UNIVERSITY OF CALIFORNIA, SAN DIEGO

Continuous MapReduce: An Architecture for Large-scale In-situ Data Processing

A Thesis submitted in partial satisfaction of the
requirements for the degree Master of Science

in

Computer Science

by

Christopher J. Trezzo

Committee in charge:

Alex C. Snoeren, Chair
Keith Marzullo
Amin Vahdat
Ken Yocum

2010

The Thesis of Christopher J. Trezzo is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

Chair

University of California, San Diego

2010

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my adviser Ken Yocum for his enthusiasm and invaluable guidance. I have learned a tremendous amount working on this thesis and much of that is due to him. I would like to thank Dionysios Logothetis who was always willing to answer my questions, and supplied a great deal of advice.

I would also like to thank my committee members Alex Snoeren, Keith Marzullo and Amin Vahdat for reviewing my thesis and ushering me through the final graduation process.

Finally, I would like to thank my family and friends; without their love and support this would not have been possible.

ABSTRACT OF THE THESIS

Continuous MapReduce: An Architecture for Large-scale In-situ Data Processing

by

Christopher J. Trezzo

Master of Science in Computer Science

University of California, San Diego, 2010

Alex C. Snoeren, Chair

This thesis addresses a fundamental data management challenge faced by cloud service providers: analysis of semi-structured log data generated by large-scale compute infrastructure. This analysis is a crucial aspect of a cloud provider's business, creating competitive advantage by mining user behavioral patterns and ensuring efficient use of resources. However, the amount of data produced in this environment is rapidly growing. The current approach brings data to a central location before analysis, incurring a significant cost and delaying results. As scale increases, the time and cost of data migration alone will render this approach infeasible.

We present Continuous MapReduce (CMR), an architecture for large-scale in-situ data processing. CMR is designed to be scalable, responsive, and available while processing logs across thousands of data center servers. CMR extends the MapReduce programming model to allow continuous queries over these data streams, building on concepts found in distributed stream processing. The salient architectural features include an in-situ approach, incremental processing with sliding windows, and a relaxed consistency model.

We have built a prototype CMR framework using Mortar, a distributed stream processor, and evaluated it against current batch processing systems. Our results indicate that this approach can improve result latency by 30% for batch and continuous queries. In addition, CMR's consistency model enables it to return results quickly in the face of failure, and still maintain high result fidelity. These results indicate CMR is a valuable tool for addressing the scalability issues of next generation data processing environments.

Chapter 1

Introduction

This thesis investigates an architecture for processing data distributed across large data centers. Such a facility is a key component to running large cloud services. The "cloud" is a current paradigm in the information technology industry allowing IT solutions that were previously custom built by in-house experts to be offered by 3rd party "cloud providers" as dynamically scalable services over the Internet. These cloud providers offer services covering the entire technology stack including scalable hardware infrastructure, development environments, deployment platforms, and finished products. Businesses without IT infrastructure can use these cloud services to develop and deploy their applications at a rapid pace.

This "Everything-as-a-Service" model has created a complex ecosystem of Internet-scale web services. It is common for a single customer request to communicate with over 50 different services, touching thousands of machines [14]. Service providers deal with hundreds of thousands of customer requests per second [32], and loads will only increase as the popularity of these services grow.

To handle this load in a cost effective way, cloud providers use commodity hardware to build infrastructure at an enormous scale. One organization could have hundreds of thousands of servers in tens of data centers spread across the world. Microsoft has recently opened data centers in Dublin and Chicago, each with a capacity for 300,000 servers [1]. Amazon, Yahoo! and Facebook are all estimated to have over

50,000 servers across their entire infrastructure [2]. Google has stated that they are currently designing core services with a scale of 1 to 10 million servers in mind [14].

Scalable log processing is a crucial facility to ensure efficient use of infrastructure and to gain a competitive advantage. Service providers continuously monitor many aspects of their system using semi-structured log data. Increased revenue can be generated by analyzing logs of user click streams for behavioral patterns to provide targeted ads. E-commerce and credit card companies analyze point-of-sales transactions for fraud detection. Infrastructure providers use log data to detect hardware misconfigurations, improve load-balancing across large data centers, and gather statistics about infrastructure usage patterns [7, 34, 38].

This semi-structured log data accumulates at a rapid rate throughout the distributed infrastructure, representing a huge data management challenge. Facebook produces over 25 terabytes of log data per day for analyzing user behavior and generating targeted ads [34]. Personal communications with a developer at a large cloud provider indicate that log rates can exceed 10 MB/sec per server. The current explosion of data collection is increasing at a faster rate than Moore's law, indicating that data management and analysis will only become more difficult in the future [37].

1.1 Current approach to log processing

Companies such as Google, Microsoft, Yahoo! and Facebook use bulk-processing frameworks, such as MapReduce [3, 15] and Dryad [24], to perform ad-hoc analysis of their distributed log data. These frameworks abstract away the complexities of writing distributed applications, such as parallelization, fault-tolerance, data distribution, and load balancing. This abstraction allows businesses to capitalize on cheap hardware, harnessing thousands of commodity machines to perform large data processing tasks. They are geared towards local-area network (LAN) clusters, batch-oriented workloads and optimized for throughput.

Currently, the dominant log processing architecture uses these bulk-processing

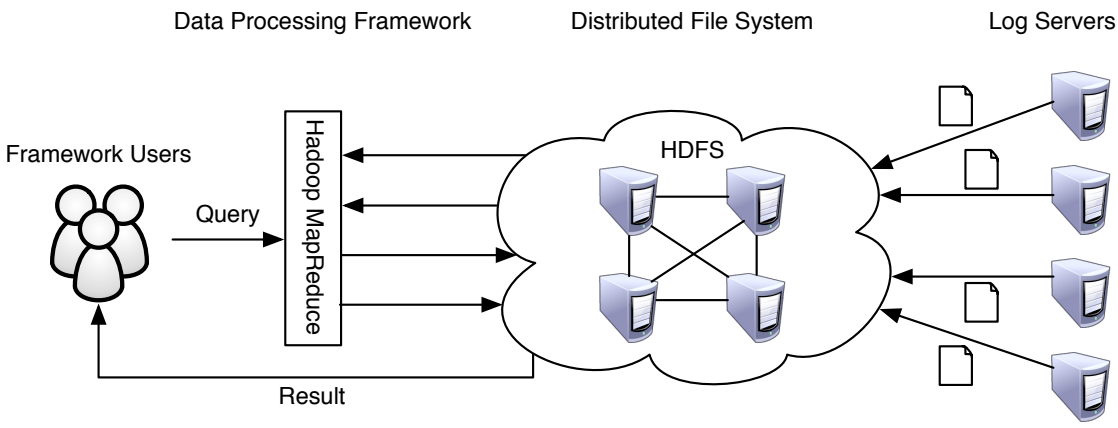

Figure 1.1: Log processing with the *store-first-query-later* model. Apache Hadoop [3] is used as an example.

frameworks in a traditional *store-first-query-later* model [17]. Companies migrate log data from the source nodes to an append-only distributed file system such as GFS [18] or HDFS [3]. The distributed file system replicates the log data for availability and fault-tolerance. Once the data is placed in the file system, users can execute queries using bulk-processing frameworks and retrieve results from the distributed file system. Figure 1.1 illustrates this model.

The current approach exhibits a number of key limitations in this environment. Migrating log data into the distributed file system will take a prohibitive amount of time for even batch-oriented analysis with loose time constraints. A simple back-of-the-envelope analysis reveals this issue. Consider 10,000 servers (Facebook currently harvests 30,000) producing log data at a rate of 10 MB/sec. This cluster generates 8 petabytes of log data per day. In our measurements, server-class machines can sink data at a rate of 30 MB/sec. It would take 3,314 dedicated HDFS [3] nodes to sink that amount of log data in one day. These machines are completely I/O bound, and are unable to perform a significant amount of data processing at the same time. Thus, all of their CPUs are left virtually idle. Servers are one of the largest fixed, depreciating costs in a data center, putting a substantial price tag on data migration alone.

This fundamental limitation has broad ramifications on the fidelity and avail-

ability of log processing queries. Companies will either scale back data analysis or use more resources to keep up with the rate of production. Both of these options are expensive. In the first case, scaling back analysis can sacrifice competitive advantage leading directly to revenue loss. In the second case, increasing resource requirements can become expensive, especially when added resources are ineffectively used. If we extend our calculation to 100,000 machines generating 80 petabytes of log data per day, it will take 33,140 dedicated servers to keep up with the rate of production. Eventually, the cost of scaling resources using this method will become prohibitive.

In addition, transferring the log data represents wasted effort and inefficient use of resources. A recent study [20] showed that these bulk-processing systems handle a workload consisting of recurring data-driven queries that are highly selective. Most queries recur daily, some weekly, and a few on a monthly basis. The queries are update-driven, running soon after new log data is available and only touching the most recent information. The queries use highly-selective filters, reducing output to 17% of the total input data in some cases. Similarly, Facebook indicates that their queries filter out 80% of accumulated log data [34]. The queries also show similarity to each other, leveraging many of the same filters and subroutines supplied by libraries.

Finally, the delay of analysis caused by data migration handicaps applications that require timely results. All of the previously mentioned applications can benefit from lower result latencies. Server logs, user click streams and point-of-sales transactions are all highly dynamic content. Online analysis of this data will create a more agile system, allowing companies to react more quickly to important events, including detecting fraudulent credit card activity, detecting when a group of servers is thrashing, identifying security breaches, or detecting popularity trends of served content.

1.2 The case for Continuous MapReduce

These fundamental limitations of the *store-first-query-later* model highlight the need for a different approach to large-scale bulk-processing. We propose Continuous

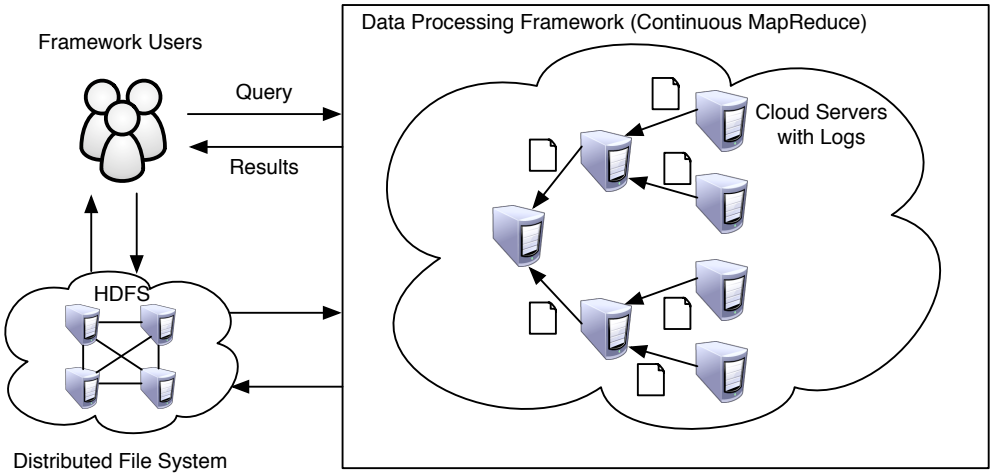

Figure 1.2: Log processing with the *query-then-store* in-situ model.

