Online Aggregation and Continuous Query support in MapReduce

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

John Gerth, Justin Talbot
Stanford University

Khaled Elmeleegy, Russell Sears
Yahoo! Research

ABSTRACT

MapReduce is a popular framework for data-intensive distributed computing of batch jobs. To simplify fault tolerance, the output of each MapReduce task and job is *materialized* to disk before it is consumed. In this demonstration, we describe 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 demonstrate 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.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.11 [**Software Architectures**]: Data abstraction

General Terms

Design

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, coordination and fault tolerance to be handled by the MapReduce framework.

The MapReduce model is typically applied to large batch-oriented computations that are concerned primarily with time to job completion. The Google MapReduce framework [4] and open-source Hadoop system reinforce this usage model through a batch-processing implementation strategy: the entire output of each map and reduce stage is *materialized* to stable storage before it can be consumed by the next stage, or produce output. Batch 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) [3], a pipelining version of Hadoop.

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

- A downstream dataflow element can begin consuming data before a producer element has finished execution, which can increase opportunities for parallelism, improve utilization, and reduce response time.

- 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* [6], can reduce the turnaround time for data analysis by several orders of magnitude. We describe our demonstration of online aggregation using our pipelined MapReduce architecture in Section 4.1.1.

- Pipelining widens the domain of problems to which MapReduce can be applied. In Section 4.2.1, we describe a system monitoring tool written as a MapReduce job that leverages HOP's support of *continuous queries*: MapReduce jobs that run continuously, accepting new data as it arrives and analyzing it immediately.

Pipelining raises several design challenges that we describe in this the demonstration. First, Google's attractively simple MapReduce fault tolerance mechanism is predicated on the materialization of intermediate state. 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 compression and pre-aggregation before communication. We discuss how the HOP design addresses this issue. Finally, pipelining requires that producers and consumers are co-scheduled intelligently; we describe our initial work on this issue.

2. BACKGROUND

MapReduce is a programming model for performing transformations on large data sets [4]. 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 a list of *records* (key-value pairs). Each job consists of two steps: first, a user-defined *map* function is applied to each 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.

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.

The Hadoop MapReduce architecture 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). A heartbeat protocol between each TaskTracker and the JobTracker is used to update the JobTracker's bookkeeping of the state of running tasks, and drive the scheduling of new tasks: if the JobTracker identifies free TaskTracker slots, it will schedule further tasks on the TaskTracker.

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 size of the input file determines the number of map tasks.

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

After a map task has applied the map function to each input record, it enters the *commit* phase. To generate the task's final output, an in-memory buffer is flushed to disk, and all of the spill files generated during the map phase are merge sorted into a single data file. The final output file is 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. 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. The *sort* phase groups records with the same key together. 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. In traditional batch-oriented Hadoop, 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 key. The reduce task merges these runs together to produce a single run that is sorted by the 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.

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).

3.1 Pipelining Within A Job

In the stock version of Hadoop MapReduce, reduce tasks gather map outputs by issuing HTTP get requests to nodes that hosted the execution of a map task belonging to the same job. The reduce task does not issue this request until it has received notification that the map task has completed and its final output has been committed to disk. This means that map task execution is completely decoupled from reduce task execution. To support pipelining within a single MapReduce job, we modified the map task to instead *push* data to reducers as it is produced.

A challenge that we faced in our pipelined architecture was choosing the right granularity for transferring data from mappers to reducers. A naïve design that eagerly sends each record as soon as it is produced 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). 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.

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 output: all the reduce task must do is merge together the pre-sorted map output for each partition. In the eager 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 [4], moving more work to the reducer can degrade performance.

We addressed these issues by buffering the mapper output until it reaches a certain record threshold. When the record threshold is reached, the mapper sorts and applies the combiner function to the buffer, sending the output to a spill file. 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.

The connection between pipelining and adaptive query processing techniques has been observed elsewhere (e.g., [1]). 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 [7] 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. In fact, a client (e.g., Pig) cannot even 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. In stock Hadoop 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, HOP supports early returns of reducer output, which enables online aggregation and continuous query pipelines. This new functionality will be the focus of our demonstration.

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 [4].

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. There are many interesting options for scheduling pipelines or even DAGs of such jobs that we plan to investigate in future.

4. HOP DEMONSTRATION

Our pipelined implementation of Hadoop enables two new features to the MapReduce model. In this section we provide a short description of these features along with a description of our demonstration of these features.

4.1 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 [10], Pig [7], and Sawzall [8].

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. However, 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 [6], but the batch-oriented nature of traditional MapReduce implementations makes these techniques difficult to apply. In this demonstration, we will show how we extended our pipelined Hadoop implementation to support online aggregation within a single job and between multiple jobs. We will visually demonstrate that online aggregation has a minimal impact on job completion times, and can often yield an accurate approximate answer long before the job has finished executing.

4.1.1 Demonstration

We demonstrate online aggregation in HOP using a data set containing seven months of hourly page view statistics for more than 2.5 million Wikipedia articles [9]. This data set is comprised of 320GB of compressed data (1TB uncompressed) that is divided into 5066 compressed files. The demonstration query counts the total number of page views for each language and each hour of the day. In other words, the query groups by language and hour of day, and sums the number of page views that occur in each group. The demonstration will show early results that are nearly accurate to the final answer in orders of magnitude less time.

4.2 Continuous Queries

MapReduce is often used to analyze streams of constantly-arriving data, such as URL access logs [4] and system console logs [11]. 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.

4.2.1 Demonstration

In this demonstration, we will show how HOP supports continuous MapReduce jobs, and visually demonstrate this feature with an implementation of cluster monitoring tool written as a continuous MapReduce query. The input data stream will be real-time system statistics and log feeds. We envision a facility where both cluster operations staff and users have access to displays of key performance metrics about the processing, disk access, and communications within the cluster in order to better understand both the cluster status as well as the progress of individual MapReduce jobs.

The monitoring tool is written in Protovis [2], which allows us to create interactive, animated, and data rich displays in an ordinary web browser without requiring a plug-in. Clients connect to a server which is accepting the output of the continuous query. Each client's web browser subscribes to the set of feeds it is interested in and visualizes the resulting data.

The overall design permits both drill-down and roll-up of information. There is a navigational display from which the user can select displays in two different perspectives. In the first perspective, data feeds are available for various levels of aggregation of the cluster resources (e.g., cluster, rack, machine). In the second, the orientation is to the progress of MapReduce jobs executing in the same cluster. In either perspective, displays may be annotated by reports of faults which can be expanded to reveal more detail.

5. CONCLUSION

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. HOP provides significant new functionality, including "early returns" on long-running jobs via online aggregation, and continuous queries over streaming data. 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 [5], with an eye toward improved scalability via parallelism.

6. REFERENCES

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[2] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, 2009.

[3] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[5] J. M. Hellerstein, R. Avnur, A. Chou, C. Hidber, C. Olston, V. Raman, T. Roth, and P. J. Haas. Interactive data analysis with CONTROL. *IEEE Computer*, 32(8), Aug. 1999.

[6] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[7] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[8] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[9] P. N. Skomoroch. Wikipedia page traffic statistics, 2009.

[10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive — a warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[11] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.