StreamReduce: Processing Data Streams and Avoiding Work

Ryan Lopopolo
Massachusetts Institute of Technology
lopopolo@mit.edu

May 7, 2012

1 Introduction

One of the fundamental problems MapReduce was created to solve was processing the growing amount of data in the world. Now with the open source Apache Hadoop and Pig [11], it is easy to process and query the terabytes of data an organization may collect. With terabytes of data, however, it is scarcely necessary to examine every entry to uncover trends, especially when one only cares about the most prominent trends or results. This type of workflow lends itself naturally to top-k-style computations, data sampling tasks, and other cases where perfect-fidelity results are not required.

How does one decide which data to process then? Selctively deciding which data to log is problematic because it can systemically mask some of the trends one would try to uncover through analysis. In this paper, I explore the possibility of discarding data randomly while trying to maintain a target accuracy level. On the subset of data that I do process, I explore using computation routines of differing fidelities to sample the data. Combining both of these techniques allows the cluster to avoid performing work.

One of the problems with MapReduce is how to incorporate new data into an already analyzed task. MapReduce simply says to rerun the job with the new data included. Others have solved this problem by batching data into chunks and continuously spawning MapReduce jobs to incorporate the new data and incrementally storing updates in an external database instead of relying on the file system for enforcing the synchronization barrier.

A simpler way to incorporate new data is to treat data not as a constant, but as a stream. By processing data on demand, one avoids repeating work involved in reprocessing data, can save resources by spinning workers up and down as necessary, and ensures results are as fresh as possible by incorporating data into the result as soon as it arrives.

The goal of StreamReduce is to provide a MapReduce-style framework for processing streams. Such a framework should include the ability to kill or discard tasks and be capable of providing workers with serveral different computation routines to choose among.

The remainder of this paper is organized as follows. Section 2 provides an overview of related work on distributed stream pro-

cessing as well as dynamic and approximate computations. Section 3 details the design of StreamReduce. Section 4 explains the methodology used to perform a case study using StreamReduce. Section 5 evaluates the effectiveness of StreamReduce in the case study. Section 6 enumerates issues with StreamReduce's current implementation.

2 Related Work

StreamReduce builds of previous work in the areas of MapReduce, Data Stream Management Systems, and approximation algorithms.

2.1 MapReduce

There is a rich research field around MapReduce. This section details work related to adapting MapReduce to a streaming environment.

The Hadoop Online Prototype [8] adds streaming to the MapReduce framework by removing the materialize phase between mappers and reducers and between phases of a MapReduce pipeline. Results are streamed from mappers to reducers as they become available. At any given point, each reducer maintains an in-progress snapshot of the MapReduce job's progress up to that point. This removes the synchronization barriers inherient in MapReduce and allows incremental progress to be observed.

This approach is in contrast to the continuous query approach taken by many other streaming systems.

2.2 Data Stream Management Systems

In their overview on data stream projects [6], a new class of data streaming applications called *Data Stream Management Systems* (*DSMS*es) is discussed. StreamReduce is a type of DSMS that includes some properties found in other systems.

The *Chronicle Data Model* [10] describes an append-only stream of tuples. Such streams lend themselves to incremental reads and incremental processing. In addition to incremental updates, the Chronicle Data Model also seeks to update results incrementally without storing the data itself.

The *Data Stream Model* [6] is a refinement of the Chronicle Data Model. It has the following properties: stream items may arrive out of order, streams are potentially unbounded in size, elements in the stream are discared after they are processed. This is the data model that StreamReduce uses.

The Aurora system [7] introduces the concept of *load shedding*, or compensating for resource overload by comparing job specific quality of service metrics with target levels. StreamReduce aims to incorporate load shedding to dynamically tune jobs in the face of impending time and resource constraints.

Aurora also introduces the non-blocking operators `filter`, `map`, and `union` for manipulating streams. StreamReduce implements these operators at the worker and fetcher nodes.