MapReduce (CMR), a data processing framework to address large-scale batch-oriented workloads where timeliness is still important.

CMR's architecture has several properties that specifically address the limitations of current approaches. It is *scalable* across large networks, enabling companies to handle the next generation of data management challenges. It is *responsive*, allowing more agile analysis of valuable log data. It is *highly available*, providing users with results in the presence of failure. Finally, it is *easily adoptable*, creating a solution that can be integrated into existing infrastructures.

These architectural properties are supported with several unique design choices. CMR's architecture uses a *query-then-store* model (Figure 1.2), taking distributed storage systems off the critical path for delivering results to a user. It processes data in-situ, or on location, eliminating wasteful migration of unused data. It uses in-network aggregation to save valuable bandwidth resources by reducing the amount of data sent across the network. It uses a push-based model and continuous queries with incremental processing windows to handle data as it arrives, accommodating the latency requirements of update-driven recurring queries. The architecture also explores a relaxed consistency model that prevents an entire query from being blocked by a small percentage of unavailable data. Lastly, CMR extends the original MapReduce program-

ming model [15], providing a familiar environment for current MapReduce developers. This allows developers to leverage their existing MapReduce code libraries for CMR applications.

1.3 Contributions

This thesis makes the following contributions:

- An architecture for continuous in-situ data processing that is scalable, responsive, and available in a distributed environment with tens of thousands of nodes.
- The extension of the MapReduce programming model to work efficiently in a continuous environment using sliding windows and incremental processing.
- The exploration of a relaxed consistency model for large-scale data processing that provides best-effort guarantees, allowing users to trade off between timeliness and result fidelity.
- A prototype and evaluation of CMR's architectural design points including the in-situ approach, continuous processing with sliding windows, and a relaxed consistency model. We use Hadoop, a popular bulk processing framework, and HOP, a recent research project that adds pipelining to Hadoop, as points of comparison.

The remainder of this document is structured as follows. Chapter two provides background for the work presented in the following chapters. We discuss the MapReduce programming model, current data processing frameworks, and the concept of continuous processing. Chapter three presents the Continuous MapReduce architecture and its important design points. Chapter four describes the mechanisms used in the implementation of the CMR framework. Chapter five evaluates the framework's performance with respect to batch queries, incremental processing, and resilience to failure. Finally, chapter six concludes the thesis.

Chapter 2

Background

This chapter gives an overview of related work as well as concepts that are relevant to the presentation of Continuous MapReduce. Section 2.1 gives a description of the original MapReduce programming model, which CMR's model extends. Section 2.2 describes Hadoop, the current state-of-the-art open-source MapReduce framework. Many companies currently use Hadoop for large-scale log processing, making it a good point of comparison for alternative approaches. Section 2.3 gives an overview of alternative MapReduce frameworks and the strategies used in their design. Finally, Section 2.4 briefly describes distributed stream-processing systems and how they relate to CMR.

2.1 MapReduce programming model

MapReduce is a programming model for large-scale bulk data processing. It is designed to simplify the parallelization of application logic enabling developers to easily leverage clustered resources. MapReduce consists of a few simple functions, making the model easily adoptable for the average programmer. MapReduce is an imperative programming model, and is based on the common *map* and *fold* functions found in languages like LISP. Finally, the model makes it easy for a programer to implement their desired functionality without having to worry about failures in a distributed environment.

```
map(String key, String value) {
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, 1);
}
```

```
reduce(String key, Iterator values) {
    // key: a word
    // value: a list of counts
    int result = 0;
    for each v in values:
        result += v;
    EmitOutput(key, result);
}
```

Figure 2.1: Word Count example using the map and reduce interfaces.

The MapReduce programming model in its simplest form consists of two functions: *map* and *reduce*. Input data is thought of as a set of semi-structured input records. First, a user-defined *map* function takes each input record and produces a set of intermediate key-value pairs. These intermediate pairs are grouped together according to their key and passed to a user-defined *reduce* function. Finally, the *reduce* function takes each key and the set of associated values, and produces the final output.

For example, consider counting word occurrences in a set of text documents. Figure 2.1 gives the pseudo code for both map and reduce functions. The map function takes the document name as a key and the contents of the document as a value. It then parses the contents and emits an intermediate key-value pair for each word in the document. The intermediate pair contains the word as the key and a count of the occurrence as the value. The intermediate pairs are grouped by word, and passed to the reduce function. The reduce function sums the occurrences, and emits the word and associated count.

In addition, MapReduce supports an optional combine function. The combine is a user-defined function that aggregates intermediate records (with the same key) to reduce data sent over the network. In the word count example, a combiner would group all intermediate pairs with “the” as a key, and sum their values. The combine function would emit one intermediate pair per intermediate key, <the,N>, where *N* is the number of occurrences of “the” encountered in the input stream from that node. If a programmer does not specify a combine function, all the intermediate values are transfered over the network.

MapReduce is an expressive programming model, and it is currently used for a wide range of applications [4]. The model can express all relational algebra operations as well as complex data mining and machine learning algorithms [5, 39]. In addition, high level frameworks have been built for data warehousing and scientific computing applications using MapReduce [10, 26, 30, 36].

2.2 Apache Hadoop

Hadoop [3] is an open-source implementation of the original MapReduce programming model and system designed by Google [15]. Hadoop has grown in popularity, and is used in industry by several companies including Yahoo!, Amazon, Facebook, and IBM. As a mature, robust platform, Hadoop makes a valuable point of comparison for alternative MapReduce frameworks such as Continuous MapReduce. This section gives an overview of the relevant design points in Hadoop.

Hadoop's framework automatically parallelizes computation in a distributed and fault-tolerant way. This allows programers to avoid the complexities of writing distributed applications enabling them to focus on program logic and leverage large cluster resources at the same time. Hadoop is designed for a batch-oriented work load and is optimized for throughput. It is geared toward large local-area clusters built with commodity hardware. Fault tolerance is built into the software framework, handling both machine and disk failure.

The Hadoop MapReduce framework leverages a distributed file system called the Hadoop Distributed File System (HDFS). HDFS is an open-source implementation of Google's Distributed File System (GFS) [18]. This file system is meant to make input data available to every node in the cluster, as well as a central place for writing output from MapReduce jobs. The design of the file system is heavily optimized for manipulating large append-only files and maximizing throughput.

A MapReduce job is a pair of *map* and *reduce* tasks executed over a distributed cluster of nodes. Hadoop implements a MapReduce job as a pull-based three-phase ex-

ecution flow consisting of map, shuffle, and reduce phases. Data flows from the map tasks to the reduce tasks. Figure 2.2 illustrates one complete MapReduce job in Hadoop. Applications may link multiple MapReduce jobs together to implement complex functionality.

At the beginning of a MapReduce job, Hadoop splits input data into a set of chunks called input splits. In the first phase, called the map phase, each map task processes a single input split. Each map task is executed in parallel on potentially many physical machines. The map tasks apply the map function to their assigned input split, and each task emits a set of intermediate key-value pairs. The map tasks group and sort their local intermediate pairs based on the intermediate key, generating key-value list pairs. If the user has implemented an optional combine function, then at this point the map tasks apply the combine function to each key-value list pair. They partition the pairs into R groups, where R is the number of reduce tasks in the reduce phase (a statically assigned number). Finally, they spool the groups to a set of intermediate files on local disk (one file per group).

The shuffle phase begins once all map tasks successfully complete. Each of the R reduce tasks request one intermediate file from each map task. As the reduce tasks receive intermediate files from the maps, they merge the contained intermediate pairs based on the intermediate key.

The reduce phase begins once all reduce tasks receive, merge and sort every intermediate file. Each reduce task processes the key-value list contained in the intermediate files in parallel. The reduce tasks apply the reduce function to each key-value list pair and emit the final output for each pair. Finally, the reduce tasks write their output to a file in HDFS.

The synchronous three-phase execution model and the use of files residing on disk creates a simple fault-tolerance mechanism. Each phase requires data from the previous one, and the phases do not overlap with each other (i.e. there is an implicit barrier between each phase). The framework monitors the execution time of each task. If there is a map task that fails or that is taking an abnormal amount of time, the framework

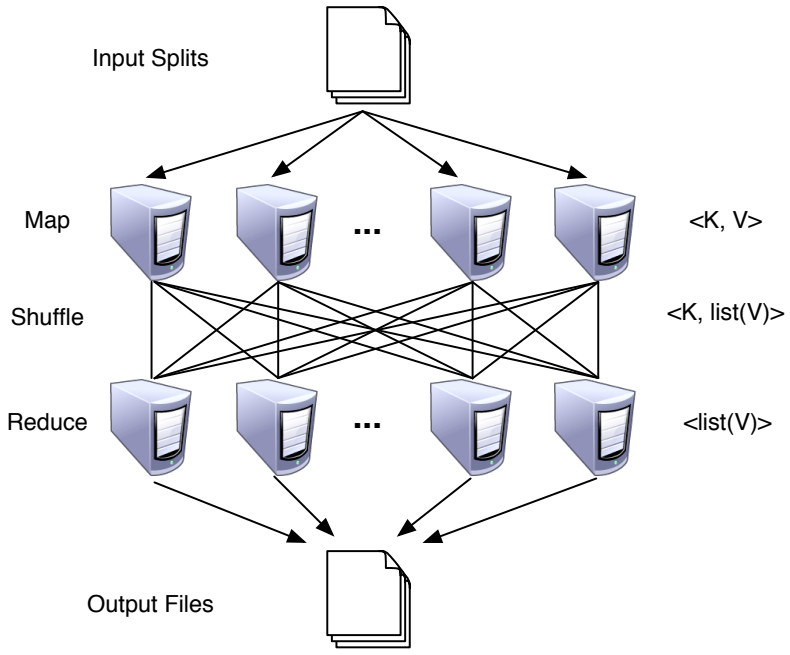

Figure 2.2: Hadoop implements a MapReduce job as a three-phase execution flow made up of the map, shuffle and reduce phases.

simply restarts that task. The exact same method is used for reduce tasks as well. This can be done because a reduce task can not start until all map tasks have finished. In other words, the reduce tasks must wait until all intermediate pairs are grouped, sorted and spooled to disk before applying reduce functions.

Hadoop uses a precise, or fully-consistent, fault tolerance model. The system will either return results corresponding to the entire input data set or will not return results at all. A consequence is that one straggling map task can block the progress of an entire MapReduce job [41]. As scale increases, a fully-consistent model might not be suitable for batch-oriented jobs that still need to complete in a reasonable amount of time. When dealing with hundreds of thousands of nodes, there is always a set of nodes that is not available due to failure. This may hamstring MapReduce jobs if the framework has to wait for all map and reduce tasks to complete successfully before returning results. Thus, knowing when (and where) to restart tasks is a complicated question that can have a large impact of job completion time [41].

