ABSTRACT

Title of dissertation: Pig Squeal:
Bridging Batch and Stream Processing
Using Incremental Updates

James Holmes Lampton, Jr.,
Doctor of Philosophy, 2015

Dissertation directed by: Professor Ashok Agrawala
Department of Computer Science

As developers shift from batch MapReduce to stream processing for better latency, they are faced with the dilemma of changing tools and maintaining multiple code bases. In this work we present a method for converting arbitrary chains of MapReduce jobs into pipelined, incremental processes to be executed in a stream processing framework. Pig Squeal is an enhancement of the Pig execution framework that runs lightly modified user scripts on Storm.

The contributions of this work include: an analysis that tracks how information flows through MapReduce computations along with the influence of adding and deleting data from the input, a structure to generically handle these changes along with a description of the criteria to re-enable efficiencies using combiners, case studies for running word count and the more complex NationMind algorithms within Squeal, and a performance model which examines execution times of MapReduce algorithms after converted.

A general solution to the conversion of analytics from batch to streaming

impacts developers with expertise in batch systems by providing a means to use their expertise in a new environment. Imagine a medical researcher who develops a model for predicting emergency situations in a hospital on historical data (in a batch system). They could apply these techniques to quickly deploy these detectors on live patient feeds. It also significantly impacts organizations with large investments in batch codes by providing a tool for rapid prototyping and significantly lowering the costs of experimenting in these new environments.

Pig Squeal:
Bridging Batch and Stream Processing Using Incremental Updates

by

James Holmes Lampton, Jr.

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Acknowledgments

It is with great pleasure that I write this section because I get to thank many of the people who made this possible. Appologies in advance to those I omit.

First and foremost I would like to thank my spouse, Andrea for all your love and support through the (longer than promised) years. This accomplishment would not have been possible without your help. I look forward to spending more time with you and Henry as I stop moonlighting as a grad student.

I would also like to thank my advisor, Professor Ashok Agrawala for acting as my guide through the mysterious forest of academia. As an undergraduate in Dr. Larry Dowdy's (thank you!) Distributed Operating Systems course, I was captivated by the elegance of the Ricard-Agrawala algorithm. Little did I know I would later take your Information Dynamics course and find such a kindred spirit. Thank you for accepting me under your tutelage, for your sage advice, and for our enlightening discussions. It has been a great pleasure to work with you.

I would like to thank my boss, Dr. Pat for planting and nurturing the idea of pursuing a Ph.D. You have a knack for seeing possible outcomes and for saying the right things at the right times to make those possibilities a reality. Thank you for your support, advice, and understanding when my mind was many places at once. I look forward to bringing my focus back to making the "impossible" a reality.

I joke that Pig Squeal is the result of a bet with a Professor – an idea so absurd it seemed doomed to failure. That Professor, Dr. Jimmy Lin, deserves special thanks. Thank for taking the time to listen to my half-developed ideas and for pushing me to form a rigorous treatment of this work.

Thank you Professor Bill Rand for serving on both my preliminary and thesis committees and for the interest in this work through the years. Thank you Professor Jeffrey Herrmann, Professor Alan Sussman, and Professor Joseph JaJa for serving on my thesis committee and for taking the time to review my manuscript and provide insights to improve this work. Thank you Professor Amol Deshpande for being my Masters advisor.

Thank you to my colleagues in the MIND Lab: Dr. Matthew Mah, Ravindra Passan, Nick Gramsky, Preeti Bhargava Shroff (you're next!), Dr. Shiv Krishnamoorthy, and Dick Baer. Thank you to the graduate staff, Jennifer Story and Fatima Bangura, for helping navigate the University's process. Thanks to the UMIACS support staff for keeping the NationMind virtual machine running.

Thank you to my work colleagues for amazing technical discussions and support – specifically Charles, Rudy, and Roger. Thank you to those who shared their thoughts on the Ph.D. experience: Dr. Mark, Dr. Tom P., Dr. Tom D., Dr. Todd, and Dr. Ed. A huge thank you to someone who helped launch my careers: Dr. LeRoy.

Finally, I owe thanks to my family: my parents Stella, Don, and Jim; my siblings April, Andy, and Mary; my grandparents Katherine, Sidney and Ada; and my aunts and uncles. You are responsible for my existence and shaping me into who I am today.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The inspiration for this work is to build systems that enhance people's lives. Information is a fundamental element in the advancement of human kind. Whereas a species may improve through evolution in time, humans have developed the ability to transfer knowledge to one another and advance at a much faster pace through our shared knowledge and application of tools. Communication is the carrier of information. Initial forms of communications came from oral traditions, apprenticeships, or written/drawn form. The written word provided resiliency and timelessness to information but requires manual and expensive duplication through transcription. The Gutenberg Press created the means to cheaply copy information and provide much wider distribution. Fast forward to where we are today: it is extremely inexpensive to produce and store information. The cost now lies in finding and consuming information.

We are amidst a tipping point in computer science. The combination of Moore's law, advancements in distributed computing, capacity, as well as ubiquitous computing and sensing have provided the circumstances necessary for this to take place. It is common to hear "This year, more data will be measured than has been ever measured in the past". New techniques in data storage and analysis have made it possible to measure, store, and reason about data at greater scales than previously possible. In addition to the ability to reliably store petabytes of data, new programming constructs have made it possible to harness large clusters of resources with comparatively less training than previously possible. Finally, providers such as Amazon Web Services[1] and Rackspace[2] provide large amounts of computing resources available to anyone for a modest price.

The evolution of the creation and consumption of information has changed very rapidly in the past twenty years. We've come a long way from getting our information from word of mouth, radio, TV news, and news papers and magazines. These forms of information transfer were at a fairly low rate and often curated by an editorial process. Even early forms of communicating across the internet were limited in scale (again due to human curation) and adoption rate. The growth of social media has changed everything.

Changes can be traced through the evolution of internet search. When searching the internet as a source of reference the information being sought is typically static ("What kills bacteria?"). However, when interacting with the internet in a social manner (e-mail, chat, etc.) the information is highly dynamic, personalized, and isn't explicitly sought ("What did George think about his breakfast?"). To handle this shift, Google has re-engineered it's backend processes to lower the latency from discovery to availability in search [58]. Systems such as e-mail, chat, application notifications, Twitter, Facebook, eBay, and Craigslist allow users to generate content and interact with one another. A major issue with various streams

[1]http://aws.amazon.com/
[2]http://www.rackspace.com/

of information is collection and aggregation of content. Thankfully, many of these applications have started providing Really Simple Syndication (RSS) feeds which can be used to handle updates in a consistent manner. The next challenge then becomes filtration, prioritization, and otherwise managing the incoming deluge of information in a personalized way (such as Google's Priority Inbox [7]).

The current use of computers and the internet, fueled by social media and a global economy, generate immense amounts of information. However, the increase in data availability comes with costs. If you combine these trends with the ubiquitous mobile platforms (smart phones), along with e-mail, social media, and various other means of notifications, it can be very easy to become overwhelmed and fatigued due to being "always on". This fatigue arises from the interruption of thought processes necessary to process a new update. These interruptions can have a significant negative effect for tasks that require concentration. Once interrupted, it takes time before a person returns to the same level of concentration and work efficiency. If these interruptions come in the middle of performing complex tasks, these context switches can lead to errors, either by missing steps in a process or by forgetting a previously established assumption. The rate of information creation should accelerate and continue to do so as adoption of ubiquitous computing platforms increases. You will spend your life sifting through this information.

The same companies that collect and catalogue information are always trying to adapt to user needs. For Google, this includes the augmenting query based indexing with more user preferential delivery information such as Google Now. This is called context aware computing which aims to deliver the right information at the right time. The key elements of a context aware system are context - know how information relates to a situation, relevance - what information is relevant to a situation, and change - context changes constantly.

Within the information management realm, systems are designed to process data in large batches and store it in a manner that makes it easier to answer questions at a later time. In “Data Finds Data” [39], Jonas and Sokol make the argument that the best time to reconcile questions of interest is as soon as the information is known by the system. The realm of answering and processing information in flight is the realm of Information Flow Processing Systems [22, 4, 9, 53]. The focus of this work is on change: handling constantly changing information by addressing the shift in architectures from batch processing to information flow processing systems.

The following section will open with an example application that is commonly found in information processing system literature. The aim is to use one of the most basic computations to describe the nature of the problem at hand and to establish nomenclature to be used throughout this work. This will be followed by a problem statement and discussions of scope and significance. The chapter will close with an outline of the rest of this dissertation.

1.1 Background

This work lies within the cross section of three major areas of computer science: batch data processing, stream data processing, and performance modeling. While more detailed discussions of the particular systems being used are available in a later chapter, we feel that a basic description of each is necessary to ground the research questions discussed in this chapter.

Batch processing is concerned with analyzing static chunks of data. Such systems usually accumulate information and execute analytics at some later time – triggered by elapsed time, accumulated data volumes, or user interaction. The defining characteristics of batch systems are: high throughput and relatively high latency. MapReduce [23, 24] is the batch processing system that this dissertation focuses on. MapReduce provides a framework for distributing computations across a cluster of commodity computers. It is usually paired with complementary systems to address the issue of latency. One such example is using MapReduce to compute inverted indexes to be host by a low latency query system. Once analysts understand how data must be manipulated to answer specific questions, other techniques such as stream processing can be used to produce answers in a more timely manner.

Stream processing is concerned with analyzing data as it arrives. Historically, streaming frameworks have focused on pattern/action pairs in Complex Event Processing systems [47, 22] or on extending the SQL dialect to include various mechanisms to manage the unbounded nature of the data, such as windowing operations [30]. Streaming has recently made a resurgence in the form of more generalized programming frameworks [53, 4, 75, 32]. Stream processing systems are chiefly concerned with the tradeoffs of space and time. Storm [4] is the stream processing framework studied in this dissertation.

Performance modeling is a deep field focused on the application of mathematical models to provide a structed analysis of system performance. The aspects of performance modeling that relate to batch systems centers around scheduling and makespan analysis [28]. Makespan is the difference in time between the start and finish of a sequence of jobs. Scheduling is the mechanism for granting access to resources by tasks to complete some measure of work. Queueing theory is a subarea of performance modeling concerned with analyzing systems with continous arrival and exit of work which is the case for streaming environments. The Storm system has a special abstraction called Trident which makes use of mini-batches to provide transactional processing of incomming data. Because of this, we use aspects of both batch processing [45, **?**, 76] and queueing theory [70, 41] to analyze the runtime of converted codes. The ultimate goal of this analysis is to understand the cause of the observed performance characteristics of this system. This knowledge will help developers focus their development time on the appropriate sections for optimization and can provide insight as to where to add or reduce resources.

While SQL has long provided a means for analysts to store, retrieve, and analyze data other mechanisms exist for describing analytics [18]. For MapReduce environments, one such language is Apache Pig [55] which decomposes user provided scripts into MapReduce stages. While various mechanisms exist for converting SQL queries to streaming queries [30, 3, 15] the conversion of Pig work flows into streaming computations is an area of open research. The major issue of direct conversion centers around the synchronization mechanism of MapReduce. When a MapReduce task starts it is assumed that all the input for the execution is available and static. This aspect of the contract is broken when data arrives continously. To better understand the impact of continous data, let us consider a simple example of data processing: word count.

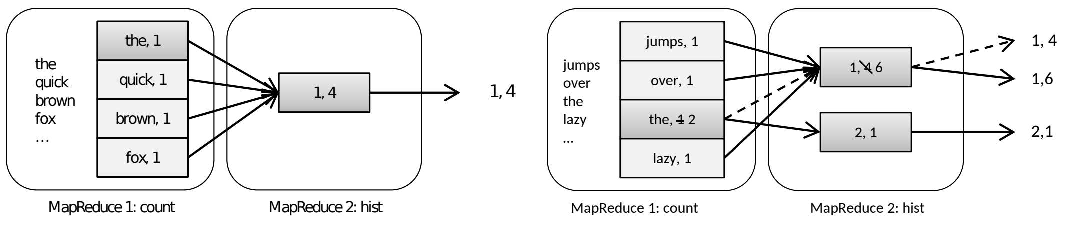
Word count is easily the “Hello World” of batch data processing. The idea is that you have a body of text that you want to analyze by examining the frequency of words within. Let us consider a particular implementation’s handling of the popular pangram “the quick brown fox jumps over the lazy dog”. The first step for word count is to *tokenize* the text – basically break the input into individual words. The second step is to group like tokens together. The final step is to count the occurrence of each to produce the answer. While the concept is simple, the devil is in the details when attempting to run any computation on a cluster of computers.

Table 1.1 provides the state of the computation through each step. The pair `(the, 1)` is a *tuple* of length two. Tuples are akin to fixed length arrays in computer science where the values can be referenced by position (zero based) or by optional field name. The columns “tokenized”, “grouped”, and “summed” represent *aliases* to *bags* of tuples. Bags are un-ordered, non-distinct, lists of tuples. The element `(the,` $\{(1),(1)\}$`)` is a tuple with a *scalar* key and a bag value containing two identical tuples. In some cases, the extra parenthesis may be omitted for brevity when the tuples are of length one – such as `(the,` $\{1,1\}$`)`. The three steps depicted here are modeled after the MapReduce [23, 24] mechanism for distributing a computation onto a cluster. The details of this mechanism will be explained in detail shortly, but for now it is important to know that results are generated by chaining together these three phases: map, shuffle, and reduce which are represented by the columns of this table. We have adopted the structure names and definitions from Apache Pig [55] (a data flow language that uses MapReduce to answer questions).

Table 1.1: Outputs for a word count algorithm.

Tokenized		Grouped		Summed	
Key	Value	Key	Values	Key	Value
the	1	brown	$\{(1)\}$	brown	1
quick	1	dog	$\{(1)\}$	dog	1
brown	1	fox	$\{(1)\}$	fox	1
fox	1	jumps	$\{(1)\}$	jumps	1
jumps	1	lazy	$\{(1)\}$	lazy	1
over	1	over	$\{(1)\}$	over	1
the	1	quick	$\{(1)\}$	quick	1
lazy	1	the	$\{(1),(1)\}$	the	2
dog	1				

Let us expand the simple word count to include a second stage that counts the frequency of the word counts for the sake of exploring chained aggregation in a streaming context (such as Storm [4]). Imagine our original input is now streamed into the system in four-word chunks. Figure 1.1a shows the counts for each word along with the resulting histogram after the first set of words are processed: there are four words that have been seen once.



(a) MapReduce states after the first four words

(b) States after the next four words with corrections

Figure 1.1: Word count histogram in a streaming context

Figure 1.1b shows the updated states and changes necessary to produce the correct results after the second set of words is seen (any unchanged states are omitted). For the first MapReduce job there are two major changes: the new count for the word “the” is two and there are three new words that occur once. These changes must be propagated to the second state: decrement "1" because the word "the" no longer contributes, increment "2" by one (again to account for "the"), and increment "1" by three to account for the newly seen hapax legomena. This results in new outputs for the second MapReduce stage which would also be propagated as: "subtract" the previously seen emission `(1, 4)` and "add" the new emissions `(1, 6)` and `(2, 1)`. Thus the correct histogram after the second batch of words is calculated.

1.2 Problem Statement

The advent of big data platforms has enabled analysis of otherwise unapproachable data sets. The largest investment in development resources has been in batch processing systems. As information consumption shifts from question oriented to alert driven interaction this leads a large technical debt in transferring existing codes and expertise from batch environments where most historical data resides into streaming environments where data first becomes available.

While techniques exist for running stream jobs in batch environments, for handling updates within batch systems [44, 60, 12, 54], and for annotation of existing codes for custom streaming environments (such as HStreaming), there are still open questions regarding the transition to streaming environments. A general solution to the conversion of analytics from batch to streaming impacts developers with expertise in batch systems by providing a means to use their expertise in a new environment. Imagine a medical researcher who develops a model for predicting emergency situations in a hospital on historical data (in a batch system). They could apply these techniques to quickly deploy these detectors on live patient feeds. It also significantly impacts organizations with large investments in batch codes by providing a tool for rapid prototyping and significantly lowering the costs of experimenting in these new environments.

This study contributes to the body of knowledge necessary to address these problems by answering the following questions: Does a general mechanism exist for converting batch MapReduce algorithms to streaming algorithms? Can such a mechanism support a wide range of computations? When converted, what are the factors that impact their performance?

1.3 Scope and Significance

The scope of this work is to discover and analyze a general solution to the problem of converting batch MapReduce codes to support streaming data through incremental updates. To demonstrate the efficacy of these mechanisms we implemented Pig Squeal: an execution engine for running Pig scripts within Storm. Pig Squeal is then used to perform a deep analysis of the runtime performance of converted scripts. The aim of this analysis is to fully understand the execution of the current implementation of Pig Squeal on Storm. This analysis can be used in future work to improve the performance of Pig Squeal to make it a viable alternative for streaming development in enterprise environments. That is to say, the current implementation is meant to be correct and analyzable while optimzing the overall performance of the system is outside the scope of this dissertation.

To understand the significance of this work, it is necessary to discuss a focused subset of the related work. The notion of tracking differences between multiple versions of state has long been used in computer science [35]. The idea of delta computations have been used to address updates within batch executions of MapReduce tasks. In MapReduce with Deltas [44, 60] Lämmel and Saile introduce the notion of delta computations for propagating updates through MapReduce computations. They implement delta computations in a few case studies and propose the possibility of a general *delta layer* to aid in the development of delta-aware computations. The Incoop [12] system implements such a delta layer along with modifications of various components of the Hadoop architecture to track dependencies between inputs and outputs to allow for partial computation on the block boundary to speed up update processing. By careful analysis of the algebraic properties of MapReduce calculations and use of key-value stores (such as Hbase[3] which resembles BigTable [17]) we demonstrate that it is possible to track dependencies without explicit memoization. Both of these works address the response time of computations within the Hadoop framework while this work exploits delta computations to convert batch codes into streaming environments.

Delta computations have been used recently in iterative processing systems. The Naiad [52] system uses differential dataflow [49] to provide a general framework for implementing numerous data processing abstractions. Differential dataflow establishes a mechanism for decomposing a computation into a series of incremental

[3]http://hbase.apache.org

updates. This work focuses on the specific issues of automatic conversion of MapReduce calculations to incremental computations for use in streaming environments. The specific structures introduced in this work (the use of counting sets and extentions to the Pig API to re-enable combine functionality) fit within the general constructs of differential dataflow (as outlined in section 6.1.1). However, the window functionality and the use of external stores through the Trident mechanism address issues of continuous expansion of state and the ability to have an online/offline approach to state management. These represent optimizations specific to running MapReduce codes in an incremental fashion.

Recent applications of updates for the Pig scripting language include the NOVA [54] workflow engine to allow developers to explicitly manage delta records. The Squeal execution environment handles delta computations natively, freeing the developer from such considerations. There has also been work at Yahoo which also converts Pig scripts to Trident topologies. They use window operations to allow the developer to reason about changes instead of using delta calculations. Window operations have issues when data can be delayed or arrive out of order. The delta computation presented in this work provides a means for calculating the correct answer in spite of delayed or out of order arrival.

1.4 Outline

Chapter two provides in depth background for the three major systems used in this work by examining implementations of word count in each. The first system is MapReduce, the second is Pig, and the last is Storm. This chapter closes with a discussion of issues encountered when implementing chained aggregate computations in a streaming environment.

Chapter three provides an analysis of MapReduce as a message passing system and establishes how data flows through the system. This model is then used to describe what changes take place as information is added and removed from the original input set. This is followed by a discussion of a generic structure for managing value lists for the reduce stage which can be used to enable delta computations for any MapReduce algorithm. We then discuss the properties necessary for enabling Combine functions in a streaming fashion. Finally, we discuss a simple windowing mechanism to enable stateful operations (such as `JOIN`) to discard information over time.

Chapter four explores the features of Squeal by examining two case studies. This includes word count and the NationMind analytics. Each implementation is used to highlight various features and a performance analysis is discussed. This chapter concludes with a discussion of how intuition from batch systems does not necessarily carry over to streaming environments.

Chapter five discusses how Trident performs it's calculations and uses this knowledge to present a simple performance model. Each element of this model is then described using the example of word count. This is followed by a discussion of the results when this model is compared to various traces.

Chapter six provides a literature review covering related papers in the areas of Big Data Processing, Streaming, and Modeling. This is followed by chapter seven which provides the conclusion, a discussion of future work, and some closing thoughts.

Chapter 2

System Reviews

This chapter provides some background on the three systems that this dissertation builds upon: MapReduce, Pig, and Storm. Each is a representative of a whole class of systems – MapReduce is a batch data processing platform, Pig is a high level language for analyzing data (by producing MapReduce jobs), and Storm is a clustered streaming environment. The following sections will discuss these three systems in greater detail using the word count example from the previous chapter. This chapter provides the foundation necessary to understand the rest of the dissertation. If you are familiar with these systems, this chapter can be skimmed for a review.

2.1 MapReduce and Google File System

From the very beginning, Google has designed its architectures for horizontal scalability [13]. The cornerstone of this architecture is a distributed storage system called Google File System [27]. Designed as a means of providing reliable storage with high throughput from a cluster of commodity gear, it enabled the company to ride the wave of cheap hardware. Developing software for distributed systems is quite challenging on highly reliable hardware; when faced with the failure rate of commodity hardware, these challenges become a significant barrier to entry. To address these issues, Google introduced MapReduce [24] as a mechanism for distributing a work flow across a cluster of machines.

The beauty of MapReduce lies within its simplicity. By providing two functions *map* and *reduce* the developer is able to process petabytes of information on computer clusters that can fill a building. The MapReduce framework manages scheduling, job placement, and failure resolution. These functions require a specific form:

$map : (key, value) \mapsto \{(key_m, value_m), \cdots\}$ Given a key and value, emit a list of new key/value pairs.

$combine : (key, \{value_0, \cdots\}) \mapsto \{(key_c, value_c)\}$ Optional function that may be applied at the mapper before transmitting data across the network. Allows for pre-processing of reduce input to cut down on network traffic.

$reduce : (key, \{value_0, \cdots\}) \mapsto \{(key_r, value_r, \cdots)\}$ Given a key and a sorted list of values, produce a list of key/value pairs. The input values are those from the map stage that shared the same key.

2.1.1 Word Count in MapReduce

To understand how these functions are used in practice, consider our word counting algorithm implemented in MapReduce. Listing 2.1 provides a simple implementation of word count in Python; the code necessary to setup this program is omitted:

```python
# MR #1 -- word count
def WCMap(k, v):
    for word in v.split():
        emit(word, 1)

def WCReduce(k, v_lst):
    emit(k, sum(v_lst))

WCCombine = WCReduce

# MR #2 -- frequency analysis
def HistMap(k, v):
    emit(v, 1)

HistCombine = HistReduce = WCReduce
```

Listing 2.1: MapReduce word count example in Python.

The `WCMap` function receives a key/value pair from the system, in this case the key is the source file (which is ignored) and the value is a line of text from that file. This function splits the string on whitespace and emits the number one for each word (which can be seen in the tokenized column of table 1.1). The `WCReduce` function received a key representing a word emitted from map and a list of integers represented the occurrences of this word (as seen in the grouped column from table 1.1). This function merely emits the sum of the input list along with the word (as seen in the summed column of table 1.1). Finally, `WCCombine` is exactly equal to `WCReduce`. Consider an input set consisting of English text: there are words that occur with significant frequency such as the word "the". Any given input slice should have a considerable number of occurrences of these frequent words which will be "compressed" by the Combiner. Thus instead of receiving a series of ones ($\{1, 1, 1, \cdots\}$) the combiner would run at the various map stages to produce partial sums ($\{4, 8, 20, \cdots\}$) which requires less resources to transmit and shares the execution load of this calculation with the mappers.

Next, we address the second stage of our example to calculate a histogram of word frequencies. Another map function is provided to set the stage for execution pipelines. The purpose of this second calculation is to calculate the distribution of word frequencies. The `HistMap` function takes the key, in this case a word from the previous execution (which is ignored), and a value, representing the word frequency, and emits the frequency along with a one. The combine and reduce functions for the secondary calculation are the same as the previous example.

2.1.2 Hadoop Architecture

Apache Hadoop is an open source implementation of the ideas presented in the Google research papers [27, 23, 24]. The 1.0 implementation provides Hadoop Distributed File System (HDFS) [62] and an implementation of MapReduce. The 2.0 implementation improved the features of HDFS and provided a generic resource scheduler called YARN [71] that allows for other computation paradigmns such as Batch Synchronous Parallel (BSP) [2], interactive workloads (such as HIVE) [66, 67], or even continous execution mechanisms such as stream processing [4].

Figure 2.1 shows the components of the Hadoop architecture. For storage, the Name Node tracks the filesystem metadata: filenames, directory structures, and block membership. The Data Nodes (DN) store the data blocks that compose the contents of the files. The Job Tracker maintains a queue of executing and pending jobs; it coordinates with Task Trackers (TT) to execute jobs. The Name Node and

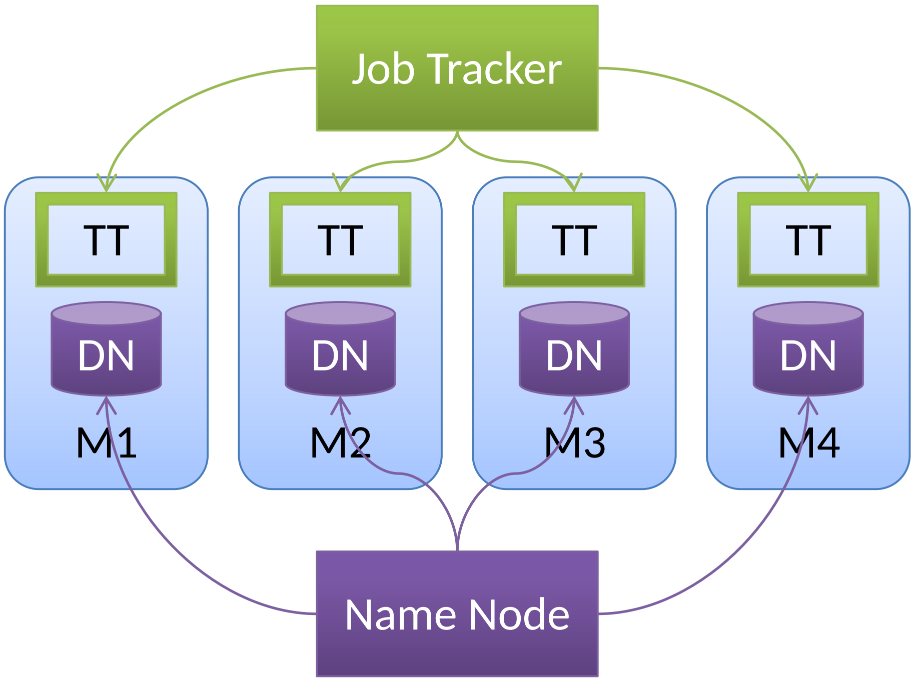


Figure 2.1: Hadoop architecture diagram

Job Trackers are the entry point for clients into the Hadoop system and provide cluster coordination. The individual machines (M1, M2, . . .) form the storage and execution resources of the Hadoop system.

Figure 2.2 shows a file stored within HDFS. In this specific case, “File 1” is made up of three sequential chunks; the chunk size is configurable, but generally ranges between 64 and 256 megabytes. When a file is inserted into the system, each chunk is sent to a data node for storage. These data nodes then begin replicating the chunks for resiliency and to increase parallelism when processing. The replication policy of Hadoop attempts to place one replica within the same rack as the initial copy while placing another replica in another rack. This balances performance (by limiting cross-rack communication) with resiliency (assuming that whole racks are susceptible to correlated failures due to power supply or switch failure).

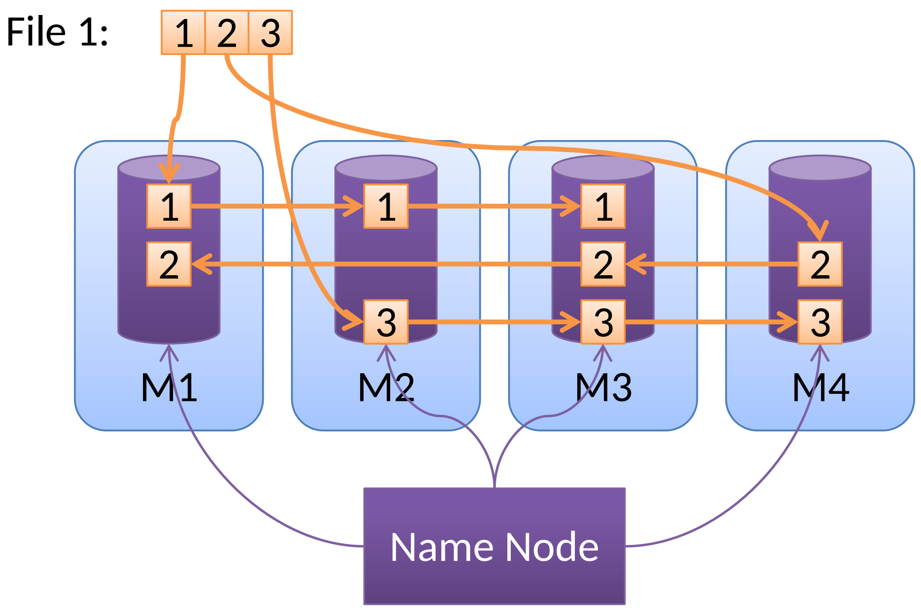


Figure 2.2: File placement in Hadoop Distributed File System

Figure 2.3 shows placement of a map phase on the example cluster. The Job Tracker attempts to assign the map task upon the same machine as the input data; absent the availability of a data-local task slot, it will then attempt to place the map task within the same rack, as a last resort it will use any available task slot. This optimizes the utilization of the intra-rack network capacity and allows for progress in less than optimal situations.

