DYNAMIC LOAD BALANCING AND AUTOSCALING IN DISTRIBUTED STREAM PROCESSING SYSTEMS

Xing Wu

A thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science

Concordia University

Montréal, Québec, Canada

April 2015

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Xing Wu**

Entitled: **Dynamic Load Balancing and Autoscaling in Distributed Stream Processing Systems**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

M. Zahangir Kabir — Chair

Chun Wang — External Examiner

Samar Abdi — Examiner

Yan Liu — Supervisor

Approved by: William E. Lynch
Chair of Department of Electrical and Computer Engineering

April 15, 2015 — Amir Asif
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Dynamic Load Balancing and Autoscaling in Distributed Stream Processing Systems

Xing Wu

In big data world, Hadoop and other batch-processing tools are widely used to analyze data and get results in minutes. However, minutes of latency still cannot satisfy the proliferated needs for real-time decision in many fields such as live stock and trading feeds in financial services, telecommunications, sensor networks, online advertisement, etc. Distributed stream processing (DSP) systems aim to process, analyze and make decisions on-the-fly based on immense quantities of data streams being dynamically generated at high rates. As the rates of data streams may vary over time, DSP systems require an architecture that is elastic to handle dynamic load. Although many dynamic load balancing and autoscaling techniques for general pull-based distributed systems have been well studied, these solutions cannot be directly applied to DSP systems because DSP systems are push-based, they process data streams with different types of operators, each running on a cluster node. One research problem is to allocate data processing operators on nodes of clusters and balance the workload dynamically. Since the data volume and rate can be unpredictable, static mapping between operators and cluster resources often results in unbalanced operator load distribution. Furthermore, the problem of making DSP system scalable requires autoscaling at runtime. In this context, the operators need to be relocated among newly provisioned nodes. The contribution of this thesis is three folds. First, we proposes a software layer that is load-adaptive between a DSP engine and clusters. The architecture allows dynamic transferring of an operator to different cluster nodes at runtime and keeps the process transparent to developers. Second, an optimization method that combines correlation of resource utilization of nodes and capacity of clusters is proposed to balance load dynamically. Lastly, we design the autoscaling mechanism and algorithm to detect overload and provision nodes at runtime. We implement our design on S4, an open-source DSP engine first developed by Yahoo!. The implementation is evaluated by a top-N topic list application on Twitter streams using clusters on Amazon Web Services. The results demonstrate a 75.79% improvement on stream processing throughputs, and a 294.47% improvement on cluster resource utilization.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Prof. Yan Liu, for the patient guidance and advice she has provided throughout my graduate study. I used to have no confidence in academic research, it was Prof. Liu who guided me how to do research. She chose a research area that I am interested in, recommended related research papers for literature survey, introduced research methodology to me, helped me design the experiments, corrected my writings, and financially supported the research. Step by step, she leaded me into the research world. I have published three conference papers and a book chapter in one and a half year. I would have never been able to achieve this without Prof. Liu's help. Besides the research, Prof. Liu also concerns about my life and career. Knowing that my objective is a career in industry, she gave me many useful advices and recommended me to an internship in Ericsson. I have had a happy and productive graduate study, and I cannot imagine a better one.

I would also like to thank Dr. Ian Gorton. I have learned a lot from his immense knowledge, enthusiasm, and optimism while we were doing research together.

Finally, I would like to thank my mentor during my intern in Ericsson, Wayne Ding, for offering me a great opportunity to apply my knowledge and experience into practice in real industry.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

High rate and large volume of online information sources enrich decision making and business values for applications in many domains. Many companies realize the significance of data and begin to utilize big data analytics. In [R+11], the authors conclude that the differences between big data analytics and the traditional business intelligence are in three aspects: 1) Data size and structure. The dataset is larger and contains more attributes. More and more semistructured or unstructured data can be included to the dataset. 2) Analytics become automatic and programmatic. Automatic analysis and decision making are used in many scenarios, such as stock trading, online advertising, personalization and etc. 3) Data-driven analysis, which means that you do not have to define a problem before working on the data. In big data analytics, you set your algorithms to work over the data and discover patterns.

Besides its large volume, big data can be also described by its velocity or speed. The frequency of data generation or data delivery is at a very high rate and it keeps increasing. For example, approximately 25 billion messages pass through NASDAQ OMX's U.S. equities and options systems on an active day, which amounts to over 2 million transactions per second during volatile periods [Cor13]. To analyze the existing transaction histories and newly generated feeds is challenging.

Hadoop is a well-known implementation of MapReduce programming model for batch-processing large data intensive applications [DG08]. Recent researches are made on Hadoop to shorten the processing time of incremental data otherwise newly updated data and the old data are both run over that makes latency proportional to the size of entire data, rather than the size of an update [BWR+11, PD10]. However, the "store and then analyze" model still cannot satisfy the proliferated needs for real-time decision in many fields such as live stock and trading feeds in financial services, telecommunications, sensor networks , online advertisement, etc. Distributed stream processing (DSP) systems aim to process, analyze and make decisions on-the-fly based on immense quantities

of data streams being dynamically generated at high rates. Yahoo built an online experiment to compute click-through rate (CTR) for a query and advertisement combination in a very low latency with their DSP platform named S4, which improved CTR by 3% and with no loss in revenue [NRNK10a].

Stream services often exhibit multi-modal and spike workloads. For example, Mickulicz et at [MNG13] presents a cloud-based sports service, called YinzCam that has various modes of workload (e.g., pre-game, in-game, post-game, game-day, non-gameday, in-season, off-season) and exhibits that the traffic during the actual hours of a game is twenty-fold of that on non-game days. To handle these multi-modal workloads, YinzCam applies and tunes Amazon Web Services autoscaling configuration to automatically scale up and down the virtual machine instances on-demand. However, such autoscaling considers a group of virtual machine instances as a whole, and does not necessarily guarantee that the workload is balanced on all instances.

Most stream processing software frameworks such as Borealis [AAB+05], SPADE [GAW+08] and S4 [NRNK10a] are operator-based, which means applications are organized as data flow graphs consisting of operators at the nodes connected by directed edges representing the data streams. The operators, which are also called PEs (Processing Elements) in some systems, are allocated to physical nodes within networked clusters.

1.1 Problem Statement

Allocating a PE to a physical node has a direct effect on scalability. For a stream processing service in where PEs are connected to each other, these dynamic changes of inputs may significantly impact the loads of certain PEs due to the different selectivities (i.e., the ratio of the input and output data rates) of PEs. Even a small raise of the input stream of a low-selectivity PE may lead to a dramatic increase of its output rate. Thus the destination PEs, the PEs that takes its output as input, have to handle a much higher input rate. As a result, the cluster nodes running destination PEs could be overloaded while the other nodes are not fully utilized. The selectivity of a PE depends on not only the business logic of the service, but also the data in the input stream. Normally, the selectivities of PEs fluctuate as the input stream varies over time, which makes the change of load distribution unpredictable. Therefore, the ability to balance the loads on different nodes at runtime is essential for a scalable and reliable stream processing service. By changing the allocation of PEs we can adjust the loads on nodes and optimize the utilization of computing resources. Meanwhile, an inefficient allocation may result in unnecessary data transfers between operators through the network that can cause extra latency.