Table 2.1: Current MapReduce frameworks compared to CMR.

Framework	Environment	Input Data Distribution	Communication	Execution Model	Fault Tolerance Model
Hadoop	LAN Cluster	Distributed File System	Message passing using files spooled to disk	Batch Oriented	Precise
HOP	LAN Cluster	Distributed File System	Message passing using files spooled to disk	Batch Oriented or Continuous	Precise
Phoenix	Shared-Memory System	Shared Memory	Shared buffers	Batch Oriented	Precise
CGL-MapReduce	LAN Cluster	Distributed File System	Pub/Sub Messaging System	Batch Oriented	Best Effort
Continuous MapReduce	***LAN or WAN***	***In-situ***	***Stream-Based***	***Batch Oriented or Continuous***	***Best Effort***

2.3 Alternative MapReduce frameworks

There have been several efforts to develop new MapReduce frameworks, each one addressing different environments and applications. This section surveys alternatives to the Hadoop framework; Table 2.1 compares these systems.

Closely related work includes the Hadoop Online Prototype (HOP) [13]. HOP is an effort to improve Hadoop by avoiding the materialization of task and job output at each phase through the use of pipelining. Pipelining is a technique where map tasks partition and send intermediate key-value pairs individually to the reduce tasks as soon as they are produced. This allows reduce tasks to group and sort intermediate data as map tasks finish, improving utilization and job completion times. HOP also provides online aggregation [21], which allows the framework to return tentative results to a user before all tasks have completed. HOP provides a fully-consistent fault-tolerance model, leaving a user the choice of using tentative results or waiting for the fully consistent ones. Finally, HOP provides basic support for continuous processing. It uses a flush API to force maps to transfer their current output to reduces. This can be done periodically to simulate a bare-bones window specification.

However, HOP still uses a store-first-query-later model and does not address the problem of data migration. Even though they support continuous processing, the

data still has to be transfered into HDFS before any analysis can be done. Furthermore, their continuous processing buffer does not function incrementally, introducing a large amount of duplicate work when the buffer size is large and the reduce function is executed at a high frequency. Lastly, as scale increases, maintaining a fully-consistent fault-tolerant model will increasingly delay results and reduce the availability of the system.

Phoenix explores the effectiveness of MapReduce for parallelizing a variety of applications on shared-memory systems [33]. Their runtime framework is designed for multi-core and multiprocessor systems. They show that given a careful implementation, for a large number of applications the MapReduce programming model can achieve similar performance to the equivalent code written with P-Threads. Similarly to HOP, Phoenix uses pipelining between phases to increase utilization and decrease job completion time. They provide fault tolerance across processor nodes, allowing the system to deal with corrupt memory as well as power and temperature related issues.

Finally, CGL-MapReduce is a system designed specifically for scientific data analysis [16]. These applications, such as clustering algorithms, require multiple MapReduce jobs to iterate over a set of input data. CGL-MapReduce is built using a publish/subscribe messaging system called NaradaBrokering [31]. NaradaBrokering is a local-area peer-to-peer publish-subscribe system that avoids file materialization by reliably streaming intermediate data from the map tasks directly to the reduce tasks. They use a distributed filesystem and check-pointing to provide fault-tolerance for map tasks, and do not address fault-tolerance for reduce tasks. CGL-MapReduce is batch-oriented and does not execute jobs over continuous streams of data.

In contrast to these systems, CMR avoids data migration through an in-situ approach. CMR does not depend on a distributed file system for data availability. Instead, it uses a best-effort consistency model and provides accountable result fidelity that can be controlled through result eviction polices. Finally, CMR adapts the MapReduce programming model to support incremental processing of continuous queries. These techniques make CMR a good system for bulk data processing in large-scale LAN and

wide-area network (WAN) environments.

2.4 Distributed stream-processing systems

CMR's architecture builds on a large base of research devoted to distributed stream-processing systems (DSPSs) [6, 8, 9, 11, 12, 22, 23, 25, 28, 35, 40]. These systems perform in-network aggregation over distributed data streams with real-time or close to real-time processing requirements. There is a large diversity in the type of DSPSs, ranging from LAN-oriented systems operating with a few complex nodes [11], to WAN-oriented systems with potentially thousands of commodity nodes [28]. DSPSs also provide the full spectrum of consistency models, using precise (fully consistent) models for applications requiring 100% result fidelity and best-effort models for real-time applications.

CMR adapts key DSPS concepts for large-scale bulk data processing. These concepts include incremental processing of data streams using windows to bound computation, per-tuple processing providing latency critical performance benefits, and a selection of various consistency models to address different application requirements. These techniques contrast sharply compared to the approaches used by current bulk data processing systems. Finally, in CMR's data center environment, the large volume and highly distributed nature of data presents unique challenges that DSPSs have not traditionally focused on.

Chapter 3

The Continuous MapReduce Architecture

This chapter presents Continuous MapReduce (CMR), a new framework for large-scale distributed data processing. First we describe the design goals of the system, and then we present the CMR programming model. We discuss CMR's ability to process continuous streams of data, its use of in-network aggregation, and its best-effort approach to fault-tolerance that ensures timely results in large-scale distributed environments.

3.1 Design overview

The Continuous MapReduce framework is designed to execute ad-hoc queries over large amounts of semi-structured log data. This section gives a brief overview of the design goals and the techniques we propose to achieve them.

Scalable: The target environment may consist of tens of thousands of nodes. A log processing architecture should execute queries spanning large portions of the data center infrastructure without using unnecessary bandwidth or overly impacting the end hosts. Our design leverages in-situ processing and in-network aggregation to minimize the amount of log data sent across the network.

Responsive: Unlike traditional batch-oriented MapReduce frameworks, a distributed log processor needs to accommodate continuous queries and streaming data sources. For example, many queries are update-driven, recurring on a daily basis over the most recent data. To support online analysis, these queries require results as soon as possible. Like distributed stream processors [11, 6, 28], CMR uses continuous queries and incremental processing windows to operate over streams of data. It avoids materialization of intermediate files to disk and the use of synchronous barriers by using a pipelined and push-oriented design.

Highly available: Large-scale data processing over large networks connecting commodity hardware, means operating in an environment where a certain percentage of nodes will always be unavailable due to failures, misconfigurations or scheduled maintenance. CMR employs a relaxed consistency model to prevent an entire query from being blocked by a small percentage of unavailable data. It also uses an eventually consistent algorithm to adjust for failed nodes and re-join restarted nodes.

Easily adoptable: CMR extends the original MapReduce programming model, giving developers a familiar environment and allowing them to leverage their existing data processing libraries. This model gives developers a simple programming API that abstracts away the complexities of writing parallel applications in a distributed environment. During design, we strived to adapt MapReduce to a continuous environment using the minimal number of changes possible.

3.2 Programming model

We have chosen to base the Continuous MapReduce programming model on the original MapReduce model (Section 2.1). We support the original map and reduce functions, an extended version of combine and reduce, as well as an additional function called uncombine. These extensions to the model support incremental processing and in-network aggregation. Users can submit standard MapReduce applications they have written for other MapReduce frameworks, or write new applications that take advantage

Table 3.1: A table comparing the MapReduce and Continuous MapReduce programming models.

Functionality	MapReduce	Continuous MapReduce	Input/Output
Input Data Processing	Reader( )	Reader( )	Raw Input → Record
Production of Intermediate Keys	Map( )	Map( )	Record → <k, v>
Aggregation	Combine( )	Combine( )	<k, [v_1, v_2]> → <k, $agg_{v_1+v_2}$>
Production of Final Result	Reduce( )	Reduce( ) or Reduce( )[range, slide]	<k, [v_1, v_2]> → <k, v>
Remove data from intermediate value	N/A	UnCombine( )	<k, [Agg_{v_n}, v_1]> → <k, $Agg_{v_n - v_1}$>

of CMR-specific functions.

A quick glance at Table 3.1 reveals how similar the CMR model is to the original MapReduce model. The reader, map and combine functions are exactly the same. The reduce function can optionally specify a processing window, and an additional uncombine function can be used for incremental processing. Otherwise, the models are identical. These small modifications enable CMR applications to run in a continuous fashion, without preventing users from running their existing unmodified MapReduce code in CMR as well.

CMR uses processing windows to bound computation over continuous streams of data. A CMR query returns results pertaining to each processing window. The reduce function specifies the type of window, which can either be time windows or logical windows. A time window defines its dimensions based on *wall-clock*, or real time. A logical window specifies its proportions based on intervals of data. For example, if a user wants to execute a query over data that was collected during a certain time period (i.e. log data collected over the last 24 hours), then they would use a time window. If a user wants to execute a query over a specific amount of data irrespective of the time it was collected (i.e. the last 200 entries in a log, or the first 500 MB of a file), then they

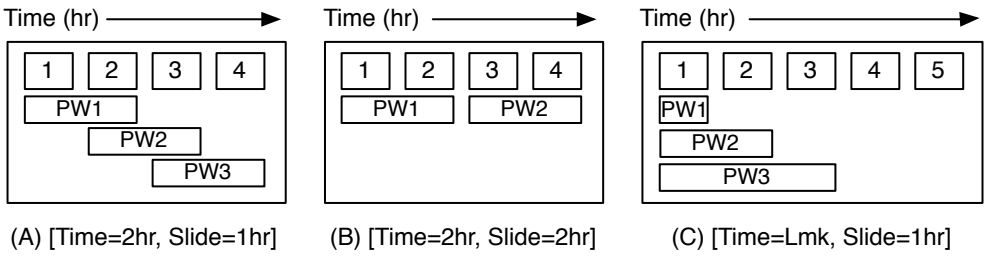

Figure 3.1: Three examples of a time processing window (PW): (A) overlapping windows, (B) hopping windows, and (C) landmark windows.

would use a logical window.

Each processing window has a window range and slide. A window range defines the size of a window, and a slide defines how often the window updates (i.e. how often results are produced). For example, if a user wants to maintain a running average for the last 24 hours of log data, which updates every hour, then they would specify a time window with a range of 24 hours and slide of 1 hour. Alternatively, if a user wants to process a file in 10 MB chunks, then they would specify a logical window with a range and slide of 10 MB.

As Figure 3.1 illustrates, windows may provide overlapping, hopping, or landmark computation. An overlapping window has a slide that is less than it's range, creating windows that overlap with each other and contain common data. This type of window lends itself to incremental processing. A hopping window has a slide equal to or greater than it's range and do not overlap. A landmark window has a fixed slide and does not specify a range. Each window cumulatively processes data added by the next slide.