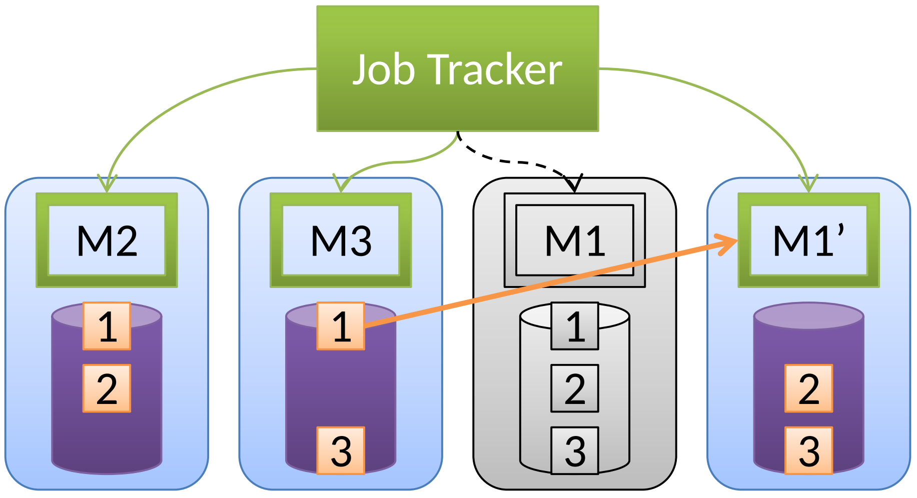


Figure 2.3: Job execution within Hadoop MapReduce

In this particular case the first execution of the map phase for the “1” input chunk had to be rescheduled to a secondary node (labeled $M1'$). This can occur because of node failure or in cases of “Preemptive Execution”. Preemptive execution is a means for compensating for slow execution times of any of the tasks (map or reduce). Within a cluster of computers, the chance of a misbehaving (i.e. slow) machine becomes quite likely. Because map and reduce tasks are supposed to be deterministic (or “acceptably non-deterministic” such as in Monte Carlo simulations) a duplicate task can be started to correct for misbehaving nodes. In this case, deterministic means that when a map or reduce function is given the same input it should provide output that is functionally equivalent. This is important if a task fails which would cause the job tracker to spawn a task to replace the missing results. Repeatability becomes important later when we start tracking changes through various stages. Preemptive execution can provide a significant performance boost as noted in [23] when you consider that the reduce phase cannot start execution until all mappers have completed.

The number of map tasks is determined by the number of chunks in the input set. If there are more tasks than task execution slots, the non-assigned tasks are queued up for execution. This provides an important distinction from typical High Performance Computing (HPC; particularly MPI) workloads. HPC computations generally require scheduling of all necessary resources for a particular step at the same time. This leads to executions that are rectangular in space and time [71]. Scheduling in this case becomes an exercise in bin-packing whereas some tasks are tall and skinny (requiring large compute resources for a short amount of time) and

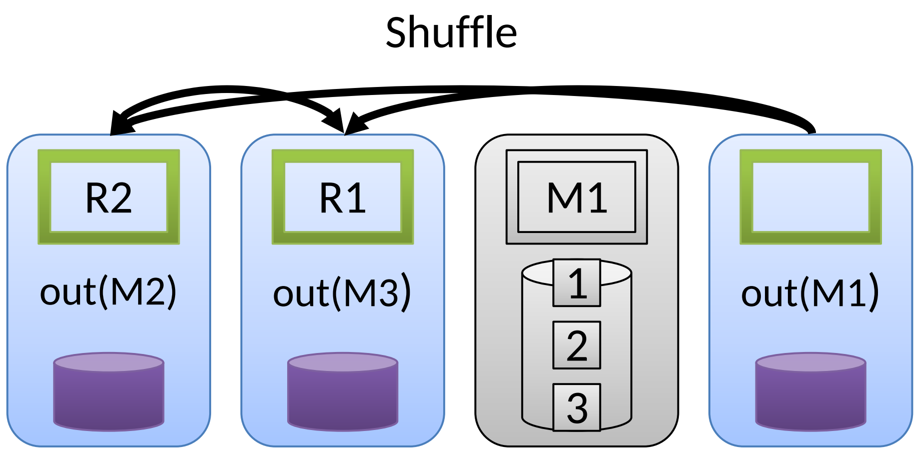


Figure 2.4: The shuffle phase of a MapReduce job

some are short and wide (requiring fewer compute resources for a long time). In multi-tenant environments systems, this necessitates prioritization and check pointing (to maintain progress in the face of preemption). Within MapReduce clusters, tasks of multiple jobs can be interleaved [71] (when using a round-robin or “fair” schedulers) with the atomic boundary of execution being the sub tasks (map or reduce). This allows for job executions that expand and contract in time and space depending on cluster utilization. Both execution paradigms have their strengths and weaknesses.

Figure 2.4 shows the shuffle stage of the MapReduce execution where intermediate results are transmitted to the reduce tasks. The map output is temporarily stored on the node that performed the map computation; if one of these nodes dies before the results can be transfered, the map task will need to be rescheduled for execution. Although the reduce tasks cannot start processing the intermediate results before all map tasks are completed, they can start to transfer these results and begin sorting them. As with the map phase, any tasks that cannot acquire a task

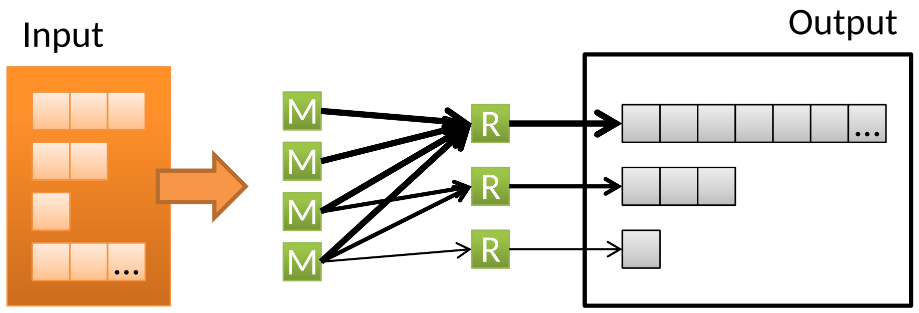


Figure 2.5: Data flow-based view of a MapReduce execution

slot immediately are queued for later execution. Upon completion, the reduce task will write its output into HDFS for follow-on processing.

Whereas the number of map tasks are determined by the size of the input set, the number of reduce tasks is specified by the developer or framework. For example, the Pig scripting language uses the following heuristic to specify the number of reduce tasks. A common mistake of Hadoop neophytes is to set the number of reducers to a significantly large number (perhaps thinking "more is better?"). However, this can cause the total runtime of the job to increase due to the latency between task assignment and execution. In a multi-tenant environment once a job has been given a task slot, it must go to the back of the wait queue. On the flip side, if a reduce task has too much data and takes a long time to execute the scheduler loses the ability to improve total cluster throughput because it cannot interleave task executions to allow short jobs to complete in a timely manner. If preemption is used to avoid such situations it can waste execution time on long tasks and lead to situations where some jobs will never complete.

Figure 2.5 provides a data flow-based logical view of a MapReduce computations. Often, this simplified conceptual view is all that is necessary to begin writing

MapReduce programs. Data flows from the left to the right – in the simplest case, data is loaded from HDFS, processed by each individual map task, shuffled to the appropriate reduce tasks, and finally processed and written back into HDFS. This figure also illustrates issues surrounding the cardinality of the reduce input groups. The top reduce task has more data being transmitted to it, the middle task less, and the bottom task less still. Ahmdal's law tells us that the execution time of a multi-process algorithm is limited by the execution time of the atomic (non-splittable) parts of the execution. Even with infinite task slots and negligible latencies for startup and data transfer the execution times of any MapReduce job is dominated by the slowest individual task.

Finally, this figure also shows that reduce tasks can have a non-uniform distribution in the sizes of the output they generate. In this particular case there is a correlation of input and output sizes, but that doesn't have to be the case (the reduce task with the smallest input could produce the most output). This doesn't create any particular issues with follow-on processing because the system will start more map tasks for any particular output file based on the chunk boundaries. However, this can be an important issue to consider later when we consider streaming computations where operations may be pipelined.

2.2 Apache Pig

Apache Pig [55] is a high-level imperative data-flow language for analyzing data using Hadoop's implementation of MapReduce. By abstracting the data operations further than Java MapReduce, Pig allows for rapid development of algorithms using significantly less code. Pig provides a set of operations for loading, organizing, manipulating, and storing information. Within each of these steps, Pig provides built-in functions for handling typical uses and provides mechanisms for extension. These built-in functions are written in a way that allows for the full use of the optimization mechanisms of MapReduce (specifically combiners) providing for efficient code with very little effort. Ultimately, this provides a softer learning curve for neophytes as well as a strong tool for more seasoned developers. The following section uses our word count example to focus the discussion of language features and execution. Subsequent chapters will provide details necessary to proceed.

2.2.1 Word Count in Pig

Listing 2.2 provides a Pig implementation of the word count example seen in section 2.1.1. Though longer than the Python code from listing 2.1, this code is complete and is ready for execution. First, an alias, *sents*, is created by loading data from the path specified in \$*input* (a parameter that must be specified at runtime). By default, Pig assumes that data is tab separated and newline terminated. The data can be compressed and processed automatically provided the filenames end with the appropriate extension (`gz` or `bz`). The data is returned as a list of tuples with a single field *sentence*. The next line creates a bag of words by splitting each *sentence* within *sents* on white space (using `TOKENIZE`) and generates a new alias *words* by flattening the bag which creates one row for each element within the bag of words. The next line creates a new alias *lowerWords* by applying the `LOWER` function to the first value in *words*, resulting in rows of lowercase words (equivalent to the tokenized column of table 1.1 – with an implicit one for each row). Line 10 groups all rows of *lowerWords* by the value in the field *word*. This results in a new alias *countGr* that has an automatically populated field *group* that represents the value of the group expression that partitions the values of *lowerWords* (similar to the grouped column of table 1.1). The second field of *countGr* is named after the grouped bag, *lowerWords* and contains all rows within the partition.

Next the alias *count* is created by iterating through each partition and counting the number of rows within the *lowerWords* bag (which is equal to the summed column in table 1.1). In a similar fashion, *histGr* is created by grouping the rows in *count* by the *wordCount* field; *hist* is created by counting the number of rows that share the same *wordCount*. Finally, these values are stored to disk in the specified output directory.

```
-- Read $input into a bag of sentences.
sents = LOAD '$input' AS (sentence:chararray);
-- Create a bag of words.
words = FOREACH sents GENERATE FLATTEN(TOKENIZE(sentence));
-- Convert the words to lowercase.
lowerWords = FOREACH words GENERATE LOWER($0) AS word;
-- Group the words together.
countGr = GROUP lowerWords BY word;
-- Count the number of times each word appears.
count = FOREACH countGr GENERATE
    group, COUNT(lowerWords) AS wordCount;
-- Group by word count.
histGr = GROUP count BY wordCount;
-- Calculate the number of times each word count occurs.
hist = FOREACH histGr GENERATE group, COUNT(count) AS freq;
STORE hist INTO '$output';
```

Listing 2.2: Word count example in Pig that calculates the histogram of word frequencies.

This script results in a two stage MapReduce execution much as the Python code. The first MapReduce calculates the *count* alias while the second calculates *hist*. The Pig framework will materialize the results of *count* temporarily within HDFS and use it as input for the second stage of execution. If the script stopped by storing *lowerWords* Pig would not use a reduce phase. Reduce stages are only necessary when interaction between rows is necessary. Thus, any operations that requires data to interact in some way (i.e. `GROUP`, `ORDER BY`, or `JOIN`) will induce a MapReduce operation within the execution plan. The next section will provide an overview of the language.

2.2.2 Language Overview

A full description of every aspect of Pig is well beyond the scope of this work, but it is necessary to explain a few key aspects as they relate to understanding this work. This section will discuss the data model for Pig along with how data is loaded, manipulated, and stored. The Pig project[1] has high quality documentation that provides examples of each of these operations in much greater detail.

Pig supports many data types[2] which can be considered scalar or complex. A subset of the scalar data types include:

`int` a signed 32-bit integer.

`float` 32-bit floating point number.

`chararray` a character array in UTF-8.

`bytearray` a raw byte array.

There are three complex (or nested) types in Pig:

`tuple` an ordered set of fields that can be referenced by position or an optional name.

`bag` a collection of tuples (think of it as a non-distinct set).

`map` a set of key/value pairs.

As demonstrated by listing 2.2 data is loaded into the system using the appropriately named `LOAD` operator. By default, Pig uses the `PigLoader` User Defined Function (UDF) to load information. The `PigLoader` function is a Pig built-in and implements the `LoadFunc` interface to interact with HDFS. The load function also

[1]http://pig.apache.org/

[2]http://pig.apache.org/docs/r0.10.1/basic.html under Data Types

translates the information into the Pig data model. By default, the `PigLoader` will treat each line (terminated by a newline) as a record with fields separated by tabs. The separator can be overridden by specifying an alternative – `PigLoader(',')` would look for comma delimited fields. Without specifying a schema, Pig will store everything as *bytearray* unless various operations require an implicit cast (such as string comparison, which would cause a cast to a *chararray*). The developer can specify the schema on the load line or throughout the script using C-style casting: `(chararray)` *expression*. The results of the load function are represented by an alias. This alias can be thought of as a representation of a bag of tuples.

Once data is loaded, it can be manipulated using various operations such as `FOREACH` or `FILTER`. The `FOREACH` allows for an expression to be evaluated against each tuple in the specified input. These expressions can be simple such as relational projection or complicated such as the application of sophisticated UDFs. Pig comes with quite a few built in functions including mathematical functions (`ABS`, `LOG`. . .), string manipulation (`INDEXOF`, `SUBSTRING`, . . .), and complex type routines (`TOTUPLE`, `TOBAG`, `TOP`. . .).

When parsing a script, Pig doesn't perform operations until a side-effect occurs either through a `DUMP` operation or a `STORE` operation. The `DUMP` operator executes the portions of the Pig script necessary to materialize the results to the screen. The `STORE` operator performs the same execution but passes the information to a storage UDF. The default `StoreFunc` is `PigLoader` which writes with the same default format as it reads. As with reading, an alternative separation character can be passed in as an argument. By default, Pig creates a directed acyclic graph (DAG) where the `LOAD` routines represent sources and the `STORE` routine represents a sink. If there are multiple `STORE` routines in a script, each is treated as an independent DAG.

All of the operations mentioned so far could operate without a reduce step – all data is processed independently of each other (there is no interaction between tuples). The real strength of MapReduce (and thus Pig) lies in grouping data together by some partitioning function and applying aggregate functions. Pig includes standard `JOIN` operations (left, right, inner, and outer) and an `ORDER BY` operation to perform a total ordering of the data set based on some expression. The operator that makes Pig unique (when compared to SQL) is `GROUP BY` (or `COGROUP BY`) which partitions one or more aliases by the specified expression(s). The word count example seen in listing 2.2 has already demonstrated grouping a single alias by an expression. Allowing developers to group or join multiple sources of information together allows Pig to mix many different types of information together to answer an analytical question.

When data is grouped together, many aggregation functions become available to the user such as `MIN`, `MAX`, or `COUNT`. These built-in functions implement the `Algebraic` interface from Pig which requires three functions: `getInitial`, `getIntermed`, and `getFinal` that return names of classes that implement `EvalFunc` and perform each step of a combinable computation. The *initial* class is passed a tuple containing the information as produced by the map function – it preps the data for no or more executions of the *intermediate* evaluation function. The intermediate function accepts bags of initial or intermediate values and returns values that can be processed by further intermediate stages or by the *final* function. The final function takes one or more of the outputs from initial or intermediate and produces the final result.

Listing 2.3 provides a simplified implementation of these constructs in Python for `COUNT`. First, notice the implementation of an `exec` method (lines 14 to 16). This is necessary if Pig encounters cases where the combine functionality cannot be used (such as when combinable and non-combinable operations are used after a `GROUP BY` operation – Pig falls back to standard MapReduce). The `initial` method (line 3-4) takes a tuple containing a bag of values and returns the size. The `intermediate` and `final` methods sum these values producing the count of the elements within. Ultimately, this allows Pig to produce very efficient execution plans from very simple user-provided scripts.

```
 1  class COUNT:
 2      @staticmethod
 3      def initial(tuple_in):
 4          return len(tuple_in.get(0))
 5      @staticmethod
 6      def intermediate(tuple_in):
 7          # tuple_in contains a bag of numbers
 8          return sum(tuple_in.get(0))
 9      @staticmethod
10      def final(tuple_in):
11          # Same as intermediate.
12          return self.intermediate(tuple_in)
13      @staticmethod
14      def exec(tuple_in):
15          # Fall back method.
16          return len(tuple_in.get(0))
```

Listing 2.3: COUNT implementation that supports combine

Pig ships with a core set of features for very basic data manipulation. However, members of the community such as Twitter[3] and Linkedin[4] have provided additional functionality through third party libraries. These libraries feature advanced functionality for built-in types such as treating Bags as Sets or `LOAD`/`STORE` routines for manipulating advanced storage schemas or mechanisms. This allows institutions to tailor Pig to their specific needs by implementing functions to access proprietary data formats or algorithms.

2.2.3 Conversion Steps

When the Pig interpreter encounters an operator that produces a side effect (such as `STORE` or `DUMP`) it begins the process of query conversion. There are three distinct instances of an execution plan: *logical*, *physical*, and *MapReduce*. The logical plan is basically a parse tree of all the operators that lead to the result. The logical plan undergoes a series of optimizations – pushing operators up or down, merging `FILTER` statements, and other adjustments to improve the performance of the execution plan. Finally, the logical plan is converted into a physical plan which is a configuration of operators that implements the logical plan.

The next phase is execution which leads to the generation of the MapReduce plan. The MapReduce plan spreads the physical plan across multiple stages of MapReduce executions as necessary to produce the results. Once converted to a MapReduce plan, further optimization occurs such as using combiners if possible. Finally, the jobs of the plan are dispatched (in parallel where possible) to Hadoop

[3]https://github.com/twitter/elephant-bird
[4]http://datafu.incubator.apache.org/

for execution. The map and reduce tasks for Pig are parameterized to instantiate a section of the physical plan to perform the calculations of the Pig script. The input is unmarshalled and placed at the top of the sub-tree and the results are drained from the bottom, marshalled, and passed to the MapReduce framework.

2.3 Apache Storm

Storm [4] is part of a new wave of stream processing frameworks along with IBM InfoSphere Streams [9, 26, 32] and Yahoo's S4 [53]. These platforms run on local clusters of computers and are geared for producing results at low latencies. Nathan Mars (one of Storm's initial creators) advocates what he calls "the lambda architecture"[5]. The purpose of streaming in this architecture is to bridge the hour long gap before the batch processing system is run. The results of the batch system are treated as the source of record so any hiccups that may occur in the streaming system only have a short impact. Storm was selected due to ease of deployment, access to source code, and a feature set that eased the development (notably the transaction handling and state management). The techniques presented in this work are applicable to other streaming frameworks and for other incremental update mechanisms such as Google's Percolator [58]. The following sections describe the Storm architecture, discuss a streaming word count implementation using the Trident APIs, and discuss a fundamental issue with updates that is the core of this research.

[5]http://lambda-architecture.net/

2.3.1 Storm Architecture

Figure 2.6 shows the architecture of the Storm processing system. The main components are Nimbus and Supervisors (labeled SV). Nimbus is similar to the Hadoop Job Tracker in that it maintains a list of running topologies and manages failures. Supervisors are similar to the Hadoop Task Trackers in that they host the processing components and manage communications between each. These components rendezvous using a system called Zookeeper [40] (labeled ZK) which is designed to mimic Google's Chubby [14] locking service.

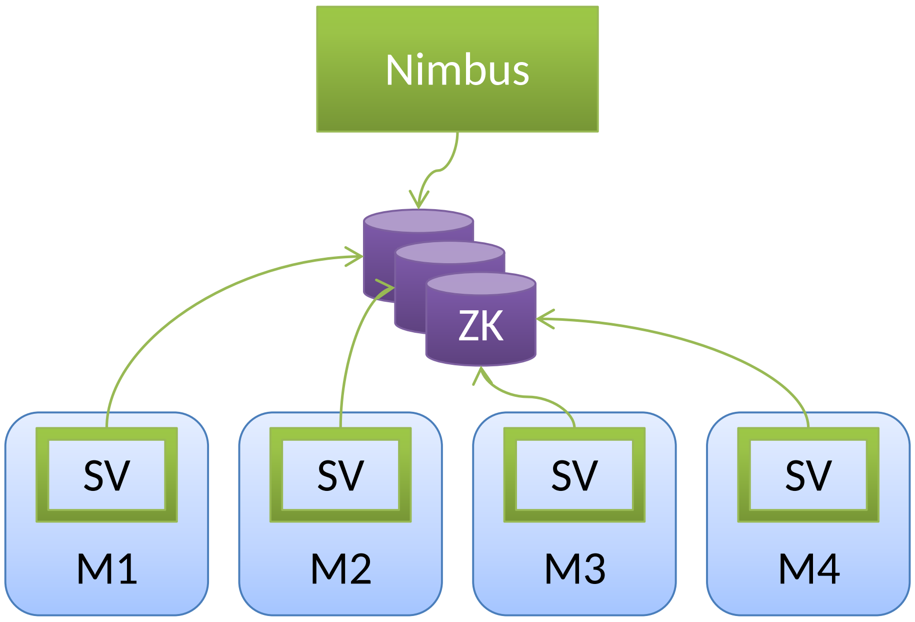


Figure 2.6: Storm architecture diagram

Figure 2.7 shows a hypothetical Storm topology. Storm applications consist of *topologies* of *spouts* (labeled S) and *bolts* (labeled B). When launched, Nimbus examines the topology and assigns the components to worker slots (labeled W) on the Supervisor nodes. A worker slot represents a JVM containing a communication channel and thread pools for servicing requests. Spouts represent sources of information, they provide a `getNext` routine which is called by the worker to pull a tuple into the system. As with Pig, Storm's tuples are ordered, named, relations of data. The data within can be of any type supported by Java provided it is serializable or has a serialization mechanism registered with Storm. Tuples are carried on *streams* which are unbounded sequences of tuples that originate from spouts and bolts. Spouts are declared within a topology by providing an instance of the spout to the topology builder along with a name and a parallelism hint (which determines how many instances to create when the topology is instantiated). Spouts also implement the `declareOutputFields` routine which specifies the output streams and field names produced within each stream.

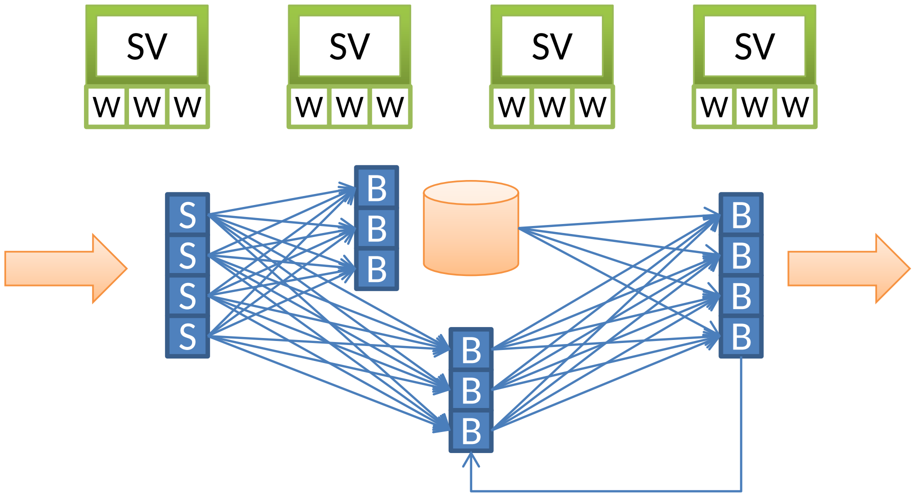


Figure 2.7: Storm sample topology

Much like spouts, bolts are declared using the topology builder by providing an instance, name, and parallelism hint. Bolts also implement a `declareOutputFields` routine like spouts. However, unlike the spout, the bolt can subscribe to other streams of information by performing a *grouping* on a named source (spout or bolt) and stream. A subset of the grouping mechanisms provided by Storm includes:

all grouping – send each tuple to every instance of the receiver.

shuffle grouping – send each tuple to one randomly chosen instance of the receiver.

fields grouping – send each tuple to one instance of the receiver based on the value of the specified fields (1-to-1).

custom grouping – send each tuple to destinations based on the specified grouping function.

In MapReduce, the default partitioning style is equivalent to Storm's fields grouping (where the field would be the key). Also like MapReduce, Storm provides the ability to specify a custom grouping function. In MapReduce such functions partition the data into separate bins whereas Storm permits more general specification allowing a tuple to be routed to multiple nodes if necessary (such as the system provided all grouping which broadcasts tuples to every receiver).

While tuples are pulled from spouts, they are pushed to bolts via the *execute* routine. Tuples provide routines to determine their source along with various access routines for the data within. Bolts ultimately produce new tuples by emitting them via the *collector* provided when the bolt is *prepare*-ed during the topology setup. These emissions don't have to be directly tied to an execute. There are cases where a bolt may delay emission such as time or size based windowing. However, if Storm's at least once semantics are used, any new messages must be transmitted before any of the influencing messages are acknowledged.

Storm has a unique acknowledgement mechanism worth mentioning[6]. When

[6]http://storm.incubator.apache.org/documentation/Guaranteeing-message-processing.html

a tuple is emitted from a spout or bolt, it is given a random 64-bit id. For each spout tuple, a special bolt called the “acker” keeps a map of id to a pair of numbers representing the source spout (for transmitting acknowledge or fail messages) and an “ack value”. The ack value is initialized to zero. When a message is received at a bolt, it can be used to chain any emissions to the original source message. For each chained message, a new random id is created and this value is XORed to the id of the inbound message. When the inbound message is acknowledged, this combination of ids is sent to the acker which XORs this combined value to the ack value. If the ack value ever becomes zero, we know (with a high degree of certainty) that the changes introduced by the source message have been successfully processed.

If a timeout occurs for the message (due to overload or intermediate bolt failure), the acker transmits a failure response to the spout (which may have died). In any case, the message would be retransmitted at a later time – this may apply the information more than once if the message (or one of its ancestors) was successfully consumed. There is also a remote possibility that the ack value can become zero out of chance. This would only be an issue if there was a failure along with this rare event.

These are the building blocks for anything running in Storm. While it is possible to write highly efficient (and complicated – and unmaintainable!) code using raw spouts and bolts, there are other mechanisms for achieving these results. The following subsection describes the Trident APIs which provide a higher level abstraction for building analytics on Storm.

2.3.2 Trident, Transactions, and Streaming Word Count

Trident is included with the Storm distribution and provides a different way to compose topologies. Listing 2.4 shows an implementation of word count using Trident. For this example, we will be processing a stream of tweets that are loaded in line 7. Lines 10-13 calls the `TriTokenize` function on the *twitterJSONb* field from the tweets stream. This function's output will be appended to the tuple with the field name "word". When there are multiple emissions for a given input, Trident duplicates the rest of the tuple and appends the new values to each. Thus, the string "the quick brown" will produce three tuples – each with a copy of the input along with the unique tokens as seen in table 2.1 (which is an expanded for of the tokenized column of table 1.1).

Table 2.1: Example of tuple expansion within Trident.

twitterJSONb	word
the quick brown	the
the quick brown	quick
the quick brown	brown

```
 1 // Java boilerplate omitted...
 2
 3 public class WordCount {
 4     public static TridentTopology setupTopology(String queueName) {
 5         TridentTopology topology = new TridentTopology();
 6
 7         Stream tweets = topology.newStream(
 8             "tweets", new TwitterRMQSpout(queueName));
 9
10         Stream tokens = tweets.each(
11                 new Fields("twitterJSONb"),
12                 new TriTokenize(),
13                 new Fields("word"));
14
15         Stream counted = tokens.groupBy(new Fields("word"))
16                 .persistentAggregate(
17                         new LRUMemoryMapState.Factory(1000),
18                         new Count(),
19                         new Fields("word_count"))
20                     .newValuesStream();
21
22         Stream hist = counted.groupBy(new Fields("word_count"))
23                 .persistentAggregate(
24                         new LRUMemoryMapState.Factory(1000),
25                         new Count(),
26                         new Fields("freqency"))
27                     .newValuesStream();
28
29         return topology;
30     }
31 }
```

Listing 2.4: Trident implementation of word count histogram

Line 15 performs a global partitioning of the tuples by the field “word”. This will send all tuples with the same “word” to the same worker (similar to the grouped column of table 1.1). Lines 16-19 perform a persistent aggregation – effectively counting the occurrences of the specific word. Trident provides a state tracking mechanism which can be seen in this segment of code. For this example, we are using the LRU Memory Map State with 1,000 memory slots. When a new key comes in, Trident attempts to fetch the previous value (for the key) from the state. If no previous state exists, the aggregation functions *zero* method is used. Updated as applied in bulk to this value and staged for commit.

Trident supports many types of aggregation including basic reduction and combination. Count, as we've seen previously, is combinable and is implemented as such. Once the count has been produced, a new value stream is created (which is equal to the summed column of table 1.1). Lines 22-27 perform the second grouping on the count field to produce the histogram. If the tuple successfully reaches the end of the calculation, Trident performs a commit phase to push all updates in bulk to the state system. Upon failure, Trident is able to discard the updates and mark the input tuples for re-processing. Because of this two-phase commit mechanism, Trident can provide at most once execution semantics (there are additional constraints on the spout and state implementations but this is beyond the scope of this work).

2.3.3 Trouble in (Streaming) Paradise

The issue with the code in listing 2.4 is that there is no coordination between the count and histogram phases. Reconsider figures 1.1a and 1.1b – when the second batch of words enter into the system, the intermediate values undergo changes which must be propagated to the second stage. In our current implementation of word count within Trident, only the "positive" information is transmitted. Thus, when the word "the" arrives in the second batch, it will be reported with a count of two. This will then be passed to the histogram section of code with will report that there is one occurrence of the frequency two. After the second batch of words is seen, the histogram results will be `(1, 7)` (not (1, 6)!!!) and `(2, 1)`. This leads us to the core contribution of this dissertation: propagating changes through aggregation pipelines.

Chapter 3

Streaming Aggregates

This chapter will focus on describing a generalized solution to the issue of change propagation and explain how it can be applied to running Pig scripts in a streaming environment. First, we will describe MapReduce as a message passing system and describe how information flows through the system. We will then examine what happens when information is added to the input set and track the influence of this information through our flow description followed by a similar analysis of the influence of removing information. This will be followed by a discussion of implementing MapReduce with deltas using a counting set to maintain value lists to be used by opaque reduce calculations. This will lead into a section that generalizes these techniques to combinable calculations. We will conclude with a brief discussion of a simple window algorithm for managing unbounded accumulation in intermediate results.

