UNIVERSITY OF CALIFORNIA, SAN DIEGO

Architectures for Stateful Data-intensive Analytics

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Dionysios Logothetis

Committee in charge:

Kenneth Yocum, Chair
Rene Cruz
Alin Deutsch
Massimo Franceschetti
Alex Snoeren
Geoffrey M. Voelker

2011

The dissertation of Dionysios Logothetis is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2011

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Dr. Kenneth Yocum, for his guidance during my graduate studies. I am grateful for his support over these years and for always trying to teach me how to become a better researcher. I have learned a lot from him.

I would like to thank all the members of my thesis committee for accepting to evaluate this work. I want to specially thank Dr. Alin Deutsch, Dr. Geoff Voelker, and Dr. Alex Snoeren for their advice and valuable feedback on many aspects of my research. I would also like to acknowledge Chris Olston and Ben Reed for their contribution to a large body of this work. Finally, I would like to thank my colleagues Chris Trezzo and Kevin Webb for their collaboration. It was great fun working with them.

Chapters 3 and 4, in part, are reprints of the material published in the Proceedings of the USENIX Annual Technical Conference 2011. Logothetis, Dionysios; Trezzo, Chris; Webb, Kevin C.; Yocum; Ken. The dissertation author was the primary investigator and author of this paper.

Chapters 5 and 6, in part, are reprints of the material published in the Proceedings of the ACM Symposium on Cloud Computing 2010. Logothetis, Dionysios; Olston, Christopher; Reed, Benjamin; Webb, Kevin C.; Yocum Ken. The dissertation author was the primary investigator and author of this paper.

VITA

2004	Diploma in Computer Science and Engineering, National Technical University of Athens, Greece
2007	Master of Science in Computer Science, University of California, San Diego
2011	Doctor of Philosophy in Computer Science, University of California, San Diego

PUBLICATIONS

Dionysios Logothetis, Chris Trezzo, Kevin Webb, Kenneth Yocum, "In-situ MapReduce for Log Processing", *USENIX Annual Technical Conference*, Portland, OR, June 2011.

Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin Webb, Kenneth Yocum, "Stateful Bulk Processing for Incremental Analytics", *1st ACM Symposium on Cloud Computing*, Indianapolis, IN, June 2010

Dionysios Logothetis, Kenneth Yocum, "Data Indexing for Stateful, Large-scale Data Processing", *5th International Workshop on Networking Meets Databases*, Big Sky, MT, October 2009

Dionysios Logothetis, Kenneth Yocum, "Ad-hoc Data Processing in the Cloud", *34th International Conference on Very Large Databases (demo)*, Auckland, New Zealand, August 2008

Emiran Curtmola, Alin Deutsch, Dionysios Logothetis, K.K. Ramakrishnan, Divesh Srivastava and Kenneth Yocum, "XTreeNet: Democratic Community Search", *34th International Conference on Very Large Databases (demo)*, Auckland, New Zealand, August 2008

Dionysios Logothetis, Kenneth Yocum, "Wide-Scale Data Stream Processing", *USENIX Annual Technical Conference*, Boston, MA, June 2008.

Yang-Suk Kee, Dionysios Logothetis, Richard Huang, Henri Casanova, Andrew Chien, "Efficient Resource Description and High Quality Selection for Virtual Grids", *5th International Symposium on Cluster Computing and the Grid*, Cardiff, UK, May 2005

ABSTRACT OF THE DISSERTATION

Architectures for Stateful Data-intensive Analytics

by

Dionysios Logothetis

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Kenneth Yocum, Chair

The ability to do rich analytics on massive sets of unstructured data drives the operation of many organizations today and has given rise to a new class of data-intensive computing systems. Many of these analytics are update-driven, they must constantly integrate new data in the analysis, and a fundamental requirement for efficiency is the ability to maintain *state*. However, current data-intensive computing systems do not directly support stateful analytics, making programming harder and resulting in inefficient processing.

This dissertation proposes that *state* become a first-class abstraction in data-intensive computing. It introduces *stateful groupwise processing*, a programming abstraction that integrates data-parallelism and state, allowing sophisticated, easily parallelizable stateful analytics. The explicit modeling of state abstracts the

details of state management, making programming easier, and allows the runtime system to optimize state management. This work investigates the use of stateful groupwise processing in two distinct phases in the data management lifecycle: (i) the extraction of data from its sources and online analysis, and (ii) its storage and follow-on analysis. We propose two complementary architectures that manage data in these two phases.

This work proposes In-situ MapReduce (iMR), a model and architecture for efficient online analytics. The iMR model combines stateful groupwise processing with windowed processing for analyzing streams of unstructured data. To allow timely analytics, the iMR model supports reduced data fidelity through partial data processing and introduces a novel metric for the systematic characterization of partial data. For efficiency, the iMR architecture moves the data analysis from dedicated compute clusters onto the sources themselves, avoiding costly data migrations.

Once data are extracted and stored, a fundamental challenge is how to write rich analytics to gain deeper insights from bulk data. This work introduces Continuous Bulk Processing (CBP), a model and architecture for sophisticated dataflows on bulk data. CBP uses stateteful groupwise processing as the building block for expressing analytics, lending itself to incremental and iterative analytics. Further, CBP provides primitives for dataflow control that simplify the composition of sophisticated analytics. Leveraging the explicit modeling of state, CBP executes these dataflows in a scalable, efficient, and fault-tolerant manner.

Chapter 1

Introduction

1.1 The "big data" promise

Data is emerging as a new science, a result of the unprecedented increase in the amount of digital information produced today and the realization of innovative ways to extract value from it. Technological advances have led to an abundance of digital information sources that constantly generate data: Internet users producing valuable content (browsing history, e-mail exchanges, online purchases, social network interactions etc.), ubiquitous digital devices generating information (mobile phones, cameras, RFID sensors etc.), and advanced scientific instruments producing hundreds of terabytes of data [48]. The information hidden in this vast amount of data has the potential to transform human endeavors, like science, business, and finance, in ways that were previously unimagined. For instance, public health organizations now monitor trends in search engine queries to detect flu epidemics [40]. Social networks analyze user interactions to do targeted advertising [27] and retailers track sales data to understand consumer behavior [63], recommend products, and increase profit. Banks analyze financial and personal information to detect fraud [24], while stock traders base their decision making on the analysis of real-time financial data [78].

Realizing this potential with traditional data management approaches has been challenging. For decades, Database Management Systems (DBMSs) have been the dominant approach for data storage and analysis. Databases were de-

signed to store data using well-defined structure, a schema, carefully determined in advance. DBMSs model data as relations [30], essentially tables, and typical uses of a DBMS include the transactional access of these data to ensure data integrity. For instance, banks use DBMSs to store tables of bank accounts, and guarantee account information integrity (e.g. during concurrent account withdrawals). Relational algebra [30] and the SQL language have been the main tool for manipulating this type of data. The strength of SQL is mainly in updating or retrieving data from tables and producing simple reports through data summaries. In fact, years of work on parallel databases and relational query optimization have resulted in DBMSs that can store and analyze large amounts of structured data efficiently.

However, meeting the promise of big data depends on a fundamentally different kind of analysis. First, there is a growing demand for complex analytics on unstructured data, such as text (e.g. web pages, e-mails, server logs), video (e.g. surveillance videos, YouTube uploads) and images (e.g. photo albums, MRI scans, satellite images). Such data are often difficult to represent as relational tables, the core abstraction in DBMSs. Furthermore, transactional access is often unnecessary since these data are archived and analyzed in bulk, and these analytics may not be expressible with relational algebra [72, 35]. For instance, several graph analyses found in social networks or web mining applications are hard and sometimes impossible to express using relational algebra [80]. Other such applications include machine learning [29, 64] and natural language processing [20] algorithms. A constrained programming model may hinder users from unlocking the potential value of their data through sophisticated analysis [72, 71].

Second, there has been an enormous increase in the scale of the data, with scientists estimating that digital data are growing at least as fast as Moore's Law [12]. Studies have shown that this unstructured data is accumulating in data centers at three times the rate of traditional transaction-based data [65]. For instance, Facebook reported in 2009 that it collected data at a rate of 15TB per day [59], a number that within less than two years has increased to 130TB per day [49], while YouTube integrates 24 hours of new video a minute [4]. Traditional DBMSs often cannot manage data at this scale, or in some cases cannot do so in

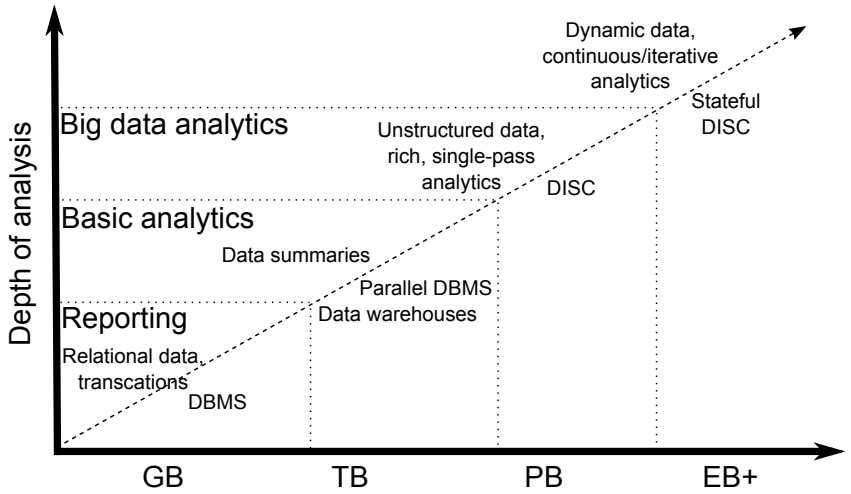


Figure 1.1: An illustration of the evolution from traditional data processing to "big-data" analytics, including the evolution in the relational data management technology.

a cost-efficient way. Figure 1.1 shows this evolution on data analysis, from simple reports on small-scale data to what we call today "big data analytics": rich analytics on very large unstructured data.

1.2 Data-intensive computing

The emergence of big data analytics has given rise to a new class of *data-intensive scalable computing (DISC)* [21] systems that address the above challenges. Systems like MapReduce [34], Hadoop [8] and Dryad [46] are representative of this class. These systems have turned a challenge, the scale of the data, into an advantage by enabling users to obtain deep insights from large data. They are used for a wide variety of data analytics by organizations ranging from educational institutions and small enterprises to large Internet firms that manage petabytes of data [3, 34, 46].

These systems assume no structure in the data and support sophisticated analytics. Unlike the declarative model of the SQL language, in these models users express the application logic using general-purpose languages, like Java, that allow arbitrary data structures and computations. This model is more powerful and

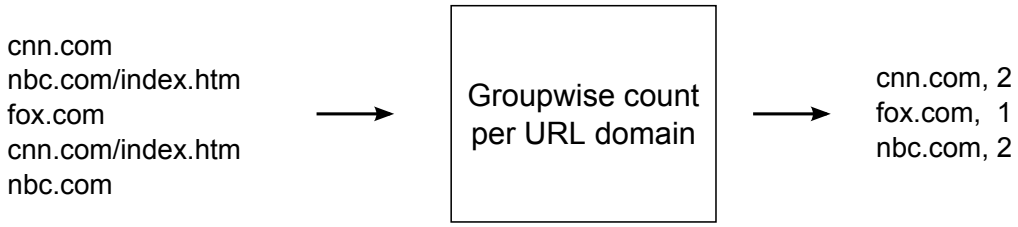
$$\text{group } (r_{in}) \rightarrow \text{key}$$
$$\text{process } (\text{key}, \{r_{in}\}) \rightarrow r_{out}$$

Figure 1.2: Groupwise processing requires users to specify: (i) how to group input records, and (ii) the operation to perform on each group. To group input records, users specify a function that extracts a *grouping key* for every record. To specify the operation, users implement a generic operator that receives as input the grouping key and all the input records that share the same key.

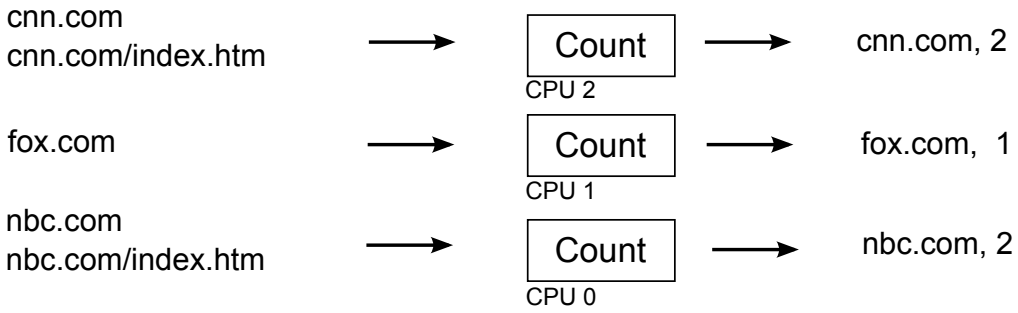
intuitive for a variety of applications. It has been used to implement analytics like machine learning [5, 64], natural language translation [20, 58] and graph mining [31, 51].

At the core of these models is *groupwise processing*, a programming abstraction that makes it easy for users to express the data-parallelism inherent in many analytics. Many analyses can be expressed as the same computation on multiple independent *groups* of data. Groupwise processing requires users to specify only (i) how to partition the data into groups, and (ii) the computation to perform on each group. Users specify these operations through two functions illustrated in Figure 1.2. For instance, search engines extract URL links from web pages, group them according to their URL domain, and count the URLs in each group to measure the popularity of different URL domains. Figure 1.3(a) illustrates this groupwise count operation.

This abstraction allows these DISC systems to analyze massive amounts of data by distributing the computation across large compute clusters. For instance, Figure 1.3(b) shows how groupwise processing divides the count operation to independent operations that can execute in parallel on separate CPUs. As opposed to using reliable, high-performance hardware, these systems scale by using pools of commodity machines in a fault-tolerant manner. They employ a simple fault tolerance model that can restart individual computations when machines fail. This allows DISC systems to scale to virtually any amount of data simply by employing more machines.



(a) Logical representation of a groupwise count



(b) Parallel evaluation of the groupwise count on multiple CPUs

Figure 1.3: Groupwise processing is a core abstraction in DISC systems. Figure (a) shows the semantics of a groupwise count operation. Here input URLs are grouped according to the domain they belong to and the operation on each group is a count. Figure (b) illustrates the physical execution. Grouping allows independent operations to execute in parallel.

1.3 Stateful data-intensive computing

While DISC systems can scale to large data, it is becoming apparent that scalability alone is not sufficient for certain applications. Many applications must now manage large data by following a fundamentally different programming approach. As the rightmost part of Figure 1.1 shows, several applications no longer view data analytics as a "one-shot" process, rather as a process that must constantly update the analysis. For instance, search engines must crawl new web pages and update their indices to deliver up-to-date results, while iterative graph mining analytics, like PageRank [62], must repeatedly refine the result of the analysis across iterations.

However, current DISC systems are not designed for update-driven analytics, often resulting in inefficient data processing. For instance, these DISC systems can re-generate a search index from the entire web corpus simply by adding com-

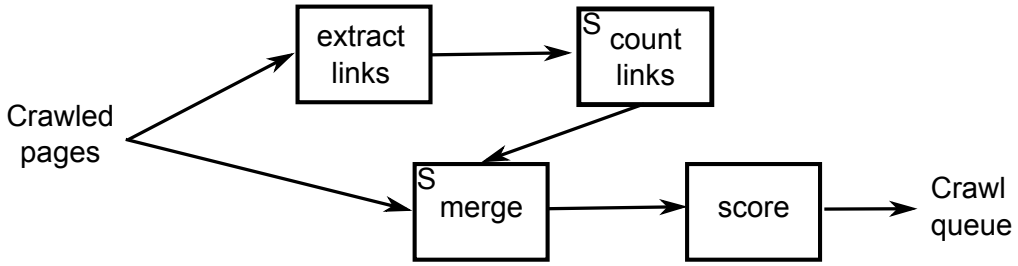


Figure 1.4: A dataflow for computing a web crawl queue. Stateful stages are labeled with an S.

pute power when more pages are crawled. Processing petabytes of data only to update a small part of the index each time is inefficient [66]. This approach discards prior computation, wasting CPU cycles and network resources, and increases energy consumption and monetary cost.

Instead, to run these analytics efficiently, users have to program these applications in a different manner. Critical to the efficiency of these analytics is the ability to maintain *state*, derived data that persist and are re-used across executions of the analysis. Stateful computations arise in at least three kinds of analytics: incremental, continuous, and iterative. For instance, analytics that are amenable to incremental computation re-use previous computations (e.g. the previously computed search index) to update the analysis instead of recomputing from scratch when new data arrive. Iterative analytics refine the result of the analysis by repeatedly combining the same input data with the results of prior iterations. Examples of such iterative applications include PageRank [62], data clustering [37], social network analytics [77], and machine learning algorithms [5].

1.3.1 An example: web crawl queue

To illustrate the importance of state, we use a toy web data-mining application. In general, such analyses consist of multiple processing *stages* that comprise a larger *dataflow*. In particular, we use a *crawl queue*, a common and important dataflow in search engines that determines the crawl frontier, the pages to crawl. At a high-level, the dataflow parses crawled web pages to extract URL links and counts the frequency of these extracted URLs. Subsequently, it compares the ex-

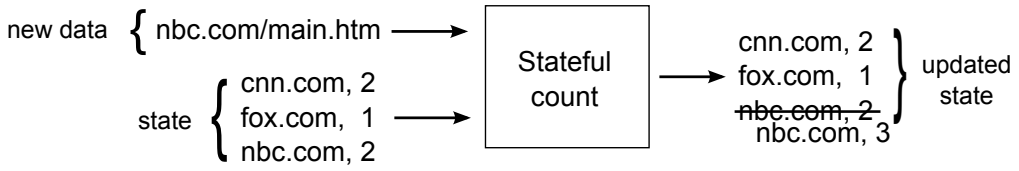


Figure 1.5: An incremental approach to update the URL counts in the crawl queue dataflow. This approach re-uses previously computed URL counts.

tracted URLs against the already crawled pages and outputs uncrawled pages with the highest score (URL frequency). Figure 1.3.1 shows the different stages of this dataflow.

Since the web changes over time (e.g. users upload new pages), a search engine must continuously run this dataflow. This means, for instance, updating all the counts computed by the *count links* stage. One approach is to repeatedly use the same dataflow every time new pages are crawled, re-processing all web pages from scratch. However, this approach is inefficient since a large fraction of the analysis may not be affected by newly crawled pages. In the *count links* stage, only a few counts may be updated.

Instead, a user may program such an application to re-use prior results by maintaining previously computed URL counts as state. Figure 1.3.1 shows this approach. This way, when new web pages become available for processing, we can simply increment the counts of the affected URLs instead of re-computing all counts from scratch. Such a stateful approach produces the same result, reduces the processing time and uses fewer compute resources.

1.3.2 Challenges in managing state

Stateful analytics present new data management challenges. Current DISC systems where designed for "one-shot" processing as they did not consider update-driven computations. Attempting to retrofit stateful programming in these systems has the following implications.

- **State management:** To turn a stateless application into a stateful one, users often modify their analysis to manually manage state by storing it to external stor-

age systems, like distributed file systems or databases. This makes programming more error-prone and less portable, since the details of state management are not abstracted from the user. The user has to take into consideration the specifics of the storage system, which may vary among computing environments. For instance, porting an application to a different environment depends on the availability of the same storage systems.

- **Fault tolerance:** While DISC systems are designed to handle failures transparently, storing state to a storage system external to the processing system may require extra effort from the user to handle storage failures. This adds programming complexity and distracts users from specifying the data processing logic.

- **Dataflow orchestration:** To develop continuous dataflows, applications must manually orchestrate these potentially complex multi-step computations. For instance, as new data arrive, applications must decide when to execute each stage, how much data to process, and synchronize execution across stages. Implementing ad-hoc synchronization logic on large dataflows can be an error-prone task. Instead, this task should be simplified and automated through programmatic control.

- **Efficiency:** Retrofitting stateful programming techniques in systems that are not designed for this purpose results in inefficient processing. As pointed out in the example in Section 1.3.1, often only a small fraction of the state needs to be updated when new data arrive. However, existing DISC systems treat state just like any other data input. As they repeatedly execute the analysis, they must re-read and re-process the entire state. This wastes resources and results in processing time that is proportional to the size of the constantly growing state, not the amount of changes to the state. Instead, to allow efficient processing a system should directly support stateful analytics.

1.4 Architectures for stateful analytics

This dissertation addresses the above challenges in large-scale stateful analytics. This work aims to build systems for stateful analytics in different scenarios

$$\text{group } (r_{in}) \rightarrow \text{key}$$
$$\text{process (key, } state_{old}\text{, } \{r_{in}\}) \rightarrow (state_{new}, r_{out})$$

Figure 1.6: Stateful groupwise processing extends the groupwise processing construct by integrating state in the model. A user-specified operation now has access to state that can be used to save and re-use computations.

that process unstructured data. The goal of this dissertation is to design (i) programming models that allow users to easily write sophisticated stateful analytics and (ii) scalable, efficient and fault-tolerant runtime systems.

This dissertation proposes that state become a first-class abstraction, and introduces *stateful groupwise processing*, a programming abstraction that integrates data-parallelism and state. Figure 1.4 shows the integration of state in the groupwise processing model. While groupwise processing hides the details of parallel execution, stateful groupwise processing abstracts the details of state management. Tasks like reliably storing and accessing state are left to the underlying runtime system, making stateful applications easier to program and more portable. At the same time, by explicitly modeling state, this abstraction gives the system the opportunity to optimize state management, resulting in reduced processing times and efficient resource usage.

In this dissertation, we explore the application of stateful groupwise processing in two distinct phases of the data management lifecycle: (i) the extraction of data from the sources and online analysis, and (ii) the storage and follow-on analysis. The analytics in both phases present the general challenges outlined above: they must perform sophisticated, update-driven analysis in a scalable, efficient, and fault-tolerant manner. However, the analytics in these scenarios serve different purposes and admit different solutions.

1.4.1 Continuous ETL analytics

The first phase in data analytics is the extraction of data from their sources and preparation for storage and follow-on analysis, what is often referred to as an *Extract-Transform-Load (ETL)* process [69, 76]. The ETL process collects raw data from sources, like click logs from web servers and news feeds from social

network servers. Such raw data must usually undergo an initial analysis that includes filtering, transforming data into an appropriate format, and summarizing data before they are stored.

The ETL process is often used to obtain quick insights by analyzing data in an *online* fashion [44, 32]: online analytics often require the analysis to return results before all data have been collected, to provide timely results. Online analysis is important for applications like failure detection that must be responsive even when not all data are available. A fundamental challenge in online analytics is the ability to assess the accuracy of the analysis when data are incomplete. At the same time, unlike "one-shot" batch processing, the ETL process must continuously update the result of the analysis as data are generated. A challenge here is how to program ETL analytics on data streams.

To address these challenges, this dissertation proposes *In-situ MapReduce (iMR)*, a model and architecture for online ETL analytics. iMR (i) provides a programming model that combines stateful groupwise processing with windowed processing for unstructured data and (ii) introduces a novel metric that helps users assess the accuracy of their analysis in online analytics and allows applications to trade accuracy for timeliness, and (iii) proposes an architecture for moving ETL analytics from dedicated compute clusters to the data sources themselves to avoid costly data migration.

1.4.2 Stateful bulk data processing

Once data are collected by the ETL process, users run analytics to obtain deeper insights. In this phase, the interest shifts from the ability to obtain quick insights toward the ability to do richer analysis on bulk data. Unlike ETL analytics that filter or summarize data, applications now run sophisticated dataflows that must continuously integrate large batches of data in the analysis. Iterative analytics, like PageRank [62], must make multiple passes over large data sets. A fundamental challenge that programmers face now is how to program these sophisticated analytics and how to execute them efficiently on bulk data.

To address these challenges, this dissertation introduces *Continuous Bulk*

Processing (CBP), a programming model and architecture for sophisticated stateful analytics. CBP (i) introduces a model for stateful groupwise processing on bulk data, (ii) provides primitives that allow users to orchestrate sophisticated stateful dataflows, and (iii) introduces an architecture that leverages the programming model to execute these dataflows in a scalable, efficient, and fault-tolerant manner.

1.5 Contributions

In summary, this dissertation presents two complementary architectures that incorporate state to address the distinct challenges in the two phases of data management. This dissertation makes the following contributions:

- **State as a first-class abstraction:** This dissertation introduces *stateful groupwise processing*, an abstraction that integrates data-parallelism and state, allowing users to write sophisticated, easily parallelizable stateful analytics. By abstracting the notion of state, users no longer have to manage state manually, and the underlying system can optimize state management.

- **Model for online ETL analytics:** This dissertation presents *In-situ MapReduce (iMR)*, a programming model that supports stateful groupwise processing tailored for continuous ETL analytics on unstructured data. The iMR model extends the MapReduce [34] programming model with a *sliding window* construct, a concept that arises naturally in analytics on data streams. By combining state and windows, the model allows efficient processing through incremental updates to the analysis.

- **Architecture for in-situ processing:** The iMR architecture processes data in-place, obviating the costly and often unnecessary migration of data from their sources to centralized compute clusters. This approach reduces resource usage and allows faster analysis by processing data as they are generated. We evaluate our in-situ architecture and show how it can provide efficient analysis in real application scenarios.

- **Metric for systematic characterization of incomplete data:** To support online analysis, this dissertation introduces a novel fidelity metric that (i) allows users to assess the impact of incomplete data on the fidelity of the analysis and (ii) provides a flexible way to express fidelity requirements and trade fidelity for result availability. Further, we provide general guidelines on using the metric. Through our evaluation we validate its usefulness in real applications.