CMR uses the user-defined combine function to aggregate intermediate data at several places throughout the data flow, reducing the amount of data sent across the network. Similar to Dryad [24], CMR can specify multi-level aggregation trees (Figure 3.2 Part A) to aggressively aggregate data when possible. Each MapReduce partition uses a separate aggregation tree with a reduce function placed at the root. Multiple trees are used for multiple partitions. In comparison, Hadoop is restricted to a one-level

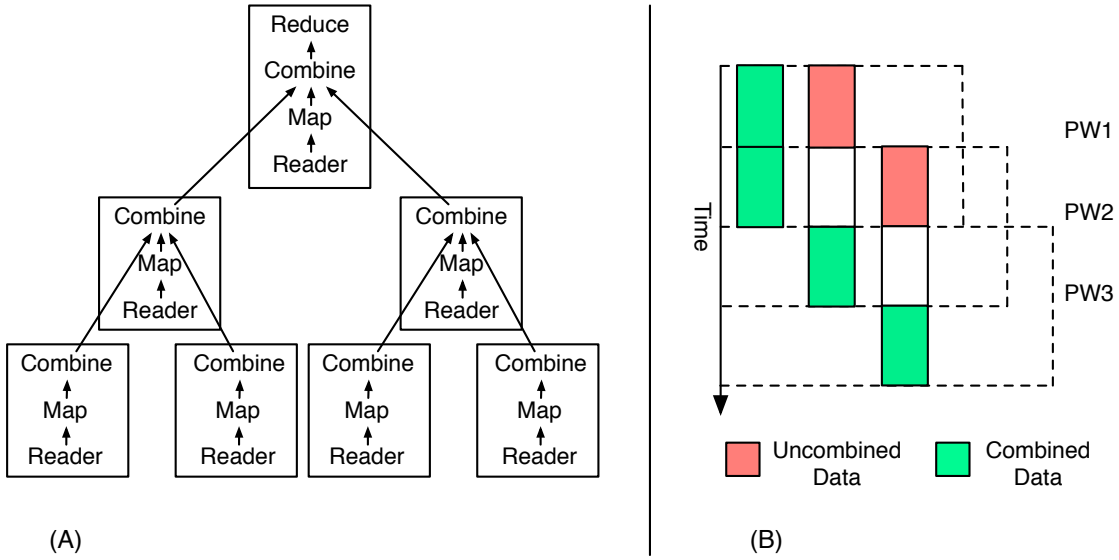

Figure 3.2: Part A shows a multi-level aggregation tree corresponding to one MapReduce partition. Part B illustrates the incremental processing of time windows using the combine and uncombine functions.

aggregation tree (Figure 2.2).

Figure 3.2 Part B shows the combine function working with the uncombine function to incrementally process a sliding time window. This ensures that overlapping work completed during a previous window is leveraged to prevent duplicate effort. As the window slides, the combine function adds new data and the uncombine function removes expired data from the window. Currently, both Dryad and Hadoop do not support incremental processing and recompute the entire window for each slide. For windows with a large range and a small slide, incremental processing can greatly increase efficiency of the system.

The effectiveness of incremental processing depends upon the characteristics of the aggregate function implemented by the user in combine and uncombine. Aggregate functions have traditionally been classified by the amount of state required to represent intermediate data; these categories include *distributive*, *algebraic*, and *holistic* functions [19]. Distributive functions are aggregated into intermediate values that are of a constant size. The contents of these intermediate values is the final result for that sub-

set of the total data. Some examples of distributive functions include sum, minimum, or maximum. Algebraic functions are aggregated into intermediate values that are of a constant size. These intermediate values represents a subset of the total data, but are not the data itself. Processing the intermediate data from each subset generates the final output. Some examples of algebraic functions include finding the average, standard deviation, or TopK of a set of integers. Finally, holistic functions require intermediate values that are the same size as the input or larger. The intermediate data is not reduced in size during aggregation. Some examples of holistic functions include finding the median or sorting a set of data.

Finally, in-network aggregation and incremental processing are both optimizations. As such, if the users data processing functions are holistic or can not incorporate data in a piece-meal fashion, then CMR will execute the query and process the data without these features.

3.3 Relaxed consistency model

CMR's best-effort consistency model enables the user to trade off between timeliness and result fidelity. A user may specify a maximum amount of processing time, or a minimum result fidelity, allowed for each CMR job. This allows CMR to flexibly address use cases that have time requirements (e.g. a report must be delivered no more than one hour after the data is generated), and use cases where a fully consistent model is not required (e.g. an accurate estimation of the results can be made from processing a small percentage of the total data). The policies that support this model are described in Section 4.5.

We chose to use a best-effort approach because it has been shown to provide higher availability and lower result latency when executing global one-shot queries in a data center environment with 10^3 to 10^9 end hosts [29]. When operating at this scale there is always a significant percentage of nodes that are not available at any given time. In an enterprise network, this percentage varies periodically and can range from 15%

to 25% depending on things like time of day and scheduled maintenance [29]. A more relaxed consistency model allows results to be returned to the user even if the entire data set is not available.

The CMR framework provides best-effort consistency guarantees using a gap recovery model [22]. In a gap recovery model, the framework reports on the results that it receives. Data missing because of a failed node or network loss is simply not included in the final result. When deciding on the fault tolerance guarantees that CMR would make, we considered several techniques used in research done on distributed stream processors [22, 23, 35]. These mechanisms provide stronger guarantees, but impose significant overheads.

There are several advantages to gap recovery at a large scale. First, it allows CMR to continue processing local data asynchronously. A node can continue processing data regardless of the progress made by other nodes. In a scenario where a percentage of nodes is always unavailable, this model can improve result latency and prevent a small amount of unavailable data from delaying the entire application. Second, best effort consistency is highly scalable, requiring virtually no overhead. There is no extra communication between nodes or replicas to maintain. This is attractive for queries that only care about the most recent data, where late data that has been replaced by new data is of no use. In addition, nodes do not have to store any extra state, making it low impact on the servers. Finally, best effort consistency provides low latency failure recovery. As soon as a node is rejoined into the query tree it can begin processing new data immediately.

Chapter 4

Implementation

In this chapter we discuss the implementation of Continuous MapReduce in detail. We give an overview of Mortar, the underlying system used to implement CMR, and the modifications to its core functionality to support the semantics of CMR and the MapReduce programming model. Finally we give a detailed explanation of the components and policies specific to the CMR framework, including the mechanisms for dealing with failure and the policies that determine when to deliver results to the user.

The major components of our prototype include:

- Implementation of the Continuous MapReduce API using generic map and reduce Mortar operators.
- Addition of support for CMR framework internals that provide grouping, sorting, and partitioning of key-value pairs.
- Modification of Mortar's fault tolerant mechanisms including boundary tuples, sliding windows, and tuple timestamps.
- Addition of new window eviction policies to accommodate CMR's new result delivery semantics.

4.1 Leveraging Mortar

Instead of building Continuous MapReduce from the ground up, we decided to leverage a pre-existing system called Mortar [28]. Mortar is a distributed stream processing platform for building in-network aggregates across federated systems. Considering the functions that make up the MapReduce programing model, Mortar was a natural choice to use as a substrate for building CMR; both combine and reduce are typically aggregate functions and map is an in-network filter.

Mortar's implementation is also in line with CMR's design philosophy. It is scalable, targeting an environment with up to tens of thousands of nodes and processing data using an in-situ approach. It is responsive, designed for real-time applications and placing a large importance on result latency. And it is highly available, using a relaxed consistency model, a best-effort and gap recovery policy, to return results to the user even when there are many failures.

Mortar is a data-driven system that processes data as it arrives. Each query consists of a single operator, or aggregate function, which Mortar replicates across a set of nodes to ensure that there is an operator at every data source. In Mortar, because it processes data at the source, all operators are "pinned" [8]. This placement enables in situ, or on location, processing of data. Any Mortar node can accept, compile, or inject new queries into the system. Each query is defined by its operator type and produces a single, continuous output data stream. Operators push, as opposed to the pull-based method used in Hadoop, tuples across the network to other operators of the same type.

There are two types of queries in Mortar, local and in-network queries. A local query is one that subscribes to and processes a local data stream at a single node. This data stream is generated by a data source residing on the same Mortar node. An in-network query is one that performs an aggregation of data across multiple nodes. Like a local query it may also subscribe to a local, raw data source, or to the output of an existing query. In-network queries aggregate the partial results of upstream operators in the same query.

These query types can be composed together to accomplish data processing tasks. For example, one may write a local operator that filters CPU load statistics from server log data. Mortar replicates and installs the operator across all nodes in the system. Then, one may write an in-network operator that aggregates the filtered CPU load statistics and calculates the average across the system. Mortar processes this log data continuously as it is generated.

Like other stream processors that operate continuously, Mortar uses processing windows to bound computation. Each query can create their own window specification. A window specification includes a range, how much time or data to compute across, and a slide, how often the window should be updated.

Mortar provides a simple API to facilitate programming aggregate operators and enable incremental processing. An operator only needs to provide a *merge* function, that the runtime calls to inject a new tuple into a window, and a *remove* function that the runtime calls to remove tuples from a window. An operator adds data to the window as it arrives, and removes data that is no longer within the window range. For example, an operator calculating a sum would add new values and subtract expired values from the aggregate sum.

Even though Mortar has many features that align with CMR's design, it is still necessary to modify Mortar's core functionality to accommodate the MapReduce programming model and CMR's primary application of log processing. We modify Mortar's timestamp mechanism, eviction policy, fault-tolerant mechanism and the Mortar Stream Language (MSL). We explain these modifications in detail throughout the rest of this chapter.

4.2 The anatomy of Continuous MapReduce

A basic CMR job (a single MapReduce job) consists of two Mortar queries, one that defines a map operator and one that defines a reduce operator. The Map operator is a local query; it processes all the records a data source (log file) produces at a single

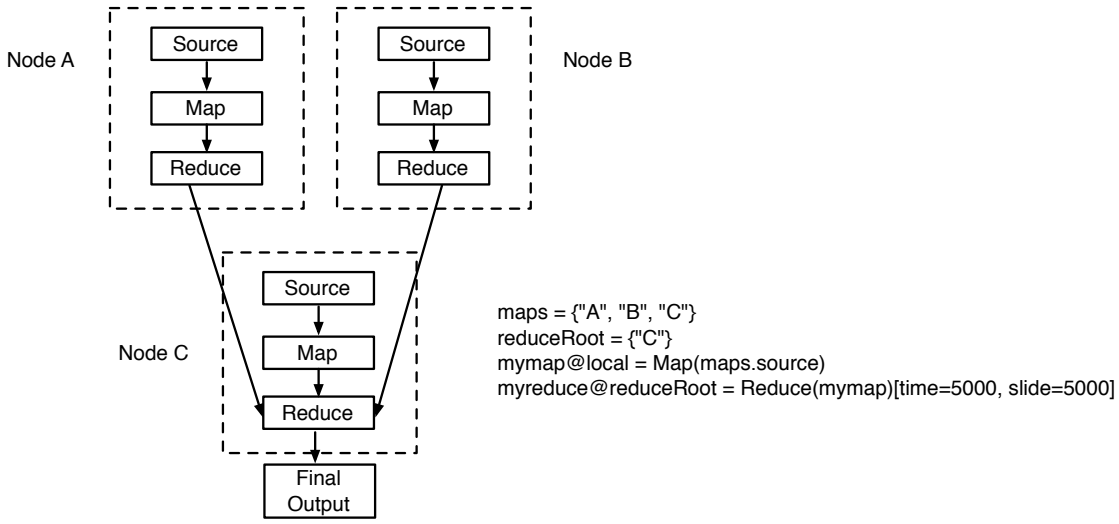

Figure 4.1: A basic CMR job consists of two Mortar queries. The MSL code that generates this job specifies the type of operators, the set of nodes participating in each query, and the window specification. A time window is specified with a range and slide equal to 5 seconds (5000 ms).

node. The Reduce operator is an in-network query; it aggregates data across all the Map nodes, grouping key-value pairs with a common key. We implement Map and Reduce by implementing the merge and remove API for each operator type; these functions up-call the user-defined map, partition, combine, uncombine and reduce functions (Section 3.2).