3.1 MapReduce as Message Sequences

A general solution emerges from close examination of the way information propagates through a MapReduce calculation. The input to a map phase is usually produced from a chunk of a file in HDFS. While each chunk represents an ordered sequence of key/value pairs, this discussion will consider the more general case of unordered key/value pairs. Knowledge of the underlying order can lead to optimizations such as map-side merge joins but such optimizations have little bearing on infinite streams of tuples from a streaming context. Let us define I, the input to a MapReduce job, as an unordered input sequence:

$$I \subseteq \mathbb{K}_I \times \mathbb{V}_I$$

Because there can be duplicate key/value pairs, I isn't a more restrictive set. In our word count example let us assume the input keys are unimportant (they are generally the name of the source file) and that the values are strings of words. So, our initial input using the example depicted in 1.1 would be: $\{$`(∅, ''the quick brown fox'')`$\}$.

Let us recall that map is a function that maps from key/values into a sequence (or list) of key values. Here, we define $M^i_{key}(k, v)$ to be the *ith* key in the output sequence for the input (k, v). We use p to denote the total number of key/value pairs in any distinct execution of $Map(k, v)$. Likewise, $M^i_{val}(k, v)$ is the ith value in the output sequence:

$$Map(k, v) = \left\{(M^0_{key}(k, v), M^0_{val}(k, v), \cdots, (M^p_{key}(k, v), M^p_{val}(k, v)\right\}$$

If we let $\bigcup$ represent sequence concatenation, then the map phase transforms the input sequence, I, into a new sequence, M:

$$M = \bigcup_{(k,v)\in I} Map(k,v)$$

$$M = \bigcup_{(k,v)\in I} \bigcup_{i=0}^{p} (M^i_{key}(k,v), M^i_{val}(k,v))$$

For our example, M_k would be the tokens of the values from I and M_v will be a sequence of ones (the same as the tokenized column of table 1.1). The shuffle phase partitions M using the key field into value lists vl_{key}. This is equivalent to the grouped column of table 1.1. The reduce phase operates on these partitions to produce the final output sequence, R:

$$vl_{key} = \{val | \forall (key, val) \in M\}$$

$$R = \bigcup_{key \in M_{key}} Reduce(key, vl_{key})$$

Where vl_{key} represents the sorted sequence of values from M_{val} with the same key (such as $(the, \{1,1,1,\cdots\})$). Much like Map, $Reduce$ produces a list of key/value pairs for each input. Let $R^j_{key}(key, vl_{key})$ and $R^j_{val}(key, vl_{key}$ be the jth key and value in the sequence produced by the $Reduce$ function. We use q to represent the length of the sequence produced by the reduce function for any distinct key/value list pair (like p it changes with the input). Then, the output sequence from the reduce phase is:

$$R = \bigcup_{key \in M_{key}} \bigcup_{j=0}^{q} (R_{key}^{j}(key, vl_{key}), R_{val}^{j}(key, vl_{key}))$$

The first stage of our example has a single output value $(1, 4)$. This establishes a description of basic MapReduce as a series of message sequences. Before we proceed, we will show that combiners are a simple extension of this mechanism. We will borrow the Pig constructs of $Init$, $Intermed$, and $Final$ to explain the changes. These functions are defined as:

$$Init : M_{val} \mapsto C_{val}$$

$$Intermed : \{c | c \in C_{val}\} \mapsto C_{val}$$

$$Final : \{c | c \in C_{val}\} \mapsto R_{val}$$

The $Init$ function is used directly after the map function to convert the values of M_{val} to a new structure C_{val} which is accepted by $Intermed$ and $Final$ ($Init$ can be the identify function where $M_{val} = C_{val}$). The $Intermed$ function can be called during the map phase after the values have been partitioned by key and accepts lists of values from the sequence C_{val}. The $Final$ function accepts lists of values from the sequence C_{val} and produces a value in R_{val}. The final value can then be used along with the key to produce the sequence R.

The following list provides values for the named elements after the first set of words have been seen in our example (corresponding to figure 1.1a – without combiners):

$$M_{stage1} = \{(the, 1), (quick, 1), (brown, 1), (fox, 1)\}$$

$$vl_{the} = \{(the, 1)\}$$

$$R_{stage1} = \{(the, 1), (quick, 1), (brown, 1), (fox, 1)\}$$

$$M_{stage2} = \{(1, 1), (1, 1), (1, 1), (1, 1)\}$$

$$vl_1 = \{(1, 1), (1, 1), (1, 1), (1, 1)\}$$

$$R_{stage2} = \{(1, 4)\}$$

The value lists for words other than "the" have been omitted because they do not change for the remainder of this exercise.

3.1.1 Adding Information

Now, assume that we have a new key/value pair to add to our existing input series:

$$I^+ = I \cup (k^+, v^+)$$

In our example, this corresponds to the addition of the next four words "jumps over the lazy" to the input set. Our goal is to analyze the contribution of this new data to the new result sequence, R^+:

$$R^+ = R + \Delta_+ - \Delta_- \tag{3.1}$$

Where Δ_+ are new key/value pairs seen because of the new data and Δ_- are key/value pairs from R that are not present in R^+. If Map is a function (in that it returns the same sequence every time the same input is provided), then I^+ would produce M^+ with separable values for the data from I:

$$M^+ = \bigcup_{(key,val)\in I^+} Map(key, val) = Map(k^+, v^+) \cup \bigcup_{(key,val)\in I} Map(key, val)$$

This holds because the map functions are applied independently of one another. If Map was non-functional the input from I could produce different values and would not be separable. Because of the shuffle stage, the results for R^+ are a bit more complicated:

$$vl_{key} = \big\{val | (key, val) \in M^+\big\}$$

$$R^+ = \bigcup_{key\in M^+} Reduce(key, vl_{key})$$

To simplify the following equation, let kv^+ be equal to the new pair (k^+, v^+). The series M^+_{key} is composed of three distinct partitions: [1]

[1] For two sets, A and B, the set subtraction operator has the following value: $A \setminus B = \{x | x \in A \wedge x \notin B\}$.

$$M^{+}_{key} = M^{I \setminus kv^{+}}_{key} \cup M^{kv^{+} \cap I}_{key} \cup M^{kv^{+} \setminus I}_{key}$$

The partition, $M^{I \setminus kv^{+}}_{k}$ represents the keys produced by the original input I but not by $Map(k^{+}, v^{+})$. These keys produce identical value lists as the first computation and thus produce the same values as R; we will label these results $R^{I \setminus kv^{+}}_{\varnothing}$ because they don't change their influence on subsequent computations. The keys $M^{kv^{+} \setminus I}_{key}$ represent keys seen only in $Map(k^{+}, v^{+})$; they will produce an independent portion of the new result sequence which we will label $R^{kv^{+} \setminus I}$.

The keys $M^{kv^{+} \cap I}_{key}$ represent keys seen both in the original key space and in the new key space produced by $Map(k^{+}, v^{+})$. For keys in this set, there will be updated value lists, vl^{+}, for *Reduce* to process. In our example:

$$M^{I \setminus kv^{+}}_{key} = \{quick, brown, fox\}$$

$$M^{kv^{+} \cap I}_{key} = \{the\}$$

$$M^{kv^{+} \setminus I}_{key} = \{jumps, over, lazy\}$$

Generically tracking the influence of new values through a *Reduce* computation would be challenging. However, if the original value lists are available we can generically analyze the resulting sequences to produce three distinct partitions:

$$R_{+}^{kv^{+}\cap I} = \bigcup_{key \in M_{key}^{kv^{+}\cap I}} [Reduce(key, vl_{key}^{+}) \setminus Reduce(key, vl_{key})]$$

$$R_{\varnothing}^{kv^{+}\cap I} = \bigcup_{key \in M_{key}^{kv^{+}\cap I}} [Reduce(key, vl_{key}^{+}) \cap Reduce(key, vl_{key})]$$

$$R_{-}^{kv^{+}\cap I} = \bigcup_{key \in M_{key}^{kv^{+}\cap I}} [Reduce(key, vl_{key}) \setminus Reduce(key, vl_{key}^{+})]$$

The sequence $R_{+}^{kv^{+}\cap I}$ consists of new values produced because of the change in input; the sequence $R_{\varnothing}^{kv^{+}\cap I}$ consists of values seen in both executions; finally the sequence $R_{-}^{kv^{+}\cap I}$ represents values that are no longer produced by the updated input. It is worthwhile pointing out the specifics surrounding the key "the" for the first MapReduce stage for our example:

$$vl_{the} = \{(the, 1)\}$$

$$vl_{the}^{+} = \{(the, 1), (the, 1)\}$$

$$Reduce(the, vl_{the}) = \{(the, 1)\}$$

$$Reduce(the, vl_{the}^{+}) = \{(the, 2)\}$$

These sequences can be combined to form the delta sets:

$$\Delta_{+} = R^{kv^{+}\setminus I} + R_{+}^{kv^{+}\cap I}$$

$$\Delta_- = R_-^{kv^+ \cap I}$$

Values from the $R_\varnothing$ sequences are already accounted for by previous emissions in R and do not need to be retransmitted. For our example, the values for each are:

$$R_{stage1}^{kv^+ \setminus I} = \{(jumps, 1), (over, 1), (lazy, 1)\}$$

$$R_{stage1+}^{kv^+ \cap I} = \{(the, 2)\}$$

$$R_{stage1-}^{kv^+ \cap I} = \{(the, 1)\}$$

The values of Δ_+ can be transmitted to follow-on MapReduce stages and treated by added information. It is worth noting that adding information has no bearing on the operation in MapReduce – information is always added to a combine function through the *Intermediate* or *Final* stages. The function of reduce acts in the same fashion and can be treated with the same message arithmetic mechanism presented in this section. The following section will address issues surrounding information subtraction (which is necessary before we consider the output values for the second MapReduce stage).

3.1.2 Subtracting Information

When data needs to be deleted from an input set, we can consider a single key value being removed:

$$I^- = I \setminus (k^-, v^-)$$

For this section, we will consider the single tuple $(the, 1)$ slated for removal from the first MapReduce stage. As with adding information, if Map is functional, then we can consider the contribution to the output of the map phase separately:

$$M^- = \bigcup_{(key,val) \in I^-} Map(key, val) = \bigcup_{(key,val) \in I} Map(key, val) \setminus Map(k^-, v^-)$$

As before, let us define a value to represent the subtracted key/value pair: $kv^- = (k^-, v^-)$. This leads to three distinct partitions for the series M^-_{key}:

$$M^-_{key} = M^{I \setminus kv^-}_{key} \cup M^{kv^- \cap I}_{key} \cup M^{kv^- \setminus I}_{key}$$

The keys in $M^{I \setminus kv^-}_{k}$ will produce the same partitions and value lists as in the original case and thus can be ignored. Likewise, keys in $M^{kv^- \setminus I}_{key}$ will produce partitions and executions of reduce that we will call $R^{kv^- \setminus I}$. In this case, the only non-empty set for our example is:

$$M^{kv^- \cap I}_{key} = (1, 1)$$

That is, the word “the” contributed a single count to the frequency, “1” (which is being nullified). As with addition, the challenge lies in considering the case where we have an intersection in the key space in $M^{kv^- \cap I}$. Assuming the previous value lists, vl, are available there are three distinct partitions to consider for the results when the new value lists, vl^-, are considered:

$$R_+^{kv^- \cap I} = \bigcup_{key \in M_{key}^{kv^- \cap I}} [Reduce(key, vl_{key}^-) \setminus Reduce(key, vl_{key})]$$

$$R_\varnothing^{kv^- \cap I} = \bigcup_{key \in M_{key}^{kv^- \cap I}} [Reduce(key, vl_{key}^-) \cap Reduce(key, vl_{key})]$$

$$R_-^{kv^- \cap I} = \bigcup_{key \in M_{key}^{kv^- \cap I}} [Reduce(key, vl_{key}) \setminus Reduce(key, vl_{key}^-)]$$

As before, $R_\varnothing^{kv^- \cap I}$ are values that are seen both in the original and the trimmed executions; these values can be ignored when considering changes downstream. The other values are used in our delta definitions:

$$R^- = R + \Delta_+ - \Delta_- \tag{3.2}$$

$$\Delta_+ = R_+^{kv^- \cap I}$$

$$\Delta_- = R^{kv^- \setminus I} + R_-^{kv^- \cap I}$$

Our example produces the following non-empty sequences:

$$R_+^{kv^- \cap I} = (1, 3)$$

$$R_-^{kv^- \cap I} = (1, 4)$$

That is, we are replacing the count for `(1, 4)` with `(1, 3)` because we've removed the contribution of `(the, 1)`. This differs from the results from figure 1.1b because it doesn't account for the new contribution of {`(jumps,1)`, `(over,1)`, `(lazy,1)`} which are all in Δ_+ from the previous section. As before, the elements of Δ_+ can be treated as information added to follow-on processes as considered in the preceding section. Likewise, the elements of Δ_- would undergo the same treatment as outlined in this section. The following section will consider the general case where multiple values are added and deleted in the same execution. The discussion of combiners will follow in section 3.3.

3.2 General MapReduce with Deltas using Counting Sets

In this section we will explore an algorithm for generically handling the case of adding and removing information sequences from a MapReduce calculation:

$$I' = I + \Delta_+ - \Delta_- \tag{3.3}$$

The strategy of the algorithm is to carry the "sign" of the tuples through the MapReduce calculation and to perform appropriate actions given this information. To do so, we apply an isomorphism between sequences of value pairs (partitioned by key) into what we call counting sets:

$$Init : (\mathbb{V}) \mapsto \{\mathbb{V} \times \mathbb{Z}\}$$

$$Unroll : \{\mathbb{V} \times \mathbb{Z}\} \mapsto (\mathbb{V})$$

Because this mechanism is used during the combine to reduce phase of the streaming computation, it is possible to discard the key (which is used to partition the value lists). By focusing on the management of value lists we are able to reason about this extended structure as a simple extension of any MapReduce algorithm (in which the value carries a sign value). To start, we have an *Init* function that takes raw key/value pairs and produces signed components:

$$Init(key, value) = \begin{cases} \{(value, +1)\} & \text{if } (key, value) \in \Delta_+ \\ \{(value, -1)\} & \text{if } (key, value) \in \Delta_- \end{cases}$$

In general, the first set of input consists of “positive” messages; which is to say the information is being added to the input set. However, there are cases where information may be “negative”; for instance, a stream of data from Twitter contains both “positive” tweets and “negative” delete signals. When a user deletes a tweet, a special message is transmitted that says “the tweet with id x has been deleted”. Once the information has been sent through the first MapReduce with delta stages, “negative” messages become commonplace.

The *Unroll* function takes a counting set and produces a sequence of values:

$$Unroll(S) = \bigcup_{(v,c) \in S} [\bigcup_{i=0}^{c} v]$$

When given a set, S, of signed elements *Unroll* expands into a sequence of values pairs based on the count, c, of the signed element. If c is negative or zero, no values will be emitted for that component.

Finally, we will define a binary operation, $\odot$ – or *Combine*, that combines counting set elements.

$$vals(S) = \{v|(v,c) \in S\}$$

$$getc(v,S) = \begin{cases} c & \text{if } \exists(v,c) \in S \\ 0 & \text{otherwise} \end{cases}$$

$$A \odot B = \{(v, c = getc(v,A) + getc(v,B))|\forall v \in vals(A) \cup vals(B) \text{ if } c \neq 0\}$$

The *vals* function returns the set of values for a given counting set. The *getc* function returns the count of a given value for a given counting set or zero if none exists. The $\odot$ operation combines two counting sets. For any distinct value from A or B it will pass the count along unaltered. For values in both A and B the combine function returns the sum of the count values if it is not zero.

3.2.1 Extending MapReduce

Using the counting sets structure, we can extend an existing MapReduce calculation to support delta calculations. First, all input needs to pass through the

Init function:

$$I_\delta = \{(k, Init(k, v)) | \forall (k, v) \in I\}$$

Given a *Map* function we create a *MapDelta* function:

$$M_\delta = MapDelta(k, v, c) = \bigcup_{i=0}^{p} (M^i_{key}(k, v), (M^i_{val}(k, v), c))$$

This has the effect of carrying the signed count through the map phase. Because the *MapDelta* function maintains the key/value output signature of a typical map phase, the shuffle phase is the same. The reduce phase is slightly different; first regardless of the combinable state of the native reduce calculation, the counting set is combinable. As such, the combine is applied to the outgoing (from map) and incoming (to reduce) counting sets partitioned by key at each stage:

$$M'_\delta = \bigodot_{s \in M_\delta} s$$

Where $\odot$ represents the counting set combine operator. Finally, after the shuffle, reduce proceeds as follows:

$$vl' = Unroll(\bigodot_{s \in vl} s)$$

$$R_\delta = \bigcup_{key \in M_{key}} Reduce(key, vl'_{key})$$

The output of the reduce stage is used to calculate the delta sets for any follow-on processes using equations 3.1 and 3.2.

3.2.2 Trident Functions Implementing MapReduce with Deltas

Now that we have the machinery necessary to generically handle MapReduce operations with updates, this section describes the algorithms to implement this functionality in Storm's Trident. Figure 3.1 provides a graphical representation of what follows. The calculation is broken down into three separate segments: `TriMap`, persistence, and `ReduceDelta`. The `TriMap` function takes signed key/value pairs and executes the map phase of the calculation. The persistence phase handles tracking of value lists before the reduce stage. The `TriReduce` function takes a key and a list of values and produces a set of positive and negative messages.

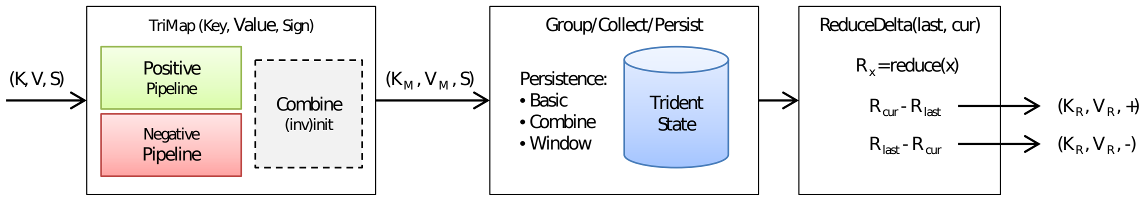


Figure 3.1: MapReduce with Deltas implementation in Storm's Trident

Listing 3.1 shows the algorithm running in `TriMap`. First we define `CollectorWrapper` which augments any emitted values with a sign. Next, we have the actual `TriMap` function which passes a wrapped collector to the real map function.

```
# Wrapper to tag emissions with the appropriate sign
class CollectorWrapper:
    def __init__(self, collector, sign):
        self.collector = collector
        self.sign = sign
    def emit(self, key, value):
        self.collector.emit(key, value + (self.sign,))
# Sign-aware map function.
class TriMap:
    def __init__(self, realMap):
        self.realMap = realMap
    def execute(self, tuple_in, collector):
        key, value, sign = tuple_in
        self.realMap.map(key, value,
                CollectorWrapper(collector, sign)
```

Listing 3.1: CollectorWrapper and TriMap implementations

If there is a reduce phase of the calculation, it is preceded with a persistence step. The Trident persistence mechanism supports combinable calculations and is thus broken into three stages: “map-side” combine (called stage zero in later chapters) within a batch of messages, “reduce-side” combination (called stage one), and emission. Listing 3.2 provides an implementation of the basic persistence mechanism based on counting sets (the combine and window mechanisms will be discussed in later sections). After the `TriMap` function has been executed on each tuple within a batch, the tuples are partitioned by destination key, initialized, and combined before transmission to the “reduce-side” node. Initialization takes each tuple and converts it into a counting set. At the “reduce-side” node, the existing state for the specified key is retrieved (from cache if available, backing store upon miss, or initialized to an empty state if previously unseen). The incoming batch of tuples is merged into the current state. The current and previous state are then passed to the `TriReduce` function.

```
class BasicReducePersistence:
    def __init__(self, backingStore):
        self.backingStore = backingStore
    @staticmethod
    def initialize(tuple_in):
        # Separate the value from the sign.
        value, sign = tuple_in
        return {value: sign}
    @staticmethod
    def merge(A, B):
        ret = A.copy()
        for value, count in B.items():
            ret[value] = ret.get(value, 0) + count
            if ret[value] == 0: del ret[value]
        return ret
    def update(self, k, incomming):
        # Caching is in backingStore
        last = backingStore.get(k)
        if last is None: last = {}
        cur = self.merge(last, incomming)
        return last, cur
        # backingStore.put(k, cur) would happen later.
```

Listing 3.2: Persistence pseudo-code

Listing 3.3 shows an implementation of the reduce phase. The reduce phase is fairly straightforward: run reduce on *last* and *cur* to determine the delta messages. The `CountingCollector` implementation first accumulates values emitted when executing against *last*. Once `reduce` returns, the `TriReduce` class switches the collector so that it can capture the values from *cur*. As values are emitted from *cur*, the collector decrements the counts from *last_res* if available, otherwise counts are collected under *cur_res*. Finally, any values in *last_res* and *cur_res* are unrolled and emitted as negative and positive records respectively.

```
class CountingCollector:
    def __init__(self):
        self.last_res = {}
        self.cur_res = {}
        self.in_last = True
    def emit(self, kv):
        if in_last:
            # Count up the times we see the specific kv pair.
            self.last_res[kv] = self.last_res.get(kv, 0) + 1
        else:
            last_c = self.last_res.get(kv, None)
            if last_c is None:
                # We are creating positive messages now.
                self.cur_res[kv] = self.cur_res.get(kv, 0) + 1
            else:
                last_c -= 1
                if last_c == 0:
                    # All previous occurrences are accounted for.
                    del self.last_res[kv]
                else:
                    # Still some outstanding occurrences.
                    self.last_res[kv] = last_c
class TriReduce:
    def __init__(self, realReduce):
        self.realReduce = realReduce
    def execute(self, tuple_in, collector):
        last, cur = tuple_in
        counter = CountingCollector()
        self.realReduce(last.getTuples(), wrapped)
        wrapped.in_last = False
        self.realReduce(cur.getTuples(), wrapped)
        # unroll last_res as negative messages.
        # unroll cur_res as positive messages.
```
Listing 3.3: TriReduce implemenation

3.3 Combiners

In section 3.1.1 we briefly mentioned that adding information does not present issues to combiner functionality. This is true because the combiner functionality is designed specifically to incorporate partial information in a manner that makes it possible to calculate a final value. Listing 2.3 provides an implementation of `COUNT` that combines values (listing 3.5 which follows has been augmented to support subtracting information).

Consider the value list for “1” from the second stage MapReduce after the first batch of information has been seen:

$$vl_1 = \{(1,1),(1,1),(1,1),(1,1)\}$$

The storage and execution overhead for this list is linear based on the input size. When using a combiner, after data leaves the map phase it is partitioned by destination key and initialized (in the case of count, it is replaced by the length of the field). Hadoop and Trident will then opportunistically execute the *intermediate* function on any lists available before transmission. The intermediate function combines values in some way (for `COUNT` the intermediate function returns the sum of the input values). Finally, once transfered to the reduce phase, the *final* method is called to produce the same value that would be produced without using a combiner (again, using sum for `COUNT`). The combined version of the previously mentioned value list is:

$$Combine(vl_1) = \{(4)\}$$

The savings for `COUNT` are quite significant in space and time. A single value is tracked instead of lists of values – this lowers network pressure between “map” and “reduce” nodes and between “reduce” and “storage” nodes. It also lowers the execution time of the calculation – updates need only consider a single value for all previously seen data. The issue we face is what to do when information needs to be removed. The counting set seen in section 3.2 provides an example of the mechanism necessary to achieve these goals. Where combiners in general MapReduce must be abelian monoids, to function under MapReduce with deltas combiners must be abelian groups. An abelian group is a set S along with a binary operation + that satisfies the following axioms (for abelian monoids, the inverse elements condition is discarded):

Closure $- \forall a, b \in S : a + b \in S$

Associativity $- \forall a, b, c \in S : (a + b) + c = a + (b + c)$

Commutativity $- \forall a, b \in S : a + b = b + a$

Identity element $- \exists id \in S : \forall a \in S, a + id = a = id + a$

Inverse elements $- \forall a \in S, \exists - a \in S : a + -a = -a + a = id$

Closure is necessary for combine operations because the combine function can be called multiple times at the map phase. *Associativity* and *Commutativity* are necessary because there is no guarantee to the boundaries between map tasks or the order of the results produced by map operators. The contract for reduce ensures that the value lists are ordered, thus the associativity restriction can be relaxed for reduce operators. *Identity elements* are generally used as initial states for the Trident storage mechanism. Also, when information is annihilated by it's negative, the identity element is used to show the absence of information. *Inverse elements* are necessary to carryout the information removal function of MapReduce with deltas.

Many of the built-in combinable functions for Pig are also invertible such as `SUM` and `COUNT`. To remove the contribution of a specific element, we transmit the negative of its value for `SUM` or negative one for `COUNT`. Others, such as `MIN` and `MAX`, do not have inverse elements; when you remove specific items (such as the current min or max value), there is no way to properly recover. Others can be converted to abelian groups by introducing some overhead. This includes the use of counting sets to implement `DISTINCT` or to use counting Bloom filters in place of traditional Bloom filters. Such changes require re-implementation of the intermediate and final functions to produce the same results as the traditional implementations. Finally, in the worst case, counting sets provide an abelian group that can produce value lists to be used with any reduce operator. In the next section, we will discuss a solution for applying abelian group combiners in Trident.

3.3.1 Generalized Combine for Abelian Groups

Listing 3.4 provides an outline of a general technique for handling abelian group combiners in Trident. Before information is passed into the *merge* routine, it must be initialized. This corresponds to translating values from those seen from map output M_{val} to combine-able values C_{val}. If the tuple represents negative information, the inverse initializer is used. After converting to C_{val}, the positive/negative distinction must be carried within the values themselves. There are two distinct phases of combination: those that occur before data has been transmitted to the reduce node (stage zero) and those that occur right before `TriReduce` is called (stage one). The combine phase before network transmission has no observable side effects or emissions beyond the store, thus it is incorrect to track the last state (which is used to determine what delta emissions are necessary to correct any side effects). This implementation has a flag that will disable tracking of the last state for use before network transmission occurs. However, when used at the reduce node, it becomes necessary to track the last state so it can be properly passed to `TriReduce`.

```
class CombineWrapper:
    def __init__(self, agg, trackLast):
        self.agg = agg
        self.trackLast = trackLast
    def initialize(self, tuple_in):
        value, sign = tuple_in
        if sign > 0:
            return {'cur': self.agg.initial(value)}
        return {'cur': self.agg.initialInv(value)}
    def merge(self, A, B):
        ret = {}
        if self.trackLast:
            # Assume A came from backing store.
            ret['last'] = A['cur']
        ret['cur'] = self.agg.merge(A['cur'], B['cur'])
        return ret
```

Listing 3.4: TriCombine implemenation for general combiners.

Listing 3.5 provides an invertible implementation of Pig's `COUNT` function. The difference between this function and the non-invertible one seen in 2.3 is the addition of the `initialInv` function. Because the negative values will naturally update the integer count of the objects, it is unnecessary to alter the intermediate and final routines.

```python
class COUNT:
    @staticmethod
    def initial(tuple_in):
        return len(tuple_in.get(0))
    @staticmethod
    def initialInv(tuple_in):
        # The following is necessary to support
        # information removal.
        return -len(tuple_in.get(0))
    @staticmethod
    def intermediate(tuple_in):
        # tuple_in contains a bag of numbers
        return sum(tuple_in.get(0))
    @staticmethod
    def final(tuple_in):
        # Same as intermediate.
        return self.intermediate(tuple_in)
    @staticmethod
    def exec(tuple_in):
        # Fall back method.
        return len(tuple_in.get(0))
```

Listing 3.5: COUNT implementation that supports subtracting information

The program in listing 3.6 demonstrates issues with our current implementation of COUNT. Imagine we have a text file named `seq.100.txt` containing the integers in the range $[1, 100]$. The first line creates a bag containing these integers. The second line determines if the number is even ($is\_even = 1$) or odd ($is\_even = NULL$). The third line groups the values based on this property. The fourth line counts the even and odd numbers using basic `COUNT` as well as `COUNT_STAR`. The final line dumps the results.

The output of this program is: `(1, 50, 50)` for the even values and `(NULL,0,50)` for the odd values. The reason for the difference (and the existence of `COUNT_STAR`) is because the basic `COUNT` routine only counts non-null values. For our histogram example this presents a significant problem: assume the next batch of data only contains the word "the", this will update the histogram for this key to be `(the, 3)` which will cause the values {`(2,-1), (3, 1)`}. When the histogram for "2" is updated in the second stage, the result will be `(2,0)` when the resulting tuple should no longer exist. The solution to this issue is to track the number of tuples that contribute to a combined value – if it ever reaches zero, `ReduceDelta` should produce the empty sequence as the current output for reduce. This has yet to be patched in the current code base and is an outstanding bug.

```
seq = LOAD 'seq.100.txt' AS n:int;
parity = FOREACH seq GENERATE (n%2==0?1:NULL) AS is_even;
gr = GROUP parity BY is_even;
counts = FOREACH gr GENERATE group,
    COUNT(parity), COUNT_STAR(parity);
dump counts;
```

Listing 3.6: Pig code to demonstrate issues with converting basic COUNT to an invertable streaming computation.

The mechanism presented in this section provides a means for more efficient executions when processing streams of data. As stated, some functions that would combine under general MapReduce cannot be combined when negative messages are used. However, the base reduce functionality using counting sets does provide an abelian group that handles value lists that allows any reduce function to work in a streaming environment. The following section proves that counting sets provide an abelian group.

3.3.2 Abelian Group Proof of Base Reduce Structure

The counting set has the nice property of being combinable – this allows some positive/negative pairs to annihilate each other before being transmitted across the network. The compression-like functionality has the potential to recover some of the storage savings lost when functions are not invertible – consider a `MAX` call against highly repetitive data – the counting set will accumulate like values and transmit them only once. This section provides a proof that the counting set is an abelian group.