- **Model for stateful bulk data analytics:** The core component of *Continuous Bulk Processing (CBP)* is a generic stateful groupwise operator that allows users to write incremental and iterative analytics. By abstracting state management, CBP simplifies programming. CBP extends the concept of grouping with new constructs to efficiently support a wide range of iterative analytics, such as graph mining. Further, the CBP model provides primitives that allow the composition and orchestration of sophisticated stateful dataflows.

- **Efficient stateful groupwise processing:** The fundamental mismatch between current DISC models and stateful computations results in inefficient processing. The CBP runtime leverages the explicit modeling of state, to optimize state management. Through a series of optimizations CBP improves processing times and reduces computation and network usage. By comparing against state-of-the-art DISC systems, our evaluation validates the benefits in performance and efficiency that the explicit modeling of state offers. In many cases, CBP reduces the running time and the network usage by at least 50%.

1.6 Outline

The rest of the dissertation is organized as follows. Chapter 2 presents basic concepts and systems that this work builds upon, like groupwise processing and DISC architectures, and reviews related work on stateful analytics. Chapter 3 introduces the iMR programming model for continuous ETL analytics, while Chapter 4 presents the implementation and evaluation of the iMR prototype. In Chapter 5, we introduce the CBP model for stateful analytics on bulk data. Chap-

ter 6 describes the design and implementation of CBP, as well as an evaluation with real-world applications. Finally, Chapter 7 summarizes the dissertation.

Chapter 2

Background and related work

To provide scalable analytics, this work builds upon the concept of data-parallelism and the architectural principles of DISC systems, like MapReduce and Hadoop. This chapter reviews the basic programming abstractions and architectural principles that allow scalable analytics on unstructured data. We describe the groupwise processing abstraction for expressing data-parallel analytics in the context of the MapReduce programming model. Further, this chapter gives a brief overview of the design of DISC systems, like MapReduce and Hadoop. Lastly, this chapter discusses related work in stateful large-scale analytics.

2.1 Groupwise processing

Grouping is a concept that appears naturally in many real-world data analysis scenarios [26] and allows parallel execution of the analytics. In Section 1.2 we illustrated this with a simple web analytics example. Because of its importance in data analysis and the opportunity for scalable execution, grouping is a core construct in various programming models. The `GROUP BY` operation in the SQL language is an example. Grouping also underlies the programming models in DISC, like MapReduce [34] and Hadoop [8], but also in higher-level, SQL-like languages layered on top of these processing systems [82, 81, 61, 9].

While groupwise processing appears in various flavors in these programming models, there are two basic parts of every groupwise operation: (i) specifying how

to partition input records into groups and (ii) specifying the function to apply on the records of every group. These operations were shown in Figure 1.2. The output of a groupwise operation is a list of group-value pairs, with every pair corresponding to a distinct group and the result of applying the user-specified function on the group. This was shown in the example of Figure 1.3(a).

For instance, in the SQL language a user partitions an input table by specifying one of the table columns as the partitioning attribute. Part of the group operation specification is also an aggregate function, like `SUM`, to apply on each group. The result of a `GROUP BY` query in SQL is a table with a row for each group that contains the result of the aggregate function. Parallel DBMSs may leverage this construct to distribute the evaluation of `GROUP BY` aggregate queries across multiple machines.

Notice that the `GROUP BY` construct in SQL assumes some structure in the data. For instance, the grouping operation assumes the existence of a column in the table that is used as the partitioning attribute. Next, we show how the MapReduce programming model extends the basic groupwise processing abstraction with constructs that allow analysis on unstructured data.

2.2 The MapReduce programming model

MapReduce [34] is a programming model designed by Google to mine large web datasets, like crawled pages and has been used for tasks like generating Google's search index and rank web pages. The motivation for the MapReduce model was the need for a simple abstraction that can leverage the power of hundreds of thousands of machines available in large data centers, without exposing the complexities of parallel processing in such an environment. At its core, a MapReduce program performs groupwise processing, providing the tools for partitioning data and performing arbitrary computations on a per partition basis.

A MapReduce program consists of two user-specified functions, the *Map* and the *Reduce* function. Input data in a MapReduce program consist of a set of input records, modeled as key-value pairs. The Map function takes an input

map (k,v) → {(k',v')}
reduce (k', {v'}) → {v'}
combine (k', {v'}) → {v'}

Figure 2.1: A MapReduce computation is expressed using two main functions. The *Map* is used to extract information from raw data and determine the partitioning of data into groups. The *Reduce* function is used to aggregate data in the same group. The *Combine* function is used as an optimization to reduce the size of the Map output.

```
                                    REDUCE(Key k, Value vals[])
MAP(Key k, Value text)              1    count:=0;
1    foreach word w in text         2    foreach v in vals[]
2        emit(w, "1");              3        count++;
                                    4    emit(k, count);
```

Figure 2.2: A MapReduce example program that counts word occurrences in text. The Map function receives as input the document text and extracts words from it. For every word, it emits an intermediate key-value pair, with the key set equal to word and the value set to "1", indicating a single occurrence. For every intermediate key, the Reduce function counts the number of occurrences of the corresponding word.

key-value pair and emits zero or more *intermediate* key-value pairs. MapReduce applies the Map function on every input key-value pair, and subsequently groups intermediate key-value pairs according to their key. The Reduce function accepts as input an intermediate key and all the intermediate values that share the same key. The Reduce function may aggregate or transform in an arbitrary way the group of values and emit one or more output values. MapReduce applies the Reduce function on every distinct intermediate key. Figure 2.2 shows the definition of the Map and Reduce functions.

In Figure 2.2, we show how a user can write a MapReduce program to count the frequency of each word in a large body of documents. Here an input key-value pair represents a document, where the key is the document name and the value is the document contents. The Map function extracts the words from the documents

and outputs a set of intermediate key-value pairs, one for each word, where the key is the word itself and the number of occurrences of each word, here the number '1', indicating a single occurrence. These intermediate key-value pairs are grouped according to key. The Reduce function sums the number of occurrences and emits the total sum for every word.

While the Map and Reduce function comprise the core of the model, users may optionally specify a *Combine* function. The MapReduce system may use the Combine function as an optimization to decrease the amount data shipped across the network. As Section 2.3 shows, MapReduce systems send the output of the Map functions across the network from the processes that execute the Map function (Map tasks) to the processes that execute the Reduce function (Reduce tasks). MapReduce uses the Combine function to calculate partial results from the Map output before they are passed to the Reduce function. Often, the same key appears multiple times in the output of a Map tasks. Reduce functions that are commutative and associative allow the final result to be computed by combining partial results. The Combine function may decrease the size of the Map output before the Reduce function is applied.

The Map function is generally used to extract information from raw data (e.g. words from text), while the Reduce function is typically used to aggregate groups of values from the intermediate data (e.g. count or sum values). Although in the previous example the Map and Reduce functions perform simple operations, a user may implement arbitrary application logic. This flexible model gives users the ability to implement sophisticated analyses, leveraging the capabilities of general purpose languages, like C++ and Java. This makes the MapReduce model more expressive than SQL, allowing a wider range of data analytics. For instance, it has been used to do satellite image stitching, render map images, and create inverted indices from web documents [34, 35].

Programmers also use the Map function to determine how to partition input data into groups. The Map function essentially implements the *group* operation shown in Figure 1.2 by specifying the intermediate key. Unlike grouping in relational databases, where the grouping attribute must be one of the table columns,

the Map function can be used to group input data in an arbitrary way. For instance, a Map function that extracts URLs from web pages may specify the URL itself as the grouping key (e.g. to count per URL), or it may extract the URL domain as the grouping key (e.g. to count per domain), as shown in Figure 1.3.

This flexibility is a key aspect of the MapReduce model. It allows users to write analytics without assuming any structure on the data. Unlike databases, where data have a well defined schema, determined in advance, MapReduce allows users to interpret input data in ad-hoc manner.

2.3 MapReduce architecture

While the MapReduce programming model allows users to express parallelism in the analysis, the role of the MapReduce runtime is to hide the details of parallel processing from the user. The runtime system handles tasks like program execution, machine communication and fault tolerance transparently, allowing programmers focus on the application logic. In this section, we review basic design principles of the MapReduce architecture that allow MapReduce to execute programs in a scalable and fault tolerant manner.

2.3.1 Program execution

The main tasks of a MapReduce runtime are to (i) execute the Map function on the input data, (ii) group the intermediate data according to key, and (iii) call the Reduce function for every group.

The execution of the Map function on the input data is a data-parallel operation. Input data are typically partitioned into *input splits* that are distributed across machines. An input split may be, for instance, a document. For every such input split a *Map task* is responsible for reading the split, converting it into input key-value pairs, and calling the Map function for every input pair. This way, the Map task generates a set of intermediate key-value pairs that are stored on the local disk. Every machine outputs a subset of all the intermediate key-value pairs that must collectively be grouped and reduced.

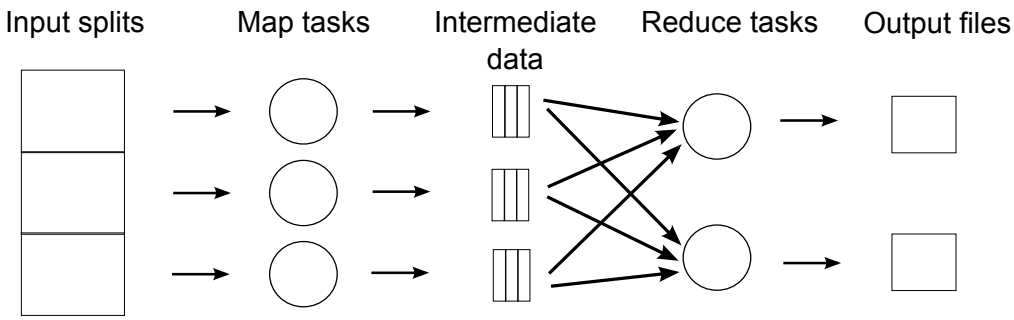


Figure 2.3: The execution of a MapReduce program. Data flow between Map and Reduce tasks. Map tasks process input splits in parallel and output intermediate key-value pairs. Intermediate key-value pairs from each Map output are partitioned across multiple Reduce tasks according to their key. Reduce tasks are also executed in parallel.

In order to parallelize the grouping and reduction as well, the system executes in parallel a number of *Reduce tasks* with each Reduce task being responsible for grouping and reducing a *partition* of all the intermediate key-value pairs. Every Map task partitions its intermediate key-value pairs into R sets according to a partitioning function, usually a hash on the key. The system starts R Reduce tasks and assigns each Reduce task to a partition. Every Reduce task fetches its corresponding partitions from all the Map tasks, and groups key-value pairs in them according to key. This operation is called the *shuffle* phase. After the shuffle phase has completed, the Reduce task can execute the Reduce function on every group of values. Every Reduce task writes the output in its own output *partition file*. Figure 2.3.1 displays the execution of a MapReduce program.

Because both the Map and Reduce operations are data-parallel, the system can easily scale to large data by using more machines. As the input data increases, the system distributes the input splits across more machines. The runtime can then divide the execution of the Map function across more Map tasks. Similarly, to scale the Reduce phase, the runtime can increase the number of partitions and the corresponding Reduce tasks.

2.3.2 Fault tolerance

The MapReduce system scales by distributing the analysis to potentially thousands of commodity machines. Because failures are common in such an environment, it is critical for the runtime to tolerate failures.

To provide fault tolerant computations, the MapReduce system must ensure that (i) input data are not lost due to failures, and (ii) a MapReduce program can complete despite failures. To ensure input data availability, MapReduce architectures leverage distributed file systems like the Google File System (GFS) [39] and the Hadoop Distributed File System (HDFS) [8] that replicate data across multiple machines.

To ensure that a program completes in the event of failures, MapReduce employs a restart mechanism. A task is the granularity at which MapReduce restarts processing. MapReduce leverages the partitioning of the analysis to independent operations, executed by Map and Reduce tasks, and handles failures by re-executing individual tasks. The MapReduce runtime monitors the status of both machines and running tasks across the compute cluster, and can re-start all the tasks that did not complete successfully on a new machine.

The MapReduce fault tolerance mechanism ensures that a program will always make progress toward completion even under high failure rates. Map tasks typically save their output to local disks and in the event of a Reduce task failure, the restarted Reduce task can simply re-fetch the Map output. This simple mechanism prevents long running programs from aborting execution and re-starting from scratch because of single machine failures.

To achieve fault tolerance at a scale of thousands of commodity machines, a basic assumption of the MapReduce fault tolerance model is that the execution of Map and Reduce tasks is free of side-effects. For instance, tasks are not allowed to communicate with each other through files, or with external services. This simplifies fault tolerance and ensures that re-starting a task does not require coordination with other tasks or external services, a process that can be complicated and error-prone.

However, the MapReduce model allows users to implement arbitrary logic

inside a MapReduce program. For instance, as pointed out in Section 1.3.2, users may abuse this flexibility to implement stateful computations by storing state to external storage systems. This introduces side-effects, and may leave a program in an inconsistent state in the event of a task failure and restart. Ensuring correctness under failures is now the responsibility of the user, adding programming complexity. In Chapter 6, we show how the explicit modeling of state eliminates this problem.

2.4 Stateful analytics on unstructured data

While the MapReduce system provides a scalable platform for large-scale analysis, it is not designed for stateful computations. Recent work has investigated the use of state for efficient incremental or iterative analysis on large-scale unstructured data. These approaches vary in the target application scenarios and the way they incorporate state in the programming model. Several approaches are targeted toward iterative analytics, while others focus on efficient processing through incremental analytics. Additionally, certain approaches incorporate state in manner transparent to the user, while others allow user-specified stateful programs. This section gives an overview of these approaches and compares them with the proposed solution of this thesis.

2.4.1 Automatic incrementalization

Incremental analysis through computation re-use is critical for the efficient processing of data that change over time. Various systems aim to incrementally update the analysis in an automatic manner [43, 42, 67, 22]. Unlike the approach proposed in this thesis, these systems do not require users to write custom incremental computations using state. Instead, users specify analytics as one-shot dataflows. The runtime systems are responsible for maintaining any necessary state and for automatically transforming the dataflow to an incremental one.

These approaches are based on the concept of *memoization*, the caching and re-use of intermediate results. In general, these systems represent analyt-

ics as dataflows consisting of multiple sub-computations and cache the results of individual sub-computations. When only part of the input data changes across dataflow invocations, the input of some sub-computations in the dataflow may remain the same as in previous invocations. Memoization avoids re-running these sub-computations by re-using the cached results.

These techniques may be applied, for instance, to MapReduce programs. A MapReduce program can be represented as a dataflow consisting of Map and Reduce tasks as in Figure 2.3.1. The output of such a program can be incrementally updated by memoizing the output of Map and Reduce tasks. For example, if only one of the input splits changes, we need to re-execute only the corresponding Map task. Reduce tasks can re-use the memoized output of the rest of the Map tasks. Similarly, because only a fraction of the intermediate data may change, we may need to re-run only a fraction of the Reduce tasks. While these approaches do not require users to devise incremental analytics, making programming easier, they restrict the range of workloads that may benefit from incremental processing. For instance, even if a single input split changes in a MapReduce program, it is possible that all Reduce tasks have to be re-executed despite the caching of intermediate results. As we show in Chapter 5, explicit incremental programming using state, as described in the example of Section 1.3, can improve efficiency in such cases by incrementally evaluating the Reduce functions using previous results.

2.4.2 Iterative analytics

Iterative analysis, like several machine learning algorithms, are inherently stateful. They repeatedly refine the analysis by combining a static input set with the analysis result of previous iterations. These analytics must re-use both the static input set and the analysis result across iterations.

The Spark [83] system supports iterative analysis by allowing users to specify distributed read-only state, used to store the static input set. The modeling of this static state allows the system to optimize iterative analytics by avoiding the re-loading of the input data from a file system in every iteration. Instead, Spark maintains this distributed data set in memory, allowing processing tasks to

access it fast. Twister [36] is an extension to the MapReduce system that also supports iterative analytics. Like Spark, it gives programs access to static state. In Twister, Map and Reduce tasks persist across iterations, amortizing the cost of state loading.

While these approaches improve the efficiency in iterative analytics, they restrict the ability to re-use computation only to the static input data set. Instead, our approach provides a more general abstraction of state that allows the update of the state. Iterative analytics may use this state abstraction to model not only the static input set, but also the iteratively refined result of the analysis.

2.4.3 Models for incremental analytics

Unlike the approaches described above, certain systems require users to write custom incremental programs. Like the solution proposed in this thesis, these systems give users access to state, allowing them to store and re-use computation results. Because transparent incremental analysis is not possible for certain analytics, the explicit access to state allows a wider range of analytics to leverage the opportunity for efficient processing through incremental analysis.

The Percolator system [66] adopts this approach. Percolator is motivated by the need to incrementally maintain an index of the web as web documents change over time. In Percolator, programs are structured as sequences of *observers*, pieces of code that are triggered when user-specified data are updated. For instance, a change to an existing document may trigger an observer that updates the link counts, as in the example of Section 1.3.1, of the documents affected by the change. An observer can access and modify the data repository. Such modifications may subsequently trigger downstream observers. Percolator allows multiple observers to access and modify the repository in a shared manner. Because multiple observers may be triggered at the same time, Percolator provides transactions, to allow users to reason about the consistency of the data.

In Percolator, data are stored in tables that persist across time. For instance, the web indexing application may use the tables to store web documents. Analytics can access and modify these tables when changes in the input happen.

Tables provide random access, allowing the selective update of analysis results when small changes occur in the input data. For example, newly crawled web documents might affect only part of the stored documents in the index. This avoids accessing the entire web document body when small changes happen. As we also show in Chapters 5 and 6, the ability to selectively access state is critical for efficient incremental processing.

While Percolator allows multiple applications to share the same state, stateful groupwise processing provides a simpler state model where state is private to a single application and concurrent state modifications are not allowed. This simplifies programming and obviates the need for transactions that can limit scalability [66]. At the same time, our model leverages the groupwise processing abstraction to easily expose data-parallelism in analytics.

Chapter 3

Stateful online analytics

The purpose of an ETL process is to continuously extract data from its sources and prepare data for follow-on analysis. Analytics in this phase include filtering, transforming raw data into an appropriate format for storage, and summarizing data. ETL analytics are intended to allow applications to prepare data. By analyzing data in an online fashion, the ETL process also allows users to get quick insights from the data.

The dominant approach for ETL analytics today is to migrate data from their sources to centralized clusters dedicated to running analytics. Typically, applications store data on distributed file systems, like the Google File System (GFS) [39], or the Hadoop Distributed File System (HDFS) [8] and execute the analytics with batch processing systems, like MapReduce and Hadoop. This approach is illustrated on the left-hand in Figure 3.1.

However, this approach has certain drawbacks. First, these batch processing systems are not designed for continuous data processing. Their programming models support only one-shot processing on well defined input, making programming continuous applications harder and less efficient. For instance, users may have to devise custom and often complicated application logic to repeatedly update the analysis with new data. Instead, continuous ETL analytics require programming models that directly support processing on streams of unstructured data. At the same time, these analytics are update-driven and present opportunities for efficient processing through incremental computation that one-shot programming models

fail to leverage.

Second, transferring bulk data to a centralized compute cluster limits the timeliness of the analysis and is inefficient. Due to the size of the data it is often impossible to migrate all data in a timely fashion. Even with well-provisioned infrastructure, this bulk data transfer stresses both the network and the disks as data are migrated from the sources to the dedicated cluster. Furthermore, this bulk data transfer is often unnecessary since the ETL process may eventually reduce the size of the data through filtering or summarization, making this approach inefficient.

Third, these approaches do not support online analysis through partial data processing. Often data may become unavailable due to load at the sources or failures. For instance, server logs may become unavailable when servers become unreachable. In this case, current approaches must sacrifice result availability and timeliness, waiting until all data are available. Alternatively, current systems may choose to blindly return partial results, degrading arbitrarily the accuracy of the results. However, applications have no means to assess the impact of partial data on the accuracy of the analysis, rendering the results practically unusable.

To address these challenges, this chapter introduces In-situ MapReduce (iMR), a programming model and architecture for online ETL analytics. iMR leverages the flexibility and inherent parallelism of the MapReduce model, and extends the model for continuous unstructured data. The iMR model combines (i) stateful groupwise processing to improve efficiency through incremental processing and (ii) *windowed processing* to simplify the programming of continuous data. To support online analytics, the iMR model allows processing of partial data and introduces a novel, general purpose metric that allows applications to (i) assess the analysis fidelity, and (ii) explicitly trade fidelity for latency. To avoid expensive data transfers, the iMR architecture moves the ETL analytics from dedicated clusters on to the data servers themselves.

In the rest of this chapter, we show how the iMR model allows sophisticated ETL analytics on continuous data and describe the mechanisms that allow scalable, efficient, and timely execution of the analytics in-situ.

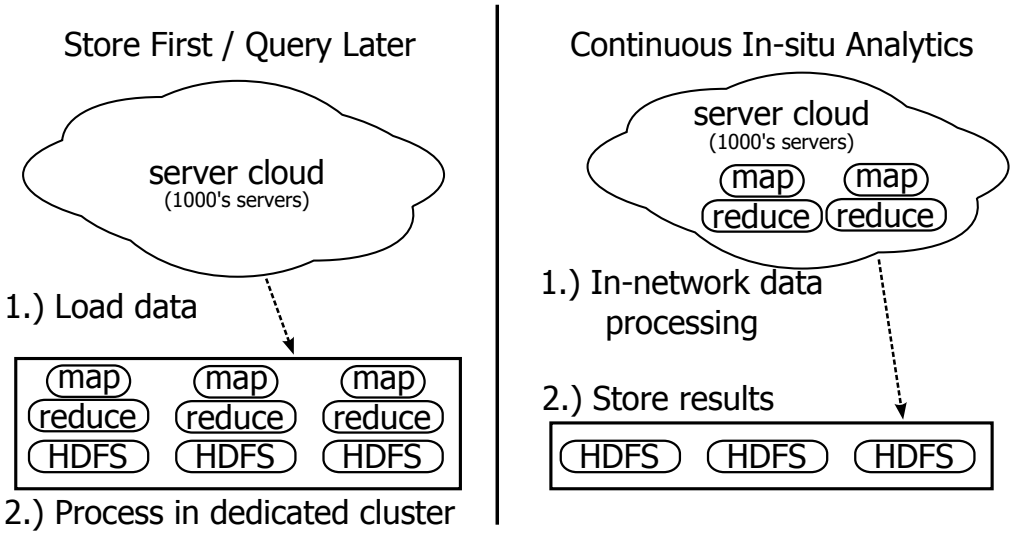


Figure 3.1: The in-situ MapReduce architecture avoids the cost and latency of the store-first-query-later design by moving processing onto the data sources.

3.1 Design

iMR is designed to provide scalable, online ETL analytics. It is meant for analytics that filter or transform data either for immediate use or before loading it into a distributed storage system (e.g., HDFS) for follow-on analysis. Moreover, today's batch processing queries exhibit characteristics that make them amenable to continuous, in-network processing. For instance, many analytics are highly selective. A 3-month trace from a Microsoft large-scale data processing system showed that filters were often highly selective (17 - 26%) [43], and the first step for many Facebook log analytics is to reduce the log data by 80% [11]. Additionally, many of these queries are update-driven, integrate the most recent data arrivals, and recur on an hourly, daily, or weekly basis. Below we summarize the goals of the iMR system and the design principles to meet these goals:

Scalable: The target operating environment consists of thousands of servers in one or more data centers, each producing KBs to MBs of log data per second. In iMR, MapReduce jobs run continuously on the servers themselves (shown on the right in Figure 3.1). This provides horizontal scaling by simply running in-place, i.e, the processing node count is proportional to the number of data sources.

This design also lowers the cost and latency of loading data into a storage cluster by filtering data on site and using in-network aggregation, if the user's reduce implements an aggregate function [41].

Responsive: Today the latency of ETL analytics dictates various aspects of a site's performance, such as the speed of social network updates or accuracy of ad-targeting. The iMR architecture builds on previous work in stream processing [15, 23, 13] to support low-latency continuous data processing. Like stream processors, iMR MapReduce jobs can process over sliding windows, updating and delivering results as new data arrives.

Available: iMR's lossy data model allows the system to return results that may be incomplete. This allows the system to improve result availability in the event of failures or processing and network delays. Additionally, iMR may proactively reduce processing fidelity through load shedding, reducing the impact on existing server tasks. iMR attaches a metric of result quality to each output, allowing users to judge the relative accuracy of processing. Users may also explicitly trade fidelity for improved result latency by specifying latency and fidelity bounds on their queries.

Efficient: A data processing architecture should make parsimonious use of computational and network resources. iMR explores the use of sub-windows or *panes* for efficient continuous processing. Instead of re-computing each window from scratch, iMR allows incremental processing, merging recent data with previously computed panes to create the next result. And adaptive load-shedding policies ensure that nodes use compute cycles for results that meet latency requirements.

Compatible: iMR supports the traditional MapReduce API, making it trivial to "port" existing MapReduce jobs to run in-situ. It provides a single extension, *uncombine*, to allow users to further optimize incremental processing in some contexts (Section 3.1.4).

3.1.1 Continuous MapReduce

Programming in the iMR system is in principle similar to the MapReduce programming model. The iMR model maintains the flexibility and ability to easily expose parallelism that comes with MapReduce. However, unlike traditional MapReduce programs that process a well defined input data set, in our application scenarios, data are continuously generated and iMR must continuously update the analysis.

Analyzing such infinite data streams requires a way to bound the amount of data that iMR can process. iMR borrows the concept of *windows* used in stream processors [15]. A window specification consists of a *range* and a *slide*. A window range R specifies the amount of data processed each time. Typically, users define the window range in terms of time. For instance, in a web log analysis scenario, a user may require to count user clicks generated over the last 5 minutes (R=5 min). A window range may even be defined in terms of data size in bytes (e.g. the last 10MB of click data), record count, or any user-defined logical sequence number. A window slide S defines how frequently to update the analysis with new data. In the previous example, a user may specify that the click count must be updated every minute (S=1 min). Just like the input, the output of an iMR program is also an infinite stream of results, one for each window of data.