To define a query in Mortar, a user needs to explicitly specify where they run. Map queries simply run on all the nodes that produce data (i.e. where the logs are). For the reduce query, a user specifies where the query draws its data and where the system should deliver the final aggregate result (i.e. the root of the query). Mortar will then arrange the reduce operators into a tree based on the source and root nodes specified in the query.

Figure 4.1 illustrates a basic CMR job constructed using Mortar with a map and reduce query. Notice that because operators are pinned, an instance of a reduce is placed at each mapper. This benefits in-situ processing as it gives CMR the opportunity to actively filter and reduce intermediate data (using the optional MapReduce combine function) before it is sent across the network.

Figure 4.1 also displays the Mortar Stream Language (MSL) code used to generate the query tree. MSL enables users to specify queries in a boxes-and-arrows fashion, allowing queries to subscribe to one another. MSL has two types of statements; a node set (i.e. *maps* and *reduceRoot*) and query definitions. In Figure 4.1 *mymap* is a local map query placed on nodes in the *maps* node set. *Myreduce* is an in-network reduce query with a root on the *reduceRoot* node that subscribes to the *mymap* query. If an in-network query requires multiple reducers, additional nodes can be added to the *reduceRoot* node set.

4.2.1 A Mortar operator

To understand how we implement Map and Reduce using the merge and remove API, we must first give a quick overview of a Mortar operator. An operator consists of two major components, the pv-list and the time-space list. These components preform two primary tasks; the aggregation of input data records (raw values) into window slides (partial values), and the merging of window slides across multiple nodes (in-network aggregation). The pv-list and time-space list use the user-defined merge function to perform both of these tasks. Figure 4.2 illustrates a Mortar operator in detail.

The pv-list aggregates raw values incrementally based on the operator's window specification. As the pv-list receives data tuples, it calls the merge function on each tuple, aggregating the raw values into one partial value. This partial value represents aggregated data for one processing window. As soon as the pv-list receives a raw value belonging to the next processing window, it sends a tuple containing the partial value to the local time-space list. The pv-list then advances its processing window and calls the remove function to incrementally remove raw values that, according to the operator window specification, are no longer in the processing window. The pv-list generates one partial value per slide of a window.

Mortar operators use the time-space list to merge partial values originating from themselves and upstream nodes. The partial values have an immutable timestamp,

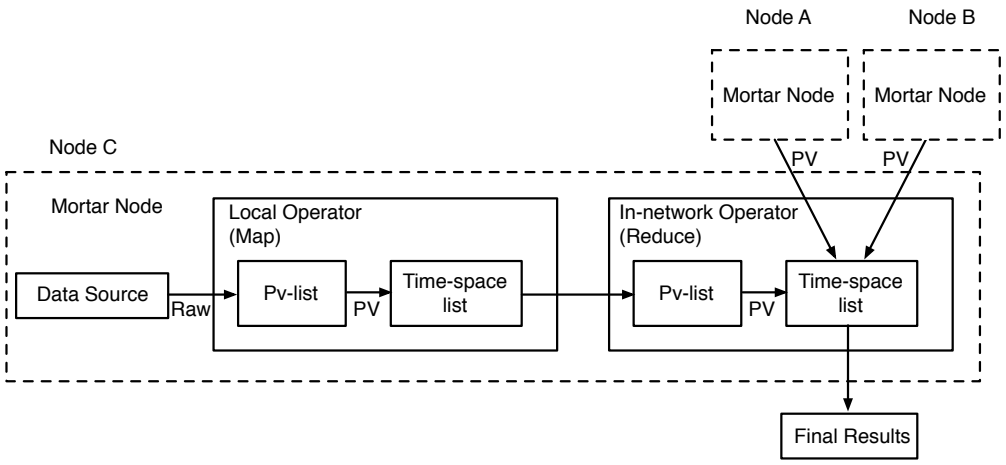

Figure 4.2: A Mortar node with two operators showing the pv-list and time-space list components.

allowing the time-space list to aggregate values based on the processing window they belong to. The time-space list does this by calling the merge function as partial values arrive.

To account for differences in processing time and network congestion, the time-space list maintains a collection of buckets, implemented as a sorted linked list, where each bucket contains values for one open processing window. A processing window is open when a time-space list has received data belonging to that window, and the time-space list has not evicted it yet. The time-space list continues to keep a processing window open and aggregates partial values in its bucket until all partial values are received from upstream operators or until the window violates Mortar's eviction policy. On eviction, the time-space list closes the window and pushes a tuple containing the aggregated values to downstream operators.

Because Mortar is a real-time system, it uses dynamic timeouts for its window eviction policy. The time-space list expires entries after a timeout based on the longest delay a tuple experiences on a path to the current operator. Since CMR is geared toward large-scale data processing, a significant amount of delay might be introduced due to processing. This delay depends on the amount of data in each window, and might vary

significantly between windows. As a result, CMR does not use the dynamic timeout mechanism, and instead we implemented a set of different eviction policies described in section 4.5.

4.3 Operator modifications

We provide the Continuous MapReduce API by implementing generic map and reduce operator types with custom merge and remove functions. These custom merge and remove functions call the user-defined MapReduce functions at the appropriate time, and properly group the key-value pairs. We also modified operator internals to provide features such as sorting and partitioning of key-value pairs and eviction of processing windows.

4.3.1 The map operator

The map operator's merge calls the user's map function each time an upstream data source pushes a tuple to the operator. The merge expects these tuples to contain a single log entry, applies the map to the log entry, which either emits one or more key-value pairs or nothing at all. We optimized the map operator by permanently assigning it a tuple window specification with a range and slide equal to one. Furthermore, since the map operator is always a local operator, we modified internal methods to push partial values directly to downstream operators, allowing partial values to skip the time-space list.

Mortar allows operators to push values to other co-located operators. To support multiple MapReduce partitions, we modified the map operator to partition key-value pairs across subscribed reduce operators (one operator per partition). Figure 4.3 illustrates the new partitioning semantics. Previously, a Mortar operator would copy the entire partial value to each subscribed operator.

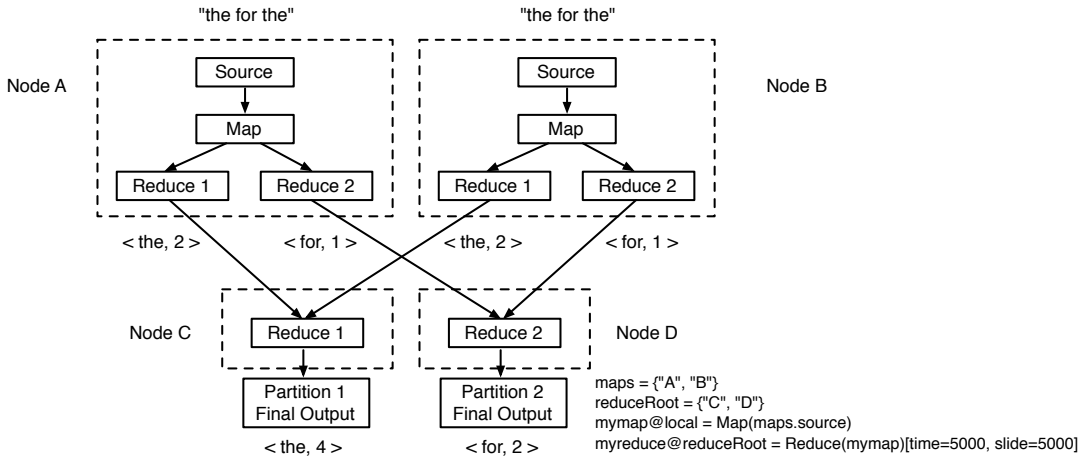

Figure 4.3: A CMR job containing two MapReduce partitions. A word count example shows the partitioning of key-value pairs across multiple reduce operators. The MSL code that generates this job is also displayed.

4.3.2 The reduce operator

The reduce operator handles all the in-network functionality of CMR including the grouping, combining, sorting and reducing of key-value pairs. The operator's merge either calls the combine or reduce based on where the calling operator is located in the query tree. The operator calls the reduce function if it is the root of the tree, otherwise it calls the combine function instead. This prevents the reduce operator from calling the reduce function before merging partial values from all available map operators. Grouping of key-value pairs is handled efficiently using a hash map. If the user requires sorted final results, the reduce operator uses a sorted hash-map instead. This maintains the grouped key-value pairs in sorted order (by key) throughout the system.

The combine and uncombine functions allow the pv-list to process data incrementally. The merge applies the combine function on each group of key-value pairs just before the pv-list pushes the partial value to the time-space list. After the pv-list advances the window, the merge will call the uncombine function to remove key-value pairs that are no longer in the window.

When the time-space list receives a tuple, it inserts the tuple into the list ac-

cording to the tuple's timestamp. If a bucket for the window already exists, the time-space list inserts the new key-value pairs into the bucket's hash-map, merging them with the pre-existing key-value pairs by calling combine. If a bucket does not exist, the time-space list creates one and inserts the new key-value pairs. Once a bucket is ready for eviction (based on the modified CMR eviction policies described in Section 4.5), the contained partial value is sent to subscribed operators and the bucket is removed from the time-space list. If the operator is at the root, then the operator's merge calls reduce on the window's key-value pairs and the final results are delivered to the user.

4.3.3 Timestamp propagation

To adapt Mortar for a large-scale log processing application, we need to change the way timestamps travel through the system. Mortar is a real-time data processing system, and it processes tuples as they arrive at an operator. As new tuples are produced, the operator treats them as an entirely new data stream and assigns a new timestamp to the output tuples. As a result, the same data may be given multiple timestamps due to network congestion or processing time. This is perfectly fine in Mortar as each query is logically separate. However, in CMR queries are logically grouped together to form MapReduce jobs, and data must retain the same timestamp across multiple queries.