Theorem 1. *Counting sets are commutative under combine.*

Proof. Consider:

$$A \odot B = \{(v, c = getc(v, A) + getc(v, B))|\forall v \in vals(A) \cup vals(B) \text{ if } c \neq 0\}$$

Due to commutativity of integer addition and set union:

$$A \odot B = \{(v, c = getc(v, B) + getc(v, A))|\forall v \in vals(B) \cup vals(A) \text{ if } c \neq 0\} = B \odot A$$

Thus:

$$A \odot B = B \odot A$$

□

Theorem 2. *The identity element of the counting set is the empty set $id = \varnothing$.*

Proof. Consider the definition of the combine operator for some counting set $A \in S$:

$$A \odot B = \{(v, c = getc(v, A) + getc(v, id)) | \forall v \in vals(A) \cup vals(ids) \text{ if } c \neq 0\}$$

Now, the union of values between A and id is exactly the values in A (because $vals(id) = \varnothing$).

$$A \odot B = \{(v, c = getc(v, A) + getc(v, id)) | \forall v \in vals(A) \cup \varnothing \text{ if } c \neq 0\}$$

$$A \odot B = \{(v, c = getc(v, A) + getc(v, id)) | \forall v \in vals(A) \text{ if } c \neq 0\}$$

Likewise, $getc$ on the empty set will always return 0 for any value in A.

$$A \odot B = \{(v, c = getc(v, A) + 0) | \forall v \in vals(A) \text{ if } c \neq 0\}$$

$$A \odot B = \{(v, c = getc(v, A)) | \forall v \in vals(A) \text{ if } c \neq 0\}$$

The $getc$ function will always return the c component for any $(v, c) \in A$. Because $vals(A)$ will return all possible values from A, we have:

$$A \odot id = \{(v, c) | \forall (v, c) \in A \text{ if } c \neq 0\}$$

Because $A \in S$, c is never zero:

$$A \odot id = \{(v, c) | \forall (v, c) \in A\} = A$$

Due to commutativity of the combine operator:

$$A \odot id = A = id \odot A$$

$\square$

Theorem 3. *Counting sets are closed under combine.*

Proof. Consider:

$$A \odot B = \{(v, c = getc(v, A) + getc(v, B)) | \forall v \in vals(A) \cup vals(B) \text{ if } c \neq 0\}$$

First, only valid values v will be permitted because $vals(A) \cup vals(B) \in \mathbb{V}$ by definition. Second, the sum of the counts $c \in \mathbb{Z}$ by definition. So, elements that have a non-zero count will create valid members of S. The final consideration is the case where no elements have non-zero count – this produces the identity element $\varnothing \in S$. Thus:

$$A \odot B \in \mathbb{V} \times \mathbb{Z}$$

$\square$

Theorem 4. *Counting sets are associative under combine.*

Proof. If we expand $(A \odot B) \odot C$ (and move the non-zero condition), we get:

$$expanded = \{(v, c = (getc(v, A) + getc(v, B)) + getc(v, C)) | \forall v \in (vals(A) \cup vals(B)) \cup vals(C)\}$$

$$(A \odot B) \odot C = \{(v, c) \in \textit{expanded} \text{ if } c \neq 0\}$$

With the non-zero condition in place, $vals(A \odot B) \subseteq (vals(A) \cup vals(B))$ (that is, there can be missing elements from the union). However, this expanded form will be exactly equal to the value obtained by calculating the values piecewise. Now, using the associative properties of integer addition and set union:

$$\textit{expanded} = \{(v, c = getc(v, A) + (getc(v, B) + getc(v, C)))|\forall v \in vals(A) \cup (vals(B) \cup vals(C))\}$$

$$(A \odot B) \odot C = \{(v, c) \in \textit{expanded} \text{ if } c \neq 0\} = A \odot (B \odot C)$$

□

Theorem 5. *For any counting set A, there exists an inverse:*

$$-A = \{(v, -getc(v, A))|\forall v \in vals(A)\}$$

Proof. Expanding $A \odot -A$ yields:

$$\{(v, c = getc(v, A) + getc(v, -A))|\forall v \in vals(A) \cup vals(-A) \text{ if } c \neq 0\}$$

We know that $vals(A) \cup vals(-A) = vals(A)$ by definition. We also know that $getc(v, -A)$ will return $-getc(v, A)$ for all key/value pairs in A, which yields:

$$\{(v, c = getc(v, A) + -getc(v, A) = 0)|\forall v \in vals(A) \text{ if } c \neq 0\}$$

Because the count portion always equals zero, we find that:

$A \odot -A = \varnothing = id$

The reverse is also true due to the commutativity of the combine operator. Thus:

$A \odot -A = id = -A \odot A$ $\square$

By proving that the counting set is an abelian group we have shown that it can be used as a combinable structure within MapReduce with deltas. Using this structure, we are able to generically manage the value lists passed to the reduce operator which makes it possible to remove information and propagate further changes through chained aggregations.

3.4 Windows

Windows are a natural way of handling infinite sequences in a stream. There are many different types of windows as outlined in [22]: time based, element based (sliding, cascading, batch, etc.) with many different semantics for interaction between slides. They are applied to many streaming constructs (notably streaming SQL) to put a bound on the number of elements for consideration. The delta propagation techniques presented in this chapter provide an alternative to windowing – one that requires no modification of the script to be run. However, there are certainly cases where windowing is still an appropriate mechanism for handling streams of information. The NationMind case study presents an example where you have a stream of data joined against a large static data set. Using counting sets, the value lists in this join will grow without bound, ultimately requiring prohibitive storage and execution time. To address this issue, we have introduced a windowing mechanism to the storage layer of our implementation of MapReduce with deltas. This section outlines the algorithm used and describe how it interacts with the delta propagation mechanism.

First, there is a difficulty with window operations and Trident. Unlike Storm's bolt which can emit a record at any time, a Trident operator can only produce results when input occurs. So, if you wanted to have a time-based window that closed after five minutes there could be no notification that the window has closed without external input. This problem is relatively simple to solve with the delta propagation mechanism: use non-blocking windows that produce results on every iteration. So, if you were wanted to trigger an alarm after five minutes if you didn't receive a second event, you would emit a "triggered event" with a timestamp in the future (that is calculated from the original event). If the second event comes in, the system would transmit a negative version of the "triggered event". External consumers would need to handle their own alarm mechanism and treat any trigger events as suspect until the specified time has elapsed.

Our main goal was limiting item overhead in the system, so we implemented count-based windows that age off. However, the second wrinkle to deal with is the fact that a window could fill and close while in the stage zero combiner. To solve this issue, we keep all closed windows around for some specified amount of time. Due to issues with time jitter in a distributed system it is best to set the age-off to a value that allows the window to reach the reduce phase and isn't too sensitive to delays caused by burst activity. The default timeout is 10 seconds (from the close time), once the expiry is reached, the window is discarded. We couple this with TTL's on our values in our external state mechanism to ensure that stale values that aren't updated in a timely fashion are expired at the storage layer (say after 30 seconds).

While a window is open, it can have items added or deleted. Once closed, no value changes can occur – this presents a corner case where a negative message for a specific record arrives while the original value is held in a closed window. For our current use cases, we are only windowing on positive-only streams of information.

Finally, when a window is created it is given a random id. This id is used to pair windows in *cur* and *last* persistence mechanism. These pairs are used to create independent tuple bundles to be processed by the reduce delta mechanism. Because windows share an id and an expiration if they have been closed, they will both expire at the same time without producing negative messages for downstream processing. This prevents the window mechanism from draining the state of any follow-on processes. While it is true that the data used to produce the follow-on results has expired, the side effects are allowed to perish at a different rate.

3.5 Chapter Summary

This chapter outlines one of the major contributions of this work – the algebraic foundation for handling changes through chains of aggregates. This chapter opens with a two-stage calculation to determine the histogram of word count frequencies from a stream of text that provides a concrete example for the remaining sections of the chapter. We then present a model of MapReduce calculations based on message passing and proceed to show what occurs when information is added and subtracted. What follows is an introduction of counting sets as a general mechanism for managing value lists between map and reduce. We show a simple extension of the MapReduce mechanism to incorporate attributes denoting the "addition" and "subtraction" of information. We then describe the algorithms necessary to implement this extended model in Storm's Trident. We then discuss the properties necessary to enable combiners in generic and extended MapReduce. We describe extensions to the Pig API to enable combiners in streamed MapReduce. We also provide a proof that the counting set mechanism meets the criteria of combiners for MapReduce with deltas. Finally, we discuss how window computations are necessary under certain circumstances and describe how they interact with MapReduce with deltas.

By providing a generic mechanism for handling negative messages, we are able to stream any existing MapReduce algorithm correctly. However, switching from batch to stream processing requires concessions. First, there is the change in behavior of combiners which can impact performance. For some cases, the combiners still function efficiently and no concessions must be made. However, some simple calculations (such as `MIN` and `MAX`) cannot be made to run perfectly and efficiently. In such cases, one must make a choice – approximate the solution and hope that information removal doesn't completely obliterate that answer or store more information and calculate the correct value. Such decisions are always necessary for system design. Another important consideration: when a batch job is written poorly, it just takes longer – the MapReduce system is fairly resilient to bad programming. When a stream processing algorithm is written poorly it can cause instability in feeding queues, overload backing stores, obliterate networks and system memory – ultimately failing hard (versus graceful degradation). Because of this, it is necessary to understand the performance implications of a running analytic. However, before we discuss such things, we must have case studies to ponder. The following chapter will show interesting case studies using the techniques outlined in this chapter followed by a chapter that addresses performance concerns.

Chapter 4

Squeal: Pig on Storm

Squeal[1] is an execution environment for Pig that converts Pig scripts into Storm topologies. This chapter discusses some of the capabilities of Squeal through a series of case studies using Twitter data. The case studies are presented in order of simplicity and functionality. We open with the humble word count, followed by NationMind, and finish with a discussion the limits of automatic conversion. Each section will open with a scenario motivating the study, progress into a description of implementation, and close with a discussion of runtime performance.

Twitter data is one of the most abundant sources of publicly available streamed data. For our studies, we had two sources of tweets: a random sample and a focused query. The random sample (often called the "garden hose") is the free version of the "fire hose". The focused query uses specific keywords to provide information. Due to the initial use case of tracking the response to the 2012 US Presidential Elections, the term "Obama" has always been included in this filter. Later, terms were added and removed in response to events (Superbowls, Grammys, the election of the Pope, etc.). In steady state, we collect around 100 tweets per second but have seen significant spikes surrounding events (which will be seen in the following chapter).

[1]Available on github: https://github.com/JamesLampton/piggybank-squeal

4.1 Word Count

On Monday April 15, 2013, two explosions erupted near the finish line of the ongoing Boston Marathon leaving three dead and an estimated 264 wounded[2]. After a very rapid investigation of pictures and video footage from the scene of the crime, the FBI was able to release photographs of the suspects on April 18. During the evening of the 18th, the suspects were located and confronted leading to the death of one suspect. The second suspect fled and was believed to be in the vicinity of Watertown, MA. On the evening of the 19th, the second suspect was discovered hiding in a boat stored in a back yard and apprehended.

I experienced this event, like many others, live through breaking coverage from a national news sources. Previously, I had been working on tracking the response to specific concepts in tweets but was concerned with tracking unexpected concepts as they happened. After seeing the breaking news, I decided to finally implement the most humble of text-based analytics – word count. I developed a small Pig script to tokenize tweets, count the word occurrences in time, and store them in a database. After running a test of a static batch of tweets in Hadoop, I then used Squeal to begin processing the live feed of tweets.

Friday evening I sat down to watch the news with my spouse. The reporter on CNN was recounting the events leading up to the massive manhunt underway. There was a pause as they heard what they thought was gunfire. This was followed by police cruisers racing past in the background. I decided to connect to my database

[2]http://en.wikipedia.org/wiki/Boston_Marathon_bombings

to see how people were responding on Twitter.

Figure 5.2 shows a graph of a few terms that popped out. First, the word "boat" had a very high score at the time – being a common word, I ignored it. However, I found the word "scanner" to be odd. I pulled up Twitter and fed it into the live search mechanism. During the night of the 18th, someone took a police scanner in the Boston area and wired it up to an internet streaming service. Basically, people were live tweeting the capture of the suspected Boston Marathon Bomber.

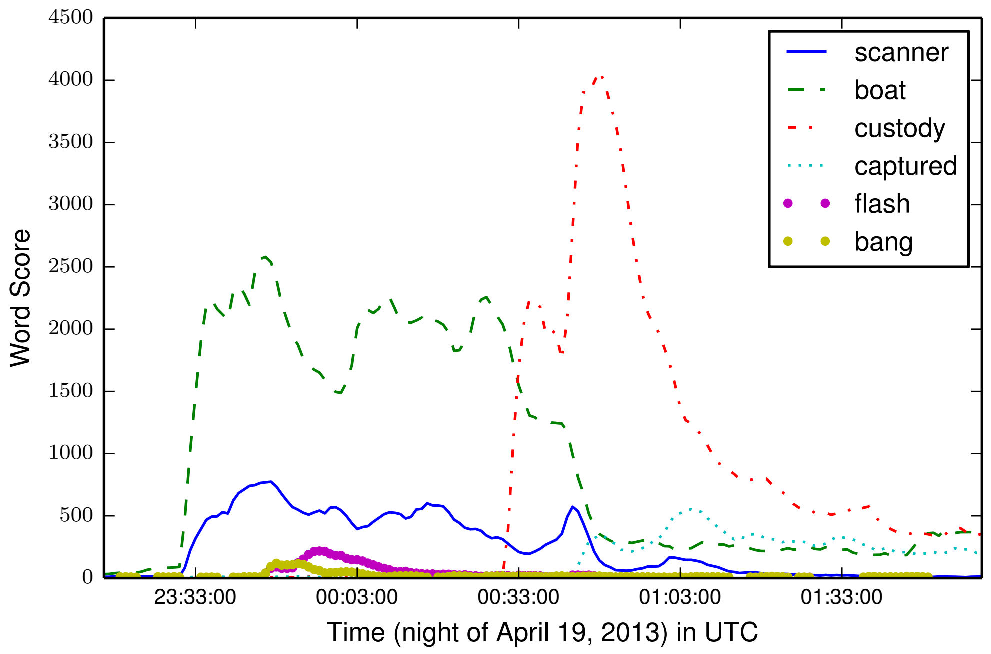


Figure 4.1: Selected word scores during Boston Marathon Take Down

CNN and other news outlets knew exactly what was going on – they had cameras focused on the scene the whole time. However, these organizations have blackout policies for these situations so they don't make them worse. The mild bumps for "flash" and "bang" would make a tactical team's blood run cold. Such devices disorient their targets and provide brief opportunities of surprise.

Another interesting feature of this graph is the initial bump of “custody” followed by a larger bump and a bump in the use of the word “captured”. The initial bump is the internet reporting that the suspect was in custody – 15 minutes before the official announcement. At 8:59PM, the Boston Police department tweeted[3] “CAPTURED!!! The hunt is over. The search is done. The terror is over. And justice has won. Suspect in custody.”

Ultimately Twitter provided unfiltered updates on the situation. The case of the Boston Marathon Bombing does provide some cautionary tales – specifically incorrect identifications of the suspects from various crowd-sourced websites. The take down itself could have gone very differently if this feed of information was exploited for ill gain. Society will have to adjust to this new normal – people will need to explicitly delay response to and scrutinize information where these were previously handled by the throughput limitations and curation of traditional media.

4.1.1 Implementation

To illustrate how completely Squeal is able to convert Pig scripts, I will explain this example in great detail. Listing 4.1 shows the code to load tweets in JSON format and extract the words therein. This specific example shows the batch version of the load statement. Switching to a streaming version requires a spout that produces tweets in JSON format. The `SpoutWrapper` loader provides hints to the compiler that the specific alias is being streamed.

[3]https://twitter.com/bostonpolice/status/325413032110989313

```
tweets = LOAD '$input' AS (tweet:chararray);
--tweets = LOAD '' USING SpoutWrapper(..) AS (tweet:chararray);

x = FOREACH tweets GENERATE JSONPath('id␣created_at␣text', tweet);
x = FILTER x BY $0#'error' IS NULL;
x = FOREACH x GENERATE $0#'id' AS id,
    $0#'created_at' AS created_at, CleanTweet($0#'text') AS text;

x = FILTER x BY created_at IS NOT NULL AND created_at != '';
words = FOREACH x GENERATE id,
    ISOToMinute(CustomFormatToISO(
        created_at, '$time_format')) AS timestamp,
    FLATTEN(BiTOKENIZE(text)) AS (word1, word2, pos, ngramlen);

words = FILTER words BY
        word1 MATCHES '^#.*' OR
        word2 MATCHES '^#.*' OR
        (LENGTH(word1) > 2 AND (
            ngramlen == 1 OR LENGTH(word2) > 2) AND
        LENGTH(word1) < 32 AND LENGTH(word2) < 32);
```

Listing 4.1: Tweet loading and text extraction

What follows is the extraction of the *id*, *created_at* and *text* fields of the tweet. If the current records contains an error it is filtered out. In a batch environment, it may be necessary to stop all processing and clean up the input data. However, such hard failures are troublesome for real time flows of information. Sources of error are numerous but mainly center around stalled connections or other edge conditions (services outages, etc.). Any surviving records are passed through the `CleanTweet` routine which strips punctuation (collapsing contractions such as “we’ll” to “well”) and URLs from the text. This is followed by tokenization by word unigram and bi-grams (annotated with position and n-gram length). Finally, any bigrams containing hashtags are kept while short unigrams and long words are discarded.

Listing 4.2 shows how common words are removed from consideration using a mechanism called “replicated joins”. In MapReduce, the stop list would be loaded into memory at the map operators allowing for the join operation to be performed without a reducer. This requires that the replicated list fit in memory and provides significant savings by preventing unnecessary network traffic associated with a reduce-based join. Squeal examines the MapReduce plan, extracts any static branches and dispatches them to Hadoop for execution. For replicated joins, the temporary load files are copied over to a configured temp path to be loaded later when the topology is executed – providing the same benefits as in MapReduce. This logic keeps any record where one of the words is not stopped. These words are concatenated and ready for further analysis.

```
stop_words1 = LOAD '$stop_list' AS (stop_word1);
stop_words2 = LOAD '$stop_list' AS (stop_word2);
words_stj1 = JOIN words BY word1 LEFT,
    stop_words1 BY stop_word1 USING 'replicated';
words_stj = JOIN words_st_join1 BY word2 LEFT,
    stop_words2 BY stop_word2 USING 'replicated';
words = FILTER words_st_join BY stop_word1 IS NULL AND
    (ngramlen == 1 OR stop_word2 IS NULL);
words = FOREACH words GENERATE
    id, timestamp, CONCAT(word1, word2) AS word,
    pos, ngramlen;
```

Listing 4.2: Removal of common words using replicated join

Listing 4.3 takes each record and expands it into five different records with varying scores. When the timestamps were extracted in listing 4.1, they were truncated to the minute. This code will generate a *score* of 1.0 for that minute (and a *real_count* of 1), a *score* of 0.8 for the following minute (with no contribution to the count for that minute), and so on. This has the effect of having a word create an impulse signal that degrades over time. These accumulated scores provides a smoothing mechanism and a value that takes into account the recent history of a word. This technique also provides an alternative to an explicit windowing function in Pig at the expense of transmitting additional records.

```
-- Generate a word count in time.
words_time = FOREACH words GENERATE
    word, ngramlen, timestamp, FLATTEN(TOBAG(
        TOTUPLE(0, 1.0), TOTUPLE(60000, 0.80),
        TOTUPLE(120000, 0.60), TOTUPLE(180000, 0.40),
        TOTUPLE(240000, 0.20))) AS (offset, score);
--words_time = FILTER words_time BY offset == 0;
words_time = FOREACH words_time GENERATE
    word, ngramlen,
    UnixToISO(ISOToUnix(timestamp)+offset) AS timestamp,
    score, (score == 1.0?1:0) AS real_count;
```

Listing 4.3: Word score generation

Listing 4.4 groups the n-grams together by minute and computes the *score* and *real_count* for each. The first few lines provide hints to the compiler on how many parallel tasks to launch for the preceding calculations which make up the first map task. In subsection 2.1.2 we mentioned that the number of map tasks was determined by the number input chunks, there is no corresponding a priori mechanism for automatically scaling the tasks for streaming data. In fact, the number of tasks will depend on the input rate and the service time of the individual tasks. Understanding these constraints is the subject of the following chapter.

```
-- These are for the map named after this alias.
set words_time_gr1_parallel '$parallel_big';
set words_time_gr1_shuffleBefore 'true';

-- Now for the combine.
%default words_time_gr1_store_opts '';
set words_time_gr1_store_opts '$words_time_gr1_store_opts';
words_time_gr1 = GROUP words_time BY (word, ngramlen, timestamp)
    PARALLEL $parallel_big;
words_count = FOREACH words_time_gr1 GENERATE FLATTEN(group),
    SUM(words_time.score) AS score,
    SUM(words_time.real_count) AS real_count;
```

Listing 4.4: Grouping words and accumulating metrics

The "shuffle before" hint is used to decouple the spout from the map operators (allowing each to have independent parallelisms). The shuffle before hint can also be useful to decouple a reduce task with non-uniform output rates from the following map operators (allowing for improved parallelism at the cost of a network transfer). An example of such disparity was illustrated by figure 2.5 on page 23. MapReduce handles this case by spawning more mappers in the following MapReduce cycle due to more input chunks being produced by that particular reducer.

The "store opts" hint is used to specify how intermediate results are to be stored using Trident's persistence mechanism. For this specific example, the MapReduce compiler will use the combiner mechanism. Because of this, the intermediate results will be a float and an integer keyed by the grouping constraints (word, n-gram length and minute). The storage mechanism will be discussed in greater detail in the following chapter.

Listing 4.5 formats the records for later consumption, filters out records that don't have a score above 6 (which represents the upper 2.1% of the scores as measured from offline analysis of historical data), and stores the results. The active `STORE` command would store the results of the calculation into HDFS. The disabled `STORE` command would stream the results out to the RabbitMQ messaging system. In this particular case, negative messages are filtered out because the records have a natural primary key: time, word, and n-gram length. Because these results are destined for a database, any updates to the score and count can replace previous values. More complicated follow-on processing may require both positive and negative messages.

```
words_count = FOREACH words_count GENERATE
    ISOToUnix(timestamp), word, ngramlen, real_count, score;
words_count = FILTER words_count BY score > 6.0;

STORE words_count INTO '$output';
--explain words_count;
/*
STORE words_count INTO '$output' USING
    NegFilterSignStoreWrapper(
        'org.apache.pig.piggybank.storage.RMQStorage',
        'amqp://$rmqup@$rmqserver/twitter', '$exch-words_count');
    */
```

Listing 4.5: Word count output

This is a good time to talk about process failure. Trident only releases the final stage of execution at the end of a final commit. Because the `STORE` routine is the last node in a Pig graph any side effects will only be seen upon a successful commit. Thus, any system consuming results from a Pig Squeal script have some guarantees that the information seen is from a clean execution. However, there is a corner case where the output function may fail. This may cause the batch to be re-executed in Trident, which would produce the same results but may have duplicate transmissions if the original output function was able to transmit some messages.

4.1.2 Performance

The following chapter will explore the metrics for word count at a very low (per processing element) level along with a full discussion of the experimental setup and trace collection setup. This section will focus on summarize word count performance on a cluster of 16 nodes on Amazon's EC2. The nodes are identical VM configurations with four vCPUs, 15 GB of memory, and two 40 GB SSD-backed drives. Unlike the following section, an extra VM was used to host the metrics collection and RabbitMQ messaging system.

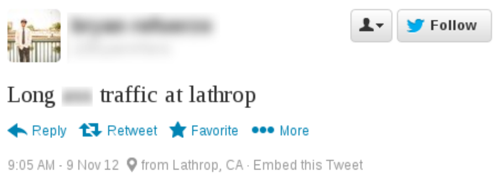
The word count example listed in the following section includes additional emissions created in listing 4.3 that aren't included in the standard word count explored in most literature. This significantly increases the workload on the network and was disabled for these tests. Also, testing was performed with an external store using HBase as well as a basic LRU ejecting in-memory store to examine the effects of the storage subsystem on performance. Finally, we have some traces with full metrics enabled but found that they negatively impacted the performance of the messaging system by consuming extra network and CPU resources. To reduce the impact of the metrics collection we only enabled tracking of the batch times for most measurements.

We found that with Storm active on only eight machines with the HBase storage system enabled (on all sixteen machines) we were able to max out the capacity of the RabbitMQ messaging system at around 3.3 thousand tweets per second. There was a peek rate of 4,000 which was the max rate allowed by our configuration of storm (eight spouts with a batch size of 250 releasing every 500 ms). We used 32 workers (one per CPU) along with 16 map tasks and 16 reduce tasks. At four thousand tweets per second the word count analytic produces 22 thousand records per second of output which was the cause of the overload. The average batch latency was 660 ms with a standard deviation of 257 ms (the storage accounted for 128 ms). This caused a load of 70-78 thousand requests per second to the HBase storage system. This also pushed the network at an aggregate of 422 MB/s.

Various other treatments were tested to examine the effects of varying different parameters but a test of the full capabilities of this system would require the use of a different messaging system such as Kafka[43]. While the overall performance of the system against known benchmarks is very important for practical use, the following chapter attempts to examine *why* an analytic performs as it does. By understanding the critical performance path it becomes possible to wisely focus optimization efforts.

4.2 NationMind

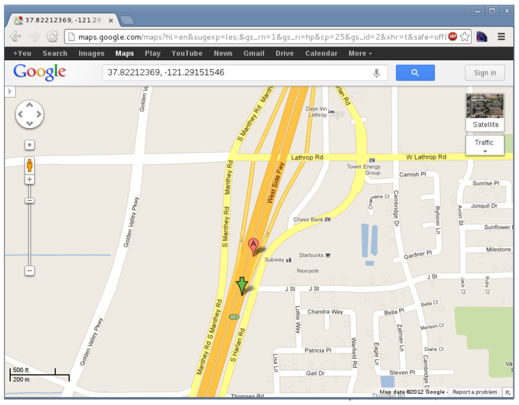
Consider the tweet: “stuck in traffic”. By itself, it informs us that someone is caught one of the hazards of modern life. If I provide you with the author’s profile, you may be able to guess what city the traffic is occurring in (barring a frequent long distance traveler). If this tweet was authored from a mobile device with GPS enabled, we can determine which intersection is having issues. Figure 4.2 shows an example along with a map of the location. While the text of a tweet is limited to 140 characters, the entire tweet available from the stream is 2.4 kilobytes on average (including profile information, location if enabled, etc.).


[redacted]

Long [redacted] traffic at lathrop

9:05 AM - 9 Nov 12 from Lathrop, CA · Embed this Tweet

(a) Geo-tagged tweet about traffic



(b) Location of tweet on Google Maps

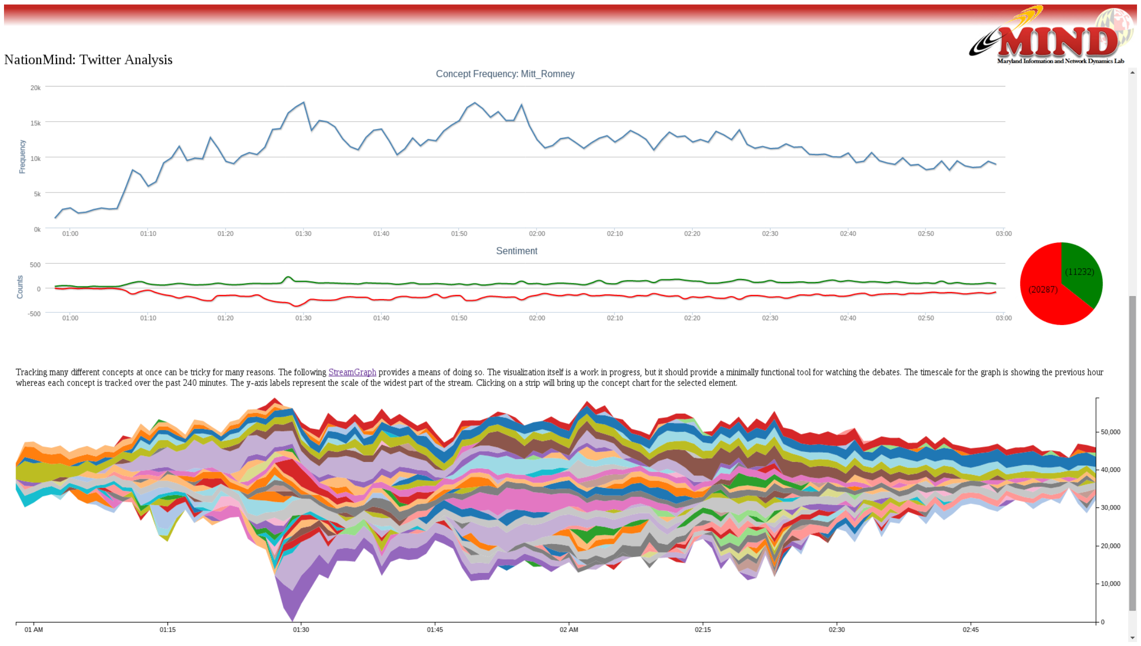
Figure 4.2: Determining context from tweets

Sadly for this branch of experimentation, the rate of geo-tagged tweets is far too small to reliably tell how bad a commute will be. During the month of October 2012, we collected 127 million tweets; of those only 2,238 tweets mentioned traffic with geo-location. However, at the same time the 2012 US Presidential Election was starting to heat up. In a speech to the 2012 Democratic National Convention, Former President Bill Clinton said the following: “People ask me all the time how we delivered four surplus budgets. What new ideas did we bring? I always give a one-word answer: arithmetic.” Twitter responded as expected with a burst of activity – the word “arithmetic” entered into modern political discourse.

There are certain things you can’t say to your advisor without expecting to complete the work. The following sentence is a prime example: "Hey! Wouldn't it be cool to track the response to the Presidential debates in real time?" Within a week I created a prototype using native Trident functions (in time for the first debate). For the Vice Presidential debate, we had a simple dynamic chart on a web page for a small cadre of users. By the second Presidential debate we had a stream graph tracking concepts from Wikipedia. Figure 4.3 shows a screen shot of the NationMind website showing the concepts discussed during the second debate[4]. The top part shows the number of tweets associated with Mitt Romney (Obama's is off-screen) while the bottom shows a stream graph summarizing the relative counts of other concepts. The debate opens with a discussion of the debate itself, followed by discussion of clean coal and taxes, which is then followed by Romney's statement "I love Big Bird, but I want to de-fund PBS" (at around $1:30$ in the figure). The lower purple stripes are for the concepts "Big" and "Bird".

In total, the original NationMind in Trident is $1,445$ lines of Java code spread across 14 modules. The current version of NationMind is 213 lines of Pig along with 559 lines of Java in four User Defined Functions (three of which have been seen in the previous example – `TwitterRMQSpout`, `CleanTweet` and `BiTOKENIZE`). The current version performs more advanced calculations than the original implementation including n-gram analysis with a maximum likelihood concept resolver. The rest of this section will discuss how these features use more of the Squeal functionality.

