("operator") of the pipeline runs serially on a different machine in the wide area, and fault tolerance deals with failures of entire operators [3]. SBON [22] is an overlay network that can be integrated with Borealis, which handles "operator placement" optimizations for these wide-area pipelined dataflows.

Telegraph's *FLuX* operator [24, 25] is the only work to our knowledge that addresses mid-stream fault-tolerance for dataflows that are both pipelined and partitioned in the style of HOP. FLuX ("Fault-tolerant, Load-balanced eXchange") is a dataflow operator that encapsulates the shuffling done between stages such as map and reduce. It provides load-balancing interfaces that can migrate operator state (e.g., reducer state) between nodes, while handling scheduling policy and changes to data-routing policies [25]. For fault tolerance, FLuX develops a solution based on process pairs [24], which work redundantly to ensure that operator state is always being maintained live on multiple nodes. This removes any burden on the continuous query programmer of the sort we describe in Section 5. On the other hand, the FLuX protocol is far more complex and resource-intensive than our pipelined adaptation of Google's checkpoint/restart tolerance model.

8 Conclusion and Future Work

MapReduce has proven to be a popular model for large-scale parallel programming. Our Hadoop Online Prototype extends the applicability of the model to pipelining behaviors, while preserving the simple programming model and fault tolerance of a full-featured MapReduce framework. This provides significant new functionality, including "early returns" on long-running jobs via online aggregation, and continuous queries over streaming data. We also demonstrate benefits for batch processing: by pipelining both within and across jobs, HOP can reduce the time to job completion.

In considering future work, scheduling is a topic that arises immediately. Stock Hadoop already has many degrees of freedom in scheduling batch tasks across machines and time, and the introduction of pipelining in HOP only increases this design space. First, pipeline parallelism is a new option for improving performance of MapReduce jobs, but needs to be integrated intelligently with both intra-task partition parallelism and speculative redundant execution for "straggler" handling. Second, the ability to schedule deep pipelines with direct communication between reduces and maps (bypassing the distributed file system) opens up new opportunities and challenges in carefully co-locating tasks from different jobs, to avoid communication when possible.

Olston and colleagues have noted that MapReduce systems—unlike traditional databases—employ "model-light" optimization approaches that gather and react to performance information during runtime [19]. The continuous query facilities of HOP enable powerful introspective programming interfaces for this: a full-featured MapReduce interface can be used to script performance monitoring tasks that gather system-wide information in near-real-time, enabling tight feedback loops for scheduling and dataflow optimization. This is a topic we plan to explore, including opportunistic methods to do monitoring work with minimal interference to outstanding jobs, as well as dynamic approaches to continuous optimization in the spirit of earlier work like Eddies [2] and FLuX [25].

As a more long-term agenda, we want to explore using MapReduce-style programming for even more interactive applications. As a first step, we hope to revisit interactive data processing in the spirit of the CONTROL work [11], with an eye toward improved scalability via parallelism. More aggressively, we are considering the idea of bridging the gap between MapReduce dataflow programming and lightweight event-flow programming models like SEDA [28]. Our HOP implementation's roots

in Hadoop make it unlikely to compete with something like SEDA in terms of raw performance. However, it would be interesting to translate ideas across these two traditionally separate programming models, perhaps with an eye toward building a new and more general-purpose framework for programming in architectures like cloud computing and many-core.

Acknowledgments

We would like to thank Daniel Abadi, Kuang Chen, Mosharaf Chowdhury, Akshay Krishnamurthy, Andrew Pavlo, Hong Tang , and our shepherd Jeff Dean for their helpful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant Nos. 0713661, 0722077 and 0803690, the Air Force Office of Scientific Research under Grant No. FA95500810352, the Natural Sciences and Engineering Research Council of Canada, and gifts from IBM, Microsoft, and Yahoo!.

References

[1] ABADI, D. J., AHMAD, Y., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., LINDNER, W., MASKEY, A. S., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. The design of the Borealis stream processing engine. In *CIDR* (2005).

[2] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. In *SIGMOD* (2000).

[3] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD* (2005).

[4] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR* (2003).

[5] CIESLEWICZ, J., FRIEDMAN, E., AND PAWLOWSKI, P. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. In *VLDB* (2009).

[6] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004).

[7] GIBBONS, P. B., AND MATIAS, Y. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD* (1998).

[8] GRAEFE, G. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD* (1990).

[9] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov. 1*, 1 (1997), 29–53.

[10] Greenplum: A unified engine for RDBMS and MapReduce, Oct. 2008. Downloaded from `http://www.greenplum.com/download.php?alias=register-map-reduce&file=Greenplum-MapReduce-Whitepaper.pdf`.

[11] HELLERSTEIN, J. M., AVNUR, R., CHOU, A., HIDBER, C., OLSTON, C., RAMAN, V., ROTH, T., AND HAAS, P. J. Interactive data analysis with CONTROL. *IEEE Computer 32*, 8 (Aug. 1999).

[12] HELLERSTEIN, J. M., HAAS, P. J., AND WANG, H. J. Online aggregation. In *SIGMOD* (1997).

[13] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys* (2007).

[14] JERMAINE, C. Online random shuffling of large database tables. *IEEE Trans. Knowl. Data Eng. 19*, 1 (2007), 73–84.

[15] JERMAINE, C., ARUMUGAM, S., POL, A., AND DOBRA, A. Scalable approximate query processing with the DBO engine. In *SIGMOD* (2007).

[16] LOGOTHETIS, D., AND YOCUM, K. Ad-hoc data processing in the cloud (demonstration). *Proc. VLDB Endow. 1*, 2 (2008).

[17] LUO, G., ELLMANN, C. J., HAAS, P. J., AND NAUGHTON, J. F. A scalable hash ripple join algorithm. In *SIGMOD* (2002).

[18] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, resource management, and approximation in a data stream management system. In *CIDR* (2003).

[19] OLSTON, C., REED, B., SILBERSTEIN, A., AND SRIVASTAVA, U. Automatic optimization of parallel dataflow programs. In *USENIX Technical Conference* (2008).

[20] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD* (2008).

[21] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *SIGMOD* (2009).

[22] PIETZUCH, P., LEDLIE, J., SHNEIDMAN, J., ROUSSOPOULOS, M., WELSH, M., AND SELTZER, M. Network-aware operator placement for stream-processing systems. In *ICDE* (2006).

[23] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4 (2005), 277–298.

[24] SHAH, M. A., HELLERSTEIN, J. M., AND BREWER, E. A. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD* (2004).

[25] SHAH, M. A., HELLERSTEIN, J. M., CHANDRASEKARAN, S., AND FRANKLIN, M. J. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE* (2003).

[26] SKOMOROCH, P. N. Wikipedia page traffic statistics, 2009. Downloaded from `http://developer.amazonwebservices.com/connect/entry.jspa?externalID=2596`.

[27] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. Hive — a warehousing solution over a Map-Reduce framework. In *VLDB* (2009).

[28] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP* (2001).

[29] WU, S., JIANG, S., OOI, B. C., AND TAN, K.-L. Distributed online aggregation. In *VLDB* (2009).

[30] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (2009).

[31] YANG, C., YEN, C., TAN, C., AND MADDEN, S. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *ICDE* (2010).