To accomplish this, CMR assigns a timestamp to data as it enters the system (either a timestamp from the log entry, or the current real time). This timestamp is immutable, and remains with the data as it travels through the system. If data is delayed due to network congestion or processing time, the data must still be placed in the window corresponding to its timestamp regardless of the amount of delay introduced by the system. Therefore, the passing of real time no longer corresponds to forward progress.

4.4 Dealing with delay and failure

CMR's relaxed best-effort consistency model allows for a highly available system, as well as accommodating for inconsistent data rates due to computational load at the nodes or network congestion. CMR's consistency model has two main facilities: the boundary tuple mechanism and the reconciliation algorithm. Both of these facilities assume a fail-restart failure model; a CMR node is either functioning correctly or not functioning at all. Byzantine, or malicious, behavior is not addressed in this work.

4.4.1 The boundary tuple mechanism

Because CMR requires immutable timestamps and new eviction polices, we modified and extend Mortar's pre-existing boundary tuple mechanism. A boundary tuple is a control message sent between operators that contains progress information. This allows the receiving operator to distinguish between a pause in the data stream and a failed node. Boundary tuples in CMR, as compared to Mortar, are now sent at different times, contain additional progress information, and trigger new actions on reception.

Mortar's original boundary tuple mechanism is similar in spirit to the boundary tuples used in the Borealis system [9]. Mortar creates boundary tuples from the pv-list, and the time-space list propagates them. For time windows, a boundary tuple's only purpose is to update a window's completeness metric (a count of the number of participants that contributed to the data contained in the window). They are always piggy-backed with a regular data tuple. For tuple windows, boundary tuples indicate that a stream has stalled and that the window must remain open in case new data arrives. They also contain a copy of the window's current partial value.

In CMR, boundary tuples are logically distinct from data tuples, and are sent once every boundary tuple period for both time and tuple windows. The boundary tuples contain progress information that allows users to specify a minimum result fidelity. The boundary tuple period, the contents of progress information, and the way these mechanisms are used in CMR are described in Section 4.5.

Because boundary tuples in CMR are now logically distinct from data tuples, when a time-space list receives a boundary tuple it does one of three things. If the tuple corresponds to an existing open processing window, the time-space list updates the boundary tuple statistics for that window (the time-space list maintains the most recent boundary tuple timestamp received for each open bucket in the list). If the tuple corresponds to a processing window that has not been open yet, it inserts a new bucket in the list for that window. Finally, if the tuple corresponds to a window that has already been evicted, the time-space list ignores the tuple.

4.4.2 The reconciliation algorithm

One of CMR's design goals is to minimize result latency at the cost of lower fidelity results. CMR is a best-effort system, and data that is lost due to failure is not recovered. However, CMR does guarantee that queries will be installed and removed on nodes in an eventually consistent manner.

CMR leverages Mortar's query persistence and pair-wise reconciliation algorithm. Mortar manages queries in a top-down fashion, allowing nodes who miss install or remove commands to reconcile with upstream or downstream nodes. Periodically, parent-child node pairs exchange summaries describing shared queries. They exchange their current set of installed queries and their current set of cached query removals. A centralized object store issues sequence numbers for each management command in a query. These sequence numbers determine which query view is most up to date.

4.5 Eviction policies

This section describes CMR's three eviction policies used to support the best-effort consistency model described in Section 3.3. These policies were chosen to balance between result fidelity and system responsiveness. They define how the time-space list closes and evicts windows when delivering results to the user. The three eviction policies include the timeout, fidelity, and failure eviction policies.

It should be noted that CMR ensures processing windows are delivered to the user in order by enforcing the in-order eviction of windows. The time-space list will only evict its oldest open window; it will also ignore any data it receives pertaining to a window that it has previously evicted.

Timeout Eviction: The timeout eviction policy delivers results based on a maximum time limit for window completion. The time-space list maintains a timeout period that represents the maximum time the system can spend processing the oldest open window. If the timeout period expires, the time-space list evicts the oldest window regardless of its completeness. The time-space list then restarts the timeout period for the new oldest window.

Fidelity Eviction: The fidelity eviction policy delivers results based on a minimum window fidelity. We calculate window fidelity based on the current percentage of the window that is processed and included in the results (i.e. the amount of total progress). Users can specify a threshold representing the minimum fidelity required for a window. Once a window reaches the minimum fidelity threshold, the window is evicted.

The pv-list calculates progress information as it incrementally processes data. In a time window, the pv-list calculates progress based on the timestamp of the most recent data it has processed. For example, if a window ranges from 5:00 pm to 6:00 pm and the pv-list has processed all data up until 5:30 pm, then the window is considered to be 50% complete. Tuple windows are conceptually similar except tuples are the base unit instead of time. The pv-list sends this progress information to subscribed nodes via boundary tuples as described in Section 4.4.1.

The time-space list receives progress information from upstream operators and merges it for each open processing window. The time-space list averages the progress information across all participating upstream operators, and forwards this information via a boundary tuple to the time-space lists residing on downstream subscribed operators. As described above, once the averaged progress of the oldest window reaches the minimum fidelity threshold, the time-space list evicts the window.

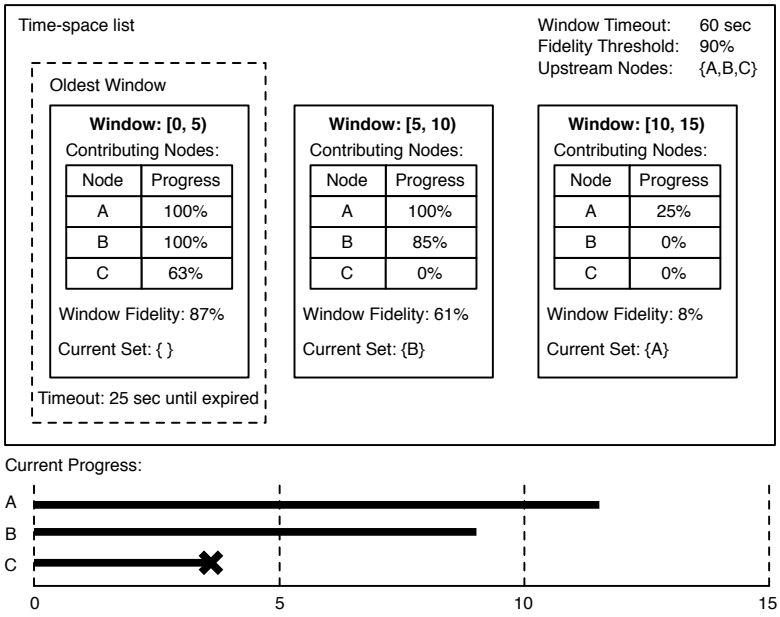

Figure 4.4: A time-space list snapshot as window [0, 5) is closed due to a failure eviction.

Failure Eviction: The failure eviction policy enables CMR to remain responsive when failure occurs. The time-space list maintains a set of contributing upstream operators for each open window. The time-space list adds an operator to a set as soon as it starts processing data for that window. The time-space list removes an operator from a set if it is no longer processing data for that window (either due to completion or failure). The time-space list monitors the cardinality of the oldest window's set. As soon as the oldest window's set is empty (i.e. there are no longer any upstream operators processing data for the oldest window), the time-space list evicts the oldest window.

These sets are maintained in a best-effort manner using boundary tuples and are updated once every boundary tuple period. The boundary tuple period is specified in real time (i.e. wall clock time), and determines the frequency at which boundary tuples are sent (one per period). The boundary tuple period enables CMR to trade off between system responsiveness and communication overhead.

Figure 4.4 is a snapshot of the time-space list at a root node subscribed to three upstream operators. In this snapshot, the upstream operator at node C has failed 63%

through window [0, 5) and has stopped sending boundary tuples. As a result, the time-space list removed node C from the [0, 5) window set, reducing the set's cardinality to zero and qualifying the window for eviction. Once window [0, 5) is evicted, window [5, 10) will become the oldest window and the timeout will be reset to 60 seconds. At the bottom of Figure 4.4, a chart illustrates the current progress of each upstream operator at the time of the snapshot: C has failed, B is 85% through window [5, 10), and A is 25% through window [10, 15).

Chapter 5

System Evaluation

This chapter discusses the evaluation of our prototype implementation of the Continuous MapReduce architecture. We focus our evaluation on the responsiveness and the availability of CMR. We evaluate our prototype along three axis:

- Result latency relative to other batch processing systems (specifically Hadoop and HOP).
- Scale and performance benefits of incremental processing and sliding windows for continuous queries.
- Tradeoffs of a relaxed consistency model in the presence of failure.

We use Hadoop 0.19.1 and HOP 0.1 as points of comparison for the current implementation of CMR. These systems represent a popular batch processing system (Hadoop) [3] and a cutting edge modified version of Hadoop (HOP) [13]. We use default settings for both systems unless otherwise stated. We use a local-area cluster consisting of 16 Dell PowerEdge SC1425 workstations as a test bed for all of our experiments. Each machine is equipped with dual Intel Xeon 2.8 GHz processors, 4 GB of RAM, and a 1 Gbps network interface card. They all run Linux CentOS 4.5 and are connected to the same 1 Gbps non-blocking switch.

For the primary test application, we implement a distributed grep using the MapReduce programing model. Leveraging the MapReduce framework, distributed

grep scans through an input data set in parallel and returns the total number of occurrences for a given regular expression. This represents a distributive application that is commonly used in large-scale text processing.

We implement the appropriate functions identically for all three test systems as follows. The reader function takes a specified input file and pushes the containing text line by line to the map function. The map function takes each line and uses the *java.util.regex* API to find occurrences of the user-specified regular expression. When an occurrence is found, the map function emits a key-value pair with the matched pattern as the key and an integer "1" representing the occurrence as the value. The combine and reduce functions are identical, and group emitted key-value pairs together and sum the occurrences. Additionally, we implement an uncombine function for CMR allowing the distributed grep application to take advantage of incremental processing. The uncombine function subtracts key-value pairs from the current partial value.

We use 48 GB of Wikipedia data as our data set throughout the evaluation. For all experiments, our distributed grep application searches for occurrences of the pattern "the" within this data set. We split the data into 16 files and spread it evenly across all nodes. In the case of CMR, we place one data file on each node. In the case of Hadoop and HOP, we load 16 data files into HDFS and allow HDFS to distribute the data amongst the nodes via its block placement policies. We set HDFS data block replication to one for the batch and incremental experiments, and three for the failure experiment.

We limit MapReduce queries to a one-level tree with a single instance of the reduce function, ensuring that all systems aggregate results to one location. We consider results "delivered to the user" once the systems write them to permanent storage either on local disk or HDFS. We place map operators on all 16 nodes during CMR experiments. We leave map and reduce task placement in Hadoop/HOP up to the respective system placement policies. These placement policies distribute data blocks and tasks evenly across all 16 nodes.