[4]A snapshot of this data is available for exploration at http://nationmind.umiacs.umd.edu/nm-debate-1/


NationMind: Twitter Analysis

Tracking many different concepts at once can be tricky for many reasons. The following StreamGraph provides a means of doing so. The visualization itself is a work in progress, but it should provide a minimally functional tool for watching the debates. The timescale for the graph is showing the previous hour whereas each concept is tracked over the past 240 minutes. The y-axis labels represent the scale of the widest part of the stream. Clicking on a strip will bring up the concept chart for the selected element.

Figure 4.3: Screen shot of NationMind from the second 2012 Presidential debate

4.2.1 Implementation

The first third of the NationMind script includes boilerplate definitions along with the same calculations necessary to create a list of bi-grams broken out by minute that we saw in the previous example (up to listing 4.2). Once we have the words broken out by tweet, listing 4.6 shows how we calculate simple sentiment. Lines 1-10 are executed in Hadoop because the sentiments are static dictionaries [46]. The alias *sent* is calculated by joining the word list with the sentiments. Each word is considered to have a positive, negative, or no sentiment. The id is reversed to make lookups more efficient in HBase (ids seem to be correlated in the higher order bits).

Lines 20-24 were added after the performance was measured while streaming. For a batch execution, this would result in an additional MapReduce execution which would result in longer execution times. However, when streaming, the lack of the additional grouping all of the words would be transmitted across the network causing the batch service time to double. Due to the nature of this specific data and computation the expansion (from tweet to words) and collapse (from sentiment values to tweet) occurs all within the map node. This leads to a 5x reduction in messages emitted due to this portion of the calculation.

```
positive = LOAD '$positive_fn' USING PigStorage(',') AS
    (sent_word:chararray, version:int);
positive = FOREACH positive GENERATE sent_word, 'pos' AS sentiment;
negative = LOAD '$negative_fn' USING PigStorage(',') AS
    (sent_word:chararray, version:int);
negative = FOREACH negative GENERATE sent_word, 'neg' AS sentiment;

sentiment_words = UNION positive, negative;
sentiment_words = FOREACH sentiment_words GENERATE
    LOWER(sent_word) AS sent_word, sentiment;

sent = JOIN words BY word LEFT,
    sentiment_words BY sent_word USING 'replicated';
sent = FOREACH sent GENERATE Reverse(id) as rev_id, timestamp,
    (sentiment IS NOT NULL AND sentiment == 'pos'?1:0) AS pos,
    (sentiment IS NOT NULL AND sentiment == 'neg'?1:0) AS neg,
    (sentiment IS NULL?1:0) AS none;

-- Additional group for streaming efficently.
set sent_gr_store_opts '$sent_gr_store_opts';
sent_gr = GROUP sent BY (rev_id, timestamp) PARALLEL $psmall;
sent = FOREACH sent_gr GENERATE FLATTEN(group),
    SUM(sent.pos) AS pos, SUM(sent.neg) AS neg,
    SUM(sent.none) AS none;
```

Listing 4.6: Tweet sentiment

Listing 4.7 loads the Google concept data [63] which provides a mapping from words to concepts represented by Wikipedia articles along with probabilities for the link, substring, and concepts. This data set has been used with great success in performance context free recovery of concepts from text. What is important for this example is that we have a static set of data being used outside the context of replicated join. The concept data is 2.6 GB compressed and doesn't lend itself well to being replicated for lookups. In this case, the compiler pulls the static portions up to the reduce (line 9). For the static execution, the reduce operator is replaced with code that stores the results into the backing store specified by the "store opts" hint. When the streaming topology begins execution, the static information is pre-populated and is used for performing the join. Another unique feature of this segment of code is the use of the window capability (which is activated by the "window opts" hint) to stream the infinite sequence of tweets across the static concept data.

```
1  concepts = LOAD 'concepts.prepped.bz2' AS (conc_word:chararray,
2      conc_prob:double, concept:chararray, support:chararray,
3      log_prob_conc:double);
4  concepts = FILTER concepts BY conc_prob > 0.21 AND
5      conc_word MATCHES '^.*[a-z].*$';
6
7  set conc_words_store_opts '$conc_words_store_opts';
8  set conc_words_window_opts '{"0":␣200}';
9  conc_words = JOIN words BY word,
10     concepts BY conc_word PARALLEL $pbig;
11 conc_final = FOREACH conc_words GENERATE id, timestamp, word,
12     conc_prob, concept, log_prob_conc, position, ngramlen;
```

Listing 4.7: Streaming distributed join

Listing 4.8 shows where the sentiments meet the concept data. In this case, `GROUP` is bringing the two data sets together in much the same way that `JOIN` does (without calculating the Cartesian products). The summing of the sentiment information is necessary to get scalar values for each. The `ConceptLatticeCompute` looks at each word position and determines if two uni-grams or a single bi-gram would provide a higher score. The lattice computation isn't combinable – even if it was Pig's optimizers don't appear to activate combine functionality for the `COGROUP` operator (if it did, the extra group by from listing 4.6 may not be necessary).

```
set sent_concept_store_opts '$sent_concept_store_opts';
sent_concept = GROUP
    sent BY (rev_id, timestamp),
    conc_final BY (Reverse(id), timestamp) PARALLEL $psmall;
tw_complete = FOREACH sent_concept {
    GENERATE FLATTEN(group),
        SUM(sent.pos) AS pos,
        SUM(sent.neg) AS neg,
        SUM(sent.none) AS unkn,
        FLATTEN(
            ConceptLatticeCompute(
                conc_final.(word, conc_prob, concept,
                    log_prob_conc, position, ngramlen)
            ).concept);
};
tw_complete = FOREACH tw_complete GENERATE Reverse($0) AS id, $1..;
```

Listing 4.8: Streaming Co-GROUP

While listing 4.8 shows two streams being grouped together (where listing 4.7 showed a stream meeting a static data set) the Trident batch mechanism along with how tweets are handled ensures this code will execute only once for each tweet. Their n-grams are generated and processed within the same batch. Couple this with the fact that the reduce operation will only run once all upstream producers have executed means that we will execute this data once for a tweet. The only way this would execute a second time (and produce negative information) is if the tweet was seen by both the filter source and the query source in a way that would span two batches. If the n-gram extraction code was generalized for word count and NationMind it is highly likely that the n-grams of a tweet would span multiple batches. This has impacts on the selection and use of backing store for key/value pairs that appear only once in a stream.

Listing 4.9 shows the final computation for NationMind – determining the number of mentions (with sentiment) of concepts in time. For this implementation, this is where negative messages are generated as many batches will span a minute. Interestingly, because the lattice computation is run once per tweet, the concepts generated will always be the same. This means that the counts for concepts in time will only increase (as in the word count example) providing for a natural primary key and simple database updates. If the n-grams for any given tweet were to span multiple batches you would see some concepts disappear from consideration (as better information comes in). In such a case, any updates to an external source would have to handle these updates in a more sophisticated manner.

```
set nm_final_gr_store_opts '$nm_final_gr_store_opts';
nm_final_gr = GROUP tw_complete BY (concept, timestamp);
nm_final = FOREACH nm_final_gr GENERATE
    FLATTEN(group),
    SUM(tw_complete.pos) AS pos_tot,
    SUM(tw_complete.neg) AS neg_tot,
    SUM(tw_complete.unkn) AS unkn_tot,
    COUNT(tw_complete) AS time_total;
```

Listing 4.9: Concepts per minute calculation

This example makes use of many of the features of Squeal: replicated join, distributed join (with static data), streaming co-group, and information updates through positive/negative messages. The following section will discuss how well this calculation performs in micro benchmarks as well as a brief discussion of scaling issues when attempting to run this analysis on a cluster.

4.2.2 Performance

Figure 4.4 shows the logical flow of information through the NationMind analytic (the actual topology of spouts and bolts can be seen in figure 5.3). The nodes are colored by type: light blue for spouts, purple for map, red for combine, green for reduce. The edges are sized by the median latency between processing nodes and colored by blending the colors of the endpoints. The critical execution path follows the lower branch through the concept computation. Interestingly, the top (sentiment grouping) nodes are hosted using the same machines (and using the same network) which implies that the latency may be caused by some internal queueing or delay (given the difference between edge widths).

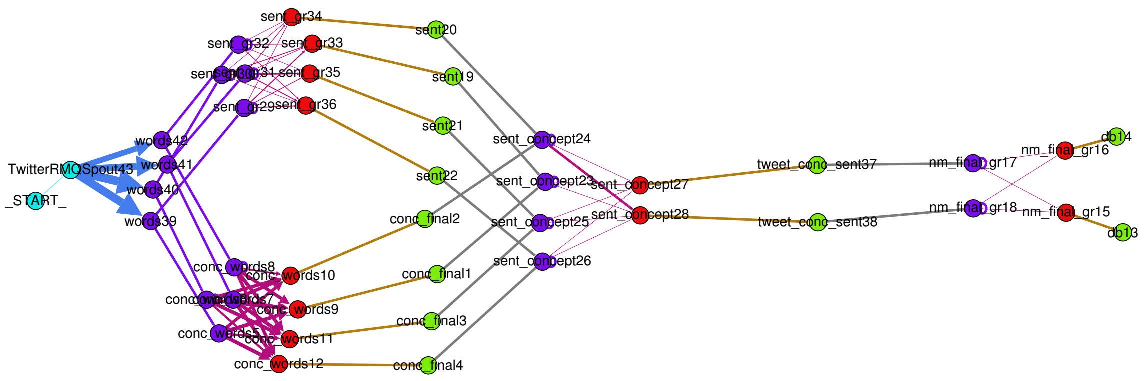


Figure 4.4: Storm topology for NationMind after optimization

The NationMind topology does not improve performance when running on multiple nodes. The performance peeks at 200 to 300 tweets per second. This corresponds to a load of around 135 thousands requests per second within HBase. The bottleneck for this computation lies in the concept and sentiment processing bolts. Using Storm's UI it was observed that one of the bolts was showing much more load than others. The notion of hot spots and stragglers can have significant impact on the runtime of MapReduce computations. Within stream processing the hot spot issue persists along with the notion of a micro-straggler[52].

The input distribution to these bottleneck nodes appears to be roughly uniform. The selectivity of the join is 0.58 (a little over half of the words produce concept hits). This is higher than those seen in small scale testing and may be producing noisy results (further grooming of the Wikipedia concept data would help). The output distribution of these bottleneck nodes is not uniform though each seems to be paired with a unique down stream process (there doesn't appear to be network pressure caused by all to one communication).

Further exploration and benchmarking would be necessary to attempt to address these issues. This work is not focused on establishing a new benchmark, so this avenue of work was not explored. One possible improvement would be the use of Bloom filters to prevent unnecessary transmissions and lower service demand during the replicated join. There is a Bloom filter UDF available in Pig which would function in a similar fashion as the map side joins seen in the word count example but this modification was not attempted. The UDF would need to be reworked if the NationMind analytic was modified to support dynamic concept updates. Optimizing the backing store for each specific operator is another avenue for improvement.

4.3 Issues Surrounding Batch Codes in a Streaming Environment

The main issue surrounding developing batch codes that later run in a streaming environment is that some assumptions and intuitions change when running in a streaming environment. The modification of NationMind to use an extra group by statement causes more work in a batch environment but cuts latency in half for streaming computations. Another case study in how intuition fails can be seen by examining the use of combiners in an expanded word count analytic. We modified our word count code to take the source (filter or sample) into consideration. The idea is that we could examine the difference in word ranks between two sources to detect events.

The basic strategy of the initial implementation was to ensure that combiners were applicable using the stock Pig functions (`SUM` and `COUNT`) at the cost of an extra MapReduce cycle. Figure 4.5 shows the topology generated by the original implementation of word count. As before, nodes are colored by type (purple for spouts, red for map, green for combine, blue for reduce) while edges are a blend of the source and destination colors and are weighted by latency of the link. Nodes are arranged vertically by the Storm worker each task was executing.

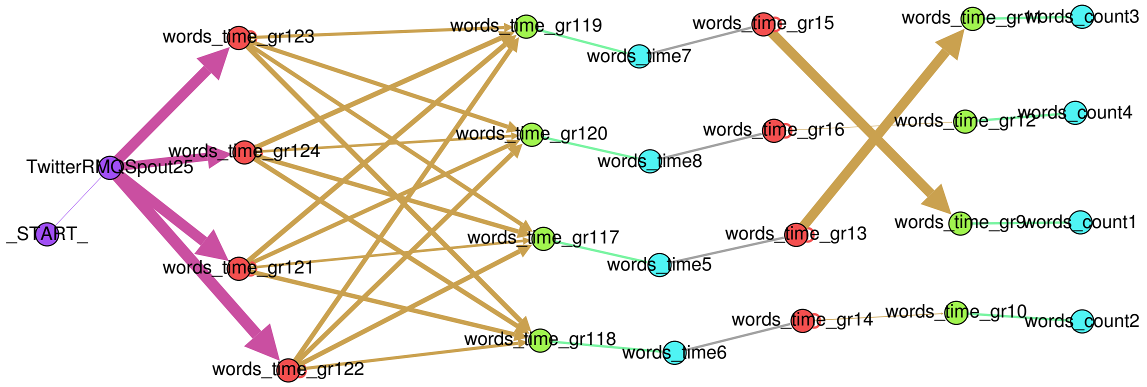


Figure 4.5: Storm topology for word count

The first MapReduce cycle groups by word, timestamp to the minute, and data source to calculate the word frequency. The second MapReduce cycle re-groups by word and timestamp to calculate a single record for word/minute. Note the imbalanced weights leaving and entering the top workers between the second map and reduce stage. The second grouping operation is being performed on a subset of the first (stripping off the source). The second from top and the bottom pathways show barely visible latencies – this is because the tuples don't leave the worker (the latency is just the queue time between threads in the same JVM). The mean latency for such transitions is 11.8 ms, with a standard deviation of 30.27 ms and a median of 2 ms. However, the top and the second from bottom pathways cross one another and provide significantly more latency with a mean of 69.6 ms, a standard deviation of 106.8 ms, and a median of 40 ms. The average latency of each batch of 250 tweets is 892.1 ms with a standard deviation of 623.1 ms and a median of 756 ms.

Figure 4.6 shows the topology after the word count script is modified to perform the calculation in a single MapReduce cycle without the use of combiners. This is achieved by grouping by word and timestamp to the minute, then filtering by source within the reduce operator and producing the merged count. The specific calculation is still combinable, but would require the implementation of a custom UDF. The nodes are positioned vertically by worker as before however, they are now colored by worker as well. The edges are weighted by latency as before and colored as a blend of the source and destination.

In this case, the latency for each batch of tweets was 594.3 ms with a standard deviation of 445.8 ms and a median of 480 ms. The median reduce execution time only increased by a few microseconds (4.1 to 6.9). While Pig Squeal provides lowers

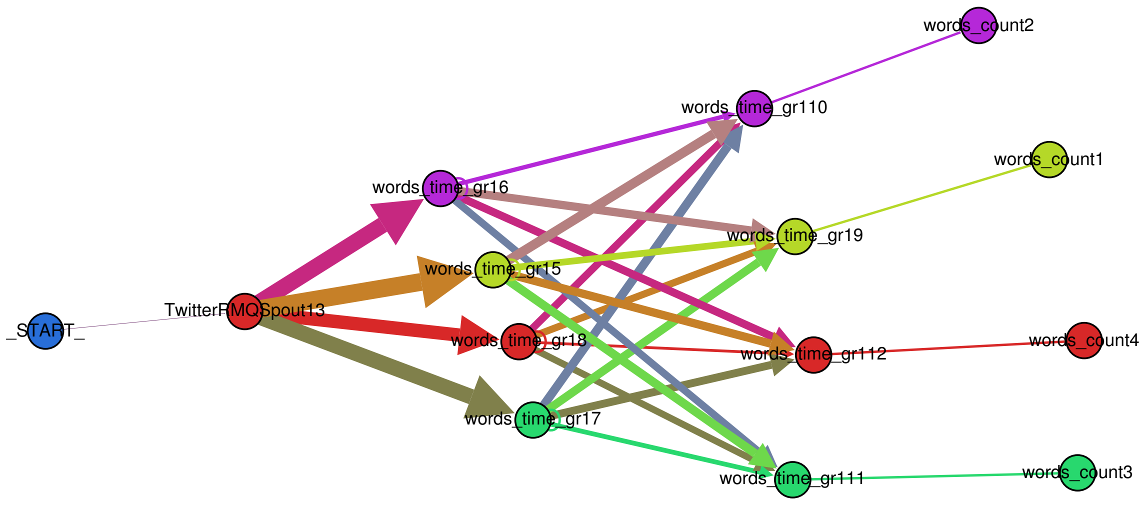


Figure 4.6: Storm topology for word count after optimization

the barriers between batch and streaming the environments are different enough to warrant additional analysis and tweaks to improve performance. The following chapter attempts to shed light on what contributes to the service time of these codes to provide better insight for optimizations.

Chapter 5

Performance Modeling

On Monday, October 29, 2012, Hurricane Sandy hit New York City flooding the streets and cutting power to parts of the city. Some argue that Hurricane Sandy significantly influenced the outcome of the 2012 Presidential Election. The first debate was rough for President Obama. Mitt Romney had just spent the past year on his toes, debating other Republicans for the chance to be the GOP candidate for President. In contrast, President Obama seemed out of character and off balance [29]. Though he was able to recover in later debates, it seemed he lost some momentum.

Figure 5.1 shows the tweets per hour for the weeks of October 2012 which happen to be the same time frame as the 2012 Presidential debates. You can see the spikes during each week corresponding to the various debates. What is even more significant is the night Sandy hit New York City – an event that would personally impact millions of people [73]. While the debates provided an opportunity for each candidate to take a stance on the various issues facing the nation, Sandy provided an opportunity for President Obama to act, well, presidential [37].

Now, contrast these rates with those seen in figure 5.2. This figure shows the average (plus and minus one standard deviation) number of tweets collected within specific hours of the week from our sample feeds. It is very easy to see the

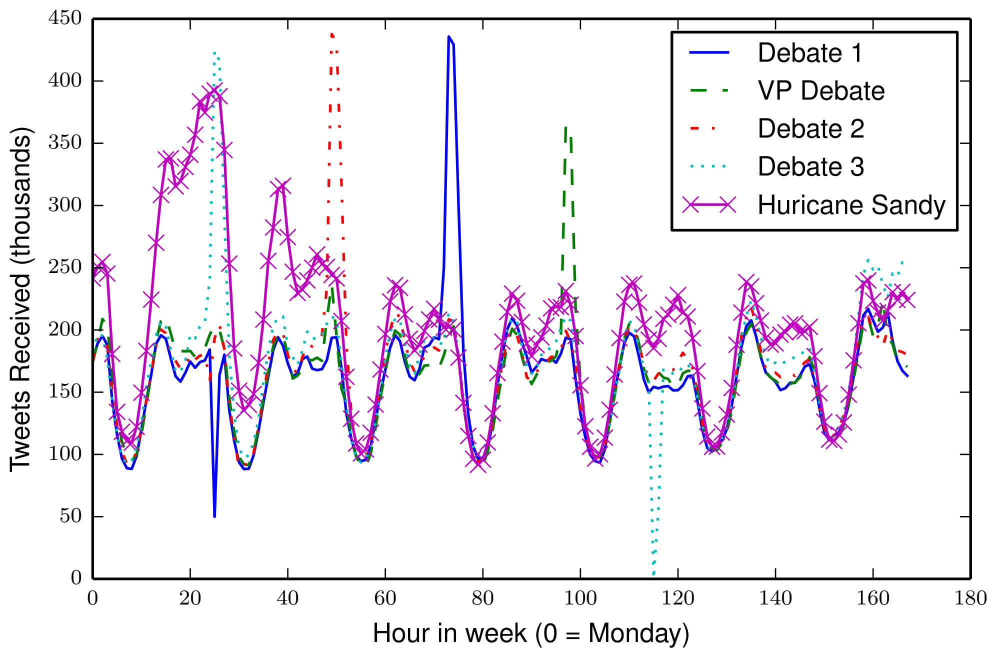


Figure 5.1: Comparing tweet rates of 2012 Presidential Debates and Hurricane Sandy

diurnal pattern of volume. You can see other trends – one looks like lunch time and another dip which may be dinner/commutes. Finally, we can see these secondary dips missing on the weekend along with a smaller dip in the night time volumes.

This data can be used to model the input rates of tweets at any given time. These rates can then be used with a performance model to understand if a particular calculation would run along with the resources necessary to consume the input. Furthermore, if you mix in a utility model (that is, what is the return on being able to process information in soft real time) you can make concessions. For instance, perhaps the demographic you're studying is only active at night – this would allow you to size your compute cluster significantly smaller than those necessary to process the feed in real time during the day. During the day you could allow messages to

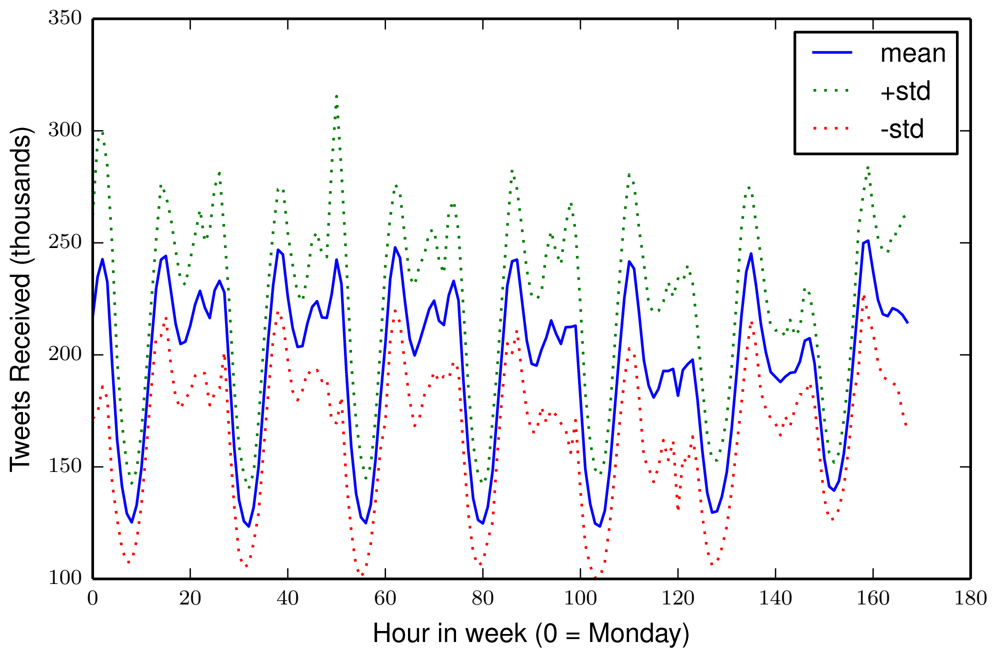


Figure 5.2: Expected tweets per hour

queue or load shed in some fashion (say with a message TTL).

This data also provides some insight into the slack necessary to handle burst traffic. As data starts to pass certain thresholds (say half a standard deviation) additional capacity could be spun up to address the potential burst. Again, consider figure 5.1 – the debate peaks are at worst an hour or four before returning to normal. Plenty of time to queue and recover as rates return to normal. If you were somehow sitting on some "I Love Big Bird" T-shirts, you may have lost some money by missing the sudden burst of interest. However, if you're in charge of Public Safety in New York City, you are facing significant volume for a long time during a Sandy-like event. Furthermore, the information within (reports of fires or transformer explosions) have a short shelf-life.

The aim of this chapter is to discuss a performance model for Squeal scripts.

The goal of this modeling is to understand why a particular calculation performs as it does and to then provide hints to get the best performance from these calculations. While tuning parameters such as number of workers or maximum outstanding messages can improve system utilization and response time these models also provide insights necessary to optimize the scripts themselves. This has certainly been true for the case studies from the previous chapter.

The following chapter will discuss the specifics of the way Trident structures computations and how that impacts modeling, this is followed by a discussion of a simple performance model, which is then followed by a detailed breakdown of the model components as they relate to word count. Finally, this model is checked against various traces of word count on a cluster which is followed by a discussions and chapter summary.

5.1 Trident Execution Algorithms

This section provides high level descriptions of the various components running in a Trident topology. Trident builds an execution graph starting from the spouts and adds a node for each operation (*each*, $filter$, *groupby*, etc). Much like Pig, Trident constructs each low level operator (Storm bolts) by chaining local operations and partitioning the graph along synchronizing operations (such as *groupby*). This removes unnecessary inter-bolt hops to perform operations that can be called directly in a chain. Trident also provides other features used by Squeal – specifically the transaction mechanism that provides at most once execution semantics and the state system that is used to track intermediate results. There are four types of nodes to consider in a Storm topology generated by Trident:

Coordinators – This includes the master and spout controllers.

Spouts – There are spouts in Trident-built topologies but they actually run in bolts.

Executors – These nodes run the part of the subgraph that can be pipelined at each point.

Acker – Exactly the same as a Storm topology.

Figure 5.3 shows the Storm topology for the NationMind case study. The "master" coordinator node is the only spout in the topology. It is responsible for starting, committing, and completing each transaction. There is a configuration option for the number of outstanding batches (which defaults to one). When a batch can be started, the master emits a message to notify the spouts to release tuples. All tuples emitted during a batch will be rooted to the batch id when tracking acknowledgements (which significantly reduces the overhead of the acker).

Each spout releases up to a maximum number of tuples, each linked by the batch id. For some reason, Trident does not chain operators directly off the spouts themselves. Oddly still, Trident uses a direct grouping to hop between the spout bolt (which may cause the tuple to travel between workers or machines). This detail cannot be seen in this particular graph and isn't really an issue in later cases because of the use of the "shuffle before" hint to spread this load.

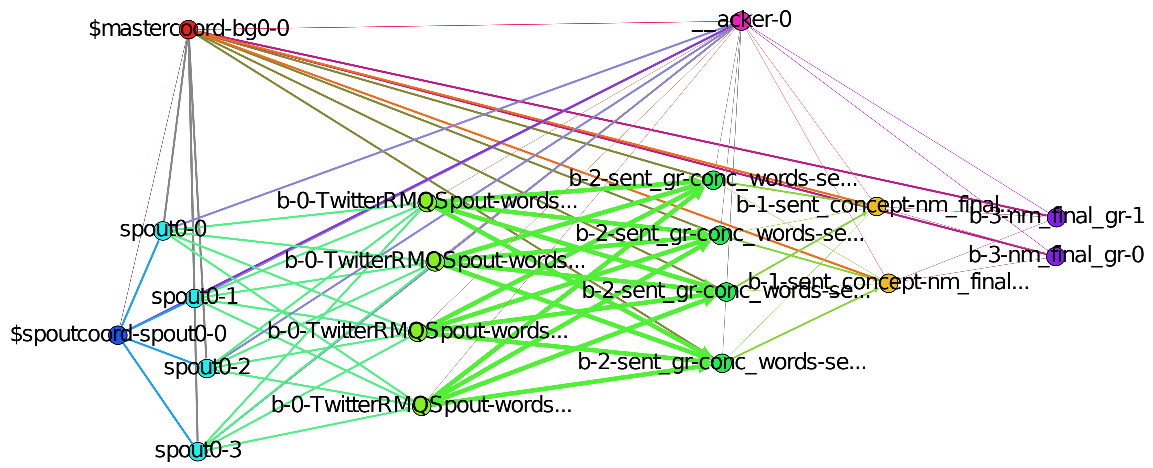

$mastercoord-bg0-0
__acker-0
spout0-0
spout0-1
$spoutcoord-spout0-0
spout0-2
spout0-3
b-0-TwitterRMQSpout-words...
b-0-TwitterRMQSpout-words...
b-0-TwitterRMQSpout-words...
b-0-TwitterRMQSpout-words...
b-2-sent_gr-conc_words-se...
b-2-sent_gr-conc_words-se...
b-2-sent_gr-conc_words-se...
b-2-sent_gr-conc_words-se...
b-1-sent_concept-nm_final...
b-1-sent_concept-nm_final...
b-3-nm_final_gr-1
b-3-nm_final_gr-0

Figure 5.3: Storm topology created from NationMind case study

The executors provide the most interesting features when modeling the performance of the topologies. It is hard to see in figure 5.3, but each bolt's name is the concatenation of the Pig aliases being calculated. Operator chaining can make it challenging to understand the performance of the specific map or reduce tasks being executed. Because of this, a more fine-grained transition graph is used to perform analysis (as we saw in figures 4.5 and 4.4).

The bolts directly following the spouts are all map operators. When a batch of tuples arrive from the spout, the map functions are applied immediately. If any of the follow-on operations are map operators, these functions are called directly in a chain including partitionging by key and partition combine. Once the subgraph of executions are complete the partition combines are completed. The combined results are then queued and transmitted across the network to the reduce nodes. Once all tuples have been processed for a particular batch, a *COORD* tuple is transmitting to successor nodes (if any). If there are no successor nodes, the first stage of a batch is acknowledged (which leads the master to enter the commit stage).

At successor (reduce) nodes, there is a different set of operations to consider. There are four stages of execution at a reduce node:

Accumulate any inbound messages are accumulated using a second-stage combiner.

Get/Update the state mechanism is activated to retrieve and manipulate the current batch of keys.

Put/Commit the updated values are committed to external storage.

Release/Reduce the reduce phase runs along with any follow-on map operators (in a pipeline).

In the accumulate phase each batch of tuples is partitioned by key and passed to another combiner immediately. Once the $COORD$ message has been received from all preceding nodes (or a $COMMIT$ message for the last executor in the graph), the keys from all partitions are retrieved using the multi-get mechanism from the Trident state. Once all keys are retrieved, the updates are applied to the existing state (in memory). In Squeal the states (with *cur* and *last*) are passed to the $TriReduceDelta$ function to process the update and produce delta messages.

This leads to the most significant observation of this chapter: Squeal topologies executing in Trident are basically running as chains of mini MapReduce operations. For the example in figure 5.3, the only active processes at a given time consist of a series of bipartite subgraphs. This is similar to the coordination between producers and consumers seen in Naiad [52]. In the first stage, the `b-0` nodes will execute and emit tuples; the `b-2` nodes will accumulate and combine these results until all `b-0` nodes have exhausted their work for the current transaction. Then, after the `b-2` nodes execute the get/update phase (which is blocking), they will enter into the release phase. Once in the release phase, the bipartite subgraph of `b-2` and `b-1` nodes will be producing and accumulating tuples. This continues until the `b-1`/`b-3` nodes complete up to the end of the accumulate phase.