2.3 Approximation Algorithms

An alternative to load shedding to cope with high stream throughput is to perform approximate computations on stream data.

Dynamic Knobs [9] describes a mechanism to explore the accuracy-performance tradeoff

space to adjust dynamically adjust program tuning parameters to changes in the execution environment. This allows programs to do things such as utilize more resources if they become available, use less power if power rates spike, or use less accurate computations in the face of an impending time constraint.

The most important piece Dyanmic Knobs offers to the DSMS space is dynamic *peak load provisioning*. Their system can adjust downward the amount of resources, machines, etc. required for normal load and quickly adjust these levels back up to handle load spikes. Combined with Aurora's load shedding, peak load provisioning can be used to create a DSMS that uses less resources in both the normal and peak load states.

Accuracy-aware transformations [12] are a class of transformations that alter a computation according to a set of probabilstic weights and an accuracy target such that the computation performs more efficiently. StreamReduce uses the subclass of transformations called *substitution transformations* in worker nodes to adjust the runtime characteristics of per-datum computation.

3 Design

StreamReduce is a distributed system for processing streams that can choose to kill tasks or limit the amount of work it performs per task. Its design is modular and there are five different types of nodes in a StreamReduce cluster: spout, fetcher, worker, collector, master.

3.1 Spout

Spouts are network-connected machines that generate data for a StreamReduce cluster to process. Examples include a machine that tails a log file or an API like the Twitter tweet stream. Fetchers depend on spouts as a source of streaming data.

3.2 Fetcher

Fetchers are responsible for communicating with spouts and returning data for the workers to process. This level of indirection allows for a variety of different configurations: A spout can emit data at such a high rate that using multiple fetchers returns more data to the cluster, multiple fetchers can each talk to a different spout, for example tailing logs from multiple machines, or fetchers can be set up redundantly to ensure that every datum emitted by a spout is processed.

Fetchers may run indefinitely or return a status code indicating the fetching task is over.

3.3 Worker

Workers perform processing on data provided by fetchers. They are similar to mappers in the MapReduce framework. Each worker receives a fraction of the total input data and each worker's tasks are independent of other workers'.

Workers are responsible for fulfilling the main design goals of StreamReduce. Workers may choose to kill a task preemptively or, if it does not complete before a timeout, greedily. Additionally, workers may choose among a variety of routines for performing computation on a datum. Routines are ranked on a [0,1] scale based on the estimated fraction of total work it requires. This rank is a proxy for the accuracy of a routine.

As it runs, a worker maintains an internal metric that estimates the fraction of work it has performed over all data it has seen. Workers use this metric when choosing which

computation routine to run for a datum. A worker will choose the lowest ranked routine such that its estimated fraction of work is above a configured threshold. The total work estimator is used as a proxy for accuracy when a worker updates this metric. This metric also takes into account the number of tasks it has killed for any reason.

Because one of StreamReduce's goals is to process (potentially infinite) streams, workers do not save their progress to disk and are forgetful; once the master queries a worker for its progress and the worker successfully returns it, the worker clears all of its state and continues processing. As a consequence, workers are easy to add or remove from the worker pool because they don't need to be initialized with long-term job state. Additionally, as long as the master polls workers frequently enough, only a small fraction of the total work is lost upon worker failure. The master queries workers every 100ms by default.

3.4 Collector

Collector nodes aggregate the results of all workers. Each collector acts as a reducer would in the MapReduce framework if MapReduce used a single reducer for all of the mappers. Collectors differ from MapReduce reducers in that they receive their data piecewise, as it is produced from workers, as opposed to after all mappers have completed. This limits the types of stateless aggregation collectors can do to online operations. (Of course, a collector can choose to retain copies of individual worker results and re-reduce them each time more results are receieved, but this is not in the spirit of the framework.)