Semantically, the result of executing an iMR program is the same as executing traditional MapReduce programs continuously on overlapping data sets. For every such window, data go through the same map-group-reduce process as in MapReduce and the final output is a list of key-value pairs. However, this naive approach of evaluating sliding windows by reprocessing overlapping data may result in inefficient processing. Section 3.1.4 shows how iMR optimizes windowed processing for more efficient analysis.

3.1.2 Program execution

The power of the MapReduce model lies not only in its flexibility, but also the ability to parallelize the analysis in a scalable manner. Here we show, how iMR analyzes data in-situ by distributing the execution of a program across the

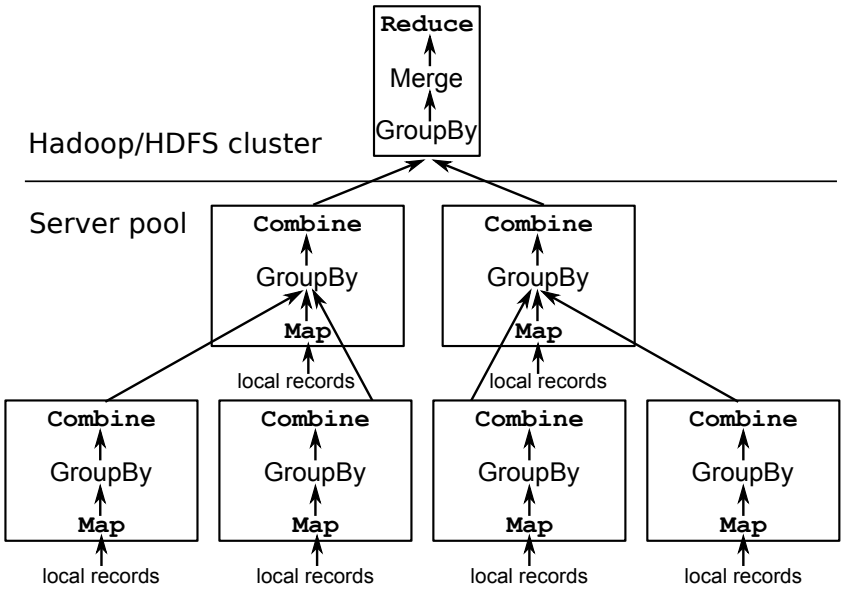


Figure 3.2: This illustrates the physical instantiation of one iMR MapReduce partition as a multi-level aggregation tree.

data sources.

In Chapter 2, we saw that a MapReduce system is tasked with three fundamental operations: mapping input records, grouping intermediate data according to key and reducing groups to produce the final output. Map tasks run in parallel, to apply the map function and produce the intermediate data. Intermediate data are shuffled, grouped according to key and partitioned, with each partition containing a subset of the keys. Subsequently parallel reduce tasks, one for each partition, apply the reduce function for every group in the corresponding partition, to produce the final result.

iMR distributes the work of a MapReduce job across multiple trees, one for each reducer partition. Figure 3.2 shows how a single tree operates. An iMR tree consists of a number of processing nodes, one for every data source. Every processing node is responsible for mapping input records that are obtained from the local source, and grouping the intermediate data according to their key. Just like in MapReduce, a processing node may also leverage the combine API, to produce partial values and decrease the size of the data that exit a processing node.

These partial values are then shipped to the *root* of the tree that is hosted,

not on the data sources, but on a dedicated processing cluster. The root is responsible for merging partial values and applying the reduce function. The final result of the reduce function is stored on the dedicated cluster.

3.1.3 Using in-network aggregation trees

Like in MapReduce, iMR leverages the combine API to decrease the data shipped from mappers to reducers. However, iMR can use multi-level aggregation trees, to further decrease the size of the data transferred by aggregating in-network, like the Dryad bulk processing system [46, 47]. This requires *decomposable* functions [55, 81] that can benefit from in-network aggregation. Operations, like a `sum` or `max` are examples of such functions that present the greatest opportunity for data reduction. Such operations may leverage the combine API to merge partial values produced by processing nodes in a tree-like manner. In contrast, *holistic* functions [41], like `union` and `median` always produce partial values with size proportional to the size of the input data and, therefore, do not benefit from aggregation trees.

Apart from reducing network traffic a multi-level aggregation tree may also distribute the processing load of the root. The root is responsible for receiving and merging partial values from all the sources. As the number of sources or the size of the partial values grows, the root may not be able to process them in a timely manner. Instead, partial values can be merged in a hierarchical way before they reach the root.

Note that the effectiveness of an aggregation tree in reducing network traffic and distributing processing load depends not only on the appropriateness of the function but also on the input data. Recall that data in the MapReduce model are structured as lists of key-value pairs. Each processing node map input data to produce such a list of intermediate key-value pairs and the combine function is applied on a per key basis. Therefore, merging key-value lists in an in-network fashion is effective only if there is a significant overlap in the intermediate keys produced between nodes. If, instead, the lists are disjoint in the keys they contain, the result of a combine function is simply a concatenation of two lists, an operation

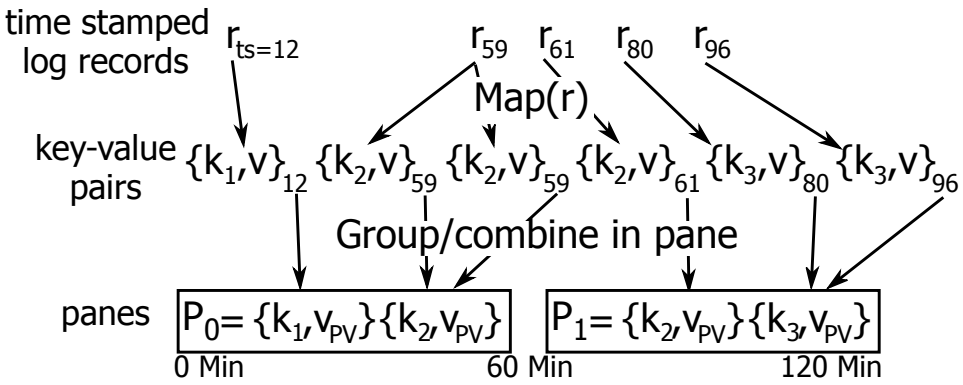


Figure 3.3: iMR nodes process local log files to produce sub-windows or panes. The system assumes log records have a logical timestamp and arrive in order.

that does not reduce the size of the data. This may actually increase the network traffic compared to a one-level tree since lists may traverse the network multiple times in the aggregation tree instead of being sent directly to the root. Sampling the input data may be used as a way to infer the key distribution and determine whether a tree is necessary.

As in MapReduce, reducers may become a performance bottleneck as they aggregate data from an increasing number of mappers. Adding more partitions and reducers addresses this bottleneck. In a similar way, iMR may run multiple such aggregation trees, one for each partition. Section 4.1.1 describes in more detail how iMR distributes the workload to multiple trees.

3.1.4 Efficient window processing with panes

Sliding windows capture a natural property in many online analytics applications: the analysis must continuously be updated with the most recent data and old data become obsolete. Recall from Section 3.1.1 that, with the exception of the sliding window specification, an iMR program is in principle similar to a MapReduce program. An obvious way to update the analysis with new data, is to continuously execute the same program iMR on every new window of data.

However, this approach may waste resources, introducing unnecessary processing and data transfers. Between successive windows there are overlapping input

records. In general, every input record may belong in R/S processing windows, where R is the range and S is the slide of the window. This means that every input record is mapped, grouped, combined and transmitted to the network multiple times by an iMR processing node.

To eliminate these overheads, iMR uses *panes*, a concept introduced in [50] for efficient window processing on single-node stream processors. Panes divide a processing window into non-overlapping sub-windows that the system processes individually, creating partial results. The system merges partial results to calculate the result for an entire window. This section shows how iMR adapts this technique for distributed in-situ execution of MapReduce programs.

Pane management

A pane is a quantum of re-usable computation. A processing node creates a pane by mapping, grouping and combining raw input records in the corresponding sub-window. A node processes raw data in a pane only once. Figure 3.3 illustrates a 2-hour window divided into two 1-hour panes. Every pane is annotated with a logical index that corresponds to the sub-window it represents. Figure 3.3 shows panes P_0 and P_1. After processing the raw input records, a pane contains a partial result that is essentially a key-value list.

Instead of sending entire windows, nodes in an iMR aggregation tree now send processed panes to their parents. Interior nodes in a tree merge and combine panes with the same. As the panes reach the root in an aggregation tree, the root combines these partial results into a window result. By sending panes up the tree instead of whole windows, the system sends a single copy of a key-value pair produced at a processing node, reducing network traffic.

The size of the panes is a parameter that affects the ability of the system to decrease network traffic. By default, iMR sets the pane size to R/S, but it can be any common divisor of R and S. By increasing the size of the pane, we allow a processing node to aggregate more data in a single pane through the combine operation, potentially reducing the amount of the data sent up the tree. The ability to reduce network traffic by increasing the pane size depends on the distribution of

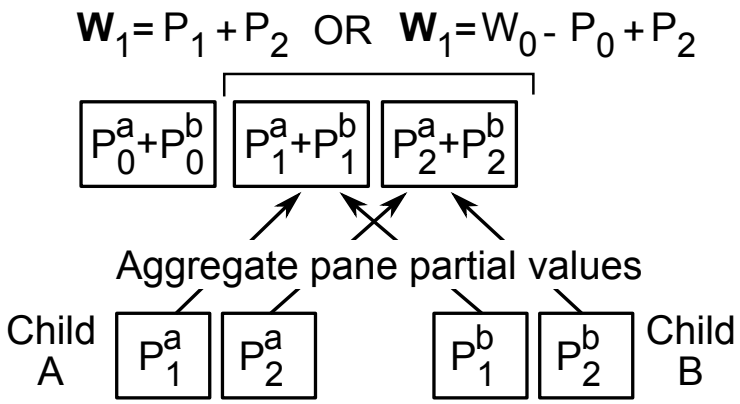


Figure 3.4: iMR aggregates individual panes P_i in the network. To produce a result, the root may either combine the constituent panes or update the prior window by removing an expired pane and adding the most recent.

the keys across a processing window. For instance, the existence of many unique keys favors small pane sizes as there is little opportunity to reduce the data size. Instead, the bigger overlap in the keys across the window, the more the system can reduce data size.

However, as we describe in Section 3.2.1, panes also represent the granularity at which iMR may proactively shed processing load, or the granularity at which failed nodes restart processing. Reducing the size of the pane gives iMR more fine control when shedding load and may reduce the amount of lost data when nodes fail and restart.

Window creation

The root of an iMR job is responsible for calculating the final result for the entire window from the individual panes. The root must group and combine all keys in the window before executing the reduce function. Figure 3.4 shows two strategies to do so. Here, a window is divided in two panes and for simplicity there are only two nodes sending panes to the root.

The first strategy leverages panes to allow incremental processing by simply using the traditional MapReduce API. In this approach, the root maintains a list of outstanding panes. Each such pane is the result of combining corresponding

$$\text{uncombine } (k', \{v'\}_{current}, \{v'\}_{old}) \rightarrow \{v'\}_{partial}$$
$$\text{combine } (k', \{v'\}_{partial}, \{v'\}_{new}) \rightarrow \{v'\}$$

Figure 3.5: iMR extends the traditional MapReduce interface with an *uncombine* function that allows the specification of differential functions. The uncombine function subtracts old data and the combine function adds new data to produce the final result.

panes from nodes in the tree as they reach the root. For instance, pane P_1^{a+b} at the root is the result of combining P_1^a from node A andP_1^b from node B. Window W_1 consists of panes P_1 and P_2 and the root can calculate the final result for W_1 by combining the key-value lists inP_1^{a+b} and P_2^{a+b}.

This improves efficiency by re-using the partial values in a pane for every window. Merging panes is cheaper than repeatedly mapping and combining input value for every window. Note that the benefit from merging panes depends again on the key distribution. The more values per key that iMR has to combine in a single pane, the more work it saves by re-using the pane. This is because the cost of creating a pane from raw data is large relative to the cost of merging panes to calculate a window. Importantly, this optimization is transparent to the user. The system leverages the same MapReduce combine API.

However, for sliding windows it sometimes more efficient to *remove* old data and then add new data to the prior window. For instance, consider a query with a 24-hour window that updates every 1 hour. This means that for every window the root must combine 24 panes, even though there is only one pane that is old and one pane that is new each time and the majority of the panes remain the same. In contrast, the root can simply remove and add a pane's worth of keys to the prior window, reducing the amount of processing for every window. Figure 3.4 shows how to compute window W_1 from W_0. Assuming that the cost of removing and adding keys to a window is equivalent, this strategy is always more efficient that merging all constituent panes in a window when the slide S is less than half the range R of the window.

However, applying this strategy requires *differential* functions [50, 14]. A differential function allows users to update the analysis by "subtracting" old data

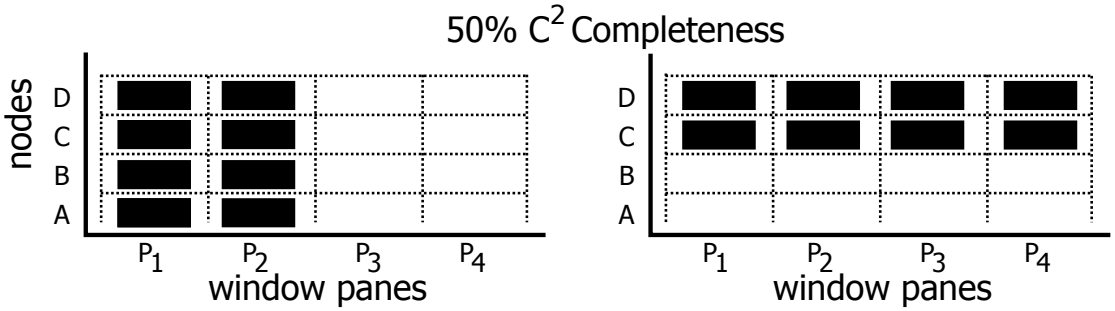


Figure 3.6: C^2 describes the set of panes each node contributes to the window. Here we show two different ways in which C^2 represents 50% of the total data: all the nodes process half their data or half the nodes process all their data.

and adding new data. The `count` and `sum` function are examples of differential functions, while `max` and `min` are not differential. To define how to subtract old data, the iMR model extends MapReduce with an *uncombine* function. Figure 3.1.4 shows how iMR can use the uncombine and combine functions to calculate a new window from the prior one.

3.2 Fidelity-latency tradeoffs

This section describes the features of iMR that allow applications to accommodate data loss. Data loss may occur because of node or network failures, or as a consequence of result latency requirements. In such cases, an iMR job may need to report a result before the system has had time to process all the data in the window. The key challenges we address here are (i) how to represent and calculate result quality to allow users to interpret partial results, and (ii) how to use this metric to trade result fidelity for improved result latency.

3.2.1 Measuring data fidelity

A useful metric for data fidelity should not only inform users that data is missing, but also allow them to assess the impact of data loss on the accuracy of the analytics. Here we introduce C^2, a fidelity metric that exposes the spatial and temporal properties of data, allowing users to better understand incomplete data.

Data are naturally distributed across space, the nodes that generate the data, and time, the processing window. The C^2 metric annotates results with spatial and temporal information about the lost data. Spatial information describes the source nodes from which data were lost. Temporal information describes the time periods during which data were lost. Such information can often provide valuable insights about the impact of the lost data on the analysis. For instance, losing data from certain nodes may be of less significance if they are not active. Similarly, there may be certain time periods of inactivity on a specific node. Being able to distinguish these cases can help users understand the impact of data loss on the accuracy of the analysis.

As a comparison, one fidelity metric that has been proposed is *completeness*, the fraction of nodes whose data are included in the result [53, 60]. Notice that such a coarse metric cannot differentiate between a node that produces data that span the entire window and a node that does not. Completeness cannot describe situations where a node fails intermittently, for a small period of time during a window, and looses only a part of the data. An alternative proposed metric is *progress*, the percentage of data processed, which is used by systems like Hadoop Online [32]. This metric too does not describe the source of the lost data or the time during which it was lost.

Instead, the C^2 metric leverages the concept of panes to describe the spatio-temporal properties of data. A pane represents a fraction of the total window and is annotated with the corresponding temporal information, the range of time within the window it represents. Additionally, a pane carries information about the source node that created the pane from raw data.

Panes are the quantum of accuracy. A pane is included in the result either in its entirety or not at all. By varying the size of a pane, we can control the granularity in the temporal dimension at which we want to measure fidelity. Smaller panes allow finer information in the event, for instance, of a very short node failure. However, as described in Section 3.1.4, a pane that is too small may hurt the ability to combine data locally, increasing the amount of data that a node ships to the network.

To inform users about the spatio-temporal properties, each delivered result is annotated with a logical *scoreboard* that indicates which panes were successfully received. Figure 3.6 shows two example scoreboards. Such a scoreboard, has two dimensions. It shows which panes across a time window (temporal dimension) are successfully included in the result for every individual node (spatial dimension).

As an example, Figure 3.6 shows two different scenarios that process the same amount of data, in this case 50% of the total data, but have different spatio-temporal properties. In the first case, all the nodes process half of their data, while in the second case, half of the nodes process all their data. In general, there are many different ways in which the system can process 50% of the data, and C^2 allows users to differentiate between them.

To maintain the C^2 metric in an aggregation tree, when interior nodes merge panes, they also merge their corresponding C^2 metric. Essentially each pane maintains a count and the IDs of the data sources that have contributed to the pane. As panes are merged inside the aggregation tree, the list of source IDs is merged as well.

Note that we can compactly summarize the IDs of the data sources using a bloom filter. This approach is useful when maintaining a large number of source IDs per pane adds significant space overhead. This may happen when the size of the panes is small and there are many data sources. While bloom filters add the possibility of false positives when testing for the existence of a source node, we can control the probability of false positives, making it practically negligible.

3.2.2 Using C^2 in applications

The goal of the C^2 metric is to allow users to better understand the impact of data loss on the analysis, but also allow users to trade result fidelity for result latency. Here, we describe how users can achieve this by appropriately specifying latency and fidelity requirements using C^2.

In general, the C^2 metric gives users flexibility in specifying fidelity requirements. For instance, a user may require simply a minimum amount of data, as the percentage of the total data, to be processed. Alternatively, users may require

the processed data to have specific spatial and temporal properties. For instance, users may simply require a minimum 50% of the data to be processed, or put more constraints. They may require that all of the nodes should return at least half of their data. This specification is illustrated in the left-hand of Figure 3.6. This is only one of many possibilities, and users may arbitrarily specify which panes they require to be processed, by specifying the exact layout of the C^2 scoreboard.

Specifications with different spatio-temporal properties, like the ones in Figure 3.6 may affect the accuracy of the analysis, but also the result latency in completely different ways, even if they specify the same percentage of data. The impact of the different C^2 specifications may depend on factors like the particular operation applied on the data and the distribution of data across nodes and across the time window. Users should set the C^2 in a way that allows them to determine the result quality when there is data loss, but also to reduce the result latency. We have identified four general C^2 specifications that allow different fidelity/latency tradeoffs and may be useful for a wide range of applications.

Minimum volume with earliest results

This C^2 specification gives the system the most freedom to decrease result latency. Users specify a percentage X% of the data they require to process and the system will return the first X% of the panes available.

This type of specification is suitable for applications where the accuracy of the analysis depends on the relative frequency of events, and the data are uniformly distributed. Here, an event is any distinct piece of information of interest appearing within the data set. As an example, consider an application that analyzes clicks logs from web servers, to count user's clicks. If events, clicks from users, are uniformly distributed across the servers and the time window, then processing any percentage of the total data can still summarize the relative frequency of the events, and give users a good estimate of the impact of data loss on the analysis.

However, this specification may provide poor estimates if data are not uniformly distributed. For instance, if events are associated mainly with the remaining panes that were not processed, this specification may not capture the relative

frequency of the events.

Minimum volume with random sampling

This C^2 specification ensures that the system will process a minimum percentage X% of the data that is randomly sampled across the entire data set. This specification gives less freedom to decrease latency since the sampled data may not be they earliest available, but it can reproduce the relative frequency of events, even if data are not uniformly distributed.

Since a pane is the quantum of accuracy in C^2, random sampling occurs at the granularity of panes. To perform random sampling, every node decides whether to process a pane with a probability proportional to the percentage X. Note that although this guarantees a random sample, it is possible that due to conditions that the system has no control of, like node or network failures, the returned result may not satisfy the user's criteria for randomness. For instance, pane sampling may appear biased toward specific nodes. However, users can still leverage the C^2 scoreboard to verify the appropriateness of the sample.

Temporal completeness

This C^2 specification ensures that a minimum percentage X% of the nodes process 100% of the panes in a window. This specification is suitable for applications that must correlate events on a per server basis. For example, an application may analyze server logs to count how often individual servers produce log errors. This implicitly partitions the analysis also on a per server basis. As a consequence, loosing data from a node does not affect the accuracy of the analysis on the rest of the servers. In the previous example, the resulting error counts for the rest of the servers will be accurate.

This specification is also useful for applications in which it is difficult to estimate the accuracy of the analysis based on incomplete data, and conclusions can be made only by processing all the data. The specification guarantees that a node will not pollute the analysis with incomplete data.

Note that the notion of temporal completeness can be applied not only on

individual nodes, but also on clusters of nodes. Often, an analysis may correlate events across specific clusters of nodes. For instance, a data center operator may analyze system logs to measure resource usage across racks of servers, or across servers owned by distinct users. Requiring temporally complete results for clusters of nodes can similarly accommodate data loss on some clusters of nodes, and at the same time guarantee the accuracy of the analysis on the rest.

This specification may, however, result in high result latency, compared to the rest of the specifications. For a result to be produced, the system must wait for nodes to process the whole window.

Spatial completeness

This specification ensures that a minimum percentage X% of the panes in the result window contain data from the 100% of the nodes in the system. The system discards any panes for which data from some nodes are missing.

This C^2 specification is useful for applications that correlate events across the whole system that occur close in time, that is, within the same pane. As an example, consider an application that analyzes click logs from web servers to characterize user behavior. A common click analysis is to find user *sessions*, groups of clicks from a user that are close in time. In load-balanced web server architectures, clicks from a single user may be served by multiple servers in the system, even during the same session. Therefore, to accurately capture sessions from a user, we need data from all the nodes in the system within the same pane.

3.2.3 Result eviction: trading fidelity for availability

iMR allows users to specify latency and fidelity bounds on continuous MapReduce queries. Here we describe the policies that determine when the root evicts results. The root has final authority to evict a window and it uses the window's completeness, C^2, and latency to determine eviction. Thus a latency-only eviction policy may return incomplete results to meet the deadline, while a fidelity-only policy will evict when the results meet the quality requirement.

Latency eviction: A query's latency bound determines the maximum

amount of time the system spends computing each successive window. If the timeout period expires, the operator evicts the window regardless of C^2. Before the timeout, the root may evict early under three conditions: if the window is complete before the timeout, if it meets the optional fidelity bound C^2, or if the system can deduce that further delays will not improve fidelity. Like the root, interior nodes also evict based on the user's latency deadline, but may do so before the deadline to ensure adequate time to travel to the root [55].

Fidelity eviction: The fidelity eviction policy delivers results based on a minimum window fidelity at the root. As panes arrive from nodes in the network, the root updates C^2 for the current window. When the fidelity reaches the bound the root merges the existing panes in the window and outputs the answer.

Failure eviction: Just as the system evicts results that are 100% complete, the system may also evict results if additional wait time can not improve fidelity. This occurs when nodes are heavily loaded or become disconnected or fail. iMR employs *boundary* panes (where traditional stream processors use boundary tuples [70]) to distinguish between failed nodes and stalled or empty data streams[1]. Nodes periodically issue boundary panes to their parents when panes have been skipped because of a lack of data or load shedding (Section 4.1.3).

Boundary panes allow the root to distinguish between missing data that may arrive later and missing data that will never arrive. iMR maintains boundary information on a per-pane basis using two counters. The first counter is the C^2 completeness count; the number of successful pane merges. Even if a child has no local data for a pane, its parent in the aggregation tree may increase the completeness count for this pane. However, children may skip panes either because they re-started later in the stream (Section 4.1.5) or because they canceled processing to shed load (Section 4.1.3). In these cases, the parent node increases an *incompleteness* counter indicating the number of nodes that will never contribute to this pane.

Both interior nodes and the root use these counts to evict panes or entire windows respectively. Interior nodes evict early if the panes are complete or the

[1]In reality, all panes contain boundary meta data, but nodes may issue panes that are otherwise empty except for this meta data.

sum of these two counters is equal to the sum of the children in this sub tree. The root determines whether or not the user's fidelity bound can ever be met. By simply subtracting incompleteness from the total node count (perfect completeness), the root can set an upper bound on C^2 for any particular window. If this estimate of C^2 ever falls below the user's target, the root evicts the window.

Note that the use of fidelity and latency bounds presumes that the user either received a usable result or cannot wait longer for it to improve. Thus, unlike other approaches, such as tentative tuples [16] or re-running the reduction phase [32], iMR does not, by default, update evicted results. iMR only supports this mode for debugging or determining a proper latency bound, as it can be expensive, forcing the system to repeatedly re-process (re-reduce) a window on late updates.

3.3 Related work

3.3.1 "Online" bulk processing

iMR focuses on the challenges of migrating initial data analytics to the data sources. A different (and complementary) approach has been to optimize traditional MapReduce architectures for continuous processing themselves. For instance, the Hadoop Online Prototype (HOP) [32] can run continuously, but requires custom reduce functions to manage their own state for incremental computation and framing incoming data into meaningful units (windows). iMR's design avoids this requirement by explicitly supporting sliding window-based computation (Section 3.1.1), allowing existing reduce functions to run continuously without modification.