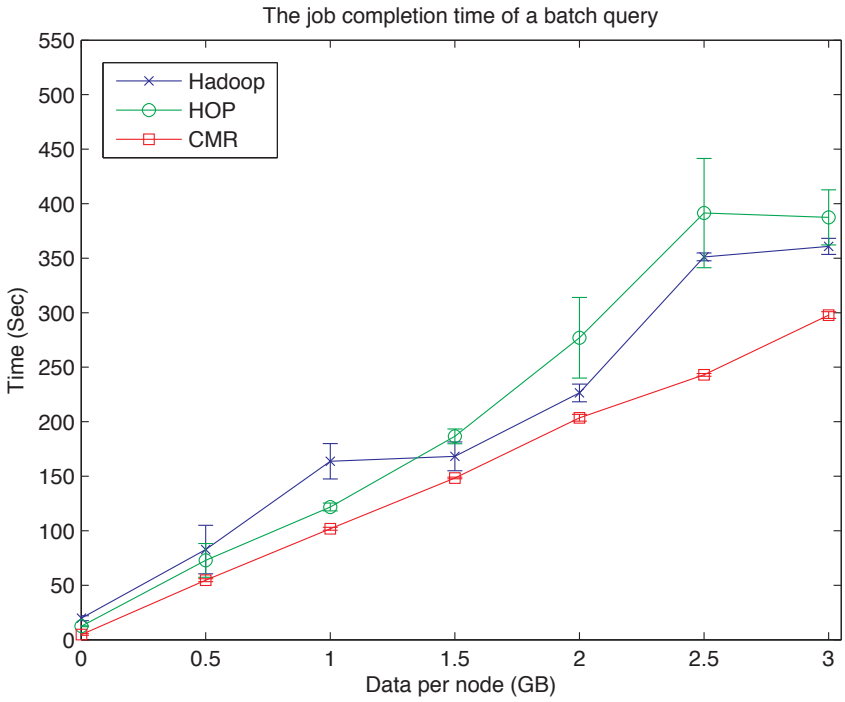

Figure 5.1: The job completion time of a batch MapReduce query with varying amounts of data per node. Error bars indicate a 95% confidence interval.

5.1 Batch queries

To test CMR's latency for processing large amounts of data, we compare it to both Hadoop and HOP by submitting one-shot batch queries over varying amounts of input data and recording the total job completion time. Job completion time includes all the time from query submission to result delivery. For CMR, this includes query installation time as well. These times do not include data migration time, assuming the input data is already in HDFS.

Figure 5.1 displays the results of our batch query experiment set. We increase the amount of input data per node by 512 MB, and average the results over five runs for each data point. CMR consistently produces lower job completion times compared to both Hadoop and HOP. One source of CMR's performance gain is from its avoidance of disk I/O as compared to Hadoop. CMR completely resides in memory, and does not

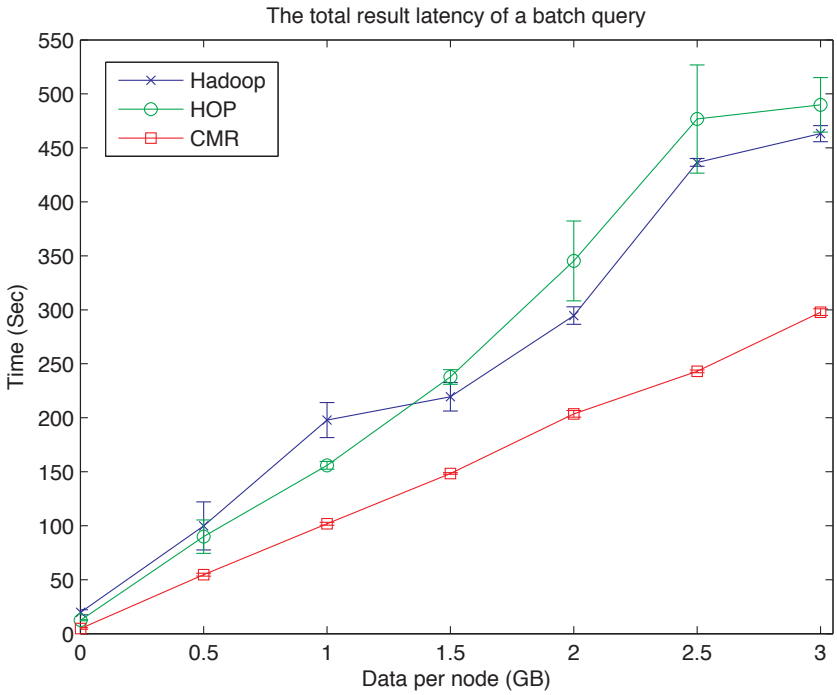

Figure 5.2: The total result latency (i.e. the sum of job completion and data migration time) of a batch MapReduce query. Error bars indicate a 95% confidence interval.

spool intermediate data to disk.

CMR shows comparable scalability to Hadoop and HOP scaling linearly with respect to the data per node. With 3 GB of data per node, CMR shows a consistent per node data processing rate of over 10 MB/sec, versus Hadoop's 8.5 MB/sec and HOP's 7.9 MB/sec data rate. CMR's data rate meets or exceeds current log data accumulation rates per node (see section 1.1). We expect CMR to continue this performance trend until the data per window at each node exceeds the size of memory. At which point, its performance will degrade. Approaches to relieving memory pressure are a subject of future work (Section 5.4).

If we take data migration time into consideration, it is clear that CMR's query-then-store model can significantly increase the responsiveness of large scale data processing. Figure 5.2 displays the total result latency as data per node increases. The total result latency is the sum of job completion time and data migration time. We assume

Table 5.1: The effectiveness of incremental processing with various amounts of window overlap.

Percentage Overlap	Add Values (msec)	Remove Values (msec)	Total Time (msec)	Number of Operations
0	1883	0	1883	N
25	1386	1706	3093	1.5N
50	916	1135	2051	N
75	441	581	1022	.5N

that all nodes in the cluster are able to sink data to HDFS at a rate of 30 MB/sec in parallel, and that data processing can not be pipelined with migration. We measured this per node rate during previous experiments. At 3 GB of data per node, CMR delivers results 35% faster than Hadoop. This performance gain will only improve as the amount of data increases.

5.2 Continuous queries and incremental processing

Next we evaluate CMR's ability to process continuous queries. For this set of experiments we keep the window range equal to 512 MB of data and vary the slide. By changing the size of the slide we see how the amount of common data between windows, or reusable computation, changes the effectiveness of incremental processing.

As described in Section 4.3, the work required to incrementally update a partial value for the next window can be divided into two distinct parts; removing expired values that are no longer in the window, and adding new values that have just entered the window. We instrument CMR and precisely record the amount of time an operator spends doing both of these tasks and average the measurements over the creation of 5 processing windows. We vary the amount of overlap by steps of 25% from 0% overlapping (hopping window) to 75% overlapping.

In these micro benchmarks incremental processing may not always be advantageous. Table 5.1 displays the results from these experiments, which include the the actual time spent on each task, the total amount of time spent on both tasks, and the

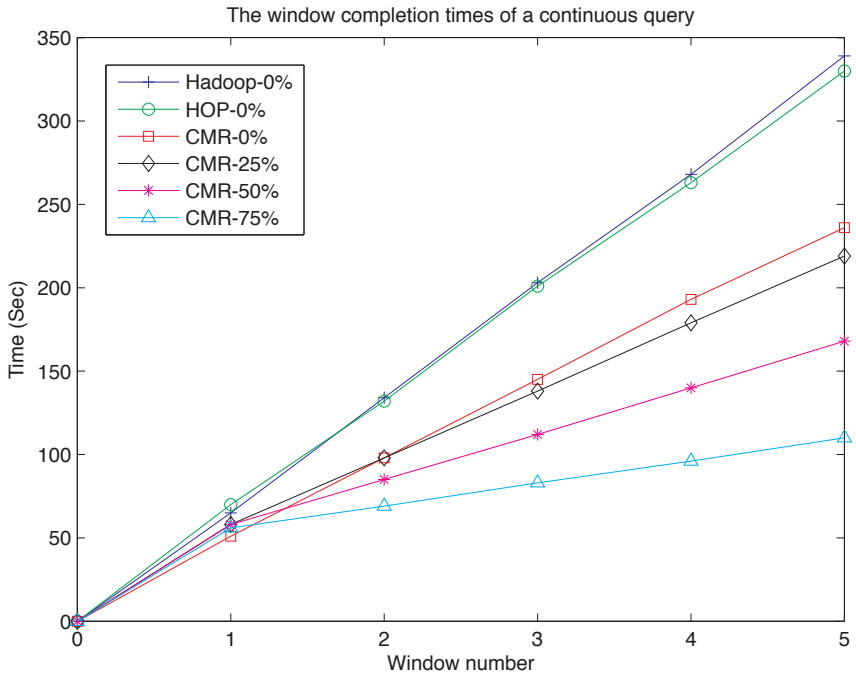

Figure 5.3: The window completion times for a continuous MapReduce query.

expected number of operations required to update a partial value for each window specification. As the table indicates, if adding and subtracting values from the partial value take relatively the same amount of time, it is advantageous to use incremental processing for window specifications with more than a 50% overlap. Otherwise, incrementally adding and subtracting values takes more operations than recomputing the entire window from scratch. However, using a smaller slide reduces the amount of data that must be reprocessed by the map and transfered across the network to produce the next window.

Figure 5.3 shows the completion times for each window in a continuous query with a range of 512 MB. We start timing as soon as we submit the query, which includes query installation in CMR, and we assume that data is already in HDFS or on the local hard disk. Each data point represents the total elapsed time at the moment the system delivers results for that window. For Hadoop, we simulate the ability to handle continuous queries by submitting back-to-back MapReduce jobs over different 512 MB chunks

of data. Each CMR line corresponds to a different amount of overlap between windows (i.e. the specifications displayed in Table 5.1).

The first thing to notice is that both the 25% and 50% overlapping windows perform better than the 0% hopping window. This result is interesting because Table 5.1 shows that more work is actually being done in the 25% and 50% cases, indicating that performance should be worse. Instead, the total elapsed time seems to correlate directly to the size of the slide. This correlation is because as the size of the slide increases, the map must reprocess and send more data across the network to the reduce operator. The map must complete all of this reprocessing before the reduce can deliver results for the next window. This processing time is the dominating factor compared to the time spent adding and removing values from the window.

5.3 The impact of failure

Finally, we evaluate the effect failure has on the job completion time and result fidelity for CMR and Hadoop. Our experiment consists of running a batch query with a window range of 512 MB. We permanently fail an increasing number of nodes (ranging from zero to six) for each repetition. We select these nodes at random, and fail all of them once 50% of the map phase is complete. We record the job completion time and the result completeness. We define result completeness as the percentage of input data the results take into account. We assume the input data is already in HDFS or on local disk.