Collectors also expose an API endpoint for accessing the results of a job. This endpoint can be used by another spout to chain StreamReduce jobs together.

3.5 Master

There is one master node per StreamReduce network. It has a function similar to the jobtracker in MapReduce. New jobs are scheduled with the master and it initializes other nodes in the cluster with job state. For each job, the master schedules roles among nodes in a round robin manner to minimize the total number of outstanding roles on any one node.

The master is the only well-known node in the cluster. Other nodes in the network exclusively communicate with the master. The master and other nodes communicate with each other by issuing HTTP requests to a simple server on each node. I decided to make use of a library [4] providing the server functionality because there was a low barrier to getting inter-node communication functioning. This design choice presents some problems and will be discussed further in Section 6.

Another consequence of the master being the only well-known node is that all data in the cluster must flow through it. The master maintains two queues per job for buffering job progress from fetchers and workers and periodically pushes this state out to workers and collectors, respectively.

3.6 Configuring a Cluster for a Job

Apart from spouts, all nodes in a cluster modify their behavior based on the job they are running. The master must know the appropriate number of nodes to spin up; fetchers need to know how to talk to spouts; work-

ers need to know how to compute results; and collectors need to know how to combine batched worker results.

StreamReduce achieves this configuration with a Jobfile. Jobfiles are instances of a special class provided by the StreamReduce framework. This class exposes several methods that can be invoked by a given node to provide configuration parameters or execute code on its behalf. Most of the routines that advance the progress of a StreamReduce job are wrappers around these Jobfile methods.

StreamReduce sends the source of the Jobfile to every node participating in the job and uses eval to obtain a local instance of the Jobfile class. An example jobfile can be found at [2].

The use of eval to instantiate Jobfile objects and the implementation language's (ruby) ability to monkeypatch classes means that jobs can be modified on the fly just by sending an updated Jobfile to job participants. For example, a job can be made lower fidelity in the face of an impending time constraint, be made higher fidelity if more workers are added to the cluster, or gracefully kill all fetchers which allows the job to end after all of the various job queues are empty.

3.7 Fault Tolerance

StreamReduce can sustain the failure of any node other than the master. Because the master is responsible for shuffling data between fetchers, workers, and collectors, a job cannot make progress without the master.

Fetchers can fail. The only consequence of fetcher failure is that some of the input stream is lost forever. Because StreamReduce uses the Data Stream Model, it does not attempt to recover this data.

Worker failures may entail some lost work. Having the master poll each worker frequently for its progress mitigates the consequences of worker failure. For jobs that run for minutes or hours, 100ms of lost work is $\ll 0.1\%$ of the total. StreamReduce can simply treat these lost tasks as preemptively killed, which is already expected by the cluster.

Each collector is an exact replica of all others. Collector failure does not impact correctness of the results delivered by any other collector. If no collectors are online, the data meant for them sits in a queue on the master. Bringing another collector online will allow these and future results to be aggregated.

3.8 Killing Tasks and Working Less

A design goal of StreamReduce is to compute meaningful results from a stream and to do so while not processing every datum one-to-one. StreamReduce achieves this goal in two different ways. First, workers may kill tasks preemptively to avoid the time and resource costs associated with processing that task. Second, workers rely on the Jobfile to provide them with several different computation routines of differing fidelities that perform a fraction of the total work on a datum.

These behaviors appeal to the law of large numbers. It assumes that trends will still be visible in spite of looking at only a fraction of the data. This means that the computation routines and the weights assigned to them must be adapted to fit the distribution of the data before the job starts (or updated *in situ* via a new Jobfile).