Like iMR, HOP also allows incomplete results, producing "snapshots" of reduce output, where the reduce phase executes on the map output that has accumulated thus far. HOP describes incomplete results with a "progress" metric that (self admittedly) is often too coarse to be useful. In contrast, iMR's C^2 framework (Section 3.2) not only provides both spatial and temporal information about the result, but may be used to trade particular aspects of data fidelity for decreased

processing time.

Dremel [57] is another system that, like iMR, aims to provide fast analysis on large-scale data. While iMR targets continuous raw log data, Dremel focuses on static nested data, like web documents. It employs an efficient columnar storage format that is beneficial when a fraction of the fields of the nested data must be accessed. Like HOP, Dremel uses a coarse progress metric for describing early, partial results.

3.3.2 Log collection systems

A system closely related to iMR is Flume [6], a distributed log collection system that places *agents* in-situ on servers to relay log data to a tier of collectors. While a user's "flows" (i.e., queries) may transform or filter individual events, iMR provides a more powerful data processing model with grouping, reduction, and windowing. While Flume supports best-effort operation, users remain in the dark about result quality or latency. However, Flume does provide higher reliability modes, recovering events from a write-ahead log to prevent data loss. While not discussed here, iMR could employ similar *upstream backup* [16] techniques to better support queries that specify fidelity bounds.

3.3.3 Load shedding in data stream processors

iMR's load shedding (Section 4.1.3) and result eviction policies (Section 3.2.3) build upon the various load shedding techniques explored in stream processing [23, 75, 74]. For instance, iMR's latency and fidelity bounds are related to the QoS metrics found in the Aurora stream processor [23]. Aurora allows users to provide "graphs" which separately map increased delay and percent tuples lost with decreasing output quality (QoS). iMR takes a different approach, allowing users to specify latency and fidelity bounds above which they'd be satisfied. Additionally, iMR leverages the temporal and spatial nature of log data to provide users more control than percent tuples lost.

Many of these load shedding mechanisms insert tuple dropping operators

into query plans and coordinate drop probabilities, typically via a centralized controller, to maintain result quality under high-load conditions. In contrast, our load shedding policies act locally at each operator, shedding sub-windows (panes) as they are created or merged. These "pane drop" policies are more closely related to the probabilistic "window drop" operators proposed by Tatbul, et al. [75] for aggregate operators. In contrast, iMR's operators may drop panes both deterministically or probabilistically depending on the C^2 fidelity bound.

3.3.4 Distributed aggregation

Aggregation trees have been explored in sensor networks [55], monitoring wired networks [79], and distributed data stream processing [53, 45]. More recent work explored a variety of strategies for distributed GroupBy aggregation required in MapReduce-style processing [81]. Our use of sub-windows (panes) is most closely related to their *Accumulator-PartialHash* strategy, since we accumulate (through combining) key-value pairs into each sub-window. While they evicted the sub window based on its storage size (experiencing a hash collision), iMR uses fixed-sized panes.

3.4 Acknowledgments

Chapter 3, in part, is reprint of the material published in the Proceedings of the USENIX Annual Technical Conference 2011. Logothetis, Dionysios; Trezzo, Chris; Webb, Kevin C.; Yocum; Ken. The dissertation author was the primary investigator and author of this paper.

Chapter 4

An architecture for in-situ processing

This chapter describes the design of the iMR architecture. Here, we show how iMR can execute continuous MapReduce programs in a scalable and efficient manner. We also describe the mechanisms that allow iMR analytics to run in-situ. This includes load shedding mechanisms to accommodate server load, as well as fault tolerance mechanisms. This chapter presents an evaluation of iMR through microbenchmarks. Furthermore, we validate the usefulness of the C^2 metric in understanding incomplete data and trading fidelity for timeliness in the analysis through experiments with real applications.

4.1 Implementation

The iMR design builds upon Mortar, a distributed stream processing architecture [53]. We significantly extended Mortar's core functionality to support the semantics of iMR and the MapReduce programming model along four axes:

- Implement the iMR MapReduce API using generic map and reduce Mortar operators.
- Pane-based continuous processing with flow control.

- Load shedding/cancellation and pane/window eviction policies.
- Fault-tolerance mechanisms, including operator re-start and adaptive routing schemes.

4.1.1 Building an in-situ MapReduce query

Mortar computes continuous in-network aggregates across federated systems with thousands of nodes. This is a natural fit for the map, combine, and reduce functions since they are either local per-record transforms (map) or often in-network aggregates. A Mortar query consists of a single operator, or aggregate function, which Mortar replicates across nodes that produce the raw data streams. These in-situ operators give iMR the opportunity to actively filter and reduce intermediate data before it is sent across the network. Each query is defined by its operator type and produces a single, continuous output data stream. Operators push, as opposed to the pull-based method used in Hadoop, records across the network to other operators of the same type.

Mortar supports two query types: local and in-network queries. A local query processes data streams independently at each node. In contrast, in-network queries use a tree of operators to aggregate data across nodes. Either query type may subscribe to a local, raw data source such as a log file, or to the output of an existing query. Users compose these query types to accomplish more sophisticated tasks, such as MapReduce jobs.

Figure 4.1 illustrates an iMR job that consists of a local query for map operators and an in-network query for reduce operators. Map operators run on the log servers and partition their output among co-located reduce operators (here there are two partitions, hence two reduce trees). The reduce operator does most of the heavy lifting, grouping key-value pairs issued by the map operators before calling the user's combine, uncombine, and reduce functions. Unlike traditional MapReduce architectures, where the number of reducers is fixed during execution, iMR may dynamically add (or subtract) reducers during processing.

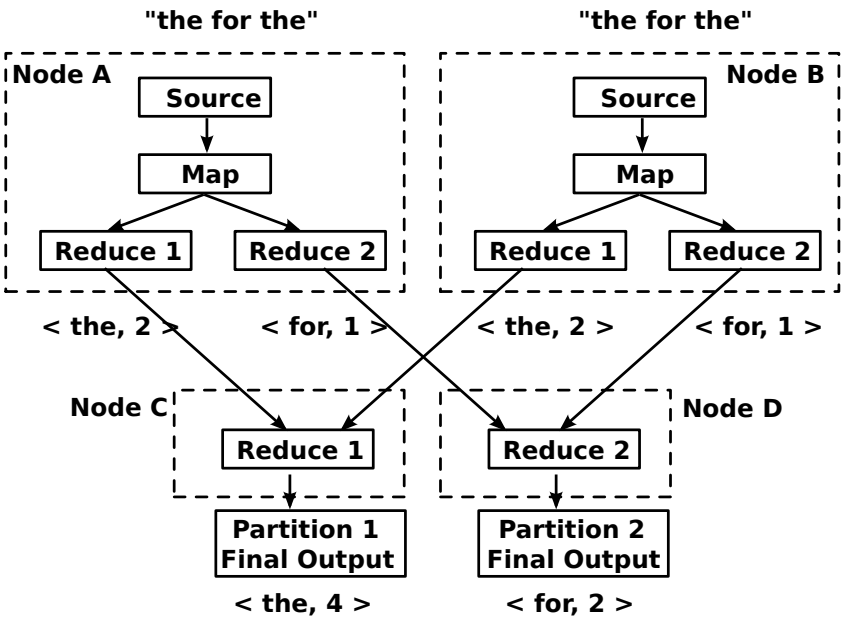


Figure 4.1: Each iMR job consists of a Mortar query for the map and a query for the reduce. Here there are two MapReduce partitions ($r = 2$), which result in two aggregation trees. A word count example illustrates partitioning map output across multiple reduce operators.

4.1.2 Map and reduce operators

Like other stream processors, Mortar uses processing windows to bound computation and provides a simple API to facilitate programming continuous operators. We implemented generic map and reduce operators using this API to call user-defined MapReduce functions at the appropriate time and properly group the key-value pairs. We modified operator internals so that they operate on panes as described in Section 3.1.4. Operators take as input either raw records from a local source or they receive panes from upstream operators in the aggregation tree. Internally, iMR represents panes as (possibly sorted) hash maps to facilitate key-value grouping.

In iMR operators have two main tasks: pane creation, creating an initial pane from a local data source, and pane merging, combining panes from children in an aggregation tree. Pane creation operates on a record-by-record basis, adding new records into the current pane. In contrast, pane merging combines locally

produced panes with those arriving from the network. Because of differences in processing time and network congestion, operators maintain a sequence of panes that the system is actively merging (they have not yet been evicted). We call this the active pane list or APL.

To adapt Mortar for MapReduce processing, we introduce immutable timestamps into the system. Mortar assumes logically independent operators that timestamp output records at the moment of creation. In contrast, iMR defines processing windows with respect to the original timestamps on the input data, not with respect to the time at which an operator was able to evict a pane. iMR assigns a timestamp to each data record when it first enters the system (e.g. using a pre-existing timestamp embedded in the data, or the current real time). This timestamp remains with the data as it travels through successive queries. Thus networking or processing delays do not alter the window in which the data belongs.

The map operator

The simplicity of mapping allows a streamlined map operator. The operator calls the user's map function for each arriving record, which may contain one or more log entries[1]. For each record, the map operator emits zero or more key-value pairs. We optimized the map operator by permanently assigning it a window with a range and slide equal to one record. This allowed us to remove window-related buffering and directly issue records containing key-value pairs to subscribed operators. Finally, the map operator partitions key-value pairs across subscribed reduce operators.

The reduce operator

The reduce operator handles the in-network functionality of iMR including the grouping, combining, sorting and reducing of key-value pairs. The operators maintain a hash map for each pane in the active pane list. Here we describe how the reduce operator creates and merges panes.

[1]Like Hadoop, iMR includes handlers that interpret log records.

After a reduce operator subscribes to a local map operator it begins to receive records (containing key-value {k,v} pairs). The reducer operator first checks the logical timestamp of each {k,v} pair. If it belongs to the current pane, the system inserts the pair into the hash table and calls the combiner (if defined). When a {k,v} pair arrives with a timestamp for the next pane, the system inserts the prior pane into the active-pane list (APL). The operator may skip panes for which there is no local data. In that case, the operator inserts boundary panes into the APL with completeness counts of one.

Load shedding occurs during pane creation. As records arrive, the operator maintains an estimate of when the pane will complete. The operator periodically updates this estimate, maintained as an Exponentially Weighted Moving Average (EWMA) biased towards recent observations ($\alpha = 0.8$), and determines whether the user's latency deadline will be met. For accuracy, the operator processes 30% of the pane before the first estimate update. For responsiveness, the operator periodically updates and checks the estimate (every two seconds). For each skipped pane the operator issues a boundary pane with an incompleteness count of one.

The APL merges locally produced panes with panes from other reduce operators in the aggregation tree. The reduce operator calls the user's combiner for any group with new keys in the pane's hash map. The operator periodically inspects the APL to determine whether it should evict a pane (based on the policies in Section 3.2.3). Reduce operators on internal or leaf nodes forward the pane downstream on eviction.

If the operator is at the tree's root, it has the additional responsibility of determining when to evict the entire window. The operator checks eviction policies on periodic timeouts (the user's latency requirement) or when a new pane arrives (possibly meeting the fidelity bound). At that point, the operator may produce the final result either by using the optional uncombine function or by simply combining the constituent panes (strategies discussed in Section 3.1.4). After this combining step, the operator calls the user-defined reduce function for each key in the window's hash map.

4.1.3 Load cancellation and shedding

When the root evicts incomplete windows, nodes in the aggregation tree may still be processing panes for that window. This may be due to panes with inordinate amounts of data or servers that are heavily loaded (have little time for log processing). Thus they are computing and merging panes that, once they arrive at the root, will no longer be used. This section discusses mechanisms that cancel or shed the work of creating and merging panes in the aggregation tree. Note that iMR assumes that mechanisms already exist to apportion server resources between the server's normal duties and iMR jobs. For instance, iMR may run in a separate virtual machine, letting the VM scheduler allocate resources between log processing and VMs running site services. Here our goal is to ensure that iMR nodes use the resources they are given effectively.

iMR's load cancellation policies try to ensure that internal nodes do not waste cycles creating or merging panes that will never be used. When the root evicts a window because it has met the minimum C^2 fidelity requirement, there is almost surely outstanding work in the network. Thus, once the root determines that it will no longer use a pane, it relays that pane's index down the aggregation tree. This informs the other nodes that they may safely stop processing (creating/merging) the pane.

In contrast, iMR's load shedding strategy works to prevent wasted effort when individual nodes are heavily loaded. Here nodes observe their local processing rates for creating a pane from local log records. If the expected time to completion exceeds the user's latency bound, it will cancel processing for that pane. It will then estimate the next processing deadline that it can meet and skip the intervening panes (and send boundary panes in their place).

Internal nodes also spend cycles (and memory) merging panes from children in the aggregation tree. Here interior nodes either choose to proceed with pane merging or, in the event that it violates the user's latency bound, "fast forward" the pane to its immediate parent. As we shall see in Section 6.6, these policies can improve result fidelity in the presence of straggler nodes.

4.1.4 Pane flow control

Recall that the goal of load shedding in iMR is not to use less resources, but to use the given resources effectively. Given some large input data at a source, load shedding changes the work done, not its processing rate. Thus, it is still possible for some nodes to produce panes faster than others, either because they have less data per pane or more cycles available. In these cases, the local active pane list (APL) could grow in an unbounded fashion, consuming server memory and impacting its client-facing services.

We control the amount of memory used by the APL by employing a window-oriented flow control scheme. Each operator monitors the memory used (by the JVM in our implementation) and issues a pause indicator when it reaches a user-defined limit. The indicator contains the logical index of the youngest pane in the operator's APL. Internally, pane creation waits until the indicator is greater than the current index or the indicator is removed. Pause indicators are also propagated top-down in the aggregation tree, ensuring that operators send evicted panes upward only when the indicator is greater than the evicted indices or it is not present.

4.1.5 MapReduce with gap recovery

While load shedding and pane eviction policies improve availability during processing and network delays, nodes may fail completely, losing their data and current queries. While traditional MapReduce designs, such as Hadoop, can restart map or reduce tasks on any node in the cluster, iMR does not assume a shared filesystem that can reliable store data. Instead, iMR provides *gap recovery* [16], meaning that the system may drop records (i.e., panes) in the event of node failures.

Multi-tree aggregation

Mortar avoids failed network elements and nodes by routing data up multiple trees. Nodes route data up a single tree until the node stops receiving heart beats from its parent. If a parent becomes unreachable, it chooses another tree

(i.e., another parent) to route records to. For this work, we use a single tree; this simplifies our implementation of failure eviction policies because internal nodes know the maximum possible completeness of panes arriving from their children.

Mortar employs new routing rules to retain a degree of failure resilience. If a parent becomes unreachable, the child forwards data directly to the root. This policy allows data to bypass failed nodes at the expense of fewer aggregation opportunities. Mortar also designs its trees by clustering network coordinates [33], and we use the same mechanism in our experiments. We leave more advanced routing and tree-building schemes as future work.

Operator re-install

iMR guarantees that queries (operators) will be installed and removed on nodes in an eventually consistent manner. Mortar provides a reconciliation algorithm to ensure that nodes eventually install (or un-install) query operators. Thus, when nodes recover from a failure, they will re-install their current set of operators. While we lose the data in the operator's APL at the time of failure, we need to re-start processing at an appropriate point to avoid duplicate data. To do so, operators, during pane creation, maintain a simple on-disk write-ahead log to indicate the next safe point in the log to begin processing on re-start. For many queries the cost of writing to this log is small relative to pane computation, and we simply point to the next pane.

4.2 Evaluation

In the section, we evaluate the scalability of iMR through microbenchmarks but also its ability to deliver high-fidelity results in a timely manner under failures or constrained computational resources. We also assess the usefulness of the C^2 metric in understanding incomplete data and trading data fidelity for result latency.

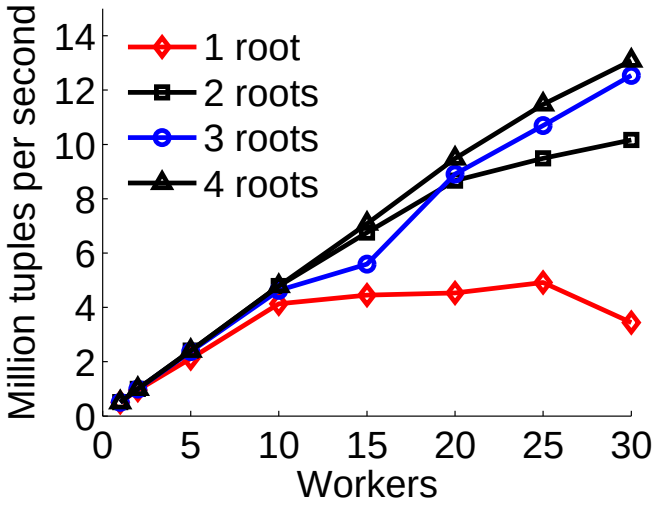


Figure 4.2: Data processing throughput as the number of workers and roots increases. When the root of the query becomes the bottleneck, iMR scales by partitioning data across more roots.

4.2.1 Scalability

One of the design goals of iMR is to scale to large datasets. Here, we evaluate the ability of iMR to scale as we increase the computational resources available. As described in Section 3.1, in the iMR in-network architecture, machines may play one of two distinct roles: they are either workers that process and summarize raw data from the local sources, or roots that combine summaries from child nodes. Increasing the number of workers should increase data processing throughput until the point that roots become a performance bottleneck. As with the MapReduce framework, iMR handles this bottleneck by increasing the number of partitions, that is, the number of roots. Therefore, we must verify the ability of iMR to scale by adding more workers and roots.

For this experiment, we evaluated iMR on a 40 node cluster of HP DL380G6 servers, each with two Intel E5520 CPUs (2.27 GHz), 24 GB of memory, and 16 HP 507750-B21 500GB 7,200 RPM 2.5 SATA drives. Each server has two HP P410 drive controllers, as well as a Myricom 10 Gbps network interface. The network interconnect we use is a 52-port Cisco Nexus 5020 data center switch. The servers run Linux 2.6.35. Our implementation of iMR is written in Java.

In this experiment, we implement a word count query that runs on synthetic input data, a set of randomly generated numbers. In our query, the map function implements the identity function, while the reducer implements a count. The query specifies a tumbling window, where the range is 150 million records. This window range corresponds to processing approximately 1GB of input data per node. We allow the query to run for five minutes and compute the average throughput across all windows computed.

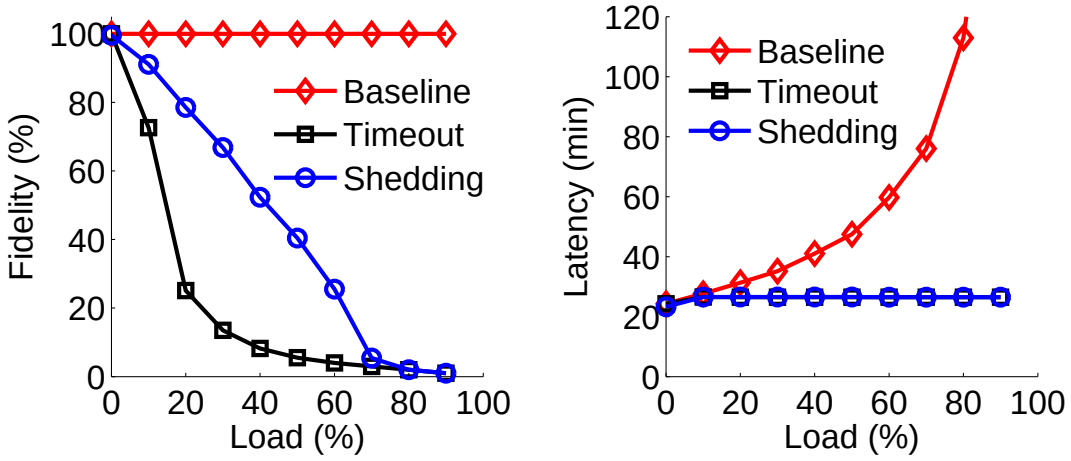
In Figure 4.2, we plot throughput, the total number of records processed per second, as we increase computational resources, that is, workers and roots. More specifically, on the x-axis we increase the number of workers and each line corresponds to an increasing number of roots. We observe that as long as the root is not a performance bottleneck, adding more workers increases throughput linearly. Notice that a single root can handle the incoming data sent by up to 10 workers. At this point, by doubling the number of roots, we can also double the throughput. Given the available resources in our experimental cluster, we were able to use up to 30 workers and we observe that three roots are enough to handle this load.

4.2.2 Load shedding

iMR employs techniques that shed processing load when nodes do not have sufficient resources due to other services and the analysis has to deliver results before a deadline. These techniques are designed to maximize result fidelity under given time constraints. Here, we evaluate the effectiveness of these techniques in providing high-fidelity results under limited CPU results. More specifically, we verify that a single node can accurately estimate the right amount of data to shed under varying CPU load.

For this and the remaining experiments we used a 30-node cluster with Dual Intel Xeon 2.4 GHz CPUs. Nodes have 4GB of RAM and are connected on a gigabit Ethernet.

In this experiment, we execute a word count query, where the map function is the identity function and the reduce function implements a count. The query



(a) Fidelity (b) Latency

Figure 4.3: Impact of load shedding on fidelity and latency for a word count job under maximum latency requirement and varying worker load.

specifies a tumbling window with 20 million records and we configure iMR to use 20 panes per window. We install the query on a single worker that delivers result to a single root. To limit the CPU available to iMR, we use the Linux `cpulimit` tool [2] on the worker. We execute the query until it delivers 10 results and report the average result latency and fidelity as we vary the CPU available to iMR.

In Figure 4.3(a) we show the fidelity of the delivered results as the CPU load increases, while Figure 4.3(b) shows the latency of the delivered results. The baseline case represents a query with no latency requirements that always delivers results with 100% fidelity. As expected, the result latency of the baseline case grows hyperbolically[2] as the load increases.

Next, we set the latency requirement of the query to the observed baseline window latency, which is 160 seconds. Based on this timeout, the ideal maximum fraction of raw data that cannot be processed within the latency requirement is equal to the CPU load percentage wise. The effectiveness of our load shedding technique is determined by how much fidelity approaches this ideal maximum.

For comparison, the timeout line represents a query that does not employ any shedding. The node keeps processing data until the timeout passes, in which

[2]Latency as a function of the CPU load x is of the type $L(x) = \frac{c}{1-x}$

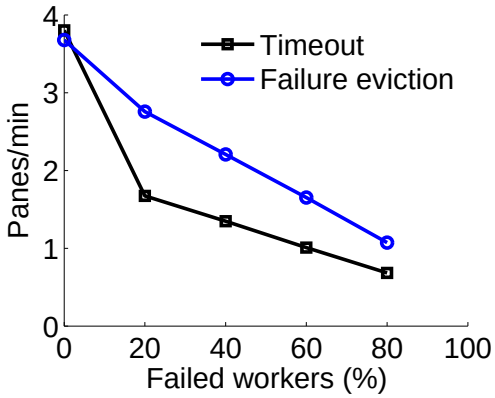


Figure 4.4: Application goodput as the percentage of failed workers increases. Failure eviction delivers panes earlier, improving goodput by up to 64%.

case the root evicts any data delivered up to that point. We observe that although results meet the latency requirement, fidelity drops quickly as the load increases. Without load shedding, the node attempts to process all raw data as they become available. However, only the first few panes can be delivered within the deadline and processing subsequent panes is useless.

Instead, by enabling load shedding workers use available CPU intelligently. They process only panes that can be delivered within the deadline. This improves result fidelity on average by 242%. Additionally, result fidelity is on average within 10% of the ideal fidelity. Notice that the higher the load is, the greater the divergence from the ideal is. Shedding data incurs some CPU overhead and increasing the CPU load results in shedding more data. Therefore, iMR spends more CPU cycles in shedding rather than useful processing, causing this divergence from the ideal fidelity.

4.2.3 Failure eviction

Apart from maximizing result fidelity through load shedding, iMR is designed to minimize result latency. Through the failure eviction mechanism, iMR detects opportunities to deliver results early when fidelity cannot be further improved by waiting for overloaded or failed nodes. Here, we evaluate the ability of the failure eviction mechanism to improve result latency when nodes fail.

In this experiment, we execute a word count query with a window of 2 million records and 2 panes per window. We set the query latency requirement to 30 seconds. We execute the query on 10 workers and emulate transient failures by stopping an increasing number of workers for 4 minutes and then resuming them. We run the query until it delivers 20 results.

Figure 4.4 plots the application goodput, the number of panes delivered to the user per time. Note that this metric does not measure how fast workers can process raw data. Instead it reflects the ability of the system to detect failures and deliver panes to the user early. The higher the metric, the less the user waits to get the same number of panes.