1.2 Objective

The goal of this thesis is to design and build a load adaptive stream processing system to achieve two features, namely dynamic load balancing and autoscaling. To achieve this goal, we propose Dynamic Operator Distribution Optimizer (DoDo), a software layer between a DSP platform and physical clusters to enable load adaptive for a DSP platform by dynamic load balancing and autoscaling. On top of DoDo, we propose an optimization method for the dynamic operator distribution of stream processing services. We assign the PEs of the same type to a logical cluster that contains a set of physical nodes. Then we can achieve dynamic operator distribution through changing the PE-cluster mapping on-the-fly. An algorithm is designed to make the mapping decision based on the system metrics we collect. In the end, we propose an autoscaling mechanism and algorithm to detect overload and provision nodes at runtime.

1.3 Contribution

The contribution of this thesis is in three folds:

1. a software architecture to enable load adaptive for a DSP platform by dynamic load balancing and autoscaling.
2. an optimization method that combines correlation of resource utilization of nodes and capacity of clusters to balance load dynamically.
3. the autoscaling mechanism and algorithm at runtime.

We implement our design on S4, an open-source DSP engine first developed by Yahoo!. The implementation is evaluated by a top-N topic list application on Twitter streams using clusters on Amazon Web Services. The results demonstrate a 75.79% improvement on stream processing throughputs, and a 294.47% improvement on cluster resource utilization.

1.4 Thesis Structure

This thesis is organized as follows.

- Chapter 2 gives an introduction about the concepts and preliminaries in this thesis.
- Chapter 3 provides related work in load balancing and autoscaling problems in distributed systems and some DSP platforms.

- Chapter 5 presents our motivation through a case study on a movie recommendation application.
- Chapter 6 introduces the dynamic operator-node mapping problems in DSP systems, and proposes an algorithm and the architecture of Dynamic Operator Distribution Optimizer (DoDo) to solve them.
- Chapter 7 provides an autoscaling mechanism that works under the DoDo architecture and evaluate our solutions through a case study. Chapter 8 concludes the thesis and outline the future works.

1.5 Publications

The research work in this thesis have been published. The publications are listed as follows.

- The case study of the movie recommendation application in Chapter 5 is included in *Big Data: Algorithms, Analytics, and Applications* (pp. 21-38) [WLG15a], a book edited by Kuan-Ching Li, Hai Jiang, Laurence T. Yang, and Alfredo Cuzzocrea.
- We build a model using the experiments in Chapter 5 to predict the processing time of Hadoop jobs. The work is presented in *Proceedings of the 11th International ACM Sigsoft Conference on the Quality of Software Architectures* [WLG15b].
- The DoDo architecture and dynamic load balancing algorithm in Chapter 6 are associated with our publications in *IEEE International Conference on Cloud Engineering* (IC2E) and *IEEE International Conference on Services Computing* (SCC) [WL14a, WL14b].

Chapter 2

Background

Distributed stream processing systems get more and more attention in big data industry nowadays. Although some basic principles of distributed stream processing systems, such as parallel computing, MapReduce model, distributed system, and etc., have been well studied for a long time, the combination of these technologies within a big data scenario brings us many new challenges on the system performance aspects. This chapter introduces the preliminaries and discusses related works in this field.

To deal with massive volume of input data on the fly, we rely on the concepts of parallel computing to process data concurrently. MapReduce, a model widely used in big data industry, can help us divide large problems into individual small problems. Distributed system is the platform to manage physical resources and perform large-scale computations.

2.1 Parallel Computing

The human brain tends to process things in a time-based sequential manner, that is probably the reason why the computers are first designed to execute commands one after another. Traditionally, a computer program contains a series of instructions, which is processed sequentially by a single processing unit. Obviously, we can fasten the processing time by assign the work to more processing units and make them work simultaneously, which is called parallel computing [Gra03].

Parallel computing is widely used because of its advantages in concurrency. Parallel computing makes it possible to solve large problems in a short time by using a bunch of processing units. In this decade, most desktops have multiple processors to better support multiple-task operating systems. And people build large clusters of computers to deal with big data problems.

Parallel computing needs to break down a problem into discrete tasks, so it can not be applied on all the problems. Many problems are only partially dividable, and usually parallel computing needs

some extra work on the overall tasks control and coordinations. Some kind of messaging interface or memory share mechanism is needed when communication is mandatory between different processing units [HX98].

2.2 Distributed System

Most of the real systems in industry are actually distributed systems, such as the back-end services of websites, data storage systems, peer-to-peer systems, and etc. Distributed systems are a collection of nodes (physical machines or virtual machines), which are connected through a network and coordinated by a middleware to cooperate on user-defined tasks and share the resources of the system. The internal coordination and nodes management are transparent to users so that users can perceive the system as a single computing facility [Jos07].

Distributed systems have many advantages comparing to centralised systems. First of all, most of distributed systems are scalable. They are designed to easily adding or removing resources. Therefore, distributed systems can achieve higher performance by adding nodes because the performance is almost linear to the number of nodes in most cases [H$^+$07]. Second, distributed systems provide higher reliability as the replication of processors and resources yields fault-tolerance [FLP85]. Third, the price performance with distributed systems is better because a cluster of nodes with commodity-class hardware are still cheaper than a mainframe that has equivalent performance[Gra08].

2.3 MapReduce Model

MapReduce is a programming model designed to process large volumes of data in parallel by dividing the work into a set of independent tasks. The term *MapReduce* is first used by Jeffrey Dean and Sanjay Ghemawat in Google, they got the ideas from the *map* and *reduce* primitives in Lisp and many other functional languages, and published a research paper about this model and the associated run-time systems [DG08]. Based on their paper, an open-source framework for MapReduce model, named Hadoop [apa14], is implemented and has gained tremendous popularity.

In general, a MapReduce program processes data with three phases: Map, Shuffle, and Reduce. The *Map* phase takes the input data and produces intermediate data tuples. Each tuple consists of a key and a value. In the Shuffle phase, these data tuples are ordered and distributed to reducers by their hashed keys. The Shuffle phase ensures that the same reducer can process all the data tuples with the same key. Finally during the Reduce phase, the values of the data tuples with the same key are merged together following the instructions of the reduce program.