4 Methodology

The cluster I used to test StreamReduce is a 5-node network consisting of 1 spout, 1

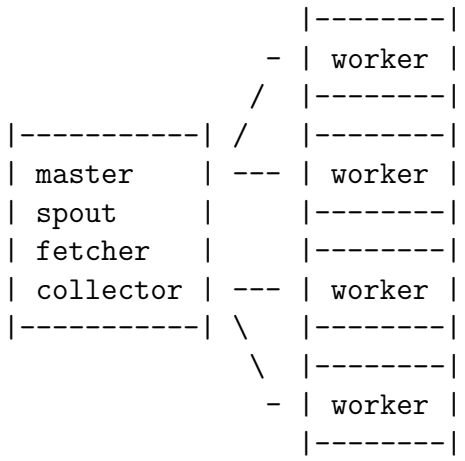

Figure 1: Topology of the test StreamReduce cluster used in the word count case study. Each box represents a single machine.

fetcher, 4 workers, 1 collector, and 1 master. Roles were partitioned among nodes as shown in Figure 1. Worker nodes were XVM [5] VMs provisioned with 128MB RAM each. The master, spout, fetcher, and collector ran on a mid-2010 Macbook Air laptop with 4GB of RAM.

The job I ran was a word counter. The collector emits a hash whose keys are words and whose values are the frequency the key occurred in the test corpus.

To run the job, I had to create a spout and find a source of data. For data, I acquired a Wikipedia dump of the Featured Articles category. I used Wikipedia's xml exporter [1] on January 9, 2012.

To serve the data, I munged it into a YAML file containing a hash of

$$\texttt{article\_id} \rightarrow \texttt{fulltext}.$$

To store fulltext, I base64 encoded it and deflated it using zlib. This requires workers to inflate and decode the fulltext before they can process it further. The server reads from this YAML file and returns the resulting hash as JSON.

Weight	Description
1.0	Deflate and decode fulltext and count words in all of fulltext
0.25	Deflate and decode fulltext and count words in approximately a quarter of fulltext
0.1	Return the results of the last computation from a cache

Table 1: Computation routines used by StreamReduce workers.

The fetcher communicates to the spout by requesting sequential article ids until it has fetched every article.

Workers are initialized with the 3 computation routines shown in Table 1.

The results of the word count job were then used in a script to compute the top 100 words. The cluster and word count job could be used in a two-stage top-k job. This script filters out SEO stopwords [3] to achieve more interesting results.

5 Evaluation

To evaluate StreamReduce, I explore the precision and recall of the word count test job as well as the speedup factor associated with varying the task kill rate and target work level.

5.1 Precision and Recall

In order for task killing to be an effective strategy for a top-k computation, the distribution of words in the corpus must be able to sustain data loss. This is the case if the distribution is long-tailed, but is not true if the distribution is uniform.

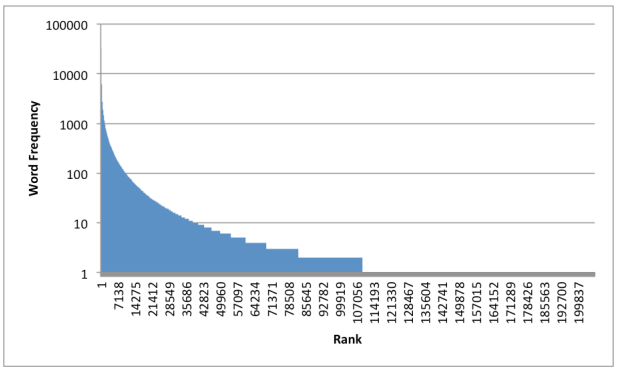

Figure 2: The frequencies of words in the Wikipedia Featured Articles corpus have a long-tailed distribution.

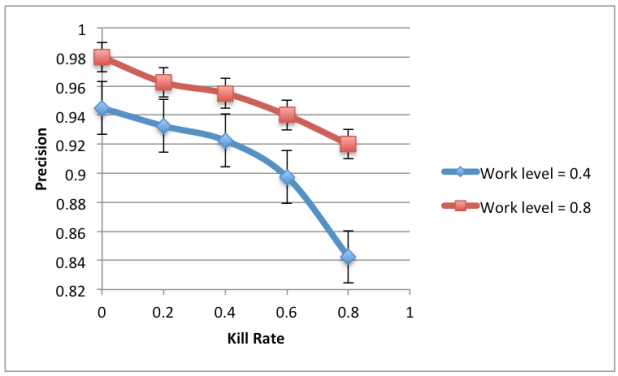

Figure 3: Fraction of the top 100 results returned in the top 100 by the word count job.