This observation has bearings on many different aspects of execution and modeling. For instance, this knowledge could be used by the Storm scheduler to exploit the execution affinity of various groups of nodes: adjacent nodes have the potential to be active at the same time whereas non-adjacent nodes will never be active at the same time (if only one transaction is allowed to run at once). Also, this allows us to break the performance model into disjoint parts that only impact each other by influencing the release time of tuples within various stages.

Another interesting point is the effect of final reduce on external side effects. The last nodes in the topology only enter the release phase during the $COMMIT$ phase of a transaction. When a transaction starts the $COMMIT$ phase, each node performs PUT operations to the backing store. Only when the $COMMIT$ message propagates to the final nodes in the graph, `b-3`, will they enter the final two phases. Because of this, we know that any side effects (e.g. $STORE$) for a Pig script in Squeal will only transmit as a part of the $COMMIT$ phase of a transaction.

The final point to consider when analyzing Trident-built topologies is that the transition doesn't complete until all of the released tuples complete. This allows us to focus our modeling efforts on the critical execution path – that which takes the most time. One interesting artifact of the determinism of MapReduce is that the trajectory of a tuple and it's dependents is fixed after the initial shuffle phase (if no "shuffle before" hints are used within the topology). This has implications when considering the minimum response time possible for a particular calculation.

5.2 Performance Model

To create a performance model for Pig scripts running in Storm, we will make use of many of the conventions from queuing theory. Specifically, we will approximate the bolts within Storm using M/M/m queues. The first M stands for Markovian (as opposed to G for General, E for Erlang, etc.) which means that the input can be modeled as a memoryless Poisson process with average arrival rate λ. The second M also stands for Markovian, which means that the m processors consuming items from the queue can be modeled as a memoryless Poisson process with mean service rate of μ. When the traffic intensity defined by $\rho = \lambda/(m\mu)$ is less than one, the M/M/m queue is said to be stable. Stable M/M/m queues are well understood [70, 41] and have closed-form solutions for many properties (expected response time, queue size, utilization, etc.). When the traffic intensity is greater than one, the queue becomes unstable and will grow without bounds.

There are many factors of the Trident execution model that cause divergence from the standard models. While data arrives at the edge of the topology at some average rate, the information is released into the topology in batches (also, a well studied topic [41, 42]) at configured intervals (defaulting at 500 ms). Queueing theory is generally concerned with the steady state of a system while Trident undergoes ramp up (where ρ may be greater than one), steady state, and ramp down (where λ goes to zero) phases every time a batch is released. Because a batch of data isn't marked as complete until all the elements within are finished, we are more concerned with the total time necessary to process a batch (as opposed to the response time of a particular "tagged" job). This presents numerous problems, specifically careful tracking of which metrics are run in parallel versus serially and also which metrics are overlapping.

Because Squeal builds topologies based on the MapReduce plan of a Pig script and because Trident has gating mechanisms around the persistence layer, a Squeal topology can be modeled as pipelines of independent executions that have five distinct phases:

Map/Stage zero (S0) Combine – tuples flow from the spout into map operators for processing, partitioned by key and combined.

Shuffle/Stage one (S1) Combine – Stage zero combine completes and tuples are transmitted to "reducers" where they undergo stage 1 combine.

Get – previous values for the current key set are retrieved or initialized.

Put – updates are stored (during the batch commit phase).

Reduce/(Map) the reduce operators are run and chained to the next map phase (if any).

The first map tasks receive their tuples from the Storm Spout, apply the map function and partition the output by key. At the map task, these keys undergo stage zero combine which performs initialization of values and combination of any values at the mapper that share the same key. When the spout has released all tuples for the current batch, the stage zero combiners are completed and the shuffle phase begins where tuples are transmitted to the reduce nodes.

At the reduce nodes, incoming tuples are grouped by key and combined. Once all tuples from the map phase have been transmitted, the stage 1 combiners are completed. Each reduce node then activates the get phase which pulls previous values from the state mechanism. Though all keys are retrieved at once, no further processing is performed until all values are available (which reduces parallelism).

After the current values are combined by key with the previous values, the state mechanism is updated. The actual order of this operation occurs after the batch has successfully finished the first stage of a two-stage commit. If there is only a single MapReduce operation, it will occur before the reduce phase. Otherwise, all the put phases (before the final MapReduce operation) will be executed during the batch commit phase. The final reduce operation executes upon a successful commit. The specific order of these events isn't important because they are non-overlapping and merely contribute to the overall batch elapsed time.

For each phase of computation, the contribution to the total batch time can be described as functions that *could* depend on the following:

$Map(\cdots)$ - Map elapsed time:

- μ_{map} - the service rate of the specific map calculation.
- λ_{spout} - the output rate of the spout.

- m_{map} - the parallelism (number of tasks) of the map operator.
- bs - the number of outstanding tuples per batch (the batch size).

$CS1(\cdots)$ - Combine Stage 1 elapsed time:

- μ_{cS0} - the service rate of the map-side combiner.
- μ_{cS1} - the service rate of the reduce-side combiner.
- $\lambda_{shuffle}$ - the effective rate of transmission from map to reduce.
- m_{map} - the number of map tasks.
- m_{reduce} - the number of reduce tasks.
- $emit_{map}$ - the number of records emitted by map.
- $emit_{cS0}$ - the number of records to survive the combiner.
- $dist_{cS0}$ - the distribution of keys (how many values are transmitted to each individual reduces – important if there is a skewed distribution of keys).

$Get(\cdots)$ - Bulk key/value retrieval elapsed time:

- μ_{get} - the service rate of retrievals (on cache failure).
- $emit_{cS1}$ - the number of keys to be retrieved.
- $cache\_rate$ - the effectiveness of the storage cache.
- $dist_{cS1}$ - the distribution of keys (which may influence storage performance).

$Put(\cdots)$ - Bulk key/value storage elapsed time:

- μ_{put} - the service rate of stores.
- $emit_{cS1}$ - the number of keys to be stored.
- $dist_{cS1}$ - the distribution of keys.

$Reduce(\cdots)$ - Reduce elapsed time:

- μ_{reduce} - the service rate of the specific reduce calculation.
- m_{reduce} - the parallelism of the reduce operator.
- $emit_{cS1}$ - the number of keys from the combine stage.
- $dist_{cS1}$ - the distribution of keys.
- $emit_{reduce}$ - the number of keys emmitted by reduce.
- $\mu_{map\ldots}$ - the service rate of any follow on map operations (if any).

It is important to note that there are only two λ's to consider in a general model for squeal scripts: λ_{spout} and $\lambda_{shuffle}$. After the first stage of execution, there is only $\lambda_{shuffle}$ because follow on maps are chained directly off of reduce. Each placeholder variable from the list above may represent a parameter to a distribution (such as the service rate μ's) or may be a function of other variables (such as the number of values produced by the map stage which is a function of the batch size: $emit_{map}(bs)$). Finally, some of these variables are directly observable (such as the service times of the various elements) whereas some can only be measured indirectly (such as $\lambda_{shuffle}$).

A complete model would allow for many interesting questions to be answered, such as: what is the performance of a specific topology as we vary the batch size, the parallelisms, or the number of outstanding batches allowed in the system (in increasing order of complexity). Unfortunately, a complete model of the system has proven elusive for the amount of time budgeted for this part of this dissertation's work. Instead, we present a simple model that attempts to reproduce the batch latencies from trace data. While not as useful as a more sophisticated model, it still provides insights into many interesting aspects of system performance: the ability to understand what parts of the execution are taking up the most amounts of times (which is important for optimization). Furthermore, it provides a basis for comparing the trade offs of various tuning parameters (such as batch size and parallelism) by ocmparing separate execution traces.

Let $Exp(x)$ represent an independent exponentially distributed random variable with rate x, then the simplified model is the sum of the following components:

$$Map(\mu_{map}, bs, m_{map}) = \frac{1}{m_{map}} \sum^{bs} Exp(\mu_{map})$$

$$emit_{map} = emit_{cS1} = \sum^{bs} Exp(\lambda_{mapEmit})$$

$$CS1(emit_{map}, \lambda_{shuffle}) = emit_{map} \cdot Exp(\lambda_{shuffle})$$

$$Get(\mu_{get}) = Exp(\mu_{get})$$

$$Put(\mu_{put}) = Exp(\mu_{put})$$

$$Reduce(\mu_{reduce}, emit_{cS1}, m_{reduce}) = \frac{1}{m_{reduce}} \sum^{emit_{cS1}} Exp(\mu_{reduce})$$

The following section will explain each component in greater detail and explore how effective it is in modeling our simple word count example.

5.3 Experiment: Word Count

This section outlines an experiment to test the efficacy of the simple performance model in simulating batch elapsed time using trace data from running topologies. The experiment is also designed to measure the strong scalability of the word count analytic as more resources are added to the topologies. Strong scaling is measured by holding the total work constant while varying the number of processors. The work load in this case is a set of 53 thousand tweets fed through a single spout with a batch size of 50. This rate is significantly smaller than what would be expected in a production run. Even the configuration for single tasks for map and reduce ($m = 1$) which is spread across multiple machines is non-ideal because there are sufficient resources on one node to handle the whole topology. The goal of these configurations was to get a "clean room" measurement of each of the various components of the model and not to run load tests (as seen in the previous chapter).

This experiment was run on a 10-node `m3.xlarge` cluster running in Amazon's Elastic Compute Cloud (EC2) over the course of a day. The `m3.xlarge` instance types has four vCPUs, 15 GB of memory, and two 40 GB SSD drives. Each test run was execute three times to account for variance caused by executing in a shared environment[1]. While there are many shortcomings of running such tests in such an uncontrolled environment, we believe that experiments run in EC2 provides a much more realistic result set than those measured in controlled lab environments.

The cluster was loaded using the Ambari management environment[2] (version 1.6.1 with storm version 0.9.1). All nodes were configured as potential work hosts (without standalone configurations of controller nodes such as Nimbus, the Hadoop Name Node, etc.). Once the topologies have been initialized, the workload on the controllers is negligible. Data was fed into the system using RabbitMQ. When a topology was started, it was allowed to initialize and run for a few seconds before the data was fed into RabbitMQ at maximum rate. Because the input rate far exceeded the service rate of the topologies, the messages were queued in memory until the topology was able to consume them. Once the work was completed, the topology was destroyed and the system was reset for the next run.

There were five treatments, each repeated three times producing 15 separate traces to analyze. For each treatment, the number of map and reduce tasks (which we call the parallelism or m) was set to the same value ($m = m_{map} = m_{reduce}$) and increased across treatments ($m = \{1, 2, 4, 8, 16\}$). HBase was used to provide the state management for the reduce phase. For each test, the number of resources (in the form of region servers) was kept constant to control storage throughput between experiments. The backing tables were pre-split based on the distribution of frequencies of the letters of the latin alphabet seen in the Wikipedia concept data.

[1]http://hbase.apache.org/book/perf.ec2.html
[2]http://ambari.apache.org/

This led to fairly uniform load across the regions except for the first and last region which carried the majority of load associated with numbers and non-latin characters. Because the input data was the same for each experiment, this imbalance effected each experiment in the same way.

For each experiment, the first minute of batches was discarded to allow the topology to complete initialization. The startup time for the word count example is greatly effected by loading the stop lists from HDFS. This occurs the first time map is called with input. There are other factors such as establishing connections to the state mechanism and the message queue system. There is also load on the queueing system while the data is being loaded into memory.

Storm has a metrics aggregation and collection mechanism that is used to populate statistics in the UI. However, to increase the granuality we opted to also collect metrics within the Squeal components themselves. One of the reasons for doing so was to get better precision in the time measurments (Storm tracks at milliseconds resolution). For tracking time within a function call (map, reduce, get, put) we made use of Java's `System.nanoTime()` function which prevides higher precision measurements from subsequent calls. There is no documentation on the precision of these measurements but most of the measurements presented here are greater than one microsecond. These metrics were transmitted across the network to the head node of the cluster (which was also running RabbitMQ). While this network traffic influenced previous measurments under load, it was not seen as a significant load for these experiments to influence the message bus. Each component measured was sampled at a five percent rate except for the batch emission and acknowledgements which were tracked at 100% rate.

Figure 5.4 shows the batch elapsed milliseconds response time for all treatments (grouped by parallelism). The same information is also summarized in table 5.1. The most significant decreases in response time can be seen when the number of workers increases from one to two and then from two to four. At a parallelism of four, we have eight tasks running on ten machines (which each runs two Storm worker processes). As we increase to a parallelism of eight, we have sixteen tasks running on ten machines – while there is machine sharing, there are still some idle worker processes. Finally, at a parallelism of 16, we have 32 tasks running on 20 CPUs – this case has both processor sharing and worker sharing in play.

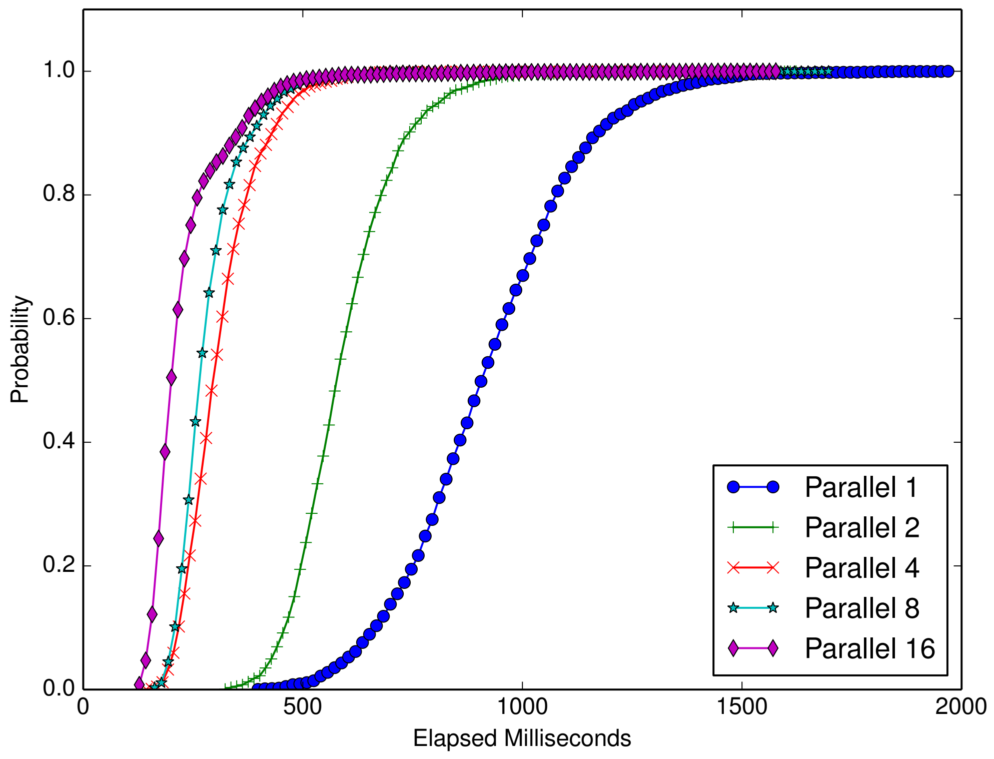


Figure 5.4: CDF of Batch Elapsed Milliseconds for varying parallelism

While examining table 5.1 quite a few changes are apparent as we increase available resources. First, there are more batches summarized for parallelisms of one and two. This can be explained by the fact that each of these configurations take longer to process each batch and thus have more surviving batches when the first minute of execution is discarded. Once a batch is processed in less than 500 ms, the system will be limited by the timer function for Trident's batch release mechanism which fires twice a second. This limits the total system throughput to 100 tweets per second at parallelism of four and above. We can also see that the variance drops until we reach a parallelism of eight and increases slightly when we double the parallelism to sixteen.

Table 5.1: Statistics for various parallelisms

		Batch response time in milliseconds						
m	Count	Mean	Std	Min	25%	50%	75%	Max
1	3,020	920	200	380	780	910	1,050	1,970
2	2,950	590	120	310	510	580	660	1,620
4	2,870	310	90	130	250	300	350	1,370
8	2,880	280	80	150	230	260	310	1,700
16	2,860	230	100	110	170	200	240	1,580

Figure 5.6 shows the contribution of each phase to the total execution time. This plot was generated by taking each class of event and grouping by their batch. The batch start time was then used to produce a batch-relative timestamp for each event and we can see the summary of the first and last executions of each event type. So, on the left we see the first stage zero (S0) combine happens directly after the first map. Next, we can see that the first map occurs nearly instantaneously when the batch starts with some delay in the slowest case. This is followed very

quickly by the last map execution (which ends around 100 ms into the batch).

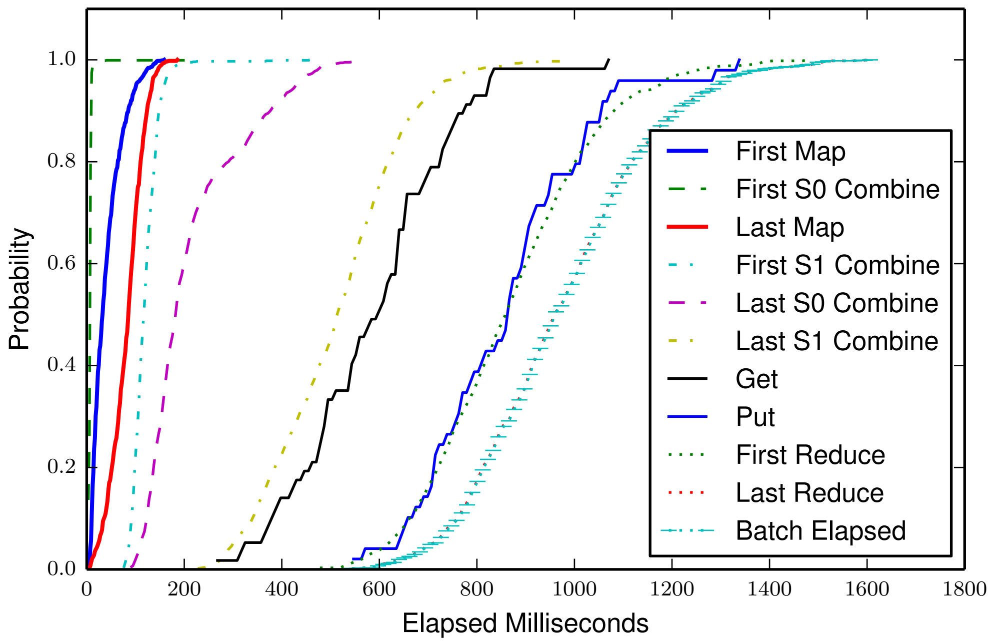


Figure 5.5: CDF of Batch Relative Start and Stop Times for Computation Stages for m=1

As the map phase finishes, the S0 combines begin to complete and transmit messages to the reduce nodes. This allows the first stage one (S1) combines to start executing. The span between the start and finish of the S1 combine is quite significant. The combine operation executes in around 14 microseconds on average, with an average map emission of 2,200 words this only leads to 61 milliseconds of CPU demand (half in stage zero, half in stage one) whereas the gap between the first and last combine is 395 ms (accounting for a third of the average batch elapsed time). The missing time is caused by the latency of moving the tuples between the nodes and presents many modeling challenges which will be discussed in a later section.

Once the final combine has executed, the get phase starts to retrieve previous states from the backing store. When the get phase completes, the batch starts the commit phase. This allows the put phase to update the state values before allowing reduce to produce side effects. Finally, the reduce phase executes – the completion of the reduce phase marks the end of the batch.

When comparing the recorded CPU time from the various components to those reported by Storm, we can account for 95 percent of Storm's recorded CPU time. The missing five percent occurs in the coordination bolts (the master coordinator, the acker) and in the spout bolts (which also perform de-serialization of input to Pig types). While we have most of the CPU time covered, there is a significant amount of batch time spent in queues between nodes. Getting direct measurements of the various queues and sub-systems that perform this functionality is challenging for various reasons (mainly due to API layering and datastructure design [65] which is designed for high performance and not easy to monitor). However, the latencies they induce is measured by tracking the absolute difference between the clocks of the Squeal components. While the machines are synchronized with NTP, there is still significant noise in these measurements.

The following sections will describe the measurements of each phase of execution along with how they are modeled. This is followed by a section discussing the results when these models are combined to simulate batch elapsed times.

5.3.1 Modeling Map

The map phase of word count takes the JSON representation of a tweet and extracts word bigrams. Because this map routine treats each tweet atomically and has no accumulation of state (aside from the initial stop list load), it is an ideal candidate to be modeled as a Poisson process. Table 5.2 shows statistics for each of the treatments. The left side of the table summarizes the service demand necessary for each tweet to be processed. We can see that for $m = 1$ the tweets take around 2.2 ms on average to process. The p-value is produced by applying Welch's T-test [3] to check the hypothesis that two populations have equal means. Welch's variation of the T-test is used when the samples have different variances. The first sample in this case is the measured elapsed time recorded from each trace. The second sample is created using an exponential distribution fit to the first sample. In effect, this gives us some notion of how well the exponential distribution models the data.

The right side of table 5.2 shows information on how many records are emitted by the bi-gram extraction process. If you recall, for each word extracted five records are produced to generate a decaying score for the word over the timespan of five minutes. This means that while there is an average of 43 records produced by the map phase, this means that we see an average of 8.6 words per tweet. The variance we see between runs can be explained by the five percent sampling which would pick up different individual tweets to sample between runs with the same parallelism. The exponential distribution was found to model this data with high p-values from Welch's T-Test.

[3]http://en.wikipedia.org/wiki/Welch%27s_t_test

Table 5.2: Map Statistics and Fitness Measures

		Service Time in Milliseconds				Positive Message Output			
m	Trace	Mean	Std	Med	p-value	Mean	Std	Med	p-value
1	1	2.159	2.993	1.625	0.254	42.671	36.722	35	0.892
	2	2.175	3.144	1.560	0.981	41.673	36.036	35	0.948
	3	2.199	3.167	1.614	0.770	43.559	36.822	35	0.718
2	1	2.392	3.306	1.678	0.626	43.294	37.872	35	0.991
	2	2.417	3.194	1.736	0.355	44.650	37.379	35	0.871
	3	2.331	2.777	1.722	0.588	43.136	36.652	35	0.975
4	1	2.493	2.830	1.771	0.805	43.717	37.703	35	0.138
	2	2.458	2.612	1.802	0.666	43.036	36.773	35	0.192
	3	2.487	3.842	1.807	0.698	43.386	36.708	35	0.568
8	1	3.587	19.606	2.077	0.890	41.968	35.871	35	0.566
	2	3.227	4.324	2.103	0.765	41.882	36.378	35	0.380
	3	4.422	43.058	2.099	0.791	43.486	37.375	35	0.808
16	1	4.797	9.474	2.651	0.844	42.540	36.349	35	0.481
	2	4.907	16.788	2.495	0.645	41.748	36.322	35	0.316
	3	5.394	17.088	2.683	0.675	42.909	36.888	35	0.640

Before we move on to the next phase it is worth pointing out the upward trend in service times as parallelism increases. While the individual map tasks don't interact with one another, they are using the same cluster resources as things such as HBase. In higher parallelisms they may also share a machine, core, or JVM (specifically the garbage collector). This has significance for any generalized model of execution (where you would assume μ_{map} is constant). Further experimentation would be necessary to understand this phenomenon. One potential avenue would be to examine the mean service demand versus the server load as reported by the operating system. This anomaly does not impact our study of the simple model because we only examine pairs of runs with the same parallelism.

Now, revisiting the formulas that relate to map:

$$Map(\mu_{map}, bs, m_{map}) = \frac{1}{m_{map}} \sum^{bs} Exp(\mu_{map})$$

$$emit_{map} = emit_{cS1} = \sum^{bs} Exp(\lambda_{mapEmit})$$

We treat the elapsed time for the map phase as a simple division of labor across the mappers uniformly. This has weaknesses if the distribution of input counts is too far from uniform (which may be the case following a reduce operator). The random shuffle in Storm from the spout to the mappers was sufficiently normal for this to hold in this case. The number of emissions from map is modeled as the sum of an exponential distribution. The reason for setting the number of combine emissions to the save value as the map emissions is explained in the reduce subsection.

5.3.2 Modeling Combine

The next phase is the shuffle/combine phase. Figure 5.6 provides a detailed view of the executions during the shuffle phase. This particular plot is for a pair of bolts hooked up in tandem across the network. Focusing on stage zero, we can see that all of the executions occur before 200 ms and that most occur around 100 ms (which is the average service time for 50 tweets). After the maps complete, we see the stage zero completes begin – at this point the accumulators are finalizing results before transmitting across the network. Next we see the stage 1 accumulators start to process the incoming data (on the reduce node). Note that the peak frequency is limited around 300 messages per millisecond and that the span of the curve is much longer. This is caused by throughput capacity and queueing in the networking subsystem. Once all messages have been transmitted from the map phase the stage 1 combiners start to complete.

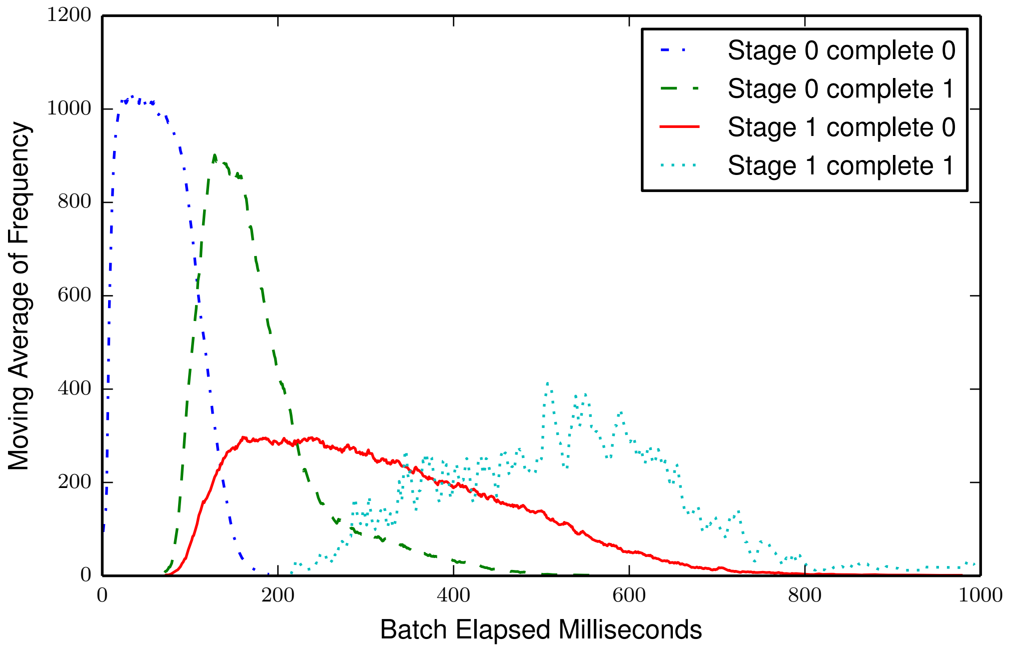


Figure 5.6: Count of Combine Events for m=1

Figure 5.7 shows the same plot for an experiment with $m = 8$. We can see the impact of additional resources in the peak rates of stage zero combines. The first spike represents the combine calls chained to the map operator. The second spike represents completion and message release. In this case, the stage one combine begins almost immediately and has very little throughput and queuing effects. The second spike occurs once the $COORD$ message arrives from all the mappers that they are complete.

One important feature of both of these plots is that the map executions and

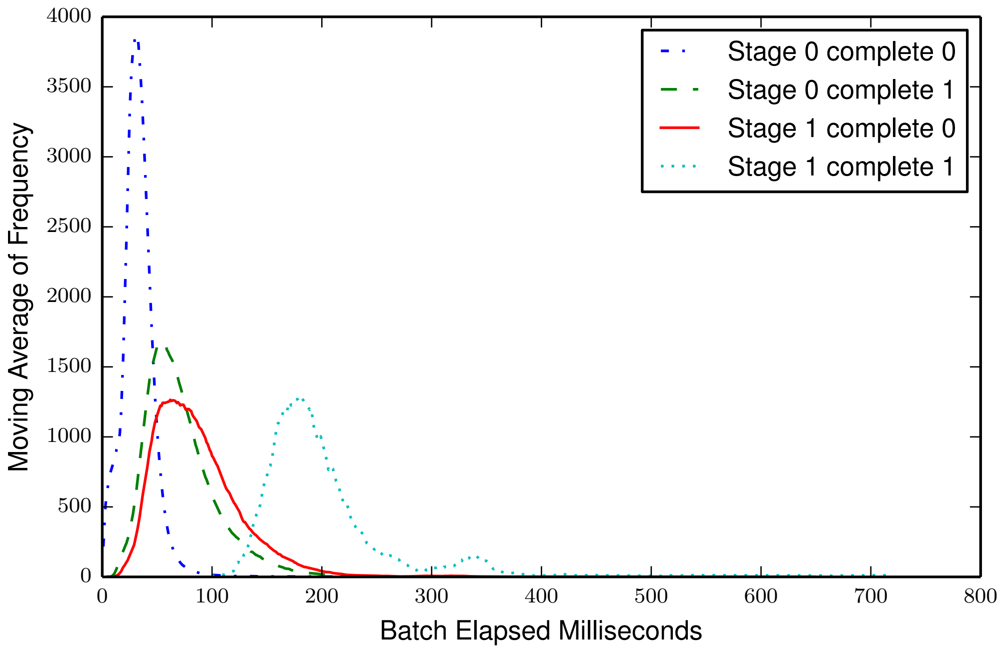


Figure 5.7: Count of Combine Events for m=8

the stage 1 completions do not overlap and are adjacent. Because of this, we can model each independently without concern of the multi-processing and network effects. This provides a very clean formula for the simplified model, but presents other limitations. Consider table 5.3: on the left we have the mean execution rate if we model the stage 1 execution as independent executions against a queue of items using the exponential distribution. The right side examines the time spans of the stage 1 execution as if it were one large job (in this case, modeled using the beta distribution).