Figure 2 demonstrate MapReduce model with a word count example. The objective is to count

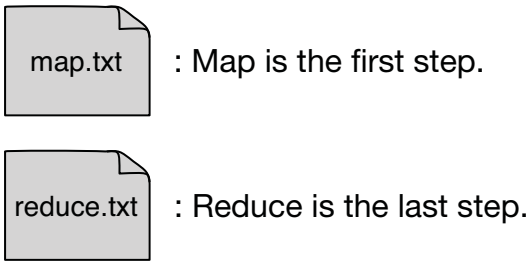


Figure 1: Word Count Input Text Files

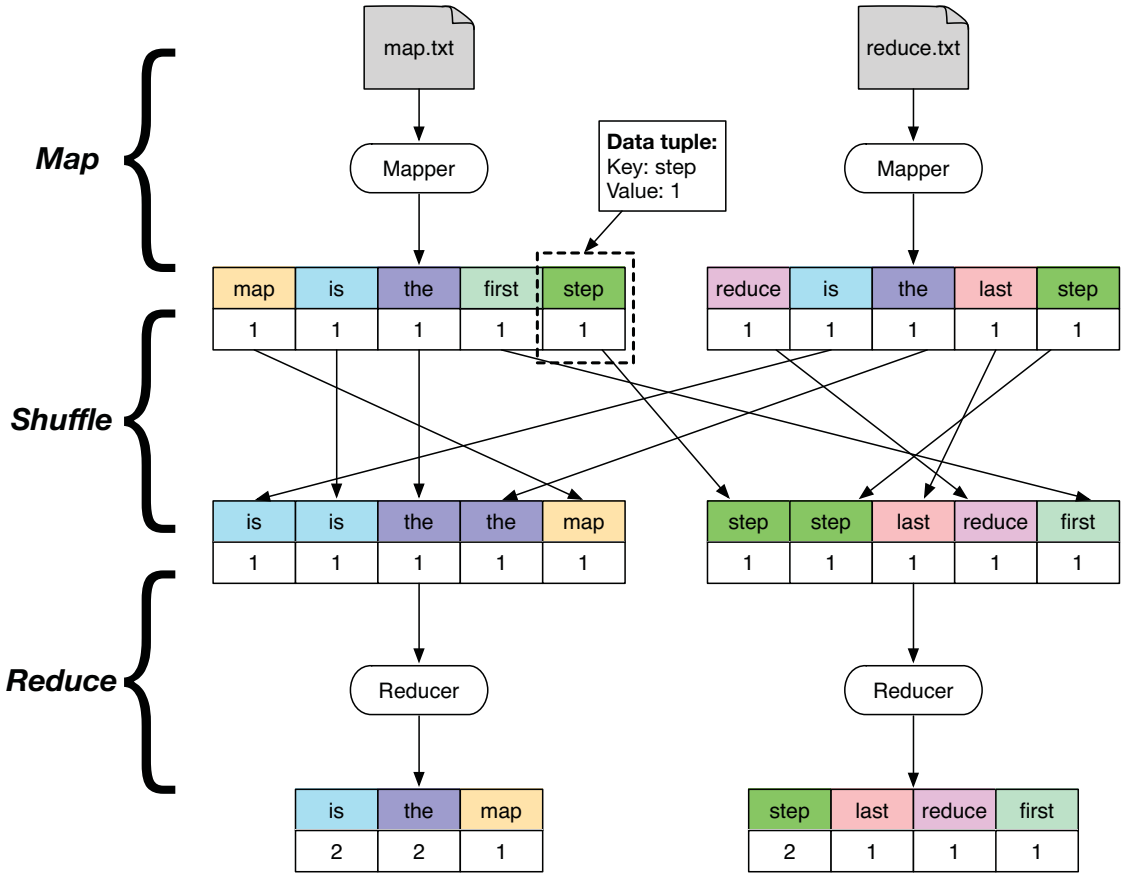


Figure 2: Word Count Example with MapReduce

the word occurrences in two text files, map.txt and reduce.txt. The content of the two files are shown in Figure 1. In this example, the word occurrences of each file are counted in the Map phase. Then in the Shuffle phase, these word occurrences are ordered and distributed to reducers by hashing the words. In the Reduce phase, the occurences of the same word are aggregated.

Listing 2.1 is the pseudo-code of the word count example. The map function extract words from

the document and associate the words with their occurrences '1'. The reduce funtion sums together all the counts for each word. Only function *map()* and *reduce()* are mandatory for a MapReduce job. The user of the MapReduce model can focus on how to process the data, the MapReduce framework will take care of the shuffling and partitioning.

Listing 2.1: Word Count

```
1  function map(String name, String document):
2    // name: document name
3    // document: document contents
4    for each word w in document:
5      emit (w, 1)
6
7  function reduce(String word, Iterator partialCounts):
8    // word: a word
9    // partialCounts: a list of aggregated partial counts
10   sum = 0
11   for each pc in partialCounts:
12     sum += ParseInt(pc)
13   emit (word, sum)
```

Many problems can be transformed into a single MapReduce job or a series of MapReduce jobs. The authors of [CKL+07] implement almost all the common machine learning algorithms using the MapReduce model, including Locally Weighted Linear Regression, Naive Bayes, Gaussian Discriminative Analysis, k-means, Logistic Regression, Neural Network, Principal Components Analysis, Independt Component Analysis, Expectation Maximization, and Support Vector Machine. [UKOVH09] presents a scalable distributed solution for large-scale Semantic Web reasoning based on MapReduce. The ability to parallel process large-scale data makes MapReduce popular in big data scenarios.

2.3.1 Hadoop and S4

Apache Hadoop is an open-source implementation of the MapReduce model [apa14]. It implements a scalable fault-tolerant distributed platform for MapReduce programs. With the Hadoop Distributed File System (HDFS), which provides high-throughput access to application data, Hadoop is reliable and efficient for big data analysis on large clusters.

To make the platform more flexible, Hadoop expands MapReduce model with combine functions and partition functions. The combine function is similar to the reduce function. It is a local aggregation between the map phase and shuffle phase. Take the word count program as an example,

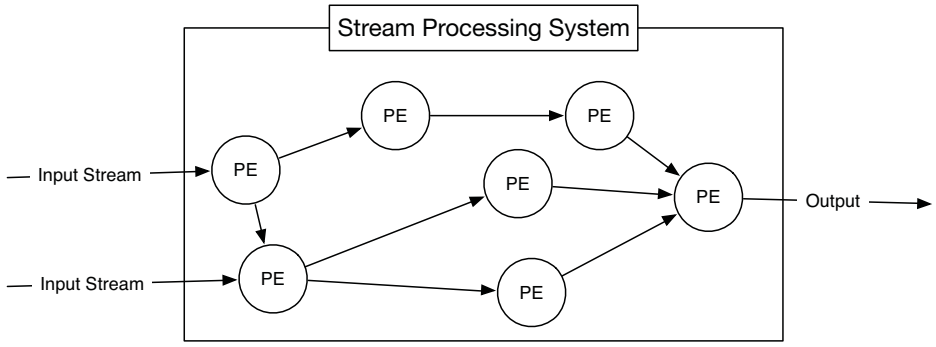


Figure 3: Data Flow Graph of Stream Processing System

after the map function, instead of sending out '(w, 1)' tuples, we can perform a local aggregation to sum the occurrences on each node so that we can merge some local data and avoid unnecessary data transportation. The source code of this combine function can reuse the code of the reduce function in Listing 2.1. The partition function is responsible for assigning intermediate key-value pairs to reducers. The partition functions are usually hash-based, they need to ensure a even distribution of the keys. They are used in the shuffle phase to determine where to send those data tuples generated by the mapper.

Hadoop is widely used because of three major advantages: 1) the ability to horizontally scale to petabytes of data on thousands of commodiy servers; 2) easy-to-understand programming semantics; 3) a high degree of fault tolerance. Hadoop allows developers with no specific knowledge of distributed programming to run MapReduce functions across multiple nodes in the cluster, and also gathers the files from multiple nodes to return a single result.