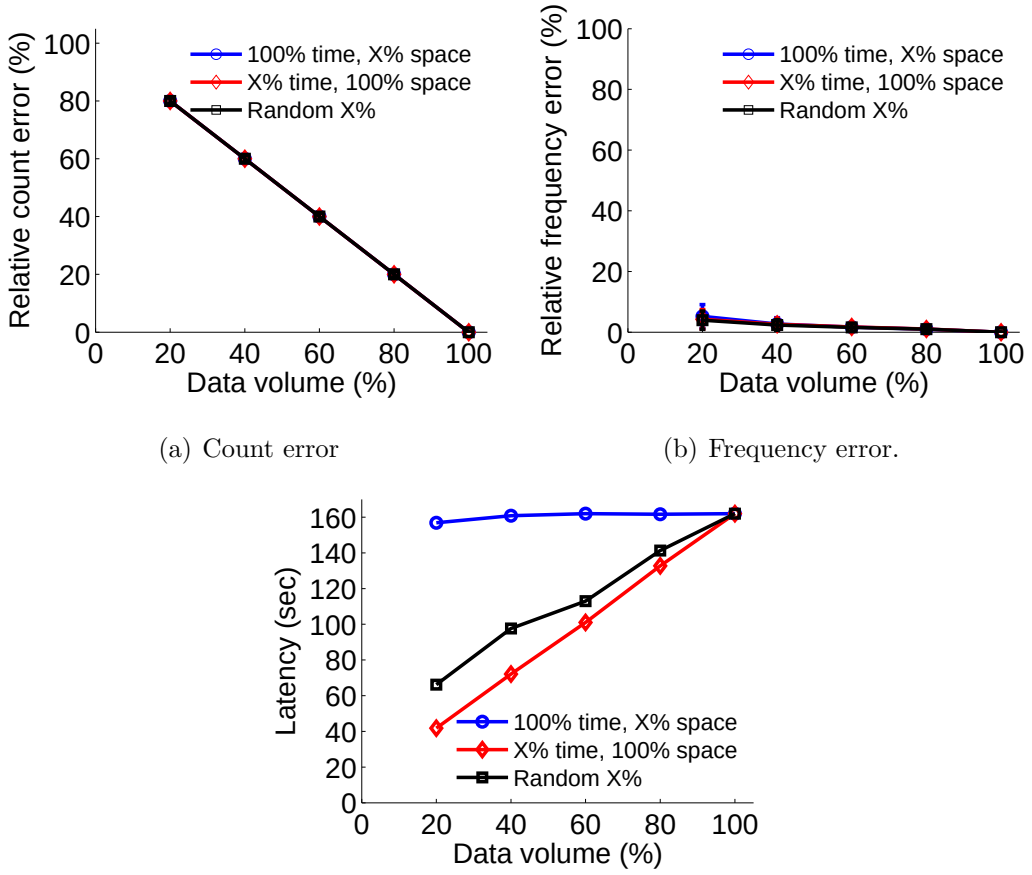
Without failure eviction, the root waits until the 30-second timeout before it delivers incomplete results, even if all live nodes have delivered their data and fidelity cannot improve. With failure eviction enabled, the root detects failed workers and delivers results before the timeout, improving goodput by 57-64%.

4.2.4 Using C^2

In this experiment, we show how using the C^2 metric users can trade result fidelity for latency. We display how depending on the application requirements and data distributions, users may appropriately set the C^2 specification. In particular, we explore the use of three general classes of C^2 specifications: temporal completeness, spatial completeness, and minimum volume with random pane selection. We also show how choosing the right C^2 specification allows users to make more useful conclusions for incomplete data than with coarser metrics, like simple progress. We perform experiments with three different application scenarios: a word count with varying word distributions, click-stream analysis, and an HDFS anomaly detector.

Word count

Here, we execute a word count query on synthetic data. We vary the total percentage of data included in the result and measure how the accuracy of the result changes. We repeat this for the three different classes of C^2 specifications.



(a) Count error

(b) Frequency error.

(c) Result latency.

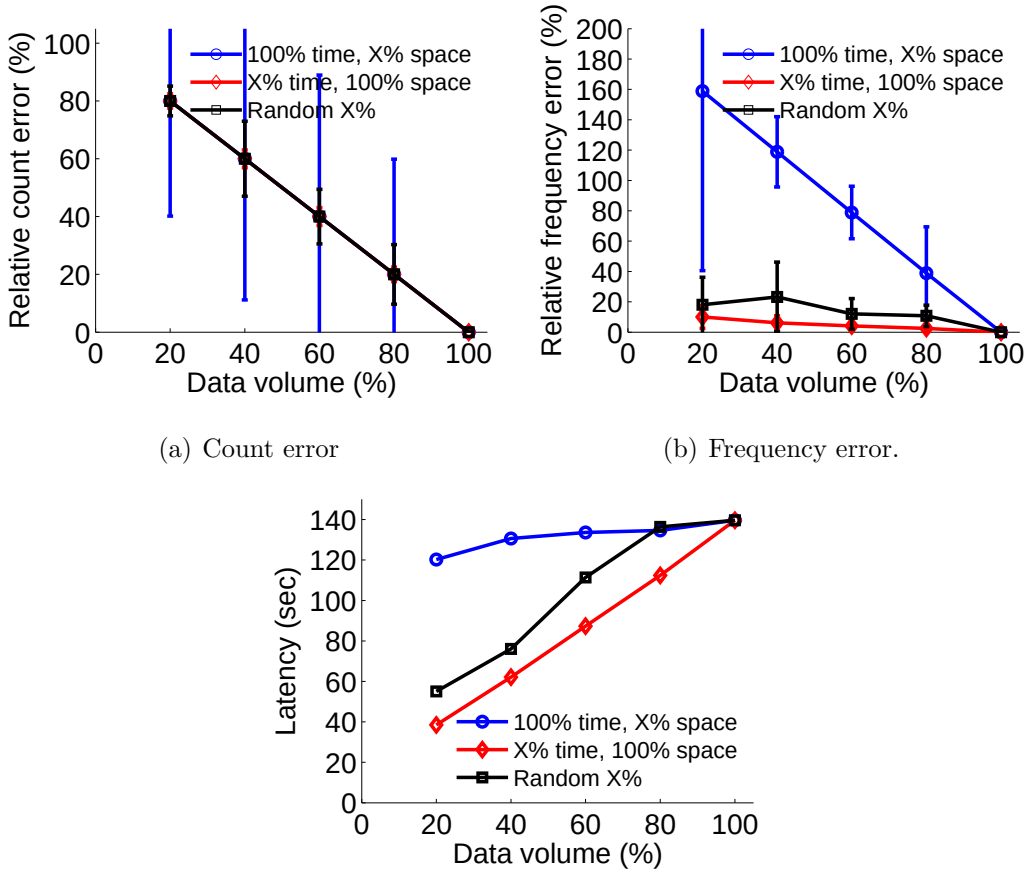
Figure 4.5: The performance of a count statistic on data uniformly distributed across the log server pool. The relative count error drops linearly as we include more data. Because of the uniform data distribution, both the count and the frequency do not depend on the C^2 specification.

We report the accuracy of two application metrics, the count of the words and the relative frequency. For both metrics we report the relative error[3]. Additionally, we report the result latency.

The query specifies a tumbling window with a size-based range of 100MB. Each window consists of 10 panes. In this experiment, there is no latency bound. The query is executed on 10 workers.

The data set is text consisting of random words chosen from a pool of 100K words. We distribute the words across the 10 workers and experiment with

[3]The relative error of a measurement X with respect to an ideal value Y is $100\frac{Y-X}{Y}\%$.



(a) Count error

(b) Frequency error.

(c) Result latency.

Figure 4.6: The performance of a count statistic on data skewed across the log server pool. Because of the spatial skew, enforcing either random pane selection or spatial completeness allows the system to better approximate count frequencies than temporal completeness, and lower result latency.

different data distributions, to display how the C^2 specification impacts our ability to understand the results in different scenarios. We change the distribution of the word frequency but also the distribution of words across different nodes. While this is a simple text processing application, the results are relevant to any application that counts, instead of words, events with similar distributions.

First, we create a data set where word frequency is uniformly distributed and also words are spatially distributed uniformly across the workers. Figures 4.5(a) and 4.5(b) plot the relative count and frequency errors respectively, while Figure 4.5(c) plots the result latency as the data volume included in the result in-

creases. The error reported is the average across all words in the result. The vertical bars are equal to plus or minus one standard deviation.

As expected, the relative count error decreases linearly as more data are include in the result. Because of the uniform spatial distribution, all C^2 specifications result in the same error. Due to the uniform word frequency distribution, the standard deviation of the error across all words is very small, since the accuracy of every word count is equally degraded.

Furthermore, due to the uniform distribution, the relative frequency error is low and also exhibits a small standard deviation. This is because the absolute count of every word is reduced by the same percentage as the total volume of the data. In Figure 4.5(c), we notice that requiring temporally complete results implies that we have to wait until at least one of the workers processes all data in a window, thus, increasing result latency. In contrast, by specifying spatially completeness or random pane drops, we can reduce the result latency. Therefore, when data are uniformly distributed, this C^2 specification is preferred since it reduces result latency and achieves result fidelity equal to the other specifications.

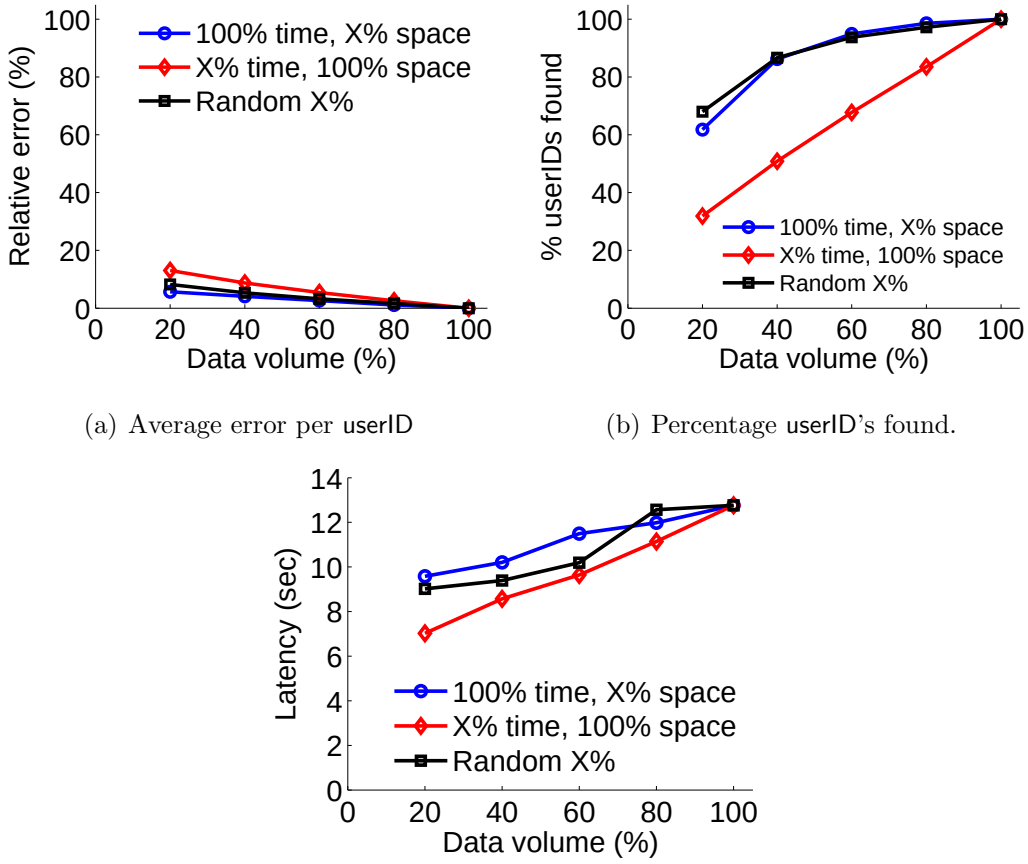
Next, we change the spatial distribution of the data. We skew the distribution so that some words are more likely to exist on some workers than others. As Figure 4.6(a) shows, while the average relative count error is similar to the previous case, notice that the standard deviation is much higher here, especially for the C^2 specification that requires temporally complete results. In this case, windows that are not temporally complete are dropped in their entirety, and data from the corresponding workers are completely lost. Therefore, words that are located on those nodes exhibit a higher relative count error, which impacts the standard deviation.

The effect of the spatial skew is more obvious in the relative frequency error, shown in Figure 4.6(b). By removing entire windows from specific workers, we reduce the count of the words that exist on those workers more than the count of the words on the rest of the workers percentage wise. This adds to the relative frequency error of these words. Instead, with spatially complete windows or randomly dropped panes, the system samples keys from the entire server pool and the

relative frequency error remains low as in the previous case.

We observe that different classes of C^2 specifications, which return the same volume of data, may return qualitatively different results. The C^2 metric exposes the spatio-temporal characteristics of data, allowing users to either understand the effect of data loss on their analysis, or set the C^2 specification based on knowledge about the data distribution.

Click-stream analysis

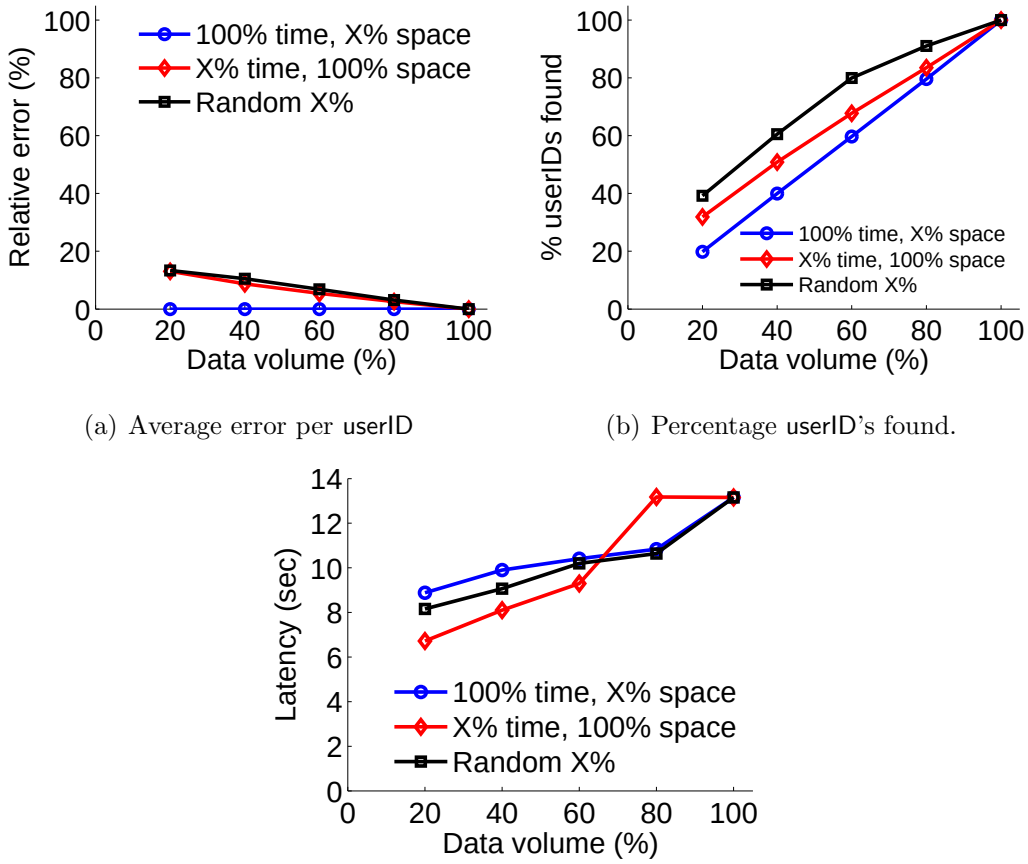


(a) Average error per userID

(b) Percentage userID's found.

(c) Result latency.

Figure 4.7: Estimating user session count using iMR and different C^2 policies. We preserve the original data distribution, where clicks from the same user may exist on different servers. Random pane selection and temporal completeness provide higher data fidelity and sample more userIDs than when enforcing spatial completeness.



(a) Average error per userID

(b) Percentage userID's found.

(c) Result latency.

Figure 4.8: Estimating user session count using iMR and different C^2 policies. Here we distribute data so that clicks from the same user exist on a single server. Temporal completeness returns sessions that are accurate, but samples the smallest percentage of userIDs. Instead, random sampling can sample a larger space of userIDs.

Here, we implement a common click-stream analysis application. In particular, we develop a query that takes as input web server logs that contain user clicks and computes user sessions. Clicks are the result of users browsing on a site and are usually defined by (i) a userID, the identity of the user browsing, (ii) the page that they clicked, and (iii) a timestamp that denotes the time of the click. Although these are the most essential information, click logs may contain other useful information as well.

A user session is a period during which a user is using a site and click

sessionization is the method of finding distinct user sessions by analyzing clicks. Such an analysis is useful to understand user behavior, for instance, how much time users spend on a site for every visit. A common method to sessionize clicks is to compare the timestamps of subsequent clicks and group them in the same sessions if they occurred withing a maximum amount of time.

We implemented an iMR query in which the map function extracts clicks from the logs and the reduce function performs the sessionization by grouping clicks as described above. The corresponding MapReduce query is also described in [38]. In this experiment, the reduce function, summarizes the calculated sessions by reporting the number of sessions found per user. As with the previous experiment, we want to evaluate how the different C^2specifications affect the accuracy and the latency of the analysis for varying amounts of data loss. Here, we measure accuracy as the relative error on the computed session count.

The data set consists of 24 hours of sever logs from the 1998 World Cup web site [1]. These represent logs from 32 web servers that comprise the site infrastructure, and have a size of 4.5GB in total. The query window is set to 2 hours, and we set the pane size to 6 minutes, allowing for 20 panes per window. We run the query for the entire data sets, which corresponds to 12 windows.

As with the previous experiment, we also want to explore how the data distribution affects the accuracy of the analysis for the different C^2 specifications. Here the main characteristic of interest is the distribution of a user's click across servers. Clicks from the same user may either be distributed across servers, or exist on a single server. Initially, we retain the original data distribution, where a user's clicks exist across different servers.

In Figure 4.7(a), we plot the relative error in the session count as the percentage of data included in the analysis changes. We compute the average error across all users in a result. Note that this average does not take into account users for which data are completely missing from the result. However, to quantify the missing users, in Figure 4.7(b) we plot the percentage of userIDs discovered in the data. This graph shows the ability of the system to sample a wide range of the user space under the different C^2 specifications.

We see in Figure 4.7(a) that although all three specifications provide a relatively low relative error, temporal completeness and random sampling provide a lower error than spatial completeness. In particular, temporal completeness reduces the relative error by approximately 50% with respect to spatial completeness. Because of the non-uniform distribution of the clicks across time, by requiring spatially complete panes, we may miss panes with larger amounts of clicks. Figure 4.7(b) shows that spatially complete results are not effective in sampling a large number of keys. Instead random sampling and temporally complete results can return a better representation of the user space.

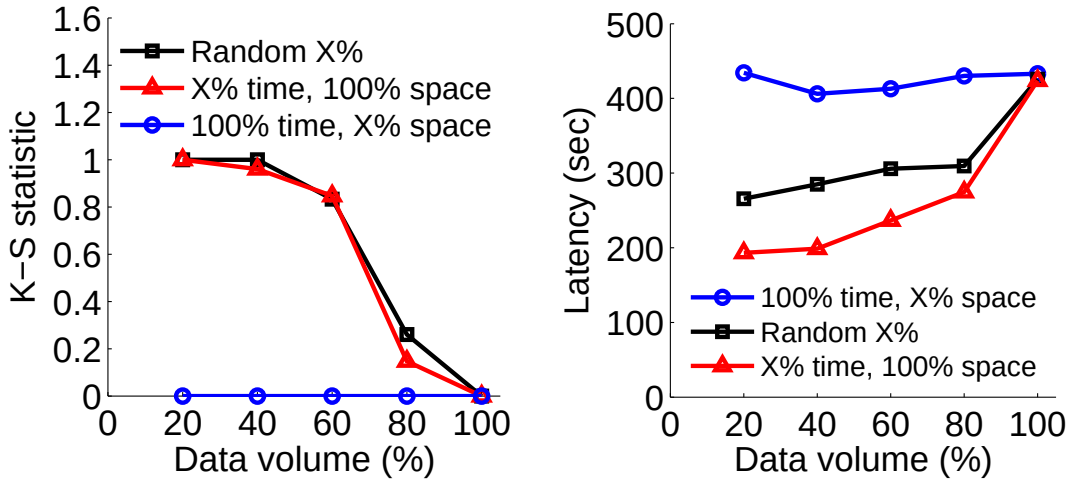
In Figure 4.7(c) we plot the corresponding latency. While random sampling and spatial completeness allow the analysis to finish earlier, as expected, due to the small data size the differences in latency are not significant.

Next, we experiment with a different data distribution, where clicks from a single user exist on a single server. Figure 4.8(a) shows that temporal completeness returns session counts that are perfectly accurate. Since a user's clicks are local to a server, by retrieving all data from a server we are able to accurately reconstruct the user's sessions. Instead, spatial completeness and random sampling incur errors similar to the previous experiment, as expected, since only the distribution across space has changed. However, in Figure 4.8(b) we see that temporal completeness samples less userIDs. As the graph shows, we may improve the percentage of userID discovered by 30-50%, simply by requiring random sampling.

We see that C^2 provides a flexible way for applications to reduce fidelity depending on data distribution and the objective. In this scenario, we may choose to trade the fidelity of the results computed per user, for a wider sample of the user space.

HDFS log analysis

Here, we implement a query that analyzes logs from the Hadoop Distributed File System (HDFS) [8] service, to detect analysis. In the HDFS system, multiple file servers, accept requests from clients to store blocks of a file locally. Among other events, HDFS servers log the time it takes to serve client requests to write



(a) KS-test (b) Latency

Figure 4.9: (a) Results from the Kolmogorov-Smirnov test illustrate the impact of reduced data fidelity on the histograms reported for each HDFS server. (b) For HDFS anomaly detection, random and spatial completeness C^2 improve latency by at least 30%.

blocks locally. By computing the distribution of these times on a per server basis, we can compare pairs of servers and detect anomalies where the distributions differ substantially [73].

We implemented an iMR query with a map function that filters server logs to find all entries that signify the beginning or end of a block write operation. The reduce function matches the beginning and end of a block write operation for each unique block, calculates the duration of such an operation, and maintains a histogram of the block write service times for every server. The resulting histograms represent the distribution of the service times, and are compared according to [73], to detect anomalies.

Similarly to the previous experiments, we measure how data loss affects the accuracy and the latency of the analysis for different C^2 specifications. In this experiment, data loss affects the calculated service time distribution of a server and, essentially, the ability of the application to detect anomalies, resulting either in missing anomalies or in false alerts. We measure the accuracy of the analysis by comparing the observed distribution for every server with the ideal one, that

is, the distribution when there is no data loss. We compare distributions using the Kolmogorov-Smirnov (KS) statistical test [10]. The KS-test determines whether the observed distribution is different from the ideal one.

The data set in this experiment consists of a 48-hour HDFS log trace generated by running the GridMix Hadoop workload generator [7] on a 30-node cluster. Each server generated approximately 2.5GB of data, resulting in a total of 75GB. The iMR query specifies a window of 60 minutes with 20 panes per window.

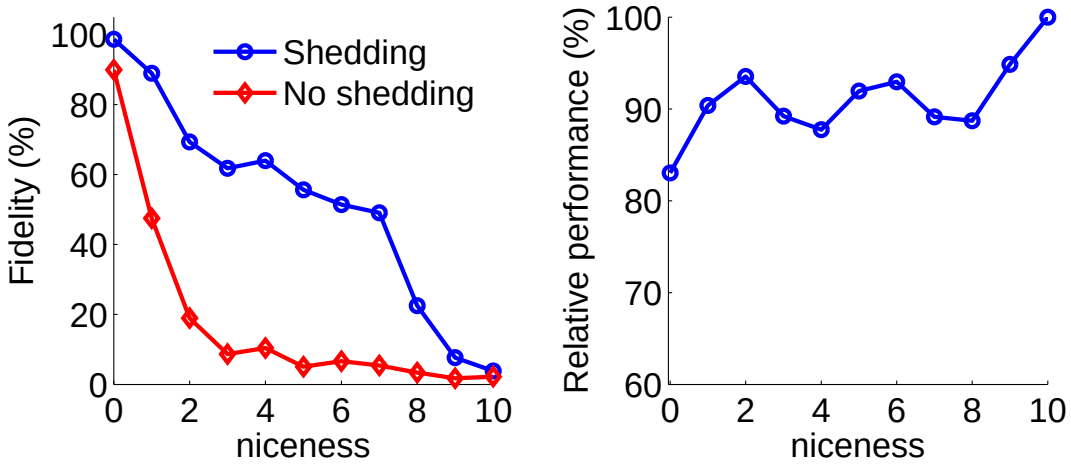
In Figure 4.9(a), we show the fraction of the observed histograms that are different than the ideal ones. We see that since data distributions are computed on a per server basis, temporal completeness returns perfectly accurate data. Even though data form some nodes may be completely lost, the rest of the nodes report all their data, resulting in accurate distributions. Notice that the other C^2 specifications require more than 80% of the data to be processed in order to return results with good quality. However, these specifications can reduce the result latency by approximately 30% at that data volume.

Similarly to the previous applications, central to the choice of the right C^2 specification is prior knowledge about the spatial properties of the data, in this case, that time distributions are calculated on a per server basis. The C^2 metric allows users to leverage such knowledge when choosing the right C^2 specification. At the same time, users have the choice to trade analysis accuracy for reduced latency.

4.2.5 In-situ performance

One of the main design principles of iMR is the co-location of the data analysis with the services generating the data. This implies that CPU resources may be limited for data analysis, since services are typically allocated the majority of the resources. Here, we want to verify the feasibility of running analytics in-situ. Specifically, we will evaluate (i) the ability of iMR to deliver useful results while running side-by-side with a real service, and under time constraints for the analysis, and (ii) the impact on the co-located service.

In particular, we run iMR side-by-side with a Hadoop installation on a 10-



(a) Fidelity

(b) Hadoop performance

Figure 4.10: Fidelity and Hadoop performance as a function of the iMR process niceness. The higher the niceness, the less CPU is allocated to iMR. Hadoop is always given the highest priority, nice = 0.

node cluster. We submit to Hadoop a workload that consists of a variety of Hadoop jobs, generated by the GridMix workload generator [7]. Hadoop is configured to use all the nodes in our compute cluster. At the same time, iMR executes a word count query on the synthetic data set used in Section 4.2.4. The query specifies a window with 2 million records, 20 panes per window, and a 60-second timeout.