First, we can see that the time spans and average rates do not decrease linearly with the increase in resources. There are numerous effects discarded from the model that may account for this discrepancy. Chief among them is a model of the networking and task dispatch subsystems. This system has presented modeling issues due

Table 5.3: Combine Statistics and Fitness Measures

		Mean Service Time in ms				Span Time in ms			
m	Trace	Mean	Std	Med	p-value	Mean	Std	Med	p-value
1	1	0.199	0.066	0.201	0.930	405.219	134.398	406	0.635
	2	0.192	0.065	0.195	0.626	391.141	132.275	391	0.860
	3	0.193	0.063	0.197	0.509	395.346	131.610	400	0.996
2	1	0.179	0.049	0.171	0.525	364.223	103.037	347	0.670
	2	0.126	0.038	0.120	0.372	255.620	78.545	243	0.416
	3	0.177	0.052	0.168	0.629	362.038	104.146	346	0.492
4	1	0.105	0.043	0.096	0.837	211.883	83.258	196	0.704
	2	0.079	0.029	0.072	0.565	160.050	57.490	146	0.838
	3	0.091	0.035	0.082	0.748	184.448	73.546	167	0.997
8	1	0.093	0.044	0.083	0.976	189.087	85.477	168	0.818
	2	0.096	0.045	0.085	0.960	194.297	89.185	171	0.541
	3	0.090	0.051	0.080	0.566	182.492	91.806	163	0.035
16	1	0.081	0.056	0.065	0.743	163.920	110.139	131	0.709
	2	0.080	0.055	0.066	0.357	161.300	105.251	133	0.536
	3	0.081	0.052	0.067	0.673	163.771	98.168	136	0.756

to complexities surrounding message bundling between hosts and lack of visibility into the underlying queues. The closest we came to being able to track the impact of these systems is through tagging message between the hosts to track absolute time differences between release and receive. A working network model is key to answering questions about the impact of increasing or decreasing specific resources in a topology (such as having different values for m_{map} and m_{reduce}). However, such a model has proven to be elusive and unobtainable in the time budgeted for such efforts.

The resulting equation is as follows:

$$CS1(emit_{map}, \lambda_{shuffle}) = emit_{map} \cdot Exp(\lambda_{shuffle})$$

The stage one execution time is actually influenced by the effectiveness of the stage zero emissions but because the batch size was so small it had very little effect (as discussed in the following section on reduce). Because $\lambda_{shuffle}$ was estimated by dividing the time span of the stage one combine divided by the number of values we decided to treat it as a service rate. If you switch over to using the value as a service demand and sum the times expected it does not work as a good estimator of the time span. Most of the model design was spent in discovering and attempting to further analyze this section of the execution time. Treating this element as a more traditional (unobserved) M/M/m queue was quite problematic due to lack of direct visibility. After estimating parameters for the hidden queue it was found to overestimate the span times significantly. This element of the model is a prime candidate for future research – solving this element would provide the necessary components to perform much more general analysis and prediction.

5.3.3 Modeling State Management

The Trident state management system allows stream processing systems to benefit from off-line storage. After the stage 1 combiner completes, we have new values to be merged with the previously known values. Trident dispatches a bulk pull for the values associated with the currently active keys. A simple LRU cache can be used to decrease the pressure on the backing storage system. During these experiments, the cache hit rate depended on the parallelism (more workers meant more cache space). For $m = 1$ the cache hit rate was around 1.7% for all keys and 5.1% for keys that had values. That is, when a get issued, it missed 98.4% of the time. However, most retrievals returned no results – for keys that had values, the miss rate was 94.9%. For $m = 16$ the cache hit rate was around 15.6% for all keys and 47% for keys that had values. This means that the cache missed 84.4% of the time for all keys but only 53% of the time for keys that had values. The simple model disregards the cache rate because for word count it isn't viewed as a significant factor.

Table 5.4 shows the service demands for get and put across the various treatments. The times in this table represent the amount of time spent retrieving and storing intermediate results per batch. The times are spent serially at each reduce task (roughly 60 ms spent per batch retrieving data for $m = 1$ and only 10 ms spent each batch for $m = 16$). The difference in time can be explained by the number of keys per retrieval for each which can be seen in the right side of table 5.5. When $m = 1$ all keys for a specific batch are retrieved and stored from a single machine. When $m = 16$ the keys are spread across the reduce tasks and executed in parallel. With more workers, the system is able to use more of HBase's bandwidth and increase the parallelism of the process because the per-node retrieval time is much shorter.

For the simple model, we draw a single value to represent the time necessary to complete this phase of execution. A more correct function would be to draw m exponents and take the max, but the time contribution wasn't seen as that significant to warrant further investigation. Furthermore, the specific model for storage and retrieval will vary significantly based on the underlying storage implementation.

Table 5.4: Get/Put State Statistics and Fitness Measures

		GET Times in Milliseconds				PUT Times in Milliseconds			
m	Trace	Mean	Std	Med	p-value	Mean	Std	Med	p-value
1	1	61.948	69.186	47.002	0.969	286.653	40.717	284.821	0.785
	2	59.417	32.542	46.036	0.934	93.186	52.987	84.466	0.972
	3	78.454	98.021	45.029	0.942	287.778	49.253	273.531	0.904
2	1	34.506	39.972	25.357	0.897	64.937	74.891	49.072	0.981
	2	25.085	11.773	22.975	0.863	105.403	61.573	118.773	0.671
	3	30.293	18.844	25.371	0.842	78.545	35.577	82.136	0.877
4	1	18.954	23.355	14.248	0.950	46.438	44.055	29.132	0.718
	2	16.937	23.232	13.480	0.912	31.434	20.889	25.841	0.915
	3	15.334	9.469	12.910	0.769	40.132	82.411	27.812	0.941
8	1	10.391	9.513	8.503	0.837	32.999	27.771	28.102	0.850
	2	12.712	50.987	8.811	0.944	33.716	28.112	28.987	0.564
	3	10.519	13.464	8.458	0.918	32.359	33.058	19.796	0.277
16	1	9.878	8.178	8.075	0.052	29.654	45.977	26.209	0.941
	2	9.115	15.033	7.417	0.884	25.169	38.174	19.117	0.817
	3	9.624	12.200	7.713	0.650	26.947	42.435	22.087	0.998

There are many different clustered storage mechanisms available, we chose HBase because it came pre-installed with an Ambari setup. There are many studies that consider the performance of other clustered storage mechanisms [1]. Provided they can be wrapped in a key/value interface they should allow for drop in replacement.

The formulas for get and put times are as follows:

$$Get(\mu_{get}) = Exp(\mu_{get})$$

$$Put(\mu_{put}) = Exp(\mu_{put})$$

These could be easily generalized to include the distribution of keys and counts to allow for more general analysis (across parallelisms and cases where

$m_{map} \neq m_{reduce}$). This would be done by modeling the effect of key size on storage performance as evident in the variation of mean service time across treatments.

5.3.4 Modeling Reduce

Reduce in word count can be modeled by a simple Poisson process using an exponential distribution. Table 5.5 shows the service times along with p-values for exponential distributions fit to the trace data. On the right side we see summaries for the number of keys put into the storage mechanism at each node. In a more general calculation, the storage key count provides a good estimate of the number of keys surviving the combine phase. For word count, the combine phase didn't produce a significant cut down on the number of keys (consider map produces 2,200 records on average while the storage mechanism manipulated 2,040 records on average).

For the simplified model, we set the number of executions to the number of map output records. This over-estimates the reduce time, but doesn't appear to impact the results for such a small batch size. The amount of time spent in reduce is small relative to the amount of time spent during the stage one combine. As the batch size increases, the combine would become more effective in decreasing the number of records bound for reduce. This would impact the combine stage one span as well as the reduce. This results in the following formula:

$$Reduce(\mu_{reduce}, emit_{cS1}, m_{reduce}) = \frac{1}{m_{reduce}} \sum^{emit_{cS1}} Exp(\mu_{reduce})$$

Table 5.5: Reduce/Key Count Statistics and Fitness Measures

		Reduce Times in Milliseconds				PUT Key Count			
m	Trace	Mean	Std	Med	p-value	Mean	Std	Med	p-value
1	1	0.042	0.015	0.040	0.895	2050.262	257.976	2042	0.842
	2	0.050	1.041	0.039	0.848	2038.750	240.581	2012	0.834
	3	0.044	0.476	0.039	0.871	2067.980	276.711	2045	0.960
2	1	0.054	0.321	0.045	0.364	1033.452	112.299	1028	0.973
	2	0.048	0.508	0.041	0.801	1045.111	149.813	1029	0.912
	3	0.056	0.880	0.044	0.692	1030.455	137.163	1014	0.951
4	1	0.057	0.942	0.043	0.981	517.874	61.469	511	0.981
	2	0.045	0.394	0.039	0.770	509.112	68.713	507	0.987
	3	0.053	0.994	0.041	0.921	510.649	68.042	504	0.826
8	1	0.059	0.696	0.044	0.735	254.453	33.695	249	0.736
	2	0.062	0.741	0.044	0.863	252.198	30.838	249	0.938
	3	0.056	0.348	0.044	0.058	256.873	35.538	253	0.386
16	1	0.069	0.326	0.051	0.647	128.821	18.961	128	0.679
	2	0.072	0.734	0.049	0.602	128.622	18.712	127	0.531
	3	0.069	0.212	0.050	0.303	128.489	18.523	128	0.155

5.3.5 Results

Table 5.6 provides summary statistics for all traces and models generated. The left side of the chart provides similar metrics as those seen from table 5.1 without rolling up all the traces into a single metric per parallelism. We can see the variance of mean within various treatment levels (such as $m = 1$ trace two). We can also see that the model results are appropriately correlated to these changes (a significant drop in mean for the trace leads to a drop in the modeled mean). Such correlations are not perfect: trace one and three would swap places if ranked by mean under measured or modeled.

Figure 5.8 shows the CDF of this data for $m = 1$. The differences between the traces becomes much more stark. Though they share the same basic shape, they

Table 5.6: Measured and Modeled Summary Statistics for Response Time

		Measured			Modeled		
m	Trace	Mean	Std	Med	Mean	Std	Med
1	1	991.978	182.158	977	975.169	163.800	941.527
	2	785.135	180.370	771	766.979	160.599	733.512
	3	974.506	178.840	960	996.590	170.433	960.680
2	1	599.416	114.799	582	605.260	126.890	582.448
	2	545.632	113.008	531	521.964	109.027	500.860
	3	630.909	116.982	616	605.041	126.985	575.516
4	1	367.208	90.130	352	355.410	102.033	331.516
	2	274.390	70.968	260	271.401	69.759	251.340
	3	291.847	75.540	277	312.293	86.711	289.182
8	1	274.350	74.285	258	274.912	93.095	252.489
	2	275.961	97.291	252	286.851	106.692	260.263
	3	297.165	76.355	279	283.028	113.258	253.241
16	1	243.773	114.224	212	239.562	101.632	211.109
	2	216.156	84.316	190	232.118	103.746	203.384
	3	221.035	83.336	195	237.195	89.891	214.058

are offset by roughly 200 ms. We can see that the model curves share similar shape but appear to under estimate the service rates in some places. Overall, these are fairly close values and likely to be useful to practitioners, however we must analyze things further to determine how significant these numbers are.

In the preceding sections we have used Welch's version of the T-test to determine if a sample drawn from a specific distribution provides a good estimate of the measured data. Table 5.7 provides an analysis of the T-test results for determining how similar the various traces relate to each other within a specific treatment. What we find is that very few trace pairs meet the conditions to pass the T-test. Within $m = 1$ we find that trace one and three have the highest p-value of 0.041 (the measure is symmetric) which isn't surprising given their proximity in 5.8. Furthermore,

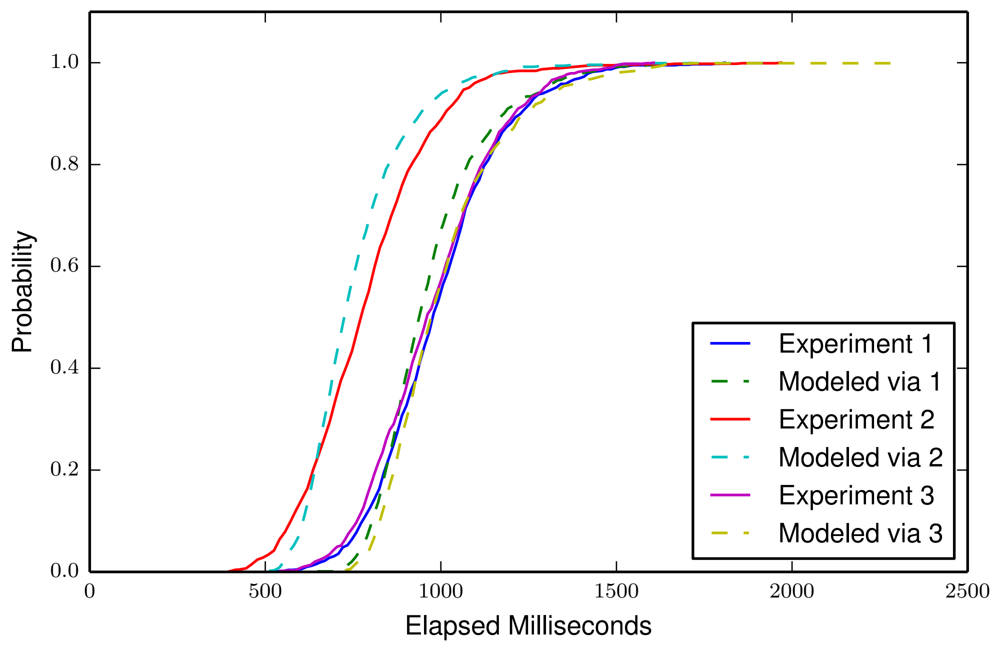


Figure 5.8: CDF of Measured and modeled batch elapsed ms for m=1

many of the models fail the T-test when compared to the generating trace or to other traces within the same parallelism. Oddly for $m = 1$ we have cases such as the model generated from trace one failing against that same trace but showing a strong p-value with trace three. Given that the traces themselves don't show statistically significant similarity we would be hard pressed to expect the same from the models. However, there are other measures to consider when comparing these results.

To put things in perspective we also provide calculations for the percentage errors (as defined by $\frac{|obs-exp|}{exp}$) for the means. Table 5.8 shows the results when calculating percentage errors against the measured and modeled data sets. For these calculations the row's mean was used as the expected value while the column's mean was used as the observed. While high p-values are good for the T-test, lower

Table 5.7: Result T-test Comparison

		Measured to Measured p-value			Modeled to Measured p-value		
m	Trace	1	2	3	1	2	3
1	1	1.0	0.0	0.041	0.0	0.0	0.233
	2	-	1.0	0.0	-	0.003	0.0
	3	-	-	1.0	-	-	0.134
2	1	1.0	0.0	0.001	0.078	0.0	0.0
	2	-	1.0	0.0	-	0.0	0.0
	3	-	-	1.0	-	-	0.0
4	1	1.0	0.0	0.0	0.001	0.0	0.0
	2	-	1.0	0.0	-	0.013	0.0
	3	-	-	1.0	-	-	0.04
8	1	1.0	0.79	0.001	0.001	0.0	0.0
	2	-	1.0	0.003	-	0.089	0.0
	3	-	-	1.0	-	-	0.0
16	1	1.0	0.0	0.001	0.0	0.689	0.228
	2	-	1.0	0.523	-	0.059	0.005
	3	-	-	1.0	-	-	0.67

percentage errors are actually better. This can be seen in the diagonal values where the expected and observed means are the same. Using this metric, we can see that the means for trace one and three (where $m = 1$) are within two percent of each other. Furthermore, the model from trace one provides a near match for the mean for trace three (which explains why this pair passed the T-test). We can also see that a large percent error between pairs of traces generally leads to a large percent difference in model performance.

Figure 5.9 shows the CDF for the case where $m = 8$ which shows the best model performance when measured using percentage error. We can see that the traces are very tight fitting and that the model curves are close. The worst performing model to measured pair (one to three) is still within 10 percent (the rest are within

Table 5.8: Result Percentage Error Comparison

		Measured to Measured Mean %-error			Modeled to Measured Mean %-error		
m	Trace	1	2	3	1	2	3
	1	0.0	20.9	1.8	1.7	19.5	0.1
1	2	26.3	0.0	24.1	29.3	2.4	27.1
	3	1.8	19.4	0.0	0.5	21.2	2.2
	1	0.0	9.0	5.3	1.0	9.9	4.2
2	2	9.9	0.0	15.6	14.8	4.5	20.9
	3	5.0	13.5	0.0	0.9	9.8	4.3
	1	0.0	25.3	20.5	3.3	22.8	17.9
4	2	33.8	0.0	6.4	35.3	1.1	7.5
	3	25.8	6.0	0.0	17.6	12.1	6.5
	1	0.0	0.6	8.3	0.2	0.4	8.1
8	2	0.6	0.0	7.7	4.4	3.8	3.6
	3	7.7	7.1	0.0	3.1	2.5	5.0
	1	0.0	11.3	9.3	1.8	9.8	7.7
16	2	12.8	0.0	2.3	5.0	6.9	4.8
	3	10.3	2.2	0.0	2.8	8.9	6.8

5 percent). Ultimately, these results demonstrate the viability of modeling the runtime performance of Pig scripts running in Storm. Also, by demonstrating that large variance between traces leads to similar variance in modeled results we have a means of judging the quality (or uniqueness) of a particular trace. The following section will summarize the chapter and discuss more implications of this work.

5.4 Chapter Summary

In this chapter we discuss the variance in streaming rates seen from Twitter and how it can impact the service requirements of a streaming computation. We then discussed the specific execution algorithm used by Trident to provide coordination

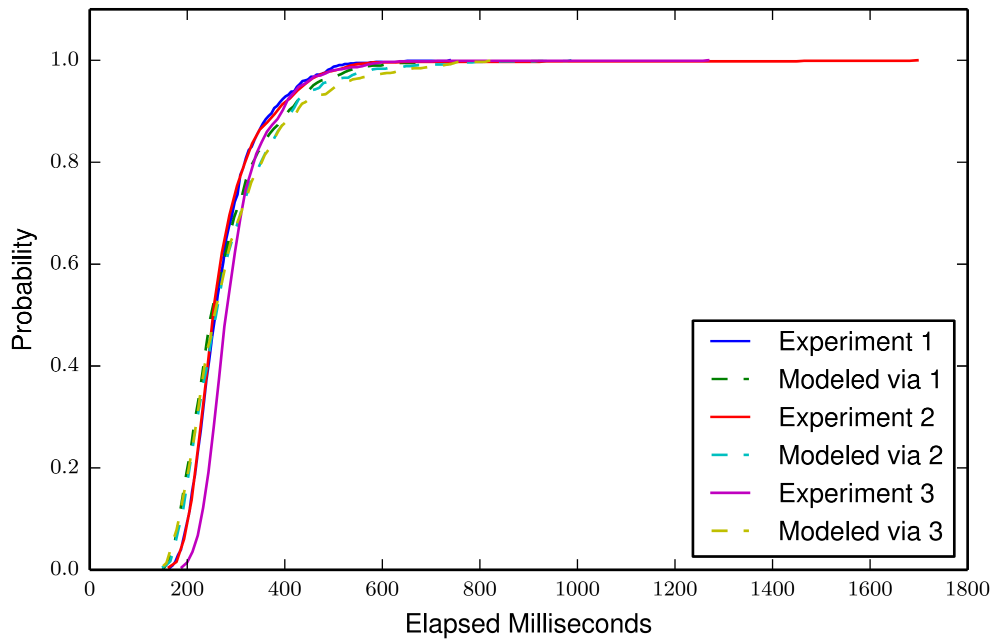


Figure 5.9: CDF of Measured and modeled batch elapsed ms for $m = 8$

between stages within a topology and how this lends itselfs to modeling MapReduce stages independently. We then discuss a simplified model that attempts to simulate execution times seen in trace data. We discuss each component of the model in great detail and show cases where generalization would be problematic (service rates changing as resources are added). Finally, we show that the model is able to generate mean performance that usually lies within 10% of the measured performance. The next chapter will provide a focused literature survey that describes related work.

Chapter 6

Literature Review

This work lies within the intersection of three major topics: Big data (batch) processing, Stream processing, and Performance Modeling. There is also a relationship to the incremental view maintenance which is a subcategory of database systems. A deep review of any of these topics would take a large part of this document, instead this review will focus on a more surgical view of the papers that most influenced and relate to this work. For a broad examination of each of these topics, I will recommend the surveys in each of the sections.

We open the chapter with a discussion of big data processing with a focus on incremental updates. This is followed by a discussion of languages and APIs to ease development which will focus on modifications to the Pig[55] dataflow language After that we will have a discussion of stream processing and conclude with a small section of performance modeling.

6.1 Big Data Processing

One would be hard pressed to write about big data and not cite the "Google trifecta": Google File System [27], MapReduce [23, 24], and BigTable [17]. Google File System is a cluster-based file system with reduced POSIX semantics (read, write, append, no overwrite) that was born from the experiences of building a web search engine [13]. MapReduce is a method for distributing a computation across a cluster of nodes. MapReduce takes care of many of the issues such as task scheduling and placement, data moving and synchronization, check pointing and failure. BigTable is a distributed B-tree that supports vertical partitioning, name spaces, and versioning. This system is used at Google to modularize aspects of it's business in a way that separate teams (web crawl, link analysis, indexing and search etc.) can focus on specific issues but still collaborate. These papers inspired the creation of Apache Hadoop[1] which has HDFS, MapReduce APIs as well as HBase[2] which mimics BigTable. They are not without their critics [64], but they are here to stay.

6.1.1 Incremental Updates

The research surrounding big data processing with MapReduce is quite a hot topic and has produced numerous survey papers [25, 61, 11]. In [25], Doulkeridis and Nørvåg present a taxonomy of MapReduce improvements for efficient query processing which can be seen in figure 6.1. Using this taxonomy, this work falls primarily within the "Interactive, Real-time Processing" domain with a secondary goal being "Avoidance of Redundant Processing" due to the conversion of MapReduce calculations to incremental streaming processes. What follows is a discussion of the most closely aligned works that deal with incremental updates. Incremental updates is one of the techniques used within DBMS to perform materialized view maintenance [77].

[1]http://hadoop.apache.org/
[2]http://hbase.apache.org/

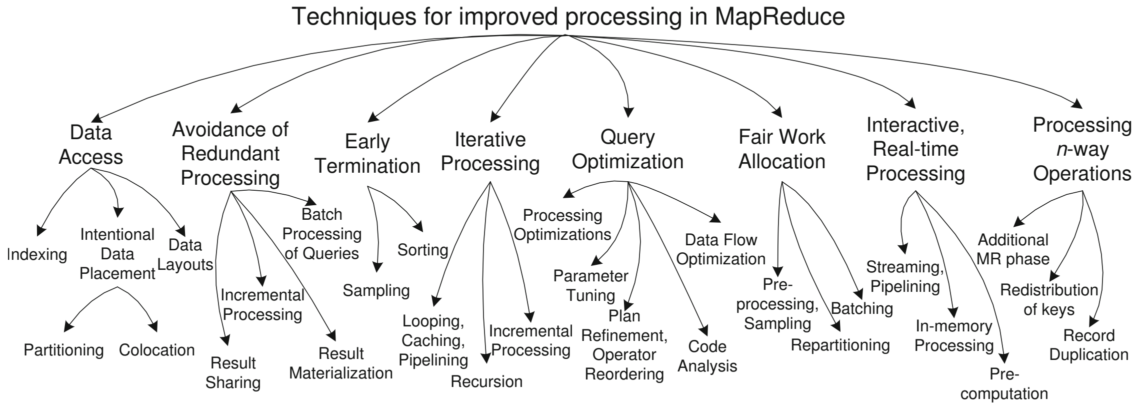


Figure 6.1: Taxonomy of MapReduce improvements from Doulkeridis and Nørvåg.

Differential dataflow [49] establishes a mechanism for decomposing a computation into a series of incremental updates. By maintaining the input and outputs as a series of deltas they are able to update and propagate changes as new information arises. There are two core formulas that relate to this work:

$$A_t = \sum_{s \leq t} \delta A_s \tag{6.1}$$

Which is equivalent to the recursive expansion of equation 3.3 seen in chapter 3. The formula:

$$\delta B_t = Op(A_{t-1} + \delta A_t) - Op(A_{t-1}) \tag{6.2}$$

Is equivalent to the reduce delta operator seen in chapter 3 where the Op function would be the reduce function and the output would be key/value sequences. In fact, the difference values δB for reduce delta operator is equal to the concatenation of the sequences: $R^{+} + R^{-}$ as defined by equations 3.1 and 3.2.

The core idea behind handling of incremental updates is the same. While differential dataflow provides a generic description of this calculations, Squeal is focused on automatic conversion of MapReduce calculations to be incremental calculations. The counting set construct presented in chapter 3 establishes a generic structure for maintaining value lists to be fed to opaque reduce operators. More efficient implementation of delta computations is enabled by extending Pig's `Algebraic` interface to include an inverse initialization function which is used in an extended combine calculation. While differential dataflow maintains B_t which is equivalent to the message sequences produced by the reduce operators, Pig Squeal currently discards these values. The current run times of the reduce operation is not seen as a significant source of execution time to warrant the complexities of maintaining the outputs given the constraints of the Trident state management.

Naiad [52] is a data workflow system that executes parallel, cyclic data flow programs which implements iterative computing using differential dataflow computations. At it's lowest level it provides callbacks on worker vertices that react to messages transmitted within the system. These executions produce deltas which are propagated through the graph until a fixed point is reached (which ends the execution of a batch of input). This abstraction was then used to provide higher level libraries that implement other well known programming abstractions such as LINQ, Bloom [21], and Pregel [48]. They also implemented a distributed system for scheduling and executing Naiad computations which resembles the Dryad [36] system without some of the synchronization steps.

Neither of these works discuss the issue of discarding state or having online/offline state. This is mainly because the aim of such systems is on performing iterative processing on data structures in memory. Consider the word count example where we track the occurrence over time (as in Chapter 4). There is a continuous process of keys coming into existence (as time changes), are highly active (during that minute), and become dormant (as the next minute becomes active). With a cache enabled storage mechanism, Trident should be able to keep the active set in memory and eject dormant keys as time passes. Furthermore, Pig Squeal isn't sensitive to out of order delivery of many window mechanisms. Ultimately, this coupled with the window functionality within Squeal should allow for long running processes that discard values to long term storage and ultimately forget old data as storage time-to-lives expire.

MapReduce with Deltas [44, 60] is a very interesting work that discusses how delta computations (inspired by diff [35]) could be used handle incremental updates within Hadoop. This work inspired me to formalize my description of the counting set used to manage *sign* values for generic MapReduce programs to implement what Saile calls a *delta layer*. They mention "streaming" as an I/O optimization to avoid materializing some of their data set but it is still performed within the confines of Hadoop. Much like the following work, there is significant cost incurred by calculating the changes based on the content of the data being processed (one of their examples uses two generations of a web crawl data). This does not effect this work because updates are streamed in and can be marked with the sign values from upstream providers. To manage changes of a web page it would be simple to group by the URL and keep the top content value by time. Upon update, the reduce delta mechanism would produce the appropriate delta information.

Incoop [12] appears to be the full realization of the delta layer for Hadoop mentioned in [60]. They apply task-level memoization to track dependencies between map inputs and reduce tasks. They use content fingerprinting techniques to generate diff markers. The Incoop system includes a custom implementation of HDFS to track difference markers, a memoization server to track dependencies from map chunks to reduce output, custom hooks at storage to maintain output differences, and a custom scheduler to exploit storage affinity and provide a means for task movement to address stragglers. They were able to run Pig incrementally by re-compiling the Pig executables with the custom HDFS libraries. They were also able to show speedups against various compute and data intense applications.

There are many differences between the Pig Squeal implementation and Incoop. When map or reduce tasks produce the same output for any given input, there is no need to explicitly track the dependencies between map tasks and reduce tasks. This allows Pig Squeal to use finer grained key-level memoization to track maintain the last inputs to the reduce stage. By implementing Pig Squeal as a Pig execution environment, it is able to run on stock Hadoop and Storm which has significant benefits to maintainability and introduction into enterprise environments (by relying on existing, battle-tested infrastructure). It would be possible to have Pig Squeal run in other environments as well including Hadoop, Spark/Naiad, and Percolator-like environments. The key to such executions is the use of a fine-grained key value storage mechanism. Storage affinity could be regained in Hadoop by using a storage-specific partitioning function (such as HBase which is used throughout this work). However, Pig Squeal would suffer from the same scheduling issues as Incoop or other small-job executions if using Hadoop directly.

NOVA [54] is a workflow engine that tracks incoming batches of data and generates template load functions to allow Pig developers to receive delta records from HDFS. Unlike Incoop, no modifications were made to the underlying implementations of Hadoop, Pig, or Oozie[3]. NOVA implements changes on a block/execution level and performs delta computation using join operations injected with the template into the user specified Pig script. By using the Zebra[4] columnar store, they are able to perform map-side joins which perform well up to a point (20 million records). The authors do note that low-latency processing is the realm of stream processing and mention auto-rewriting of non-incremental workflows to implement in an incremental fashion.

MapReduce Online [20] is a technique for continuously running MapReduce tasks within a modified version of Hadoop that supports pipelined operations. While Hadoop materializes intermediate results to disk to be *pulled* during the shuffle phase, MapReduce Online changes the communications methods to *push* values from map to reduce as soon as they are available. This requires enough reduce slots to be available when the map phase is running as well as a TCP connection between every mapper and reducer. They refined this technique to materialize data to disk and pull in the traditional way if a reduce task must be delayed due to lack of task slots. They also configured the reducers to pull from only a bounded number of mappers at a time. Within Storm, all tasks must be online to process the data in

[3]http://oozie.apache.org/
[4]http://pig.apache.org/docs/r0.8.1/zebra_overview.html

the topology and connections are only made between workers (which host numerous tasks).

Eager transmission of results from map to reduce does not allow for combiner use and also puts more work on the reducer because normal map operations send sorted results (which can use a more efficient merge-sort on the reduce side). They address this by buffering the data on the map side until a threshold is met where they apply the combiner and sort the results. Trident currently accumulates all results until the preceding tasks in the topology complete at which point the aggregators running combine are completed and results start to flow to the next task set in the graph. They discuss the possibility of pipelining from reduce into the next map phase of a chained computation (which is commonplace with Pig). They transmit “snapshot” outputs to represent partial answers at any given time (the notion of partial answers is related to online aggregation [31] and is implemented in some recent interactive analysis systems such as Dremel [50]). These snapshots require recomputing of any follow-on MapReduce stages in a chained execution to produce the answer. They mention that optimizations are possible if the reduce operators in the chain have the appropriate algebraic properties (as outlined in chapter 3).