However, Hadoop also has some shortcomings: 1) Hadoop jobs have high startup costs, which can be tens of seconds on a large cluster. This asserts that Hadoop can never be used for real-time analytics; 2) Data skew can occur in the reduce phase, which will create stragglers in the cluster and significantly drop the performance. 3) As MapReduce jobs are isolated from each other, sometimes it is difficult or even impossible to implement algorithms in Hadoop. 4) The shuffle phase might bring overhead in network transportation.

Besides Hadoop, MapReduce model is also adopted by many distributed stream processing systems. Processing elements (PE), also called operators, are the computation units in stream processing systems as Figure 3 shows. PEs process input data tuples one by one and generate output data tuples. The MapReduce programming model can be applied to process stream data.

Apache S4 (Simple Scalable Stream System) is an open-source distributed stream processing platform that has applied the MapReduce model to its PEs(Processing Elements) [s4]. The data tuples of streams are named events in Apache S4. Each event is a key-value pair. Events are

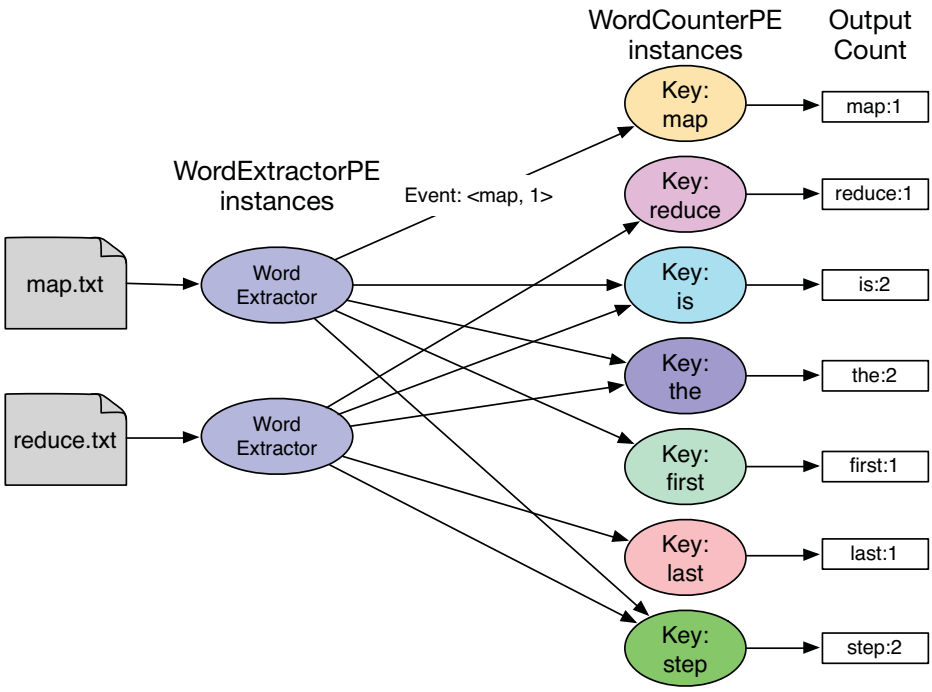


Figure 4: Word Count Example in S4

dispatched to PE instances based on the hashed keys. As a result, the events with the same keys are processed by the same PE instance, which is similar to the Reduce phase of the MapReduce model. After processing the input event, PE generates a new event, which is also a key-value pair, for the next stage of processing. The events generating scheme in PEs is akin to the Map phase of the MapReduce model.

To implement applications, developers need to design custom PEs and link PEs by events. Figure 4 is the word count example implemented in S4. There are two types of PEs in this application, namely WordExtractorPE and WordCounterPE. The WordExtractorPE reads the text input file and generates key/value events such as <map, 1>. The WordCounterPE stores current occurrences of words and updates the counts when new event comes.

The motivation of S4 is to provide a highly scalable software solution (akin to Hadoop for batch data processing) to operate at high data rates and process massive amounts of data. The design of S4 has already considered the limitation of previous work in stream processing projects and commercial engines, which are either lack of integration with cloud programming models or restricted to highly specialized applications[NRNK10b]. The S4 design shares many attributes with IBM's stream processing middleware. Both systems are designed for big data and capable of mining

information from continuous data streams. The main difference is IBM's design follows a model-based approach with a streaming language and distributed runtime (InfoSphere Streams) supporting this language [GA12]. S4's design is a combination of MapReduce and the Actor model (a formalized model of concurrency [Agh85]). Each actor has an independent thread of control and communicates via asynchronous message passing. The limitation of S4 is the mapping between PEs and the nodes are static. The current version of S4 does not support balancing the workload by dynamically transmitting PEs from one node to another, which limits its ability to scale in and out computing nodes on demands.

Chapter 3

Related Work

A variety of approaches have been proposed in the literature to improve the performance of distributed systems. We categorize distributed systems as pull-based distributed systems and push-based distributed system and discuss them separately. In pull-based distributed systems, when a client send a request to the system, the system distributes the request to relevant nodes, compose a response and send it back to the client (e.g., distributed database systems [HY79], Hadoop). In contrast, clients in push-based distributed system push data into the server and do not expect responses. Distributed stream processing systems are typical push-based distributed systems.

3.1 General Distributed Systems

Brewer's CAP theorem [Bre12] states some features and their trade-offs in distributed systems. It asserts that distributed systems can have only two of these three features: *consistency*, *availability*, and *paritition tolerance*. *Consistency* means that the data is exactly the same on any nodes at any time. *Availability* guarantees that every request receive its response. *Paritition tolerance* stands for the tolerance of partial failure in the sytem (e.g., nodes crash, network failure). These features are highly related to the performance of distributed sysytems [Bre00, SC11].

In most distributed database systems, many features are optional and can be switched on or off through configurartion files. MySQL[MyS95], a traditional relational database widely used to create horizontal-scaled database sytems, has over 200 different options including cache size, replication, etc. Therefore, the first step for performance optimization in such systems is to study your usage scenario and optimize those parameters [SZT12]. As for the state-of-the-art NoSQL databases such as Cassandra [LM10] and MongoDB [mon15], many configurations are available to optimize the performance [vdVvdWM12, LM13].

Performance optimization can occur during the processing procedure of individual requests as

well. In SQL-based databases, a lot of query optimization techniques have been studied [Cha98, JK84]. These approaches aim to analyze the given query and make an efficient query plan instead of executing the query sentence literally. However, such optimization approaches are problematic for Hadoop-based systems as the map and reduce functions can be written in arbitrary programming languages instead of well-structured SQL. MANIMAL [JCR11] is an attempt to overcome these difficulties through the analysis of compiled Hadoop programs to detect optimization opportunities. The code analyzer in MANIMAL looks for functionality in the *map()* that is equivalent to an SQL SELECT or PROJECT statement, and seeks data compression opportunities. It also generates an indexing program that builds a B+ Tree based index over the program's input data. The output from the optimizer is an execution descriptor that informs the MANIMAL execution fabric, a modified version of Hadoop, which applies optimizations and indexes during execution. Validation of the approach yield 3x to 11x speedups on a sample set of benchmarks. This shows the potential of the approach, but its ability to detect optimizations in arbitrary complex Hadoop programs has not been explored.