Figure 5.4 displays the job completion times for both Hadoop and CMR regardless of whether results are actually delivered. CMR's relaxed consistency model produces far lower job completion times and is unaffected by the number of nodes we fail. Hadoop on the other hand, produces drastically higher completion times when we fail a single node.

It is important to note that Hadoop can be tuned to classify a failed node based on a maximum number of reconnection tries and a maximum backoff time. We use the

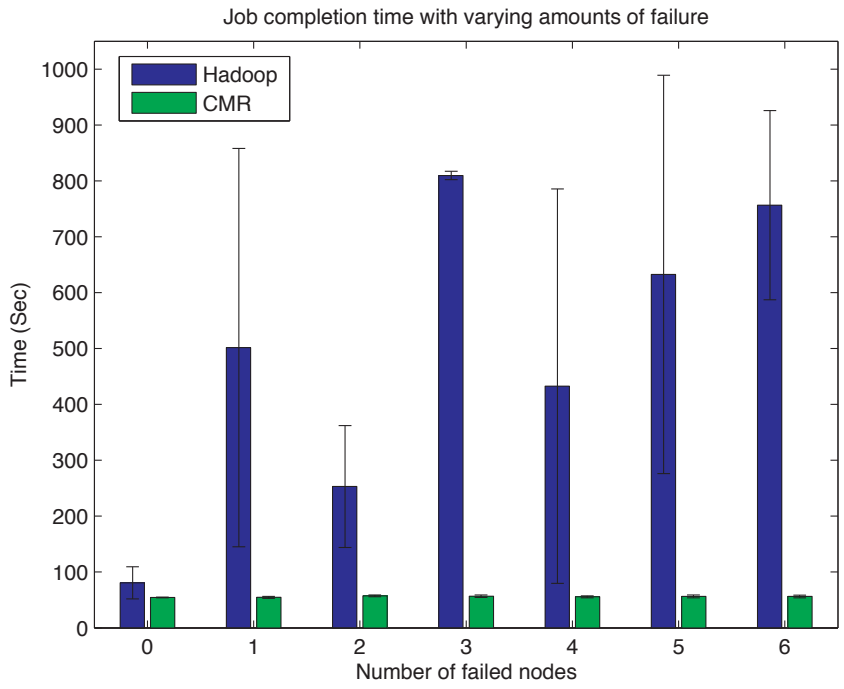

Figure 5.4: The job completion time of a batch MapReduce query with varying amounts of failure. Error bars indicate a 95% confidence interval.

default values for both parameters, but they could be tuned more aggressively. These two parameters control the exponential back-off that a reduce task carries out when unsuccessfully requesting data from a node. It is only after this back-off that the reduce task notifies the job tracker and the node is classified as failed. However, if these parameters are tuned too aggressively, this will force Hadoop to falsely classify nodes as failed and greatly increase job completion time when no failures are present. In a large-scale homogeneous environment, we expect these parameters to be difficult to tune correctly.

CMR's relaxed consistency model also produces more complete results in a high failure environment. Figure 5.5 shows that as the failure rate increases, CMR gradually returns less complete results directly corresponding to the amount of failure. Since we distribute data evenly across all nodes during our experiments, each failed node reduces CMR's result completeness by 6.25%. Hadoop, on the other hand, does not return anything if it can not return fully consistent results. In the case where we fail

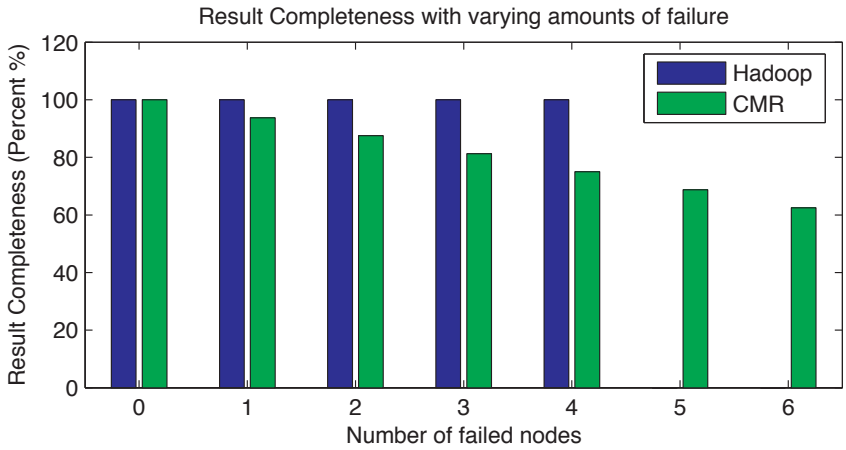

Figure 5.5: The result completeness of a batch MapReduce query with varying amounts of failure.

five or six nodes, Hadoop's job completion time increases by over ten times the failure free case. In addition, for both these cases Hadoop does not return any results to the user due to unavailable data blocks. In contrast, when we fail 6 nodes CMR returns results that are 62% complete in the same amount of time as the failure free case.

Hadoop can not successfully complete the MapReduce job in the high failure cases because of unavailable data blocks. Even though each data block is replicated three times and spread across multiple data nodes, there is still a chance that all replicas of a single data block may fail. Once a single data block is unavailable, Hadoop's fully consistent model prevents the system from returning any results. To maintain usability of the system in a high failure environment, Hadoop administrators are forced to increase data block replication. As the accumulation of log data grows, replication may not be feasible.

5.4 Future work

Based on our evaluation of Continuous MapReduce, there are several promising areas that we would like to explore in future research. One promising direction is the use of window panes to relieve memory pressure and to return early results to the

user [27]. In this approach, CMR divides overlapping windows into disjoint panes, or sub-windows, and processes them separately. As soon as an operator finishes processing a pane, it pushes the pane to subscribed operators. The root reduce operator aggregates all panes that belong to the same window. This technique relieves memory pressure at in-network nodes, and would allow CMR to process windows that exceed the size of memory without materializing intermediate data to disk. The use of panes also facilitates the delivery of early results to the user by enabling the root reduce operator to cumulatively process panes as they arrive. Finally, the use of panes would also provide higher utilization throughout the system, which would presumably reduce query latency as well.

Some other promising areas include dynamically scaling the number of map and reduce operators to accommodate changes in data rate for continuous queries, evaluation of different data processing applications, and exploration of CMR's performance in wide-area networks spanning multiple data centers.

Chapter 6

Conclusion

This thesis presents Continuous MapReduce (CMR), an architecture for parallel data processing in a large data center environment. This architecture adopts an in-situ approach to avoid costly data migration. This approach allows for scalability, responsiveness and availability in a large heterogenous setting where node failure is always present. CMR accomplishes this by taking distributed file systems off the critical path, handling continuous queries efficiently using incremental processing, and relaxing the consistency model.

We implemented a new prototype MapReduce framework for data processing that uses the CMR architecture. The prototype is implemented on top of Mortar [28], and adapts mechanisms and techniques from distributed stream processors to enable bulk processing over continuous streams of data. We have modified a substantial amount of Mortar including its programing API, timestamping mechanism, and data eviction policies to provide the expected semantics of a MapReduce framework.

We evaluated the CMR prototype and compared it to a production MapReduce framework Hadoop [3], a popular open-source project developed in industry based on Google's proprietary MapReduce implementation. We showed that CMR can significantly improve result latency not only when failure is present, but when batch processing in a failure-free environment as well. We highlighted incremental processing as an effective way of reusing computation by leveraging common data between processing

jobs. We also displayed that in a high failure environment, CMR maintains a low result latency with accountable fidelity.

Finally, in this thesis we have created a foundation for more research investigating large-scale distributed data processing with Continuous MapReduce. We have identified future research directions to pursue including the use of panes [27], the evaluation of different data processing applications, and the exploration of CMR's performance in wide-area networks spanning multiple data centers. Continuous MapReduce is a promising approach to large-scale distributed data processing and will hopefully enable cloud providers to address the next generation of data management challenges.

Bibliography

[1] http://www.datacenterknowledge.com/microsofts-dublin-data-center-room-to-grow/.

[2] http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/.

[3] Apache hadoop. http://hadoop.apache.org.

[4] Apache hadoop applications. http://wiki.apache.org/hadoop/PoweredBy.

[5] Apache mahout. http://lucene.apache.org/mahout/.

[6] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *Proc. of the Second Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[7] S. Agarwal. Volley: Automated data placement for geo-distributed cloud services. UCSD Center for Networked Systems Lecture Series, Nov 2009.

[8] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 456–467. VLDB Endowment, 2004.

[9] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 13–24, New York, NY, USA, 2005. ACM.

[10] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa: A large-scale monitoring system. In *Cloud Computing and its Applications*, pages 1–5, Chicago, IL, USA, October 2008.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, New York, NY, USA, 2003. ACM.

[12] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[13] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[14] J. Dean. Large-scale distributed systems at google: Current systems and future directions. Keynote #3: LADIS 2009, Oct 2009.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, page 10, Berkeley, CA, USA, 2004. USENIX Association.

[16] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[17] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR '09: 4th Biennial Conference on Innovative Data Systems Research*, 2009.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[19] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[20] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and Z. Lidong. Comet: Batched stream processing in data intensive distributed computing. Technical Report MSR-TR-2009-180, Microsoft Research, December 2009.

[21] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 171–182, New York, NY, USA, 1997. ACM.

[22] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 779–790, Washington, DC, USA, 2005. IEEE Computer Society.

[23] J.-H. Hwang, Y. Xing, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering*, 2007.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[25] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow.*, 1(1):574–585, 2008.

[26] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for snps with cloud computing. *Genome biology*, 10(11):R134+, November 2009.

[27] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, 2005.

[28] D. Logothetis and K. Yocum. Wide-scale data stream management. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 405–418, Berkeley, CA, USA, 2008. USENIX Association.

[29] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with seaweed. *The VLDB Journal*, 17(2):315–331, 2008.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[31] S. Pallickara and G. Fox. Naradabrokering: a distributed middleware framework and architecture for enabling durable peer-to-peer grids. In *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 41–61, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[32] R. Ramakrishnan. Data serving in the cloud. Keynote #1: ACM LADIS 2009, Oct 2009.

[33] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[34] J. Rothschild. High performance at massive scale - lessons learned at facebook. UCSD Center for Networked Systems Lecture Series, Oct 2009.

[35] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838, New York, NY, USA, 2004. ACM.

[36] 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. *Proc. VLDB Endow.*, 2009.

[37] R. Winter. Why are data warehouses growing so fast? http://www.b-eye-network.com/view/7188, April 2008.

[38] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, New York, NY, USA, 2009. ACM.

[39] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[40] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 247–260, New York, NY, USA, 2009. ACM.

[41] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI '08: 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.