6.1.2 Launguages, APIs, and Pig Modifications

There are numerous languages to provide lower barriers to entry to using big data systems [50, 59, 66, 67, 55]. Pig[55] was selected for further study because it is most representative of general MapReduce computations. Furthermore, by having

Pig available in a streaming platform, we can leverage much of the existing work available through scripts and UDFs. HIVE's [66, 67] relationship with SQL gives it a more natural transition to stream processing through the use of windows and other techniques that are described later. HadoopDB [8] is a demonstration of this strength.

There are some notable modifications to Pig: HStreaming is a technology for migrating Hadoop jobs to a proprietary streaming platform. It added a WINDOW keyword to the Pig programming language. HStreaming was purchased by Adello[5] and does not appear to be generally available.

Spork[6] is a port of Pig to the Spark [74] execution environment. They take the MapReduce plan and use it to build a corresponding Spark plan. Spark provides higher level functionality than Storm (such as `FILTER`) which Spork attempts to leverage by going deep into the physical plan and replacing operators. The focus of this work is on using Spark to provide more interactive interactions with Pig on bulk data. Updates to the data through the use of Spark Streaming [75] have not been discussed, but the techniques of Squeal could be applied to propagate changes appropriately.

At the 2014 Hadoop Summit, Mridul Jain presented Yahoo's work in converting Pig scripts to Storm topologies. They make use of Trident to process batches of tuples. Each batch is treated as an independent execution of the Pig topology and they make use of windows stores in HBase to provide across-batch synchronization.

[5]https://gigaom.com/2013/09/05/ad-platform-adello-buys-hadoop-startup-hstreaming/
[6]https://github.com/sigmoidanalytics/spork

They also have a hybrid mode which parts of the Pig script are run in Hadoop. Squeal and Yahoo's work seem to be closely related and complimentary (with a potential for collaboration).

6.2 Stream Processing

Cugola and Margara present a survey[22] in which they coin the phrase *Information Flow Processing (IFP) Systems* to describe systems that "process continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries." This definition covers both *Data Stream Management Systems (DSMSs)* and *Complex Event Processing (CEP) Systems* which have evolved separately to solve similar problems. The lineage of DSMSs can be traced back to the database community while CEP systems came more from messaging middle-ware. Cugola and Margara state that "…DSMSs focus on producing query answers, which are continuously updated to adapt to the constantly changing contents of their input data. Detection and notification of complex patterns of elements involving sequences and ordering relations are usually out of the scope of these systems." Of Complex Event Processing Systems, they state that "…they associate a precise semantics to the information items being processed: they are *notifications* of *events* happening in the external world…The CEP engine is responsible for filtering and combining such notifications to understand what is happening in terms of *higher-level* events (sometimes also called *composite events* or *situations*)." Modern IFP systems such as Streambase[7] and Esper[8] incorporate both ideas: continuous query and detection/notification.

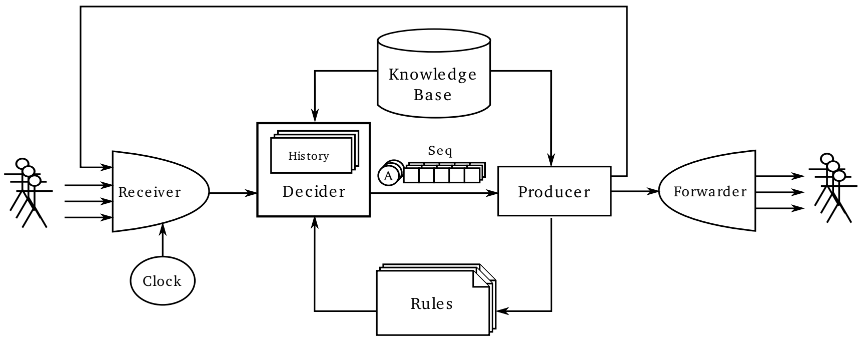


Figure 6.2: The functional architecture of an IFP system as depicted by Cugola and Margara.

Complex Event Processing (CEP) can be thought of as a means of describing a series of patterns and actions to react to data (or the passage of time). One of the quintessential examples is "If smoke is detected and the temperature is above a threshold trigger an alarm!" As the name implies, for CEP the main element of concern is an event. Luckham [47] defines an *event* as an object that is a record of an activity in a system. The events signify the activities and may be related to other events. He describes three aspects of events: form (a representation), significance (what activity the event signifies), and relativity. Relativity defines how an activity relates to other activities by time, causality, and aggregation. By explicitly encoding the causality of events, a CEP system can better reason about the state of the observed system without relying on the implicit representations of causality which may be apparent through timestamps on events. Time itself provides various issues

[7]http://www.streambase.com/
[8]http://esper.codehaus.org/

to consider, such as which clock is being observed, whether or not two clocks are synchronized, and well as various issues due to the uncertainty principle.

Mühl, Fiege, and Pietzuch [51] define an *event* as "any happening of interest that can be observed from within a computer" (similar to Luckham's *activity*). Luckham does explain that two confusions often arise with the common usage of the word "event" and CEP: mixing up an activity (or "something that happens") with an object representing that activity and confusing an event with its form, such as "An event is just a message." Luckham explains that event processing is different from message processing because it must deal with the relationships between events. Events can either be processed in a semi-ordered stream (ordered along some attribute with the possibility of out-of-order delivery, called stream disorder) or in an un-ordered event cloud (typically stored on disk). Some systems require bounds on the amount of stream disorder to be seen. These bounds typically influence the amount of buffering necessary to ensure that answers remain consistent. The notion of disorder strongly influenced my choices for switching from a window-style execution model to a delta-based approach.

I think the ideas from CEP processing are important to understand as they embody many of the concepts of context aware computing. Languages such as Pig and data stream management systems provide the means for the analysis of observed data. The notion of form, significance, and relativity are implicitly encoded into these analytics by the writer where as CEP provides makes treats these notions explicitly. While Squeal makes it possible to analyze data acting upon these results (triggering an event) is left to follow on consumers of this information. Pig itself could benefit from the inclusion of CEP concepts: manipulating data once grouped often requires a UDF (where CEP queries may cover a significant portion of these corner cases). I've done some exploration of embedding the Esper engine into Pig as a UDF. There is also a project that wraps Esper in bolts for use in Storm.

There are many papers that cover first generation DSMSs. The STREAM project[30] created CQL for describing continuous queries which is a combination of SQL for relation-to-relation operations, windowing from SQL-99 and relation-to-stream operators for creating streams from fixed tables. NiagaraCQ[19] allows for continuous queries of geographically distributed information using XML-QL. They focus on grouping queries together for efficient execution. TelegraphCQ[16] is interesting in that it addresses issues associated with running continuous queries in a cluster (routing and load balancing). They have decoupled the components of the system (query operators and interconnect) which allow independent optimizations of each. Aurora[6] is notable in its use of the *boxes and arrows* paradigm for graphically describing continuous queries and for explicitly addressing the QoS problem. Stream processing systems can become overwhelmed when a large burst of events occur. When overload occurs, load shedding or external queuing must be used to keep the system stable. Borealis[5] was created by merging Aurora with the Medusa networking stack[15]. Borealis extends the QoS ideas from Aurora into any point in a flow. By having per-message utility values Borealis is able to improve the quality of results when overload occurs. Borealis was commercialized as Streambase.

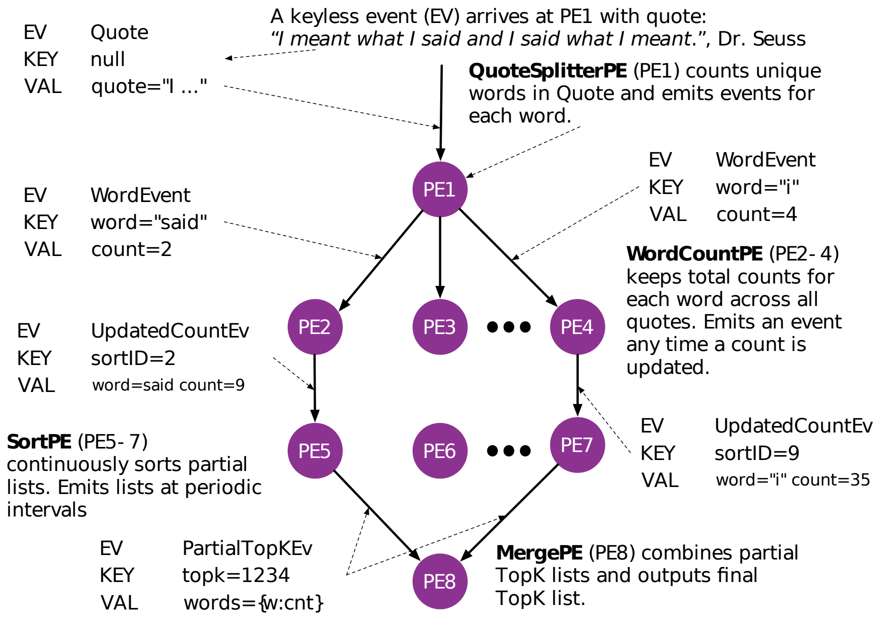
The newest generation of IFP systems focus more on programming frameworks and middleware. They are typically designed for clustered work environments to handle scale out. Recent work includes IBM's SPC[9], SPADE[26], and SPL[32], Yahoo's S4 [53], and Storm[4]. *Stream Processing Core (SPC)* creates an infrastructure that allows *Processing Elements (PE)* to be inserted into a flow graph by registering input/output ports. PEs subscribe to streams based on a name ($StreamName = EmailTraffic$) and receive selected elements from within using a predicate filter ($TransportType : value* = chat$). Results are emitted through output ports which have type information associated with them. SPADE expands SPC's functionality to include an intermediate language (SPADE) to bridge high level languages, such as Stream SQL and the lower level APIs, a toolkit of pre-made stream processing constructs, such as windowing and punctuation, and finally a broad range of input adapters (sockets, DBMS, XML, etc.). IBM's *Stream Processing Language (SPL)* provides a declarative language for describing computations as a graph-of-operators which is used to drive code generation for high performance implementations of the specified analytic. These abstractions allow for entry at the appropriate level of development based on need. Simple Stream SQL workloads can be up and running quickly but still allow for optimization as necessary. Likewise, complex data types (videos, images) can be processed using the lower level API's and composed using the stream processing constructs.

C-MR [10] is a system that adopts the MapReduce API to be used in computing results in a multiprocessor environment. They use the notion of windows to handle the notion that information is coming in a stream. They use combiners for incremental computations of sub-windows which are sent to reduce for final computation. This is a strange reference in that makes use of the semantics of MapReduce to describe a streaming computation which is more in the realm of SPL (which can describe much richer relationships and is tied directly to a first-class DSMS).

Yahoo's S4 applies similar ideas as SPC but focuses on hash partitioning to implement data parallelism across a cluster. This idea is similar to the reduce behavior in Google's MapReduce [23] and is one of the fundamental building blocks of any cluster-based stream processing framework [33]. Figure 6.3 shows a word counting program running in S4. Apache Storm also uses a graph-of-operators approach as we've seen in chapter 1. Storm differentiates the PE's that produce data (spouts) from those that process data (bolts). Though Trident alters this distinction by using spouts to manage batches of tuples which are released later in the processing graph.

There are other methods for processing updates as they arrive. Percolator [58] is a method for triggering on updates in BigTable. It has been used at Google to significantly decrease the latency from web crawl to index replacement. While it lowers the latencies of these pipelines it is still focused on batch updates for performance reasons. It also requires a re-write of the application to the new paradigm. Exploring the use of Squeal in such environments will be interesting as they become available and mature.

It is also worth noting that many of the frameworks designed for iterative in memory processing are also alternatives when using incremental updates. This has been demonstrated in Naiad [52] and Spark Streaming [75].


A keyless event (EV) arrives at PE1 with quote:
"I meant what I said and I said what I meant.", Dr. Seuss
EV Quote
KEY null
VAL quote="I ..."
QuoteSplitterPE (PE1) counts unique words in Quote and emits events for each word.
EV WordEvent
KEY word="i"
VAL count=4
EV WordEvent
KEY word="said"
VAL count=2
WordCountPE (PE2- 4) keeps total counts for each word across all quotes. Emits an event any time a count is updated.
EV UpdatedCountEv
KEY sortID=2
VAL word=said count=9
EV UpdatedCountEv
KEY sortID=9
VAL word="i" count=35
SortPE (PE5- 7) continuously sorts partial lists. Emits lists at periodic intervals
EV PartialTopKEv
KEY topk=1234
VAL words={w:cnt}
MergePE (PE8) combines partial TopK lists and outputs final TopK list.
PE1
PE2
PE3
PE4
PE5
PE6
PE7
PE8

PE ID	**PE Name**	**Key Tuple**
PE1	QuoteSplitterPE	null
PE2	WordCountPE	word="said"
PE4	WordCountPE	word="i"
PE5	SortPE	sortID=2
PE7	SortPE	sortID=9
PE8	MergePE	topK=1234

Figure 6.3: A graphical depiction of word count running in S4 from [53].

6.3 Performance Modeling

Performance modeling is a deep field with a long history with many conferences (including SIGMOD, SIGMETRICS), working groups (such as IFIP WG 7.3), and text books [70, 41, 42]. This work's foray into performance modeling was aimed at understanding what resources would be necessary to handle a specific rate for a Pig script running in Squeal. Analyzing the steady state and stability of systems with flows of work is within the realm of Queueing Theory. However, the execution of batch systems and the study of the total completion time of a batch job is within the realm of Scheduling.

The queueing theoretical model chosen to represent a stage of a converted

MapReduce program is that of a tandem M/M/m queue [70, 41] with batch release [41, 42]. There are significant differences to this model as outlined in chapter 5 namely that unlike a typical queueing model that assumes continuous release of information, the Trident system only releases a single batch at once. This means that Trident executions look more like sequentially scheduled jobs which lends itself to other modeling techniques.

Lin et al.[45] examines the optimal scheduling of MapReduce tasks from the map to shuffle phases and show that it is strongly NP-hard. They model map and shuffle as tandem M/M/1 queues which inspired the experimental design for chapter 5. They also discuss the issues of analyzing the overlapping behavior of these stages. This issue is less pronounced in Squeal because the map phase and the second stage combine phases don't overlap.

ARIA [72] provide formulas for modeling the three stages (map, shuffle, reduce) using the Makespan Theorem [28]. They used roughly fixed sized input along with recorded trace performance to measure the parameters for their formula: map task min and average times, selectivity of the map phase, the average amount of data per map task, the average and max times for shuffle tasks (they break these metrics down by first wave and "typical"), and the reduce task average, max times, and selectivity. Because these metrics were stable between runs and they were able to successfully model traces from these formula and predict the run times when the number of map and reduce slots was varied. These formula were used to find optimal resource allocations for Service Level Oriented time bounds.

Zhang et al.[76] uses models inspired by ARIA to model multi-stage MapReduce plans generated by Pig. They apply the Johnson algorithm for building an optimal two-stage job schedule [38] to reduce the non-determinism of job execution. They then examine the effects of modeling multiple stages and are successful in predicting run times. They show that the techniques from ARIA are applicable to multi-stage executions.

Tian and Chen [68] derive similar formulas to those seen in ARIA but allow for variance in the task size. They overlay a cost model using prices from Amazon's Elastic MapReduce[9] to find the resource allocation to optimize the cost of the computation. All of these models assume that the performance of the operators is independent of the number of resources assigned. The experimental results seen in tables 5.2 and 5.5 imply that the service times would need to be a function of the number of resources. Also, the shuffle performance is linearly dependent on the number of reduce resources assigned. For Trident, this relationship does not hold as we can see in table 5.3.

Other studies have examined the performance of Storm and Trident. Researchers at Ericson [56] report decent performance in a micro-benchmark (25,000 records per second on three nodes) but find that the performance dropped significantly when follow on aggregation stages are added (to 7,375 records per second). They found that performance improvement was linear to a point but started to drop off. They also found that the garbage collector caused significant delays to some executions. This is assumed to be one source of the micro-stragglers we see in our study but this has not been confirmed. This is followed up with another study [57]

[9]http://aws.amazon.com/elasticmapreduce/

that finds that hand written Storm and Trident topologies have similar performance. They also note that events spend a long time in the `waitFor` method of Disruptor [65] (the inter-thread communications used in Storm).

Toshniwal et al. [69] provide an excellent overview of Storm and discuss various operational observations, enhancements, and benchmarks. Their work uses hand written Storm topologies which they compared with purpose built streaming programs (without the use of a streaming framework). They found that with reliable messaging disabled, that Storm utilized the same amount of CPU as the custom written code. They found that three machines were necessary to process the 300K messages per second when message reliability was enabled. They present a flow control algorithm that auto tunes the number of messages active in the system at once (which is equivalent to what we have called the batch size). They also perform an empirical study of the latency for a calculation by varying the number of machines available to the topology. The physical machines used in this and the Ericson studies were much larger than those used by the studies from chapters 4 and 5.

Chapter 7

Conclusion

We are in truly exciting times. As our troves of data open up to us through the use of big data processing techniques we have started to shift the focus from scale to latency. New and complementary processing systems [74, 52] are emerging that address some of the shortcomings of MapReduce [25]. While HIVE [66] has allowed some SQL queries to migrate into these new platforms its most important contribution is that people can bring their existing knowledge and understanding of analyzing data in SQL to a new environment. DBMS and MapReduce systems are truly complementary [64] each with unique strengths and weaknesses. The same can be said about MapReduce and iterative or streaming platforms. This work, like others (such as Spork and Yahoo's Pig on Storm) is attempting to build a bridge between these systems.

What we know now is that we can bring batch algorithms described in Pig Latin over to streaming environments using incremental techniques [49]. We know that under certain conditions (map and reduce tasks that are mathematical functions) it is possible to track changes through the system without explicitly storing dependency information [12]. We also know the conditions which allow combiners to be migrated over to incremental implementations and have a simple extension to the Pig API to facilitate this change. Through our case studies we know that we can significantly lower the latency from receiving our data and having answers to our standing questions (we also know the throughput needs to be improved). Finally, we understand the contributions each phase of computation make to the service time of our system.

7.1 Discussion of Trade-offs

During the implementation of a solution to a problem there are many decisions made that have consequences on the generality of that solution. Under different conditions, different decisions may be made. The ultimate goal of this work is to provide a bridge from batch to stream processing. In doing so, it becomes less "expensive" to introduce low latency computation into an existing environment while providing additional returns on investment in batch codes. The main considerations of discussion of pros and cons centers around three dimensions: development costs, storage costs, and runtime costs.

For development costs, the solution presented in this work displays strengths in development time, maintainability, and extensibility while having weaknesses in descriptiveness and intuition issues. For a Pig developer, using Squeal significantly reduces the leap to stream processing – enabling streaming execution is a command line switch. Existing code can be tested in a streaming environment very quickly and tuned to the new operating environment. The maintainability and extensibility are a carryover from using a scripting language: less code, strengths of a domain specific language, ability to extend functionality with user defined functions (which can still be optimized under the new execution environment). The cost of these benefits is that there is an impedance mismatch of using a static/batch domain specific language in a streaming environment: it lacks functionality that is typical of the domain – pattern/action mechanisms of complex event processing systems [22] and native windowing of data stream management systems [30]. Because the execution environment is different, highly tuned codes will require some re-engineering as we've seen in our case studies. These differences require re-consideration of developer intuition – honing this intuition requires experience.

Storage costs for this implementation of MapReduce with Deltas requires a key/value storage mechanism. When compared with Incoop [12] (which implemented a block-level memoization mechanism to track the influence of updates on previous computations) there is less impact on practical deployment of these codes (because the existing enterprise systems are left intact) with the costs associated with running a key/value store. Incoop requires custom versions of various components of the Hadoop architecture which would be hard to deploy in an enterprise environment. If the number of update records within a block is small, having a key/value mechanism will be faster (having to only examine the specific updates). There is likely a threshold of updates where it is faster to recompute a full block instead of fetching the individual keys but this was not examined in this work.

When comparing this implementation to iterative computation mechanisms such as Spark [74] and Naiad [52] which assume most of the working set is in memory, Trident's caching mechanism provides a good balance of maintaining the working set in memory (if possible) while spilling to secondary, "offline", storage if necessary. This is quite important for time-based computations which have an active state (surrounding the time) and then a dormant state (once all updates have been received). An interesting property of the delta mechanism when compared to standard windowing operations is that data can arrive out of order without having significant impact on the correctness of the answer (there will be latency costs in loading "old" data from the key/value store).

The final consideration surrounding the key/value store is the storage and network costs themselves. While the combiner functionality can significantly reduce the amount of information to be transfered across the network and stored, the counting set functionality can have significant costs in memory and storage if the number of unique values is great. While compression can alleviate some of these concerns (at the cost of latency) the space/time trade-off is a core issue when considering streaming applications.

As far as runtime costs and benefits there are a few to consider: delta overhead, optimized code, and service time. Delta computations using counting sets requires two executions of the reduce function (one for the last value list and one for the current value list). Reduce functions can be quite expensive, there is the option to cache the previous execution's output (which would trade time for memory – which is used by the Naiad system [52]). However, it is possible for the message sets to be quite large (significantly larger than the input in the case of `JOIN` operations). The combine operation provides a potential for optimizing execution. By extending the combine interface within Pig to allow for inverse it is possible to enable combiners to function within a delta environment. Performing delta computations for abelian groups can greatly reduce the runtime overhead of performing updates. One of the strengths of Pig is providing highly optimized operator code to be used by script authors. The techniques described in this work allow for these optimized operators to be run in a streaming environment.

The ultimate test of this technique is whether or not the service time of the converted applications provides a stable execution – being able to service requests at a rate faster than the input rate. At this point it is very simple to deploy a code and measure the service rate of the resulting topology. There are still challenges in determining what resources to tweak to improve the service rate. First, the existing Storm User Interface only reports the CPU time of each worker. We have shown that there are significant latencies between the components. Also, further development is necessary to optimize the performance of converted topologies (which is discussed in the following section). Any investment in optimizing the service time of converted scripts will pay off in reuse across many scripts – similar to improvements in DBMS query optimizers to produce better query plans.

7.2 Future Work

One treacherous aspect of working on a Ph.D. is the realization that the thread you found and began to pull seems to go on forever. There are many outstanding questions from this work. Some derive from implementation choices – could Squeal be ported to other streaming platforms [32, 75]? Taking that a step further, could Squeal be ported into a completely different (but still data driven) framework such as Percolator[58]? There are certainly improvements that could be made to Storm: new versions provide a graphical representation of the topologies along with bolt execution latencies – tracking the latencies of the network would be hugely beneficial in understanding the performance of running topologies (this has been a long standing issue and has led the community to change the underlying communication system). Can we lean on the incremental computations more to allow tuples to flow more freely (without requiring upstream producers to finish – this would blend the pure streaming and batch-based streaming approaches)?

In the realm of Pig: what new optimizations make more sense when executing in a streaming environment? Perhaps implementing some of the known streaming optimizations [34] would help (this holds true for Storm as well). As new execution environments come online it will be interesting to see how the Pig community manages some of the conflicting goals of the various implementations – definitely some areas of study for Software Engineering and Language design. There is also an outstanding question of decomposing multiple queries to share work from common sub-problems.

The discipline of mathematical modeling will never want for new problems, but there are some interesting questions herein. There is certainly existing work for studying the execution times of MapReduce programs to automatically optimize resource allocation [72, 76, 68]. It would be interesting to see these models shift into the streaming realm to help operators understand their system. Finally, it would be interesting to feed these models back into the Pig compiler to assist in optimization or more importantly to provide user feedback as to which part of their script costs the most to run.

7.3 Closing Remarks

My hope is that this work will help organizations with large investments in Pig (through scripts, UDFs, and developer knowledge) to begin to shift their operations into streaming environments. This work is not an end in itself – much as custom MapReduce code will usually outperform a Pig script (perhaps at the cost of the developers sanity when faced with implementing `JOIN` again) the same can be said about custom written Storm topologies (or IBM InfoSphere Streams[32] or Spark[75]). Pig can provide a means of bridging the gap between developers and analysts. With a working script in hand, the developer has a concrete set of requirements to start from. Selfishly, I want this myself – I enjoy working in higher level language to quickly solve my problems (who knows, I may be able to ride this knowledge for a long time, much like COBOL programmers in 1999). Thank you for your interest in this work.

Bibliography

[1] Apache cassandra nosql performance benchmarks. http://planetcassandra.org/nosql-performance-benchmarks/.

[2] Apache giraph. http://giraph.apache.org/.

[3] Espertech - esper. http://www.espertech.com/esper/.

[4] Storm: Distributed and fault-tolerant realtime computation. http://storm.incubator.apache.org/.

[5] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[6] Daniel J. Abadi, Don Carney, Ugur etintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management, 2003.

[7] D. Aberdeen, O. Pacovsky, and A. Slater. The learning behind gmail priority inbox.

[8] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[9] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: a distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, DMSSP '06, pages 27–37, New York, NY, USA, 2006. ACM.

[10] N. Backman, K. Pattabiraman, R. Fonseca, and U. Cetintemel. C-mr: continuously executing mapreduce workflows on multi-core processors. In *Proceedings of third international workshop on MapReduce and its Applications Date*, pages 1–8. ACM, 2012.

[11] Edmon Begoli. A short survey on the state of the art in architectures and platforms for large scale data analysis and knowledge discovery from data. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, pages 177–183, New York, NY, USA, 2012. ACM.

[12] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[13] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *COMPUTER NETWORKS AND ISDN SYSTEMS*, pages 107–117. Elsevier Science Publishers B. V., 1998.

[14] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[15] U. Cetintemel. The aurora and medusa projects. *Data Engineering*, 51:3, 2003.

[16] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[17] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[18] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[19] Jianjun Chen, David J. Dewitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *In SIGMOD*, pages 379–390, 2000.

[20] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21, 2010.

[21] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[22] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys, to appear.*

[23] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53:72–77, January 2010.

[25] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, 23(3):355–380, 2014.

[26] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[27] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[28] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[29] Alan Greenblatt. Five takeaways from the first presidential debate. *National Public Radio*, Oct 2012. http://www.npr.org/2012/10/04/162265100/five-takeaways-from-the-first-presidential-debate.

[30] The STREAM Group. Stream: The stanford stream data manager. Technical Report 2003-21, Stanford InfoLab, 2003.

[31] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. *ACM SIGMOD Record*, 26(2):171–182, 1997.

[32] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. Ibm streams processing language: Analyzing big data in motion. *IBM J. Res. Dev.*, 57(3-4):1:7–1:7, May 2013.

[33] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[34] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, March 2014.

[35] James Wayne Hunt and M Douglas McIlroy. *An algorithm for differential file comparison*. Bell Laboratories, 1976.

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

[37] Frank James. Obama and romney respond to sandy with election (and katrina) in mind. *National Public Radio – it's all politics*, Oct 2012. http://www.npr.org/blogs/itsallpolitics/2012/10/29/163862168/obama-and-romney-respond-to-sandy-with-election-and-katrina-in-mind.

[38] Selmer Martin Johnson. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly*, 1(1):61–68, 1954.

[39] Jeff Jonas and Lisa Sokol. Data finds data. *Beautiful Data, The Stories Behind Elegant Data Solutions*, 2013.

[40] Flavio P Junqueira and Benjamin C Reed. The life and times of a zookeeper. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 4–4. ACM, 2009.

[41] Leonard Kleinrock. Queueing systems, volume i: theory. 1975.

[42] Leonard Kleinrock. Queueing systems, volume ii: computer applications. 1976.

[43] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, 2011.

[44] Ralf Lämmel and David Saile. Mapreduce with deltas. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA*, volume 2011, 2011.

[45] Minghong Lin, Li Zhang, Adam Wierman, and Jian Tan. Joint optimization of overlapping phases in mapreduce. *Performance Evaluation*, 70(10):720–735, 2013.

[46] Tim Loughran and Bill McDonald. When is a liability not a liability? textual analysis, dictionaries, and 10-ks. *The Journal of Finance*, 66(1):35–65, 2011.

[47] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[48] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[49] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of CIDR 2013*, January 2013.

[50] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[51] G. Muhl, L. Fiege, and P. Pietzuch. *Distributed event-based systems.* Springer-Verlag, 2006.

[52] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, November 2013.

[53] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. *Data Mining Workshops, International Conference on*, 0:170–177, 2010.

[54] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki BN Rao, Vijayanand Sankarasubramanian, Siddharth Seth, et al. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1081–1090. ACM, 2011.

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

[56] Manoj P. Ericson research blog: Trident benchmarking performance. http://www.ericsson.com/research-blog/data-knowledge/trident-benchmarking-performance/ [Online; accessed 21-Nov-2014].

[57] Manoj P. and Laszlo Toka. Ericson research blog: Comparing apache storm and trident. http://www.ericsson.com/research-blog/data-knowledge/comparing-apache-storm-trident/ [Online; accessed 21-Nov-2014].

[58] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[59] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[60] David Saile et al. Dissertation: Mapreduce with deltas. 2007. http://kola.opus.hbz-nrw.de/volltexte/2011/687/pdf/diplomarbeit_david_saile.pdf.

[61] Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Comput. Surv.*, 46(1):11:1–11:44, July 2013.

[62] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[63] Valentin I Spitkovsky and Angel X Chang. A cross-lingual dictionary for english wikipedia concepts. In *LREC*, pages 3168–3175, 2012.

[64] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53:64–71, January 2010.

[65] Martin Thompson, Dave Farley, Michael Barker, Patricia Gee, and Andrew Stewart. *DISRUPTOR:High performance alternative to bounded queues for exchanging data between concurrent threads.*

[66] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[67] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[68] Fengguang Tian and Keke Chen. Towards optimal resource provisioning for running mapreduce programs in public clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 155–162. IEEE, 2011.

[69] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[70] Kishor S Trivedi. *Probability & statistics with reliability, queuing and computer science applications.* John Wiley & Sons, 2008.

[71] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[72] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 235–244. ACM, 2011.

[73] Wikipedia. Effects of hurricane sandy in new york — Wikipedia, the free encyclopedia, 2014. http://en.wikipedia.org/wiki/Effects_of_Hurricane_Sandy_in_New_York [Online; accessed 14-Nov-2014].

[74] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[75] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

[76] Zhuoyao Zhang, Ludmila Cherkasova, Abhishek Verma, and Boon Thau Loo. Optimizing completion time and resource provisioning of pig programs. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 811–816. IEEE, 2012.

[77] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 316–327, New York, NY, USA, 1995. ACM.