The distribution shown in Figure 2 is long-tailed. This implies that killing tasks will impact the tail of the distribution more than the head.

5.1.1 Precision

Task killing should least impact the head of the word distribution. Figure 3 shows that even with a task kill rate of 0.8 and a work level of 0.4, meaning that the cluster only looks at 8% of the words in the corpus, the results are at least 80% accurate. Increasing the work level, which increases the fidelity of results emitted by workers, also increases the fidelity of the top 100 results.

5.1.2 Word Count Job Accuracy

Because I am running a top-k job, I compute the $F_{0.5}$-score of the StreamReduce results as opposed to the $F1$-score because I care more about precision of the top k than recall in the tail. Figure 4 shows that the $F_{0.5}$-score, or the weighted accuracy of the top-100 classifier, is always higher than the fraction of the input examined (task kill rate * work level). Setting the work level to 0.4 nets a greater increase in

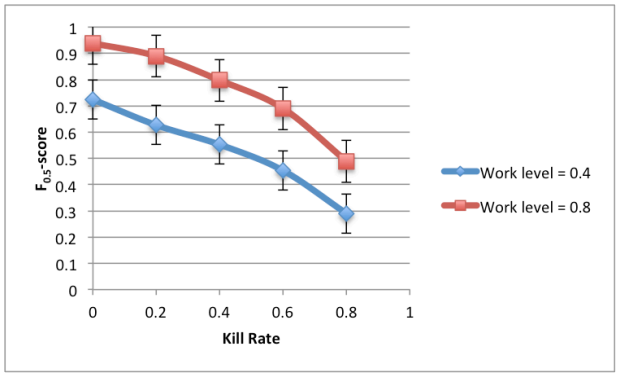

Figure 4: $F_{0.5}$-score for the top-100 word count job. I measured the F-score with $\beta = 0.5$ because the goal of the job is to run a top-k filter on it, meaning I care more about precision than recall.

job accuracy versus fraction of words examined, with accuracy approximately 1.75 times higher than the fraction of words examined.

5.2 Speedup

StreamReduce is able to achieve a speedup over a MapReduce-style computation through three mechanisms: providing multiple computation routines for the workers to choose from; varying the task kill rate; and varying the target work level, which influences the chosen frequency of computation

Task kill rate	Work level	Average execution time [minutes]
0.0	1.0	11:35.5
0.0	0.4	10:33.8
0.0	0.8	10:30.8
0.2	0.4	08:25.8
0.2	0.8	08:37.8
0.4	0.4	06:54.0
0.4	0.8	06:28.2
0.6	0.4	04:14.7
0.6	0.8	04:13.7
0.8	0.4	02:33.3
0.8	0.8	02:15.0

Table 2: Impact of task kill rate and work level on job execution time.

routines per worker.

Table 2 shows the effect of varying the task kill rate and work level. The row with kill rate = 0.0 and work level = 1.0 represents the perfect, MapReduce-style, execution of the word count job. The only consistent reduction in execution time is a result of increasing the task kill rate. Increasing the task kill rate exhibits a linear reduction in execution time as the fraction of processed data linearly decreases.

Notably, simply by killing tasks, StreamReduce is able to achieve over a 5 times speedup compared to the perfect execution while still returning about 84 of the top 100 words.

6 Implementation Notes

This section describes deficiencies in the implementation of StreamReduce used to run the case study in this paper and goals for the next version of the system.