Hadoop++ [DQRJ+10] extends Hadoop to perform index accesses whenever a MapReduce job can exploit the use of indexes, and to co-partition data to enable maps to compute joins. The overall Hadoop framework is left unaltered and benchmarks show impressive speedups. However, programmer intervention is needed and hence the approach is not transparent to Hadoop developers. HadoopDB [ABPA+09] also extends the basic framework to incorporate local relational databases, creating a parallel database system that uses Hadoop at its core. HadoopDB pushes SQL statements to the local databases and computes partial aggregates, which require a single reduce task for the final aggregation. HadoopDB therefore enforces a hybrid programming for applications, and is not useful for optimizing vanilla Hadoop codes. Simulation is a proven technique for performance prediction and analyzing design trade-offs in a large number of software and hardware domains. It is not surprising therefore to see a number of efforts that have focused on building simulations of the Hadoop framework and applications. These include MRSim [HLL+10], HSim [LLAH13], Mumak [Mur09], SimMR [VCC11] and MRPerf [WBPG09].

MRSim is based on a composition of the discrete event simulation package SimJava [HM98] and GridSim [BM02], with the latter used to simulate network topology and traffic. The core abstractions in the simulator are models of CPU, HDD and Network Interface, which are replicated to describe a topology for cluster running Hadoop. Hadoop job descriptors specify the number of map and reduce tasks, data layout, algorithm description, and the job configuration parameters. The data layout describes the location of the data splits, including replicas, on the simulated nodes. Algorithm information is coarsely specified by measures including the number of CPU instructions per record and average record sizes of the data used in map and reduce tasks. Simple evaluations on a 4 node

cluster for word count show promising results in terms of accuracy of predicting job characteristics, including latency. However, no extensive evaluation has been described for large scale cluster and complex analysis and data types.

HSim also aims to provide comprehensive Hadoop simulation capabilities. It is designed to enable a deep exploration of the design space for Hadoop jobs by supporting the modeling of an extensive number of parameters concerned with processing nodes, clusters, the Hadoop framework itself, and simulator controls that can govern simulation speeds and the fidelity of results. Cluster characteristics and Hadoop job behaviors are specified by populating a collection of parameters that are read by the simulator. Evaluation is performed firstly against a number of published industry benchmarks executed on clusters ranging from one to 100 nodes, and also against two Hadoop applications, namely information retrieval and content based image annotation. These two applications were implemented on a 4 node cluster, and comparisons against HSim predictions showed strong congruence as the number of mappers increased.

Mumak and SimMR focus on simulating the Hadoop task scheduling algorithms. Both simulate Hadoop workloads based on traces from Hadoop jobs. Mumak runs the actual Hadoop scheduler in a simulated environment, whereas SimMR simulates both the scheduler and the Hadoop jobs themselves. Due to their focus on scheduling tasks, there is no detailed simulation of the map and reduce phases in these simulators, and hence their ability to accurately simulate resource contention and the effects of various Hadoop configuration settings is limited. MRPerf [WBPG09], based on the ns-2 discrete event simulator (http://en.wikipedia.org/wiki/Ns-2), also performs detailed simulations of Hadoop jobs. MRPerf aims to provide fine-grained simulation of Hadoop framework behavior. MRPerf requires detailed specifications of a cluster topology, application characteristics, and the layout of the application input and output data, all provided in XML. A major limitation is that it assumes simple map and reduce tasks, where the computing requirements are dependent on the size of the data, and not the processing of its content. MRPerf has been validated in several studies (TeraSort, Search and Index) using a 40 node cluster, and has been used to predict the effect of topology changes on a simulated 72 node cluster. Others however have reported difficulties in using MRPerf and have not been able to produce accurate performance predictions [HLL+10].

In general, the major problem with simulation as a technique for Hadoop performance prediction is accurately modeling the complexity of the map and reduce phases. This is typically done by specifying the amount of 'work' per unit of input data, where work is represented by CPU instructions, or some unit of time depending on the precise nature of the simulator. This information can only be coarsely estimated by an application designer in the absence of a running application. Simulators are also sensitive to changes in the implementation of the Hadoop framework. These changes can invalidate the simulations, necessitating changes to the core simulation approaches in

order to accurately model framework behavior. This makes it difficult to keep simulators 'in sync' with Hadoop, and hence makes their usage of questionable value.

High fidelity performance models can potentially facilitate the prediction of Hadoop application performance, and provide faster solutions than simulation. For example, [Her11] takes a fine grained data-flow approach that provides a mathematical model of every stage of an Hadoop job. The models calculate the amount of data flowing through the different phases of a job, as well as the execution time for all tasks and phases. The model requires a large number of input parameters specifying both the job characteristics and cluster configuration. While [Her11] does not address how the parameters are obtained or model validation, [HB11] builds on this model and uses application profiling from a running application to populate the model. It then introduces a cost based optimizer that can predict the application performance given different job configuration parameters. Validation of the 'what-if' predictions is provided for a single application only, and this approach relies upon having the application code available for profiling and model calibration.

[SMH+13] presents a Hadoop job analyzer and prediction model, and uses a locally weighted regression method to train the prediction model based on historical traces. The analyzer measures data input size and the number of input records of a collection of sample applications, and tries to estimate the complexity of the map and reduce functions and the ratio of input values that are passed from the map to reduce phase. This data is then used for model training. Predicting an application's performance requires the application code to be profiled so that jobs in the training set with similar profiles can be used for prediction. A limited validation using word count shows that the method works well only when a job profile closely matches those in the training set. The paper does not describe in detail the experimental setup and conditions for prediction.

Another challenge that distributed systems need to face is the dynamic load. We usually configure the number of nodes, sharding strategies, and etc. for an ideal scenario with a maximum requests rate. However, the load could change unpredictably and we need to keep the system operate continuously and make sure no node is overloaded. Stateless distributed systems, such as non-session web server cluster, can be easily scaled out by adding more server instances [SGR00]. As for those stateful distributed systems (e.g., Cassandra, MongoDB), some states need to be transferred to the new nodes when scaling out. Consistent hashing is used to evenly distribute the load and make it convenient to scaling [KLL+97]. The load balancing and scalability in distributed systems have been well studied [SKH95] [GB99] [GL12]. In [GL12, XCL14], the authors present methods to provision service instances dynamically on a cloud and scale up automatically.

3.2 Distributed Stream Processing Systems

Unlike pull-based distributed systems, the performance optimization techniques in distributed stream processing systems have been less addressed. Recent research related to load balancing and scalability of DSP platforms has been dedicated to improving the throughputs and ability to handle high volume input data.

The load distribution problem in DSP systems is complicated. A simple and widely used approach for this problem is measuring the load of operators and then balancing the total load on different nodes. In this context, the optimization problem becomes a k-way *Number Partitioning Problem* [Kor09]. The authors of [BTO13] and [GJPPM+12] provide solutions to this problem based on the *Best Fit Decreasing* algorithm. The approach focuses on balancing the total load on different nodes follows the strategy used in traditional pull-based parallel and distributed systems, while push-based stream processing systems often have input streams that can vary over time. An algorithm presented in [XZH05] uses the correlations of load series over fixed-length time periods to determine the locations of operators. Their research observes that if the correlation coefficient of the load time series of two operators is small, then putting them together on the same node helps minimize the load variance. Through maximizing the correlations between nodes, the load are more likely to keep balanced when the input load changes.