In this experiment, we vary the CPU allocated to iMR and measure the quality of the delivered results under the given time constraint of 60 seconds. As less CPU is allocated to iMR, we expect fidelity to drop since iMR will not be able to process an entire window within the time constraint. We vary the CPU allocated by changing the iMR process *niceness*, the priority assigned by the kernel scheduler, and report the fidelity of the returned results. Here, fidelity is equal to the volume of data processed. Additionally, we report the relative change in the Hadoop performance, in terms of jobs completed per time, as the iMR CPU allocation varies. This metric measures the impact of running iMR in-situ on Hadoop.

Figure 4.10(a) shows that without load shedding iMR returns poor results and fidelity drops quickly as iMR is allocated less CPU. In this case, iMR spends

most time trying to process data that will never be delivered before the time constraint. Instead, the load shedding mechanism is able to use available resources intelligently, improving result fidelity by a factor of more than 5.6× most of the time. Notice than when the iMR niceness is set to greater than 9, CPU resources are not sufficient to process data within the 60-second timeout, and fidelity drops significantly.

Figure 4.10(b) shows the relative change performance for Hadoop as the CPU allocated to iMR varies. We measure Hadoop performance as the job completion throughput. For reference, when iMR and Hadoop are assigned the same priority by the scheduler (niceness=0), the cost in Hadoop performance is a 17%-decrease in job throughput. When the iMR niceness is set to 8, at which point iMR can still deliver good quality results, the cost in Hadoop performance is less than a 10%-decrease in job throughput. We see that iMR is able to deliver useful results even when assigned only a small fraction of the CPU. At the same time iMR incurs little impact on the co-located Hadoop system, making in-situ processing a practical approach.

4.3 Acknowledgments

Chapter 4, in part, is reprint of the material published in the Proceedings of the USENIX Annual Technical Conference 2011. Logothetis, Dionysios; Trezzo, Chris; Webb, Kevin C.; Yocum; Ken. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Stateful bulk processing

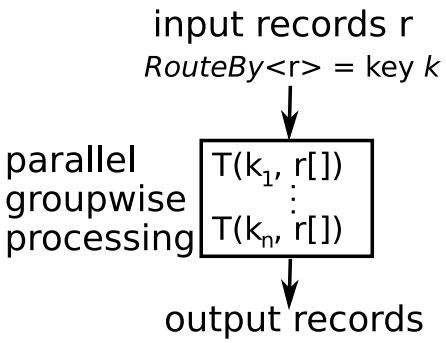
In the previous chapters we presented a system for managing data during the extraction from their sources and early analysis. After the extraction, data are stored for follow-on, richer analysis. Analytics in this phase do not simply filter or summarize data. Instead, they may perform more complex computations, like iterative graph mining and machine learning algorithms, often consisting of large, multi-step dataflows. At the same time, emphasis is given on extracting information from a large body of data, rather than obtaining quick insights over the most recent data.

In these analytics, state remains a key requirement for efficient processing. For instance, many analytics must often incorporate large batches of newly collected data in an efficient manner, and use state to avoid recomputation. This chapter presents CBP, a complementary architecture for stateful analytics on bulk data. CBP provides a programming model and runtime system for sophisticated stateful analytics. Here, we describe the basic constructs of the model and show how it allows users to program sophisticated stateful analytics. We illustrate how CBP can be used to program (i) incremental analytics that must absorb large batches of continuously arriving data, and (ii) iterative analytics.

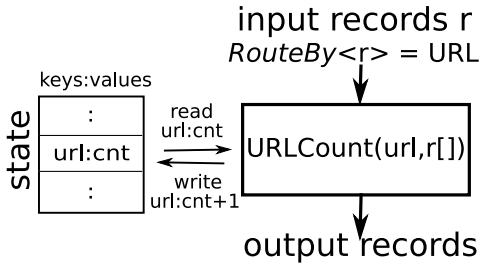
A core component of CBP is a flexible, stateful groupwise operator, *translate*, that cleanly integrates state into data-parallel processing. This basic operation allows users to write stateful analytics and affords several fundamental opportunities for minimizing data movement in the underlying processing system.

Additionally, CBP allows users to compose sophisticated dataflows using the *translate* operator as the building block, and introduces primitives for dataflow management. Continuous dataflows require control for determining stage execution and input data consumption. The CBP allows dataflow control through simple yet flexible primitives. These features simplify the construction of incremental and iterative programs for large, evolving data sets.

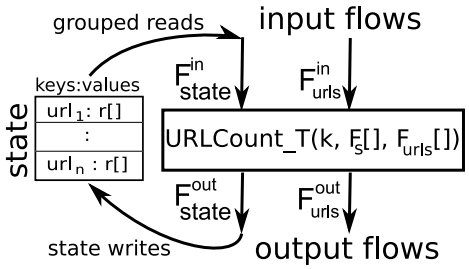
5.1 A basic translate operator



(a) Basic groupwise processing.



(b) Groupwise processing with access to state.



(c) Grouping input with state records.

Figure 5.1: The progression from a stateless groupwise processing primitive to stateful translation, $T(\cdot)$, with multiple inputs/outputs, grouped state, and inner groupings.

We begin by studying the incremental crawl queue (Figure 1.3.1) dataflow in more detail, where each stage is a separate translation operator. We illustrate translate with a simplified version of the *count in-links* stage, called *URLCount*, that only maintains the frequency of observed URLs. This stateful processing stage

has a single input that contains URLs extracted from a set of crawled web pages. The output is the set of URLs and counts that changed with the last set of input records.

For illustration, Figure 5.1 presents a progression from a stateless groupwise primitive, such as *reduce*, to our proposed translate operator, $T(\cdot)$, which will eventually implement *URLCount*. Figure 5.1(a) shows a single processing *stage* that invokes a user-defined translate function, $T(\cdot)$. To specify the grouping keys, users write a $RouteBy\langle r\rangle$ function that extracts the grouping key from each input record r. In the case of *URLCount*, *RouteBy* extracts the URL as the grouping key. When the groupwise operator executes, the system reads input records, calls *RouteBy*, groups by the key k, partitions input data (we illustrate a single partition), and runs operator replicas in parallel for each partition. Each replica then calls $T(\cdot)$ for each grouping key k with the associated records, $r[]$. We call each parallel execution of an operator an *epoch*.

To maintain a frequency count of observed URLs, the *URLCount* translator needs access to state that persists across epochs. Figure 5.1(b) adds a logical state module from which a translate function may read or write values for the current grouping key. In our case, *URLCount* stores counts of previously seen URLs, maintaining state records of the type $\{url, count\}$. However, as the next figure shows, translate incorporates state into the grouping operation itself and the semantics of reading and writing to this state module are different than using an external table-based store.

Figure 5.1(c) shows the full-featured translation function:
$T : \langle k, F_S^{in}, F_1^{in}, \ldots, F_n^{in}\rangle$, with multiple logical input and output *flows* and grouped state. As the figure shows, we found it useful to model state using explicit, *loopback* flows from a stage output to a stage input. This allows translate to process state records like any other input, and avoids custom user code for managing access to an external store. It also makes it simple for the system to identify and optimize flows that carry state records. For simple stateful translators like $URLCount$ one loopback suffices, F_S^{out} to F_S^{in}.

Figure 5.2 shows pseudocode for our *URLCount* translate function called

```
URLCount_T(url, F^in_state[], F^in_urls[])
1    newcnt ← F^in_urls.size()
2    if F^in_state[0] ≠ NULL then
3        newcnt ← newcnt + F^in_state[0].cnt
4    F^out_state.write({url, newcnt})
5    F^out_updates.write({url, newcnt})
```

Figure 5.2: Translator pseudocode that counts observed URLs. The translator reads and updates the saved count.

within this stage. With multiple logical inputs, it is trivial to separate state from newly arrived records. It counts the number of input records grouped with the given *url*, and writes the updated counts to state and an output flow for downstream stages. A translation stage must explicitly write each state record present in F_S^{in} to F_S^{out} to retain them for the next processing epoch. Thus a translator can discard state records by not propagating them to the output flow. Note that writes are not visible in their groups until the following epoch.

We can optimize the *URLCount* translator by recognizing that F_{urls}^{in} may update only a fraction of the stored URL counts each epoch. Current bulk-processing primitives provide "full outer" groupings, calling the groupwise function for all found grouping keys. Here *URLCount* takes advantage of translation's ability to also perform "inner" groupings between state and other inputs. These inner groupings only call translate for state records that have matching keys from other inputs, allowing the system to avoid expensive scans of the entire state flow. However, to improve performance this requires the underlying processing system to be able to randomly read records efficiently (Section 6.5.2).

5.2 Continuous bulk processing

We now turn our attention to creating more sophisticated translators that either iterate over an input or, in incremental environments, continuously process newly arrived data. A key question for CBP is how to manage continuous data arrivals. For example, an incremental program typically has an external process

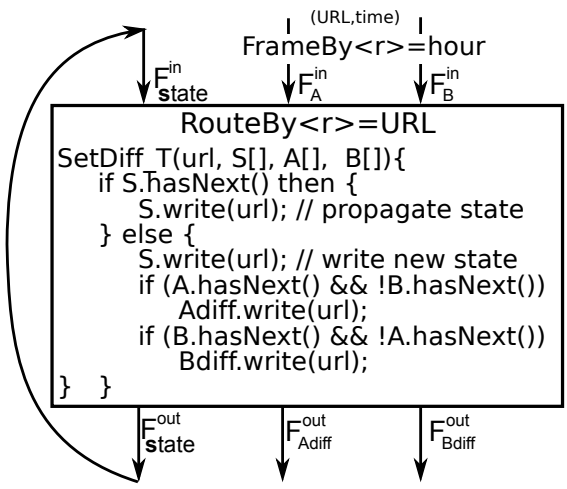


Figure 5.3: A stage implementing symmetric set difference of URLs from two input crawls, A and B.

creating input. CBP systems must decide when to run each stage based on the records accumulating on the input flows. In some cases they may act like existing bulk-processing systems, in which a vertex (a Dryad vertex or a Map-Reduce job) runs when a batch of records exists on each input. They may behave in a manner similar to data stream processors [15], which invoke a dataflow operator when any input has a single tuple available. Or they may behave in some hybrid fashion.

During each processing epoch, the translator, $T(\cdot)$, reads zero or more records from each input flow, processes them, and writes zero or more records to output flows. Thus a flow F is a sequence of records passed between two processing stages over time. The sequence of records read from a given input flow is called an *input increment*, and a special *input framing* procedure determines the sizes of the input increments. The sequence of records output to a given flow during one epoch form an *output increment*. CBP couples the framing function with a second function, *runnability*, which governs the eligibility of a stage to run (Section 6.2) and also controls consumption of input increments.

We illustrate these concepts by using a CBP program to compare the output of two experimental web crawlers, A and B. The stage, illustrated in Figure 5.3, has an input from each crawler whose records contain (url,timestamp) pairs. Similarly, there is an output for the unique pages found by each crawler. The translator

implements symmetric set difference, and we would like to report this difference for each hour spent crawling.[1]

First, the stage should process the same hour of output from both crawlers in an epoch. A CBP stage defines per-flow $FrameBy\langle r\rangle$ functions to help the system determine the input increment membership. The function assigns a *framing key* to each record, allowing the system to place consecutive records with identical framing keys into the same increment. An increment is not eligible to be read until a record with a different key is encountered.[2] Here, $FrameBy$ returns the hour at which the crawler found the URL as the framing key.

However, the stage isn't *runnable* unless we have an hour's worth of crawled URLs on *both* F_A^{in} and F_B^{in}. A stage's runnability function has access to the status of its input flows, including the framing keys of each complete increment. The function returns a Boolean value to indicate whether the stage is eligible to run, as well as the set of flows from which an increment is to be consumed and the set from which an increment is to be removed.

For our symmetric set difference stage, *runnability* returns **true** iff both input flows contain eligible increments. If both input flow increments have the same framing key, the runnability function indicates that both should be read. On the other hand, if the framing keys differ, the runnability function selects only the one with the smaller key to be read. This logic prevents a loss of synchronization in the case that a crawler produces no data for a particular hour.

Finally, the stage's translation function, SetDiff_T, is ready to process observed URLs, storing them in state records. This stage's $RouteBy\langle r\rangle$ function extracts the URL from each input record as the grouping key for state and crawler records. If there is a state record for this url, then it either was reported in a prior epoch or belongs to both crawls (the intersection). In this case the translator only needs to manually propagate the state record. Otherwise, this URL has not been seen and it is written to state. If it was seen exclusively by either crawl, we add it

[1] Note that this is the *change* in unique URLs observed; the outputs won't include re-crawled pages (though that is easily done).

[2] The use of *punctuations* [15] can avoid having to wait for a new key, although we have not implemented this feature.

to the appropriate output flow.

Framing and runnability are a powerful combination that allows stages to determine what data to present to a stage, and to synchronize consumption of data across multiple input flows. As with framing functions, runnability functions may maintain a small amount of state. Thus it may contain significant control logic. We have used it to synchronize inputs (e.g., for temporal joins), properly interleave writes to and reads from state, and to maintain static lookup tables (read but not remove an increment). Finally, applications such as PageRank can use it to transition from one iterative phase to another, as we show in Section 5.5.3.

5.3 Support for graph algorithms

Groupwise processing supports obvious partitionings of graph problems by assigning a single group to each vertex or edge. For example, programmers can write a single translator that processes all vertices in parallel during each processing epoch. In many cases, those per-vertex translation instances must access state associated with other vertices. To do so, each vertex sends "messages" to other vertices (addressed by their grouping key) so that they may exchange data. Such message passing is a powerful technique for orchestrating large computations (it also underlies Google's graph processing system, Pregel [56]), and the CBP model supports it.

Translation complements message passing in a number of ways. First, using a second loopback flow to carry messages allows an inner grouping with the state used to store the graph. Thus the system will call translate only for the groups representing message destinations. Second, message passing can take advantage of the generality of the *RouteBy* construct.

Often a computation at a single vertex in the graph affects some or all of the vertices in the graph. For example, our incremental PageRank translator (Section 5.5.3) must broadcast updates of rank from dangling nodes (nodes w/o children) to all other nodes in the graph. Similarly, an update may need to be sent to a subset of the nodes in the graph. While *RouteBy* can return any number of

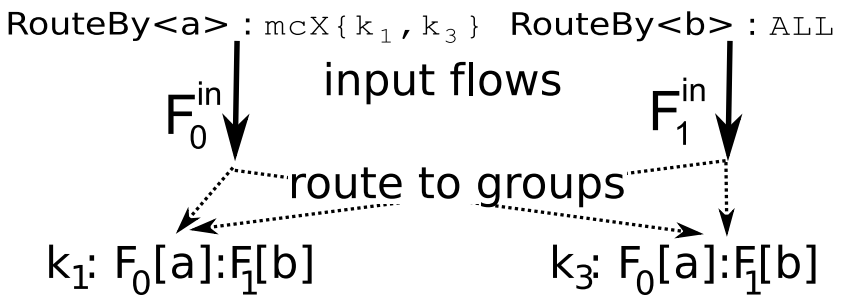


Figure 5.4: Users specify per-input flow *RouteBy* functions to extract keys for grouping. Special keys enable the broadcast and multicast of records to groups. Here we show that multicast address mcX is bound to keys k_1 and k_3.

grouping keys from within a record, there is no simple way for a translator to write a record that includes all nodes in the graph. It is difficult to know the broadcast (or multicast) keyset *a-priori*.

To address this issue, *RouteBy* supports logical broadcast and multicast grouping keys. Figure 5.4 shows *RouteBy* returning the special ALL broadcast key for the input record on F_1^{in}. This ensures that the record b becomes associated with all groups found in the input flows. While not shown, it is also possible to limit the broadcast to particular input flows, e.g., only groups found in state. Translators may also associate a subset of grouping keys with a single logical multicast address. Here *RouteBy* on input flow F_0^{in} returns a multicast address, mcX, associated with grouping keys k_1 and k_3. We describe both mechanisms in more detail in Section 6.5.3.

5.4 Summary of CBP model

Naturally, multiple translation stages may be strung together to build more sophisticated incremental programs, such as the incremental crawl queue. In general, a CBP program itself (like Figure 1.3.1) is a directed graph $\mathcal{P}$, possibly containing cycles, of translation stages (the vertices), that may be connected with multiple directed flows (the edges). Here we summarize the set of dataflow control primitives in our CBP model that orchestrate the execution of stateful dataflow programs.

Table 5.1: Five functions control stage processing. Default functions exist for each except for translation.

Function	Description	Default
Translate(Key,$\Delta F_0^{in}, \ldots, \Delta F_n^{in}$)→ ($\Delta F_0^{out}, \ldots, \Delta F_n^{out}$)	**Per-Stage:** Groupwise transform from input to output records.	—
Runnable(framingKeys, state) → (reads, removes, state)	**Per-Stage:** Determines if stage can execute and what increments are read/removed.	RunnableALL
FrameBy(r, state) → (Key, state)	**Per-Flow:** Assign records to input increments.	FrameByPrior
RouteBy(r) → Key	**Per-Flow:** Extract grouping key from record.	RouteByRcd
OrderBy(r) → Key	**Per-Flow:** Extract sorting key from record.	OrderByAny

As our examples illustrate, CBP controls stage processing through a set of five functions, listed in Table 5.1. An application may choose these functions, or accept the system-provided defaults (except for translate). The default framing function FrameByPrior returns the epoch number in which the upstream stage produced the record, causing input increments to match output increments generated by upstream stages. The default runnability function, RunnableAll, makes a stage runnable when all inputs have increments and then reads and removes each.

The default *RouteBy* function, RouteByRcd, gives each record its own group for record-wise processing. Such translators can avoid expensive grouping operations, be pipelined for one-pass execution over the data, and avoid state maintenance overheads. Similarly, the *OrderBy* function, another key-extraction function that provides per-flow record ordering, has a default OrderByAny, which lets the system select an order that may improve efficiency (e.g., using the order in which the data arrives).

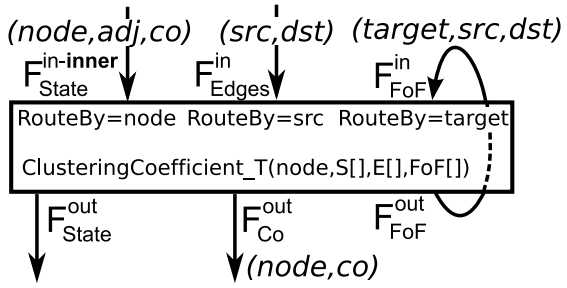


Figure 5.5: Incremental clustering coefficient dataflow. Each node maintains as state its adjacency list and its "friends-of-friends" list.

5.5 Applications

The collection of default behaviors in the CBP model support a range of important incremental programs, such as the incremental crawl queue example, which uses RunnableAll and FrameByPrior for all its stages. Here we showcase the extra flexibility the model provides by building stateful, iterative algorithms that operate on graphs.

5.5.1 Mining evolving graphs

Many emerging data mining opportunities operate on large, evolving graphs. Instances of data mining such graphs can be found in systems biology, data network analysis, and recommendation networks in online retail (e.g., Netflix). Here we investigate algorithms that operate over Web and social network graphs. The Web is perhaps the canonical example of a large, evolving graph, and we study an incremental version of the PageRank [62] algorithm used to help index its content. On the other hand, the explosive growth of community sites, such as MySpace or Facebook, have created extremely large social network graphs. For instance, Facebook has over 300 million active users (as of September 2009, see www.facebook.com/press). These sites analyze the social graph to support day-to-day operations, external querying (Facebook Lexicon), and ad targeting.

```
ClusteringCoefficient_T(node,F^in_state,F^in_edges,F^in_FoF)
1    if F^in_state.hasNext() then state ← F^in_state.next()
2    foreach edge in F^in_edges
3        state.adj.add(edge.dst);
4    foreach edge in F^in_edges
5        foreach target in state.adj
6            F^out_FoF.write(target,edge.src,edge.dst);
7    foreach update in F^in_FoF
8        state.adj[update.src].adj.add(update.dst);
9    if F^in_FoF.hasNext() then
10       recalcCo(state); F^out_Co.write(node,state.co);
11   F^out_state.write(state);
```

Figure 5.6: The clustering coefficients translator adds new edges (2-3), sends neighbors updates (4-6), and processes those updates (7-10).

5.5.2 Clustering coefficients

We begin with a simple graph analysis, clustering coefficient, that, among other uses, researchers employ to ascertain whether connectivity in social networks reflects real-world trust and relationships [77]. This example illustrates how we load graphs into a stateful processing stage, how to use groupwise processing to iteratively walk across the graph, and how messages may be used to update neighbor's state.

The clustering coefficient of a graph measures how well a graph conforms to the "small-world" network model. A high clustering coefficient implies that nodes form tight cliques with their immediate neighbors. For a node n_i, with N neighbors and E edges among the neighbors, the clustering coefficient $c_i = 2E/N(N-1)$.This is simple to calculate if each node has a list of its neighbor's neighbors. In a social network this could be described as a "friends-of-friends" (FoF) relation.

For graph algorithms, we create a grouping key for each unique node in the graph. This allows the calculation to proceed in parallel for each node during an epoch, and us to store state records describing each vertex. Figure 5.5 illustrates

the single stateful stage for incrementally computing clustering coefficients.[3] The input F^{in}_{edges} carries changes to the graph in the form of (src,dst) node ID pairs that represent edges. Records on the state flow reference the node and its clustering coefficient and FoF relation. Each input's *RouteBy* returns a node ID as the grouping key.

Figure 5.6 shows the translator pseudocode. The translator must add new graph nodes[4], update adjacency lists, and then update the FoF relations and clustering coefficients. Line 1 retrieves a node's state (an adjacency list, adj, of adjacencies). Each record on F^{in}_{edges} represents a new neighbor for this node. Lines 2-3 add these new neighbors to the local adjacency list. While that code alone is sufficient to build the graph, we must also send these new neighbors to every adjacent node so that they may update their FoF relation.

To do so, we send a record to each adjacent node by writing to the loopback flow F^{out}_{FoF} (lines 4-6). During the next epoch, *RouteBy* for F^{in}_{FoF} routes these records to the node designated by target. When the system calls *translate* for these nodes, lines 7-10 process records on F^{in}_{FoF}, updating the FoF relation and recalculating the clustering coefficient. Finally, line 11 propagates any state changes. Note that the runnability function allows the stage to execute if input is available on *any* input. Thus during one epoch, a translate instance may both incorporate new edges and output new coefficients for prior changes.

There are several important observations. First, it takes two epochs to update the cluster coefficients when the graph changes. This is because "messages" cannot be routed until the following epoch. Second, Figure 5.5 shows state as an "inner" flow. Thus translation *only* occurs for nodes that have new neighbors (input on F^{in}_{edges}) or must update their coefficient (input on F^{in}_{FoF}). These two flows actively select the graph nodes for processing each epoch. Finally, where a single input record into the *URLCount* translator causes a single state update, here the work created by adding an edge grows with the size of state. Adding an edge creates messages to update the FoF relation for all the node's neighbors. The message count (and size) grows as the size and connectivity of the graph increase.

[3]Going forward we hide the loop in state loopback flows.

[4]For ease of exposition we do not show edge deletions.

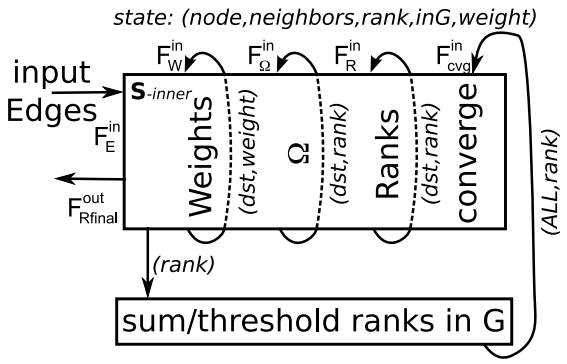


Figure 5.7: Incremental PageRank dataflow. The loopback flows are used to propagate messages between nodes in the graph.

We explore these implications further in Section 6.6.3.

5.5.3 Incremental PageRank

PageRank is a standard method for determining the relative importance of web pages based on their connectivity [62]. Incremental PageRank is important because (1) computing PageRank on the entire web graph still takes hours on large clusters and (2) important changes to the web graph occur on a small subset of the web (news, blogs, etc.). However, truly incremental PageRank is challenging because small changes (adding a link between pages) can propagate throughout the entire graph. Here we implement the approximate, incremental PageRank computation presented in [28], which thresholds the propagation of PageRank updates. This algorithm takes as input a set of link insertions in the web graph; other approaches exist to incorporate node additions and removals [28].

Figure 5.7 illustrates our incremental PageRank dataflow, which shares many features with clustering coefficient. It uses the same format for input edges, groups records by vertex, stores adjacency lists in state records, uses an inner state flow, and sends "messages" to other nodes on loopback flows. We skip the sundry details of translation, and instead focus on how to manage an algorithm that has several distinct iterative phases.

At a high level, the algorithm must build the graph W, find the subgraph G affected by newly inserted edges, compute transition probabilities to a supernode

```
IncrPageRank_T(node,F_S^in,F_E^in,F_R^in,F_W^in,F_Cvg^in,F_Ω^in)
1    if F_E^in.hasNext() then makeGraph();startWeight();
2    if F_W^in.hasNext() then sendWeightToNeighbors();
3    if F_Ω^in.hasNext() then updateSupernode();
4    if F_Cvg^in.hasNext() then resetRankState();
5    elseif F_R^in.hasNext() then
6        doPageRankOnG();
```

Figure 5.8: Pseudocode for incremental PageRank. The translator acts as an event handler, using the presence of records on each loopback flow as an indication to run a particular phase of the algorithm.