6.1 On-the-Fly Job Reconfiguration

The mechanism for altering execution of a job as it is running described in Section 3.6 is not yet implemented. Adding this feature to StreamReduce is a matter of adding an API endpoint to the servers run by each role and adding a bin-script to submit the new Jobfile.

6.2 Internode Communication

In Section 3.5 I mentioned design difficulties relating to the master and using HTTP requests for sending messages between nodes. I initially chose to use HTTP requests with POST payloads to send data and results between nodes because of its low barrier to entry; however, initiating many HTTP requests in succession is expensive. When killing jobs on worker nodes, the observed time savings was minimal because most of the time spent per datum involved shipping it from the master to the worker.

In order to mitigate the effects of this problem, I batched individual requests into groups of 5. This lowered the number of connections I was initiating but it also limited the parallelism of the cluster and removed workers from the pool for longer periods because they had more data to process at a time. A killed task on the worker no longer freed it up to receive a new task immediately. Keeping workers removed from the pool caused the queues on the master to grow longer as the workers could no longer keep up with the rate the fetcher was producing data.

Eventually, I introduced a configuration option to kill tasks at the master. The master then decides to discard a task as it is popped from the queue, before it is ever sent to a worker. Doing so improved the execution time of the cluster but also meant that work-

ers no longer had a full idea of how accurate they were being because they never killed any tasks themselves.

In order to address this problem, the master needs to keep a persistent channel open between it and all active nodes. Communication channels should be established when nodes come online. Possible solutions include writing to a socket or using a higher abstraction like AMPQ. AMPQ is appealing because it also relieves the master from handling queues.

6.3 Node Health

Currently, the master does not probe nodes to ensure that they are responsive. In the case that a node is serving a worker role for a job, this deficiency is masked. The node will simply not return the HTTP request sent to it to push it some data and the worker will remain removed from the pool. If the node is in the role of fetcher or collector, the master will stall.

All messages passed by the master to other nodes must be made non-blocking.

6.4 Job Management

There is no support for killing jobs or removing them from the master's data structures. The only way to remove a job is to kill the master and all other nodes in the cluster.

7 Conclusion

I have extended the ideas of approximate computations to a MapReduce-style framework and shown that it is feasible to tune the input parameters of a job to achieve sufficient precision and recall for a particular application.

I have taken a different approach to streaming MapReduce by treating data not as a batch, but as a stream. At any given point in the execution of a job, the collectors will have accurate (so far as the job parameters allow) results for data seen until that point. This approach guarantees freshness and makes it easier to incorporate new data into an existing computation, yet it suffers from being limited to online aggregations.

The StreamReduce source code is open source and available at:

`https://github.com/lopopolo/sr`

Acknowledgments

I would like to acknowledge Martin Rinard for giving me direction for my initial idea and guiding me to apply the data stream model to StreamReduce; Sasa Misailovic for being a sounding board for ideas and providing me with background research; Daniel Erenrich for providing information about data stream management systems; and SIPB for providing the XVM service, which made deploying a StreamReduce cluster painless.

References

[1] Export pages. `http://en.wikipedia.org/wiki/Special:Export`. Accessed January 12, 2012.

[2] lopopolo/sr. `https://github.com/lopopolo/sr`. Code repository for sr.

[3] Online seo guide: Google stop words - a comprehensive list of words google ignores. `http://www.link-assistant.com/seo-stop-words.html`. Accessed April 24, 2012.

[4] Sinatra. `http://sinatrarb.com`. Accessed May 4, 2012.

[5] XVM—virtual servers for MIT. `http://xvm.mit.edu/`. Accessed January 16, 2012.

[6] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[7] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 215–226. VLDB Endowment, 2002.

[8] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[9] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 199–212, New York, NY, USA, 2011. ACM.

[10] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '95, pages 113–124, New York, NY, USA, 1995. ACM.

[11] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[12] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 441–454, New York, NY, USA, 2012. ACM.