COLA [KHP+09] is a optimization scheme of IBM's stream processing system that considers not only the load of PEs but also the cost of communication between nodes. [KK14] has discussed the importance of balance the load between data centers. COLA analyzes the profiles of PEs and combines PEs together in compiling. However, COLA is profile-based and can not solve the dynamic operator distribution problem. In another paper [YCY+13], the authors use a genetic algorithm that considers both computation and communication cost to decide the allocation of operators in the context of mobile cloud computing.

Some stream processing engines, such as S4 [NRNK10a] and Stormy [LHKK12] are designed as a SaaS so that the application developers can focus on their business logic and use the resources in a cloud conveniently. The current version of S4 is lack of adaptability features to handle changing load. Enabling auto-scaling needs to consider more issues including the events dispatching strategy between operators when the number of nodes in the cluster has changed and the transmission problem of PE states. Esc [SHLD11] is an elastic stream processing platform. It supports provisioning nodes to adjust the capacities dynamically, but it has limitations on balancing the load on different nodes. Dynamic load balancing is also addressed in [CC09] and [DRK06], however only stateless operators are considered.

Similar to our work, *StreamCloud* [GJPPM+12] has a architecture that supports both load balancing and auto-scaling, they divide the application into operator groups and put each group in

a *Subcluster*. The allocation of operators has a limitation that each group has at most one stateful operator. *StreamCloud* parallelizes standard data stream operators, which means users need to implement a *Load Balancer* for each of their custom operators.

Chapter 4

Overview of Research Method

This chapter provides an overview of our research method. We first give the problem statements we aim to solve. Then we present a summary of the solutions and how we evaluate our solutions.

4.1 Problem Statements

To design a DSP system that is adaptive to dynamic loads, we aim to solve three problems in this thesis as follows:

1. System architecture. There are many different types of operators to process data streams in a DSP system. Each type of the operators may have a large number of instances. These operator instances process the input stream and update the status, then output a new stream to other operators. We need to design an architecture that allows very flexible bindings between the operator instances and the physical nodes that the instances are running on, with which we can easily assign a type of operator to a cluster of nodes and the operator instances can be transeferred from one node to another.

2. Dynamic load balancing. Different types of operators demand different system workload. When the input stream rate fluctuates, the load of each type of operators can vary over time. We need an alorithm to arrange the allocations of the operators to physical nodes so that the load are evenly distributed to all the nodes and the load on nodes can keep balanced when input stream changes. Dynamic load balancing can utilize the cluster resourses and avoid straggler nodes that may drag down the performance of the whole system.

3. Autoscaling. Design a mechanism to provision resources in DSP systems so that a part of the operator instances can be smoothly transferred to the new node at runtime. And design a algorithm that can make scaling decisions to keep system from overload.

4.2 Research Method

4.2.1 Empirical Analysis

To understand the features of stream processing platforms and key elements involved in the workload distribution and optimization, we first conduct an empirical analysis by running experiments and comparing the stream processing platform's scalability with the Hadoop platform. Since both platforms share the same computing model of MapReduce. We implement a movie recommendation application on both platforms with the sample data sets from Netflix Prize [net].

4.2.2 System Architecture

We construct Dynamic Operator Distribution Optimizer (DoDo), a software layer between general DSP engines and physical resources. The design principle of DoDo is to arrange a virtual cluster to each type of operators and allow different type of operators assigned to the same virtual cluster. Each virtual cluster contains a set of physical nodes and we can add or remove nodes in a virtual cluster at runtime. With this design, we are able to move operators from one virtual cluster to another and also ajust the number of nodes in a virtual cluster.

DoDo consists of six components: the *Data Flow Graph (DFG) Optimizer*, the *Health Monitor*, the *PE Transmission Controller*, the *Cluster Manager*, the *Health Client*, and *Events Dispatcher*. DoDo manages all the virtual clusters and takes care of the communication between operators. DSP engines only needs to inform which operator is the destination of the current data stream, DoDo can then route the data to the corresponding node. The communication details and route tables are completely transparent to DSP engines.

When we change the mapping of operators and virtual clusters, or adjust the number of nodes in virtual clusters, some of the operator instances need to be transferred as the number of partitions has changed. DoDo provides a mechanism to serialize and send the operator instances and deserialize and restore the operator instance at the destination node.

4.2.3 Dynamic load balancing

The goal of dynamic load balancing is to minimize the average load variance and maximize the throughputs of stream processing services. We design an optimization method applied in the mapping process to assign PEs to cluster nodes. First, we discuss the static mapping problem. The assumption is that the load on each PE is steady and the input stream rate does not change over time. In this case, the optimization objective is to find a mapping plan so that the difference between the average loads on different clusters is minimized. This is a k-way *Number Partitioning Problem*

[Kor09]. A widely used algorithm is to sort the numbers in a decreasing order, and then assign each number in turn to the subset with the smaller sum so far. In the problem of dynamic mapping optimization, the load of each PE can vary over time as the input stream rate fluctuates. In this case, we need to consider not only the current load, but also how to keep the loads balanced over time. We propose an optimization method that combines correlation of resource utilization of nodes and capacity of clusters to solve the dyanmic mapping problem.

4.2.4 Autoscaling

We aim to design an autoscaling mechanism and algorithm to detect overload and provision nodes at runtime. There are two major problems: (1) how to synchronize the new node information to all the existing PEs at runtime; and (2) stateful PE instances must be re-shuffled based on the event partition algorithm and transferred to another node when the number of nodes in this cluster has changed.

We solve the first problem by keeping tracking of the status of cluster nodes and synchronizing the state to other related nodes by means of *ZooKeeper*. In our architecture, *ZooKeeper* works as a coordinator and holds the information of all the clusters in a distributed stream processing system. Once *ZooKeeper* loses the connection with a node, the *Cluster Manager* deletes this node in related clusters and notifies the *Events Dispatchers* of all the clusters. Likewise, when a new instance is provisioned, *ZooKeeper* notifies all the connected nodes with the information of the new instance.

As for the second problem, we optimize the relocation of PEs by moving as few PEs as possible by using *Consistent Hashing* [KLL$^+$97, KSB$^+$99] as the event partition algorithm. Assume the events have K different keys and the cluster contains N nodes. Approximately K/N keys are mapped to each node. When a new node is added to the cluster, we can estimate $K/(N+1)$ keys need to be relocated. Thus the *Consistent Hashing* help reduce the number of PE states we need to transfer between work nodes when we add or remove nodes in a cluster.

4.3 Evaluation Method

The main objective is to evaluate the improvement and cluster resource utilization made by our architecture. We implement our design and apply it to S4. We choose S4 because of its pluggable architecture, with which we can easily make our software layer integrated. Moreover, S4 is suitable for comparison experiments as itself support neither dynamic load balancing nor autoscaling.