Ω ($W - G$), and then compute PageRank for G (pages in Ω retain their rank). This algorithm has been shown to be both fast and to provide high-quality approximations for a variety of real and synthesized web crawls [28].

Figure 5.8 shows high-level pseudocode for the PageRank translator. Internally, the translator acts as a per-node event handler, using the presence of records on each loopback flow as an indication to run a particular phase of the algorithm. Here the *runnability* function plays a critical role in managing phase transitions; it exclusively reads each successive phase's input after the prior input becomes empty. Thus *runnability* first consumes edges from F^{in}_{edges}, then F^{in}_{W} (to find G), then F^{in}_{Ω} (updating the supernode), and finally F^{in}_{R} (to begin PageRank on G). When doPageRankOnG converges, the second stage writes an ALL record to F^{out}_{Cvg}. This causes the translator to reset graph state, readying itself for the next set of edge insertions.

This design attempts to minimize the number of complete scans of the nodes in W by using both "inner" state flows and the multicast ability of the *RouteBy* function. For example, when calculating PageRank for G, leaves in G multicast their PageRank to only nodes in G. We discuss the multicast API more in Section 6.5.3. Finally, note that we place all the phases in a single translator. Other organizations are possible, such as writing a stage for each phase, though this may make multiple copies of the state. In any case, we envision such analytics as just one step in a larger dataflow.

5.6 Related work

Non-relational bulk processing: This work builds upon recent non-relational bulk processing systems such as Map-Reduce [34] and Dryad [46]. Our contributions beyond those systems are two-fold: (1) a programming abstraction that makes it easy to express incremental computations over incrementally-arriving data; (2) efficient underlying mechanisms geared specifically toward continuous, incremental workloads.

A closely related effort to CBP enhances Dryad to automatically identify redundant computation; it caches prior results to avoid re-executing stages or to merge computations with new input [67]. Because these cached results are outside the dataflow, programmers cannot retrieve and store state during execution. CBP takes a different approach, providing programmers explicit access to persistent state through a familiar and powerful groupwise processing abstraction.

Our work also complements recent efforts to build "online" Map-Reduce systems [32]. While their data pipelining techniques for Map-Reduce jobs are orthogonal to the CBP model, the work also describes a controller for running Map-Reduce jobs continuously. The design requires reducers to manage their own internal state, presenting a significant programmer burden as it remains outside of the bulk-processing abstraction. The controller provides limited support for deciding when jobs are runnable and what data they consume. In contrast, CBP dataflow primitives afford a range of policies for controlling these aspects of iterative/incremental dataflows.

Twister [36], a custom Map-Reduce system, optimizes repeatedly run (iterative) Map-Reduce jobs by allowing access to static state. Map and Reduce tasks may persist across iterations, amortizing the cost of loading this static state (e.g., from an input file). However, the state cannot change during iteration. In contrast, CBP provides a general abstraction of state that supports inserts, updates, and removals.

Data stream management: CBP occupies a unique place between traditional DBMS and stream processing. Data stream management systems [15] focus on near-real-time processing of continuously-arriving data. This focus leads to an

in-memory, record-at-a-time processing paradigm, whereas CBP deals with disk-resident data and set-oriented bulk operations. Lastly, CBP permits cyclic data flows, which are useful in iterative computations and other scenarios described below.

Incremental view maintenance: Traditional view-maintenance environments, like data warehousing, use declarative views that are maintained implicitly by the system [18, 68]. In contrast, CBP can be thought of as a platform for generalized view-maintenance; a CBP program is an explicit graph of data transformation steps. Indeed, one can support relational view maintenance on top of our framework, much like relational query languages have been layered on top of Map-Reduce and Dryad (e.g., DryadLINQ [82], Hive [9], Pig [61]).

5.7 Acknowledgments

Chapter 5, in part, is reprint of the material published in the Proceedings of the ACM Symposium on Cloud Computing 2010. Logothetis, Dionysios; Olston, Christopher; Reed, Benjamin; Webb, Kevin C.; Yocum Ken. The dissertation author was the primary investigator and author of this paper.

Chapter 6

CBP design and implementation

This chapter describes the design and implementation of the CBP runtime system. We describe how CBP allows the orchestration of dataflows and how it can reliably and efficiently execute them. Furthemore, we illustrate the fundamental mismatch between DISC systems and stateful computations by comparing two implementations of the CBP model: (i) a "black-box" implementation on top of a state-of-the-art DISC system, and (ii) a direct implementation of the CBP model. Our evaluation shows how the CPB runtime that directly supports the model significantly improves processing time and reduces resources usage.

The CBP architecture has two primary layers: dataflow and physical. The physical layer reliably executes and stores the results of a single stage of the dataflow. Above it, the dataflow layer provides reliable execution of an entire CBP dataflow, orchestrating the execution of multiple stages. It ensures reliable, ordered transport of increments between stages and determines which stages are ready for execution. The dataflow layer may also compile the logical dataflow into a more efficient physical representation, depending on the execution capabilities of the physical layer. Such automated analysis and optimization of a CBP dataflow is future work.

6.1 Controlling stage inputs and execution

The dataflow layer accepts a CBP dataflow and orchestrates the execution of its multiple stages. The incremental dataflow controller (IDC) determines the set of runnable stages and issues calls to the physical layer to run them.

The IDC maintains a *flow connector*, a piece of run-time state, for each stage's input flow. Each flow connector logically connects an output flow to its destination input flow. It maintains a logical, ordered queue of identifiers that represent the increments available on the associated input flow. Each output flow may have multiple flow connectors, one for each input flow that uses it as a source. After a stage executes, the IDC updates the flow connectors for each output flow by enqueueing the location and *framing* key of each new output increment. The default, with a DefaultFraming *framing* function, is for the stage to produce one output increment per flow per epoch.

The IDC uses a stage's *runnable* function to determine whether a stage can be run. The system passes the function the set of flow connectors with un-read increments and the associated framing keys, and an application-defined piece of state. The *runnable* function has access to each flow connector's meta data (e.g., number of enqueued increments) and determines the set of flow connectors from which to read, *readSet*, and remove, *removeSet*, increments for the next epoch. If the *readSet* is empty, the stage is not runnable. After each epoch, the IDC updates each flow connector, marking increments as read or removing increment references. Increments may be garbage collected when no flow connector references them.

6.2 Scheduling with bottleneck detection

The IDC must determine the set of runnable stages and the order in which to run them. Doing so with prior bulk processing systems is relatively straightforward, since they take a DAG as input. In that case a simple on-line topological sort can determine a vertex (stage) execution order that respects data dependencies. However, CBP presents two additional criteria. First, $\mathcal{P}$ may contain cycles, and the scheduler must choose a total order of stages to avoid starvation or high result

latency (makespan). Second, using the runnability function, stages can prefer or synchronize processing particular inputs. This means that increments can "back up" on input flows, and that the stage creating data for that input no longer needs to run.

Our simple scheduler executes in phases and may test each stage's *runnability* function. It can detect stage starvation and respond to downstream backpressure (a bottleneck stage) by not running stages that already have increments in all outputs. Full details of this algorithm are available in our technical report [52]).

6.3 Failure recovery

The dataflow layer assumes that the physical layer provides atomic execution of individual stages and reliable storage of immutable increments. With such semantics, a single stage may be restarted if the physical layer fails to run a stage. The executed stage specifies a naming convention for each produced increment, requiring it to be tagged by its source stage, flow id, and increment index. These may be encoded in the on-disk path and increment name. Once the physical layer informs the IDC of success, it guarantees that result increments are on disk. Dryad used similar techniques to ensure dataflow correctness under individual job failures [46].

Next, the IDC updates the run-time state of the dataflow. This consists of adding and deleting increment references on existing flow connectors. The controller uses write-ahead logging to record its intended actions; these intentions contain snapshots of the state of the flow connector queue. The log only needs to retain the last intention for each stage. If the IDC fails, it rebuilds state from the XML dataflow description and rebuilds the flow connectors and scheduler state by scanning the intentions.

6.4 CBP on top of Map-Reduce

We divide the design and implementation of the CBP model into two parts. In the first part we map *translate* onto a Map-Reduce model. This is a reasonable starting point for the CBP physical layer due to its data-parallelism and fault-tolerance features. However, this provides an incomplete implementation of the translate operator and CBP dataflow primitives. Further, such a "black-box" emulation results in excess data movement and space usage, sacrificing the promise of incremental dataflows (Section 6.6). The next section describes our modifications to an open-source Map-Reduce, Hadoop, that supports the full CBP model and optimizes the treatment of state.

The design of our bulk-incremental dataflow engine builds upon the scalability and robustness properties of the GFS/Map-Reduce architecture [39, 34], and in particular the open-source implementation called *Hadoop*. Map-Reduce allows programmers to specify data processing in two phases: map and reduce. The map function outputs a new key-value pair, $\{k_1, v_1\}$, for each input record. The system creates a list of values, $[v]_1$, for each key and passes these to reduce. The Map-Reduce architecture transparently manages the parallel execution of the map phase, the grouping of all values with a given key (the sort), and the parallel execution of the reduce phase.

We now describe how to emulate a single CBP stage using a single Map-Reduce job.[1] Here we describe the Map and Reduce "wrapper" functions that export translate $T(\cdot)$. In CBP applications data is opaque to the processing system, and these wrapper functions encapsulate application data (a record) inside an *application data unit* (ADU) object. The ADU also contains the `flowID`, `RouteByKey`, and `OrderByKey`.

While the Map-Reduce model has one logical input and output, current implementations allow a Map-Reduce job to process multiple input and write multiple output files. In CBP, the `flowID`s within each ADU logically separate flows, and the wrapper code uses the `flowID` to invoke per-flow functions, such as *RouteBy*

[1] An efficient implementation of CBP over a Map-Reduce environment requires deterministic and side-effect-free translators.

and *OrderBy* that create the routing and ordering keys. This "black-box" approach emulates state as just another input (and output) file of the Map-Reduce job.

- **Map:** The map function wrapper implements routing by running the *RouteBy* function associated with each input flow. It wraps each input record into an ADU and sets the `flowID`, so the reduce function can separate data originating from the different flows. Map functions may also run one or more *preprocessors* that implement record-wise translation. The optional Map-Reduce combiner has also been wrapped to support distributive or algebraic translators.
- **Reduce:** The Hadoop reducer facility sorts records by the `RouteByKey` embedded in the ADU. Our CBP reduce wrapper function multiplexes the sorted records into n streams, upcalling the user-supplied translator function $T(\cdot)$ with an iterator for each input flow. Per-flow emitter functions route output from $T(\cdot)$ to HDFS file locations specified in the job description. Like the map, emitter functions may also run one or more per-record *postprocessing* steps before writing to HDFS.

Thus a single groupwise translator becomes a job with a map/reduce pair, while a record-wise translator can be a map-only job (allowed by Hadoop) or a reduce postprocessor.

6.4.1 Incremental crawl queue example

We illustrate the compilation of a CBP dataflow into Map-Reduce jobs using our incremental crawl queue examples from Figure 1.3.1. This dataflow is compiled into two Map-Reduce jobs: CountLinks and DecideCrawl. Figure 6.1 shows the two jobs and which stages each wrapper function implements. In both jobs all input flows *RouteBy* the site, and order input by the URL. Otherwise all input flows use the default framing and runnability functions. The first Map-Reduce job implements both *extract links* and *count in-links*. It writes state ADUs with both site and URL routing keys to maintain counts for each. The second job

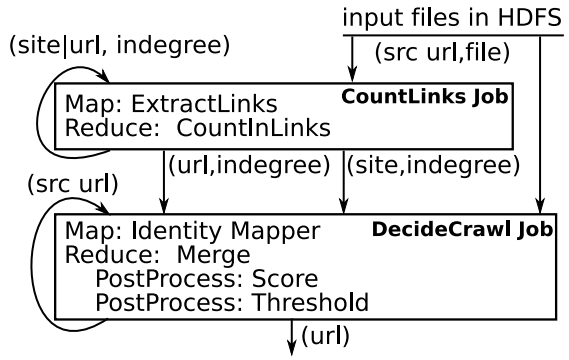


Figure 6.1: The Map-Reduce jobs that emulate the CBP incremental crawl queue dataflow.

places both *score* and *threshold* as postprocessing steps on the groupwise *merge* translator. This state flow records all visited src URLs.

6.4.2 Increment management

Map-Reduce implementations use shared file systems as a reliable mechanism for distributing data across large clusters. All flow data resides in the Hadoop distributed file system (HDFS). The controller creates a *flow directory* for each flow F and, underneath that, a directory for each increment. This directory contains one or more files containing the ADUs. As discussed in Section 6.2, when Hadoop signals the successful completion of a stage, the controller updates all affected flow connectors.

We emulate custom (non-default) framing functions as post processing steps in the upstream stage whose output flow the downstream stage sources. The reduce wrapper calls the `framing` function for each ADU written to that output flow. By default, the increment directory name is the stage's processing epoch that generated these ADUs. The wrapper appends the resulting `FramingKey` to the increment directory name and writes ADUs with that key to that directory. The wrapper also adds the `FramingKey` to the meta data associated with this increment in the input flow's flow connector. This allows a stage's *runnable* function to compare those keys to synchronize input increments, as described in Section 5.2.

6.5 Direct CBP

We now modify Hadoop to accommodate features of the CBP model that are either inexpressible or inefficient as "black-box" Map-Reduce emulations. The first category includes features such as broadcast and multicast record routing. The second category optimizes the execution of bulk-incremental dataflows to ensure that data movement, sorting, and buffering work are proportional to arriving input size, not state size.

6.5.1 Incremental shuffling for loopback flows

The system may optimize state flows, and any loopback flow in general, by storing state in per-partition side files. Map-Reduce architectures, like Hadoop, transfer output from each map instance or *task* to the reduce tasks in the *shuffle* phase. Each map task partitions its output into R sets, each containing a subset of the input's grouping keys. The architecture assigns a reduce task to each partition, whose first job is to collect its partition from each mapper.

Hadoop, though, treats state like any other flow, re-mapping and re-shuffling it on each epoch for every groupwise translator. Shuffling is expensive, requiring each reducer to source output from each mapper instance, and state can become large relative to input increments. This represents a large fraction of the processing required to emulate a CBP stage.

However, state is local to a particular translate instance and only contains ADUs assigned to this translate partition. When translators update or propagate existing state ADUs in one epoch, those ADUs are already in the correct partition for the next epoch. Thus we can avoid re-mapping and re-shuffling these state ADUs. Instead, the reduce task can write and read state from/to an HDFS *partition* file. When a reducer starts, it references the file by partition and merge sorts it with data from the map tasks in the normal fashion.

Note that a translator instance may add state ADUs whose *RouteBy* key belongs to a remote partition during an epoch. These *remote* writes must be shuffled to the correct partition (translation instance) before the next epoch. We

accomplish this by simply testing ADUs in the loopback flow's emitter, splitting ADUs into two groups: local and remote. The system shuffles remote ADUs as before, but writes local ADUs to the partition file. We further optimize this process by "pinning" reduce tasks to a physical node that holds a replica of the first HDFS block of the partition file. This avoids reading data from across the network by reading HDFS data stored on the local disk. Finally, the system may periodically re-shuffle the partition files in the case of data skew or a change in processor count.

6.5.2 Random access with BIPtables

Here we describe BIPtables (bulk-incremental processing tables), a simple scheme to *index* the state flow and provide random state access to state. This allows the system to optimize the execution of translators that update only a fraction of state. For example, a translator may specify an *inner* state flow, meaning that the system only needs to present state ADUs whose *RouteBy* keys also exist on other inputs. But current bulk-processing architectures are optimized for "streaming" data access, and will read and process inputs in their entirety. This includes direct CBP with state partition files (described above), which reads the entire partition file even if the translator is extremely selective.

However, the success of this approach depends on reading and writing matched keys randomly from a table faster than reading and writing all keys sequentially from a file. Published performance figures for Bigtable, a table-based storage infrastructure [25], indicate a four to ten times reduction in performance for random reads relative to sequential reads from distributed file systems like GFS[39] for 1000-byte records. Moreover, our recent investigation indicates even achieving that performance with open-source versions, such as Hypertable, is optimistic, requiring operations to select under 15% of state keys to improve performance [52]. The design outlined below outperforms sequential when retrieving as many as 60% of the state records (Section 6.6.2).

BIPtables leverages the fact that our CBP system needs only simple (key, ADUs) retrieval and already partitions and sorts state ADUs, making much of the functionality in existing table-stores redundant or unnecessary. At a high level,

each state partition now consists of an *index* and *data* file. While similar to HDFS `MapFile`s or Bigtable's `SSTable` files, they are designed to exist across multiple processing epochs. Logically, the data file is an append-only, unsorted log that contains the state ADUs written over the last n epochs. Because HDFS only supports write-once, non-append files, we create additional HDFS data files each epoch that contain the new state inserts and updates.

Each translate instance reads/writes the entire index file corresponding to its state partition each epoch. They use an in-memory index (like Bigtable) for lookups, and write the index file as a sorted set of key to $\{epoch, offset\}$ pairs. To support inner state flows using BIPtables, we modified reduce tasks to query for state ADUs in parallel with the merge sort of mapper output and to store reads in an ADU cache. This ensures that calls to the translate wrapper do not stall on individual key fetches. Our system learns the set of keys to fetch during the merge and issues reads in parallel. The process ends when the ADU cache fills, limiting the memory footprint, or all keys are fetched. The reduce task probes the ADU cache on each call to the translate wrapper, and misses fault in the offending key.

6.5.3 Multicast and broadcast routing

The CBP model extends groupwise processing by supporting a broadcast `ALL` address and dynamic multicast groups. Here we describe how to do so efficiently, reducing duplicate records in the data shuffle. We support `ALL` *RouteBy* keys by modifying mappers to send `ALL` ADUs to each reduce task during the shuffle phase. At this point, the reduce wrapper will add these tuples to the appropriate destination flow before each call to translate. Since the partition count is often much less than the number of groups in state, this moves considerably less data than shuffling the messages to each group. `ALL` may also specify an optional set of input flows to broadcast to (by default the system broadcasts to all inputs).

While broadcasting has an implicit set of destination keys for each epoch, we provide translator authors the ability to define multicast groups dynamically. They do so by calling $associate(k\ , mcaddr)$, which associates a target key k with a multicast group $mcaddr$. A translator may call this for any number of keys, making

any key a destination for ADUs whose *RouteBy* returns *mcaddr*. The association and multicast address are only valid for this epoch; the translator must write to this multicast address in the same epoch in which it associates keys.

Under the hood, calls to *associate* place records of {k,*mcaddr*} on a dynamically instantiated and hidden loopback flow named f_{mcaddr}. The system treats input records routed to a multicast address in a similar fashion to ALL ADUs, sending a single copy to each reduce task. That record is placed in an in-memory hash table keyed by *mcaddr*. When the reduce wrapper runs, it reads the hidden loopback flow to determine the set of multicast addresses bound to this key and probes the table to retrieve the data.

6.5.4 Flow separation in Map-Reduce

While the `FlowID` maintains the logical separation of data in the black-box implementation, the Map-Reduce model and Hadoop implementation treat data from all flows as a single input. Thus the system sorts all input data but must then re-separate it based on `flowID`. It must also order the ADUs on each flow by that flow's *OrderBy* keys. This emulation causes unnecessary comparisons and buffering for groupwise translation.

Consider emulating a groupwise translator with n input flows. A Hadoop reduce tasks calls the reduce function with a single iterator that contains all records (ADUs) sharing a particular key. Direct CBP emulates the individual flow iterators of $T(\cdot)$ by feeding from a single reduce iterator, reading the flow iterators out of `flowID` order forces us to buffer skipped tuples so that they can be read later. A read to that last flow causes the system to buffer the majority of the data, potentially causing OutOfMemoryErrors and aborted processing. This occurs in practice; many of our examples apply updates to state by first reading all ADUs from a particular flow.

We resolve this issue by pushing the concept of a flow into Map-Reduce. Reduce tasks maintain flow separation by associating each mapper with its source input flow. While the number of transfers from the mappers to reducers is unchanged, this reduces the number of primary (and secondary) grouping comparisons on the

RouteBy (and *OrderBy*) keys. This is a small change to the asymptotic analysis of the merge sort of r records from m mappers from $O(rlogm)$ to $O(rlog\frac{m}{n})$. This speeds up the secondary sort of ADUs sharing a `RouteByKey` in a similar fashion; the reduce task now employs n secondary sorts based only on the `OrderByKey`. This allows each flow to define its own key space for sorting and permits reading flows in an arbitrary order that avoids unnecessary ADU buffering.

6.6 Evaluation

Our evaluation validates the benefits of programming incremental dataflows using the CBP model. It explores how the various optimizations for optimizing data movement improve the performance of our three example programs: the incremental crawl queue, clustering coefficients, and PageRank. We built our CBP prototype using Hadoop version 0.19.1, and the implementation consists of 11k lines of code.

6.6.1 Incremental crawl queue

This part of the evaluation illustrates the benefits of optimizing the treatment of state for incremental programs on a non-trivial cluster and input data set. These experiments use the physical realization of the incremental crawl queue shown in Figure 6.1. Our input data consists of 27 million web pages that we divide into ten input increments (each appr. 30GB) for the dataflow. We ran our experiments on a cluster of 90 commodity dual core 2.13GHz Xeons with two SATA harddrives and 4GB of memory. The machines have a one gigabit per second Ethernet connection to a shared switch fabric.

The goal of our system is to allow incremental algorithms to achieve per-epoch running times that are a function of the number of state updates, not the total amount of stored state. Note that for the incremental crawl queue, the number of state record updates is directly proportional to the number of arriving input records. Thus, as our test harness feeds the incremental crawl queue successive increments, we expect the running time of each successive increment to be almost

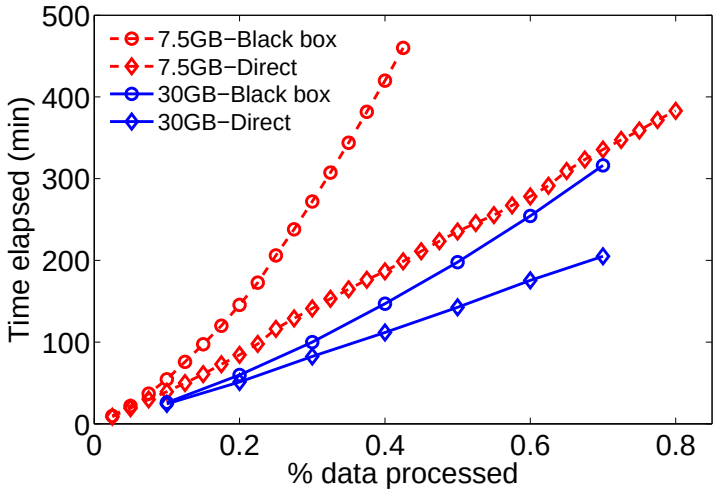
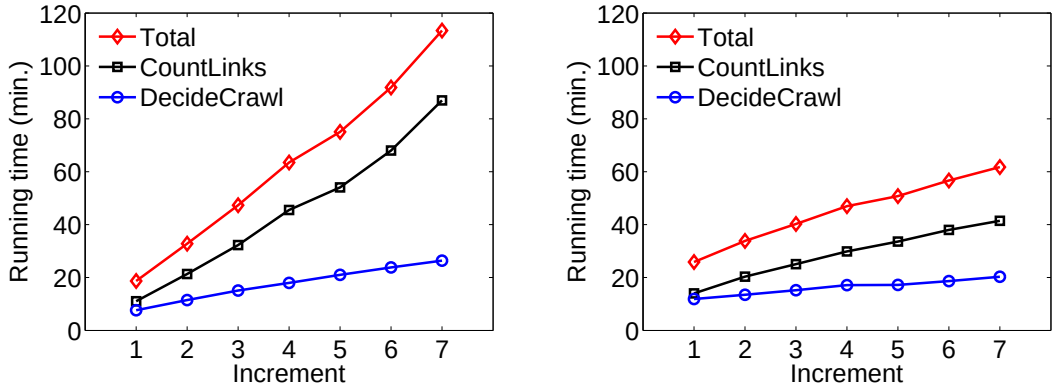


Figure 6.2: Cumulative execution time with 30GB and 7.5GB increments. The smaller the increments, the greater the gain from avoiding state re-shuffling.

constant. To measure the effectiveness of our optimizations, we compare executions of the "black-box" emulation with that of direct CBP.

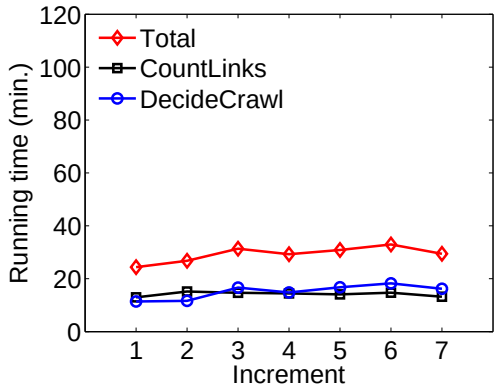
For some dataflows, including the incremental crawl queue, the benefits of direct CBP increase as increment size decreases. This is because processing in smaller increments forces state flows to be re-shuffled more frequently. Figure 6.2 shows the cumulative processing time for the black-box and direct systems with two different increment sizes: 30GB (the default) and 7.5GB (dividing the original increment by 4). Though the per-stage running time of direct CBP rises, it still remains roughly linear in the input size (i.e., constant processing time per increment). However, running time using black-box emulation grows super linearly, because the cumulative movement of the state flow slows down processing.

Figure 6.3 shows a similar experiment using 30GB increments, but reports the individual epoch run times, as well as the run times for the individual CountLinks and DecideCrawl jobs. This experiment includes the strawman, non-incremental processing approach that re-computes the entire crawl queue for each arriving increment. In this case we modify the dataflow so that runs do not read or write state flows. As expected, the running time of the non-incremental dataflow increases linearly, with the majority of the time spent counting in-links. While the incremental dataflow offers a large performance improvement (seen in



(a) Non-incremental dataflow.

(b) Incremental dataflow: black-box.



(c) Incremental dataflow: direct.

Figure 6.3: The performance of the incremental versus landmark crawl queue. The direct CBP implementation provides nearly constant runtime.

Figure 6.3(b)), the runtime still increases with increment count. This is because the black-box emulation pays a large cost to managing the state flow, which continues to grow during the execution of the dataflow. Eventually this reaches 63GB for the *countlinks* stage at the 7th increment.

Figure 6.3(c) shows run times for the direct CBP implementation that uses incremental shuffling (with reducer pinning) and flow separation. Note that state is an "outer" flow in these experiments, causing translation to access all state ADUs each epoch. Even so, incremental shuffling allows each stage to avoid mapping and shuffling state on each new increment, resulting in a nearly constant runtime. Moreover, HDFS does a good job of keeping the partition file blocks at the prior reducer. At the 7th increment, pinning in direct CBP allows reducers to read 88%

of the HDFS state blocks from the local disk.

6.6.2 BIPtable microbenchmarks

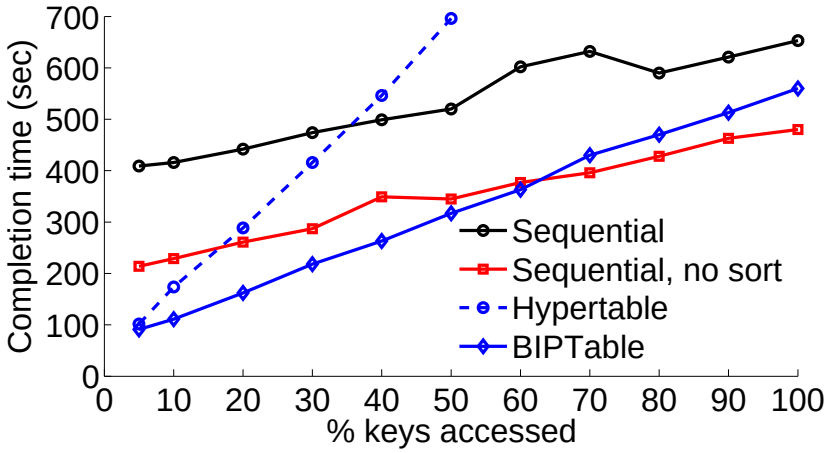
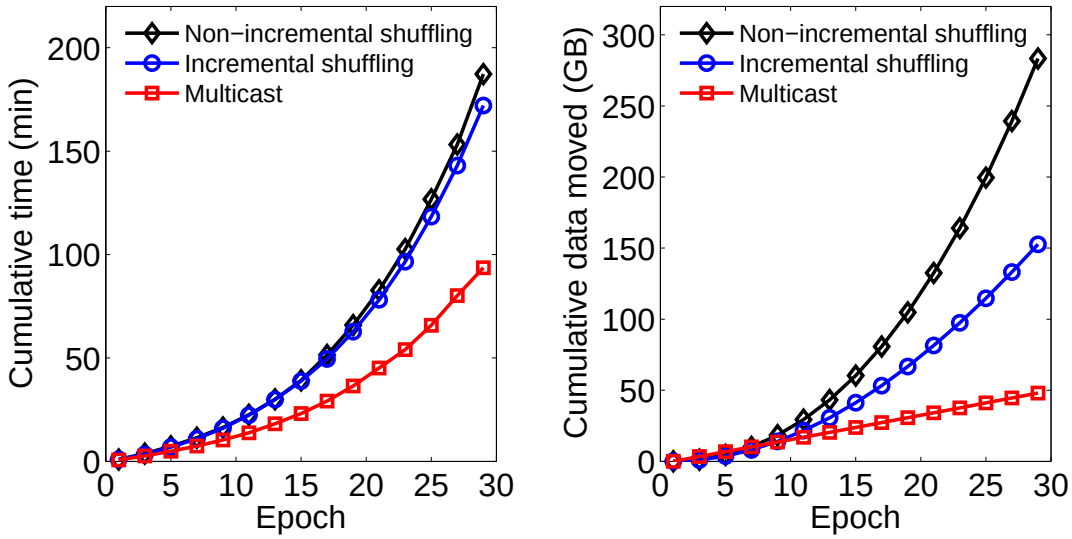


Figure 6.4: Running time using indexed state files. BIPTable outperforms sequential access even if accessing more than 60% of state.

These experiments explore whether randomly reading a subset of state is faster using BIPtable than reading all of state sequentially from HDFS. We identify the *break-even* hit rate, the hit rate below which the random access outperforms the sequential access. The test uses a stage that stores a set of unique integers in an inner state flow; input increments contain numbers randomly drawn from the original input. Changing input increment size changes the workload's *hit rate*, the fraction of accessed state. We run the following experiments on a 16-node cluster consisting of Dual Intel Xeon 2.4GHz machines with 4GB of RAM, connected by a Gigabit switch. We pre-loaded the state with 1 million records (500MB). Here translation uses a single data partition, running on a single node, though HDFS (or Hypertable) runs across the cluster.

Figure 6.4 compares running times for four configurations. *BIPtable* outperforms *Sequential*, which reads the entire state partition file, for every selectivity. One benefit is that *BIPtable* does not sort its records; it uses hashing to match keys on other inputs. To measure this effect, *sequential, no sort* does not sort the partition file (and will therefore incorrectly execute if the translator writes new



(a) Running time.

(b) Data shuffled.

Figure 6.5: Incremental clustering coefficient on Facebook data. The multicast optimization improves running time by 45% and reduces data shuffled by 84% over the experiment's lifetime.

keys during an epoch). In this case, BIPtable still outperforms sequential access when accessing a majority (>60%) of state. For reference we include a prior result [54] using Hypertable; it failed to produce data when reading more than 50% of state. Finally, it is relatively straightforward for BIPtables to leverage SSDs to improve random access performance; a design that promises to significantly extend the performance benefit of this design [54].

6.6.3 Clustering coefficients

Here we explore the performance of our clustering coefficient translator (Figure 5.6). These graph experiments use a cluster of 25 machines with 160GB drives, 4GB of RAM, and 2.8GHz dual core Xeon processors connected by gigabit Ethernet. We incrementally compute clustering coefficients using a publicly available Facebook crawl [77] that consists of 28 million edges between "friends." We randomize the graph edges and create increments containing 50k edges a piece. These are added to an initial graph of 50k edges connecting 46k vertices.

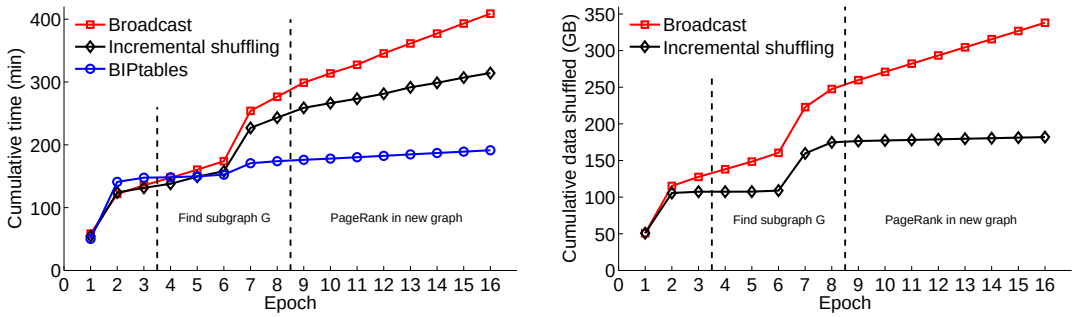
Figure 6.5(a) shows the cumulative running time for processing successive increments. We configure the translator to use full, outer groupings and successively enable incremental shuffling and multicast support. First note that, unlike the incremental crawl queue, running times with incremental shuffling are not constant. This is because the mapped and shuffled data consists of both messages and state. Recall that these messages must be materialized to disk at the end of the prior epoch and then shuffled to their destination groups during the next epoch. In fact, the message volume increases with each successive increment as the graph becomes increasingly more connected.

Additionally, map tasks that emulate multicasting (i.e, by replicating an input record for each destination) take four to six times as long to execute as map tasks that operate on state records. Hadoop interleaves these longer map tasks with the smaller state map tasks; they act as stragglers until state becomes sufficiently large (around epoch 24). At that point incremental shuffling removes over 50% of the total shuffled data in each epoch, enough to impact running times. Even before then, as Figure 6.5(b) shows, incremental shuffling frees a significant amount of resources, reducing total data movement by 47% during the course of the experiment.

For this application the critical optimization is multicasting, which both eliminates the user emulating multicast in map tasks and removes duplicate records from the data shuffle. In this case, direct CBP improves cumulative running time by 45% and reduces data shuffled by 84% over the experiment's lifetime.

6.6.4 PageRank

This section explores the impact of direct CBP optimizations on the incremental PageRank dataflow. We have verified that it produces identical results for smaller, 7k node graphs using a non-incremental version. As input we use the "indochina-2004" web graph obtained from [19]; it contains 7.5 million nodes and 109 million edges. These experiments execute on 16 nodes in our cluster (described above). Here our incremental change is the addition of 2800 random edges (contained in a single input increment).



(a) Running time.

(b) Data shuffled

Figure 6.6: Incremental PageRank. (a) Cumulative running time of our incremental PageRank translator adding 2800 edges to a 7 million node graph. (b) Cumulative data moved during incremental PageRank.

Figure 6.6(a) shows the cumulative execution time for this process. As Section 5.5.3 explained, the dataflow proceeds in three phases: computing PageRank on the original graph (epochs 1-3), finding the subgraph G (epochs 4-8), and re-computing PageRank for nodes in G (epochs 9-16). Here we have purposefully reduced the number of iterations in the first phase to highlight the incremental computation. For this incremental graph update, the affected subgraph G contains 40k nodes.

Here we evaluate the impact of incremental shuffling and inner state flows via BIPtables. Note that this dataflow required the direct CBP implementation, specifically broadcast support for propagating weights from dangling nodes. Without it, local disks filled with intermediate data for even small graphs.

Unlike clustering coefficient, incremental shuffling improves cumulative running time by 23% relative to only using broadcast support. Improvements occur primarily in the last phase as there are fewer messages and processing state dominates. After re-computing PageRank, incremental shuffling has reduced bytes moved by 46%. Finally, we see a significant gain by using inner state flows (BIPtables), as each epoch in the last phase updates only 0.5% of the state records. In this case our architecture reduced both network and CPU usage, ultimately cutting running time by 53%.

6.7 Acknowledgements

Chapter 6, in part, is reprint of the material published in the Proceedings of the ACM Symposium on Cloud Computing 2010. Logothetis, Dionysios; Olston, Christopher; Reed, Benjamin; Webb, Kevin C.; Yocum Ken. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusion

We have been witnessing an unprecedented increase in the amount of unstructured data produced today. Exploiting these big data sets requires data management systems that allow users to gain valuable insights through rich analysis. This thesis is based on the observation that data analysis is no longer a "one-shot" process, rather we view it as an update-driven process. Update-driven analytics arise in several scenarios, like continuous data processing, or machine learning algorithms that iteratively refine the result of the analysis. We argue that there is a fundamental mismatch between current data-intensive systems and update-driven analytics that makes programming big data analytics harder and inefficient. This dissertation introduces a different programming approach that captures this update-driven nature, simplifying programming and allowing efficient analytics.

We observe that the concept of *state* arises naturally in these update-driven data analytics and is a fundamental requirement for efficient processing. Based on this observation, this dissertation proposes that state become a first-class abstraction in large-scale data analytics. To this end, this thesis introduces *stateful groupwise processing*, an abstraction that integrates data-parallelism for scale with state for efficiency. We use stateful groupwise processing to efficiently manage analytics in two phases of the data lifecycle: (i) online ETL analysis, and (ii) follow-on analysis.

Critical to building a practical system for online ETL analytics is the ability to assess the impact of incomplete data on the analysis fidelity. We found

that knowledge about the natural spatial and temporal distribution of the data across their sources allows useful insights about the quality of the analysis results. By exposing this information to the users through the C^2 metric, iMR allows a wide range of online analytics applications. At the same time, we provided general guidelines for using C^2 to trade fidelity for latency depending on the different application requirements. Through our iMR prototype we validated the usefulness of the metric in a variety of real applications. We showed that it is possible to process incomplete data, to retain result availability, and still make useful conclusions from the data.

To efficiently execute ETL analytics, the iMR architecture moves the analysis from dedicated clusters to the data sources, avoiding costly data migrations. While this in-situ processing architecture reduces network traffic, it requires iMR to make careful use of available resources to minimize the impact on collocated services. iMR's load shedding techniques use available resources intelligently, and provide useful results under constrained resources or latency requirements with little impact on collocated services.

While iMR is suitable for running ETL analytics, running richer analytics requires a different programming model and architecture. iMR is designed mainly for filtering and summarizing data, operations that are common as a first step in data analytics. However, several follow-on analytics may be expressed as complex, multi-step dataflows. Additionally, some of these analytics implement algorithms, like graph mining, that must iterate over datasets.

We designed the CBP model to be expressive enough to allow a variety of sophisticated stateful analytics. We exhibited the expressiveness of CBP by building a number of real applications, including web mining analytics, and iterative graph algorithms. CBP's broadcast and multicast grouping constructs proved particularly useful in graph algorithms. Abstracting these common graph mining operations made programming easier and allowed the CBP runtime to optimize the execution of this type of analytics. Furthermore, we found that running dataflows adds to the programming complexity as users must coordinate the execution of individual steps. CBP assists users in this task by allowing the programmatic

control of dataflows through simple, yet powerful primitives.

We validated the benefits of integrating state in the CBP programming model by comparing CBP against a prototype that implements stateful computations on top of the MapReduce model. Leveraging the explicit modeling of state, CBP can optimize state management, significantly reducing processing times and network resource usage. In many cases CBP reduces both processing time and network traffic by at least a factor of 2. Further, we verified that CBP's extended grouping constructs allow huge performance gains in graph mining analytics. In certain scenarios we found that executing these iterative analytics with current DISC systems was impossible due to the amount of data that has to be saved to disks and transferred across the network.

Our work is a first step toward addressing the basic challenges in managing update-driven analytics. At the same time it creates the ground for investigating a variety of interesting issues in the future. There is a trend toward real-time analytics and a need to extract useful information from partial data. An interesting direction is to explore the use of fidelity metrics for online analysis in more sophisticated analytics, like machine learning and graph mining, not just ETL. Incomplete data impact such analytics in non-obvious ways. Understanding and characterizing this impact is a step toward making online analysis applicable to a wider range of analytics.

Like MapReduce, one of the strengths of CBP is its flexibility. Even though this allows rich analytics, it may result in a lot of custom, complicated user code that is difficult to maintain and re-use [61]. Higher-level languages, like Pig [61] and DryadLINQ [82], that are layered on top of systems like MapReduce and Dryad allow users to compose dataflows from a restricted set of high-level operations (e.g. filtering, grouping, aggregations, joins), simplifying programming. A promising direction is to investigate the integration of stateful programming in a higher-level language.

Devising incremental algorithms that use state may sometimes be a hard programming task. Recent work has explored the ability to automatically detect opportunities for computation re-use [67, 42, 43, 17]. These approaches transpar-

ently modify "one-shot" dataflows to update the analytics in an incremental way. An interesting future study would be to investigate the sweet spot between ease of programming that these systems provide and increased performance through explicitly incremental programs.

Bibliography

[1] 1998 World Cup Web Server Logs. http://ita.ee.lbl.gov/html/traces.html.

[2] CPU Usage Limiter for Linux. http://cpulimit.sourceforge.net.

[3] List of companies using Hadoop. http://wiki.apache.org/hadoop/PoweredBy.

[4] Oops pow surprise...24 hours of video all up in your eyes! http://youtube-global.blogspot.com/2010/03/oops-pow-surprise24-hours-of-video-all.html.

[5] The Apache Mahout machine learning library. http://mahout.apache.org.

[6] The Flume log collection system. https://github.com/cloudera/flume.

[7] The GridMix Hadoop Workload Generator. http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[8] The Hadoop project. http://hadoop.apache.org.

[9] The Hive project. http://hadoop.apache.org/hive.

[10] The Komogorov-Smirnoff test. http://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test.

[11] Windows Azure and Facebook teams. Personal communications, August 2008.

[12] Supercomputers: 'Data Deluge' Is Changing, Expanding Supercomputer-Based Research. http://www.sciencedaily.com/releases/2011/04/110422131123.htm, April 2011.

[13] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Conference on Innovative Data System Research*, Asilomar, CA, Jan. 2005.

[14] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *International Conference on Very Large Data Bases*, pages 336–347, Toronto, Canada, Aug. 2004.

[15] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *ACM Symposium on Principles of Database Systems*, page 1, Madison, WI, June 2002. ACM Press.

[16] M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *International Conference on Data Engineering*, pages 779–790, Tokyo, Japan, Apr. 2005. IEEE.

[17] P. Bhatodia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. Akar. Large-scale Incremental Data Processing with Change Propagation. In *Workshop on Hot Topics in Cloud Computing*, Portland, OR, June 2011.

[18] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *ACM SIGMOD International Conference on Management of Data*, volume 15, pages 61–71, June 1986.

[19] P. Boldi and S. Vigna. The WebGraph Framework I: Compression Techniques. In *International World Wide Web Conference*, pages 595–601, Manhattan, NY, 2004. In Proc. of the Thirteenth International World Wide Web Conference.

[20] T. Brants, A. C. Popat, and F. J. Och. Large Language Models in Machine Translation. In *Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, volume 1, pages 858–867, Prague, Czech Republic, June 2007.

[21] R. E. Bryant. Data-Intensive Supercomputing: The case for DISC. Technical report, Carnegie Mellon University, Pissburgh, PA, May 2007.

[22] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment*, 3(1-2):285–296, Sept. 2010.

[23] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *International Conference on Very Large Data Bases*, pages 215–226, Hong Kong, China, Aug. 2002.

[24] P. Chan, W. Fan, A. Prodromidis, and S. Stolfo. Distributed data mining in credit card fraud detection. *IEEE Intelligent Systems*, 14(6):67–74, Nov. 1999.

[25] 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. In *USENIX Symposium on Operating Systems*

Design and Implementation, OSDI '06, pages 205–218, Seattle, WA, Nov. 2006. USENIX Association.

[26] D. Chatziantoniou and K. A. Ross. Groupwise Processing of Relational Queries. In *International Conference on Very Large Data Bases*, pages 476–485, Athens, Greece, Aug. 1997.

[27] Y. Chen, D. Pavlov, and J. F. Canny. Large-scale behavioral targeting. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 209, New York, New York, USA, June 2009. ACM Press.

[28] S. Chien, C. Dwork, R. Kumar, D. R. Simon, and D. Sivakumar. Link Evolution: Analysis and Algorithms. *Internet Mathematics*, 1:277–304, 2004.

[29] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *Neural Information Processing Systems*, Dec. 2006.

[30] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 26(1):64–69, Jan. 1983.

[31] J. Cohen. Graph Twiddling in a MapReduce World. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[32] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *USENIX Symposium on Networked Systems Design and Implementation*, page 21, San Jose, CA, Apr. 2010.

[33] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *ACM SIGCCOMM Conference on Applications, Technologies, Architectures and Protocols of Computer Communications*, volume 34, page 15, Portland, OR, Oct. 2004.

[34] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation*, page 10, San Francisco, CA, Dec. 2004.

[35] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72, Jan. 2010.

[36] J. Ekanayake, X. Qiu, T. Gunarathne, S. Beason, and G. Fox. High Performance Parallel Computing with Cloud and Cloud Technologies. In *International Conference on Cloud Computing*, June 2009.

[37] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental Clustering for Mining in a Data Warehousing Environment. In *International Conference on Very Large Data Bases*, pages 323–333, New York City, NY, Aug. 1998.

[38] E. Friedman, P. Pawlowski, and J. Cieslewicz. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, Aug. 2009.

[39] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, volume 37, page 29, Bolton Landing, New York, Dec. 2003.

[40] J. Ginsberg, M. H. Mohebbi, R. S. Patel, L. Brammer, M. S. Smolinski, and L. Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, 457(7232):1012–4, Feb. 2009.

[41] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *International Conference on Data Engineering*, New Orleans, LA, Mar. 1996.

[42] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.

[43] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *ACM Symposium on Cloud Computing*, pages 63–74, Indianapolis, IN, June 2010. ACM.

[44] J. Hellerstein, P. Haas, and H. Wang. Online Aggregation. In *ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, June 1997.

[45] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *International Conference on Very Large Data Bases*, pages 321–332, Berlin, Germany, Sept. 2003.

[46] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys European Conference on Computer Systems*, volume 41, page 59, Lisbon, Portugal, June 2007.

[47] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *35th International Conference on Management of Data*, page 7, Providence, Rhode Island, 2009.

[48] Z. Ivezic, J. A. Tyson, R. Allsman, J. Andrew, and R. Angel. LSST: From Science Drivers to Reference Design and Anticipated Data Products. *Evolution*, page 29, May 2008.

[49] R. Johnson. Scaling Facebook to 500 Million Users and Beyond. http://www.facebook.com/note.php?note_id=409881258919.

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

[51] J. Lin and M. Schatz. Design Patterns for Efcient Graph Algorithms in MapReduce. In *Workshop on Mining and Learning with Graphs*, 2010.

[52] D. Logothetis, C. Olston, B. Reed, K. Webb, and K. Yocum. Programming Bulk-Incremental Dataflows. Technical report, University of California, San Diego, 2009.

[53] D. Logothetis and K. Yocum. Wide-Scale Data Stream Management. In *USENIX Annual Technical Conference*, Boston, MA, June 2008.

[54] D. Logothetis and K. Yocum. Data Indexing for Stateful , Large-scale Data Processing. In *5th International Workshop on Networking Meets Databases (NetDB'09)*, Big Sky, MT, Oct. 2009.

[55] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. In *USENIX Symposium on Operating Systems Design and Implementation*, volume 36, page 131, Boston, MA, Dec. 2002.

[56] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. *International Conference on Management of Data*, 2010.

[57] 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, Sept. 2010.

[58] D. Metzler and E. Hovy. Mavuno: A Scalable and Effective Hadoop-Based Paraphrase Acquisition System. In *KDD Workshop on Large-scale Data Mining: Theory and Applications*, San Diego, CA, Aug. 2011.

[59] C. Monash. Facebook, Hadoop, and Hive, 2009.

[60] R. N. Murty and M. Welsh. Towards a dependable architecture for internet-scale sensing. In *Workshop on Hot Topics in System Dependability*, HotDep'06, pages 8–8, Berkeley, 2006. USENIX Association.

[61] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data*, page 1099, Vancouver, BC, Canada, June 2008. ACM Press.

[62] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, Nov. 1999.

[63] C. Palmisano, A. Tuzhilin, and M. Gorgoglione. User profiling with hierarchical context: an e-Retailer case study. In *International and Interdisciplinary Conference on Modeling and Using Context*, pages 369–383, Aug. 2007.

[64] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. PLANET: massively parallel learning of tree ensembles with MapReduce. *Proceedings of the VLDB Endowment*, 2(2):1426–1437, Aug. 2009.

[65] B. Pariseau. IDC: Unstructured data will become the primary task for storage, Oct. 2008.

[66] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–15, Vancouver, BC, Canada, Oct. 2010.

[67] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *USENIX Workshop on Hot Topics in Cloud Computing*, page 21. USENIX Association, 2009.

[68] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, Sept. 1991.

[69] A. Simitsis, P. Vassiliadis, S. Skiadopoulos, and T. Sellis. Data Warehouse Refreshment. In R. Wrembel and C. Koncilia, editors, *Data Warehouses and OLAP: Concepts, Architectures and Solutions*. IRM Press, 2006.

[70] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *ACM Symposium on Principles of Database Systems*, page 263, Paris, France, June 2004. ACM Press.

[71] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all? Part 2: benchmarking results. In *Conference on Innovative Data System Research*, Asilomar, CA, Jan. 2007.

[72] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *International Conference on Very Large Data Bases*, pages 1150–1160, Vienna, Austria, Sept. 2007.

[73] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: analyzing logs as state machines. In *1st USENIX Workshop on Analysis of System Logs*, page 6, Dec. 2008.

[74] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: efficient load shedding techniques for distributed stream processing. In *International Conference on Very Large Data Bases*, page 11, Vienna, Austria, Sept. 2007.

[75] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *International Conference on Very Large Data Bases*, Seoul, Korea, Sept. 2006.

[76] P. Vassiliadis and A. Simitsis. Extraction, transformation, and loading. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*. Springer, 2009.

[77] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *ACM European Conference on Computer Systems (EuroSys '09)*, EuroSys '09, page 205, New York, New York, USA, 2009. ACM Press.

[78] H. Wu, B. Salzberg, and D. Zhang. Online event-driven subsequence matching over financial data streams. In *ACM SIGMOD International Conference on Management of Data*, page 23, Paris, France, June 2004. ACM Press.

[79] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *ACM SIGCCOMM Conference on Applications, Technologies, Architectures and Protocols of Computer Communications*, volume 34, page 379, Portland, OR, Oct. 2004.

[80] A. Yoo and I. Kaplan. Evaluating use of data flow systems for large graph analysis. In *Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 1–9, New York, New York, USA, 2009. ACM Press.

[81] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *ACM Symposium on Operating Systems Principles*, page 13, Big Sky, MT, Oct. 2009.

[82] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, San Diego, CA, Dec. 2008.

[83] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark : Cluster Computing with Working Sets. In *Workshop on Hot Topics in Cloud Computing*, Boston, MA, June 2010.