For the workload, we use the application of *Twitter's Top-N Topic List*. This application reads tweets stream, extracts topics, counts occurrences of each topic and outputs the top-N topic list periodically. The top-N recommendation application contains one event generator and three PEs in

the pipeline, namely *ExtractorPE*, *TopicCountAndReportPE* and *TopNTopicPE*. Each event generator creates 50 tweets every second at the beginning, and increasingly generates 5 more tweets every 3 seconds. Thus each event generator is able to gradually increase the tweets generation speed to 4,050 tweets per second in 40 minutes. We run the application until the throughput stops increasing, which means the system reaches its maximum throughput.

We compare the throughput and cluster resource utilization in three options, namely original stream processing engine (SPE), DoDo with dynamic load balancing (DoDo+LB) and DoDo with autoscaling (DoDo+AS).

Chapter 5

Empirical Analysis of the Scalability of Hadoop and Stream Processing Platform

5.1 Motivating Applications: Movie Ratings from Netflix Prize

Recommendation systems, which implement personalized information filtering technologies, are designed to solve the problem of information overload on individuals [ASST05], which refers to the difficulty a person can have understanding an issue and then making decisions by presence of too much information. Many popular commercial Web services and social networking Web sites realize recommendation systems to help users with enhanced information awareness. For example, movie recommendation improves the user experience by providing a list of movies that most likely covers the movies the user may like. Collaborative filtering (CF) is the most widely used algorithm for recommendation systems. It contains two major forms, namely user-based and item-based CF. The user-based CF aims to recommend a user movies that similar users like [BHK98][HKBR99][RIS+94], while the item-based CF recommends a user movies similar to the user's watched lists or high-rating movies [DK04][LSY03][SKKR01].

As users keep rating and watching movies, both algorithms cannot avoid the problem of incremental data processing, which means analyzing the new ratings and most recent watched histories and updating the recommendation lists. As the numbers of users and movies grow, a single machine

cannot satisfy the immense needs of computation and memory usage. Take the item-based CF algorithm in [SKKR01] as an example, assume there are M users and N movies, the time complexity to compute the similarity between two movies is O(M). For the final recommendation result, we have to compute the similarities for all the possible movie pairs, which has a complexity of $O(M * N^2)$. With millions of users and tens of thousands of users, the complexity will be extremely large and scalability becomes a serious problem. Implementing the CF algorithms with MapReduce model and applying them in scalable platforms is a reasonable solution. Hence a movie recommendation application on real-world data sets provides the motivating scenario to investigate the scaling needs and techniques in a cloud environment.

We apply the sample data sets from Netflix Prize [net]. The size is approximately 2GB in total. The data set contains 17,770 files. The MovieIDs range from 1 to 17,770 sequentially with one file per movie. Each file contains three fileds, namely, UserID, Rating and Date. The UserID ranges from 1 to 2649,429 with gaps. There are totally 480,189 users. Ratings are on a five star scale from 1 to 5. Dates have the format YYYY-MM-DD. We merge all the 17,770 files into one file, each line with format as <UserID, MovieID, Rating, Date>, ordered by the ascending date. For the input of the stream processing application, we implement a stream generator that read the date-ordered file line by line. We choose the item-based CF algorithm, as the implementation is comparable between Hadoop and S4 to observe the differences in scalability and cost.

In the item-based CF algorithm, we define a movie fan as a user who rates this movie higher than three stars. Mutual fans of two movies are the users who have rated both of the two movies higher than three stars. Then we measure the similarity between two movies by the number of their mutual fans. As a result, we output a recommendation list that contains top-N most similar movies to each movie.

5.2 Processing Netflix Movie Ratings Data in Hadoop

With the MapReduce model in Hadoop, the algorithm consists of three rounds of MapReduce processing as follows:

1. **Round 1**. Map-and-Sort the user-movie pairs. Each pair implies the user is a fan of the movie. Reducer is not needed in this round. Figure 5 shows an example of the input and output.

2. **Round 2**. Calculate the number of mutual fans of each movie. Figure 6 demonstrates the processing with an example. Assume Jack is a fan of movie 1, 2, and 3, then movie 1 and 2 has one mutual fan as Jack. Likewise, movie 1 and 3, and movie 2 and 3 also have one mutual fan as Jack. A mapper finds all the mutual fans of each movie and outputs a line for every mutual

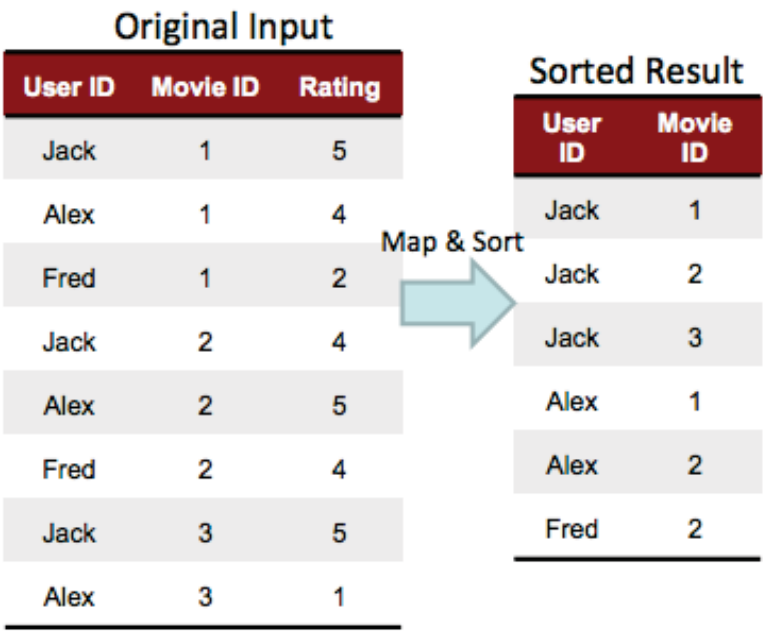


Figure 5: An Example of Round 1

fan. Then the reducer aggregates the result based on the movie pair that has one mutual fan and counts the number of mutual fans.

3. **Round 3**. Extract the movie pairs in the result of Round 2 and find the top-N movies that have the most mutual fans of each movie. The mapper extracts the movie IDs from the movie pairs and the reducer aggregates the result from the mapper and orders the recommended movies based on the numbers of their mutual fans. As Figure 7 demonstrates, movie pair 1-2 has mutual fan count as 2 and movie pair 1-3 has one mutual fan. Therefore the movie id 1 has mutual fans with movie id 2 and 3. The recommended movies for movie id 1 are movies [2,3] in descending order of the count of mutual fans.

5.2.1 The Experiments

We design experiments to evaluate the scalability and resource usage cost of running this Hadoop application and explore the insights of the empirical results using monitoring data at both the system and the platform level. The source code of the MapReduce jobs is shown in Appendix A. Our

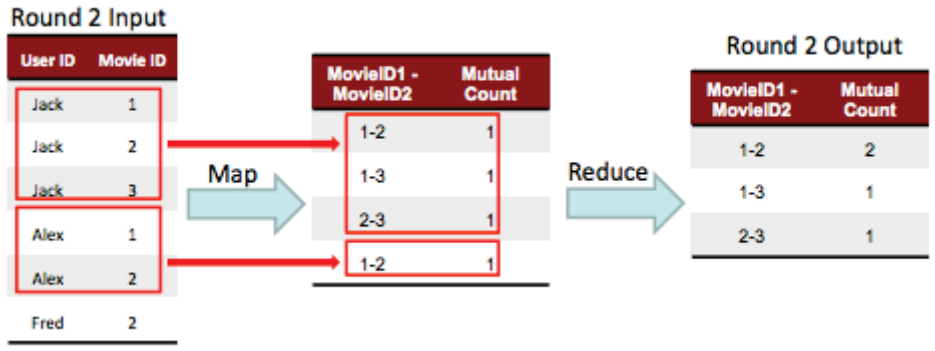


Figure 6: An Example of Round 2

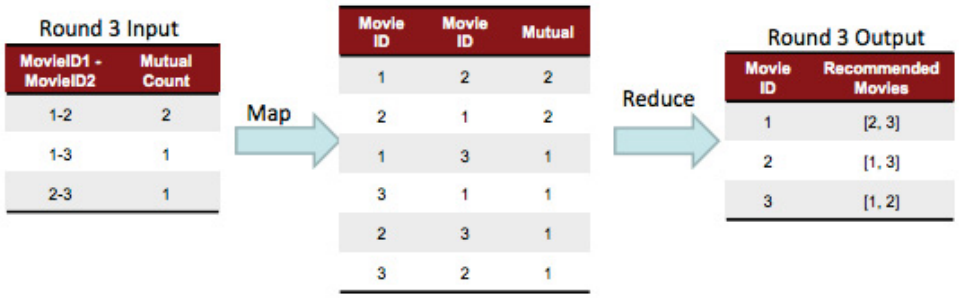


Figure 7: An Example of Round 3

evaluation is performed on Amazon Web Services (AWS), which provides the essential infrastructure components for setting up the Hadoop platform. The infrastructure components are listed in Table 1. AWS allows flexible choices on virtual machine images, capacities of virtual machine instances, and associated services (such as storage, monitoring and cost billing).

The deployment of the Hadoop architecture uses AWS Elastic MapReduce (EMR), which is a Hadoop platform across a resizable cluster of Amazon Elastic Compute Cloud (EC2) instances. The deployment architecture is shown in Figure 8. In the EMR configuration, we set up one master instance and ten core instances to run the Hadoop implementation of the movie recommendation application. The master instance manages the Hadoop cluster.It assigns tasks and distributes data to core instances and task instances. Core instances run map/reduce tasks and stores data using HDFS. All instances are in the type of m1.small, which has 1.7 GB of memory, 1 virtual core of EC2 Compute Unit, and 160 GB of local instance storage.

We use Amazon CloudWatch [Ama14] to monitor the status of Hadoop jobs and tasks and EC2 instances. For EC2 instances, CloudWatch provides a free service named Basic Monitoring, which

Table 1: Infrastructure Componentsof Movie Recommendation App

Application	Movie Recommendation
Platform	Apache Hadoop
Amazon Web Service	Simple Storage Service (S3)
	Elastic MapReduce (EMR)
Monitoring Tools	Amazon CloudWatch

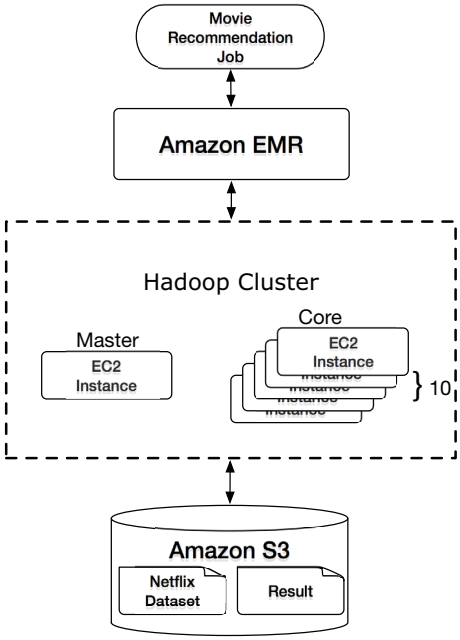


Figure 8: Deployment Architectures on AWS

includes metrics for CPU utilization, data transfer, and disk usage at a five-minute frequency. For the EMR service, CloudWatch has metrics about HDFS, S3, nodes, and map/reduce tasks. All these built-in EC2 and EMR metrics are automatically collected and reported. AWS users can access these metrics in real-time on the CloudWatch website or through the CloudWatch APIs.

The evaluation includes scenarios to emulate the incremental updates of data sets, in terms of data sizes. We consider scalability as to what extend a platform can utilize computing resources to handle increasing data sizes or data rates before a platform scales out on more computing resources. We also observe the cost incurred as the data sizes and update rates grow. To this end, we collect the following metrics to observe the performance of the Hadoop platform.

The system status metrics help to identify anybottleneck that limits performance. We explore

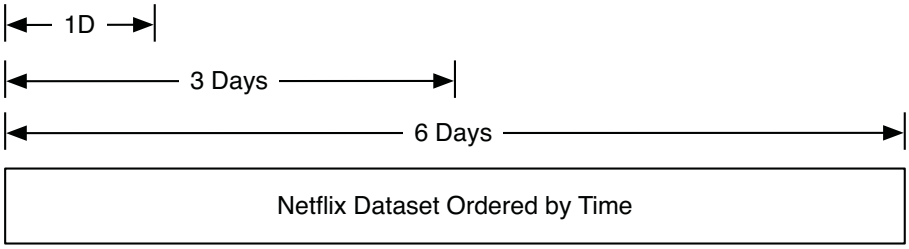


Figure 9: Input Rating File

the metrics of the platform status to observe factors at the application level that link to a bottleneck and affect performance most. The MapReduce tasks status provided by CloudWatch includes *RemainingMapTasks*, *RemainingMapTasksPerSlot*, *RemainingReduceTasks*, *RunningMapTasks*, and *RunningReduceTasks*.

In the Hadoop implementation of the movie recommendation application, the rating data are constantly increasing as users keep rating movies on the website. Therefore, Hadoop needs to process the whole set of rating data available as the size increases over time. To evaluate the scalability and cost of Hadoop processing incremental data, we create experiments that process different sized rating files. First, we order the Netflix dataset by time. Assume 200 ratings made by users every second, the dataset is of $200 * 24(hours) * 3600(seconds) * N$ records on the Nth day. We run our movie recommendation jobs over the period of days as N = 1, 3, 6 as Figure 9 and compare the results.

Understanding how minimal time interval or frequency of data processing varies according to the data update sizes helps to make scaling decisions on the optimal resource provision settings. Table 2 shows the elapsed time of Hadoop jobs with different sizes of input data. The elapsed time implies given a certain data size, the minimal time required torun the Hadoop application that updates the movie recommendation list. For example, the first entry in Table 2 means given the current capacity, it is only possible to re-run the Hadoop movie recommendation application with a frequency of once per hour for processing rating data updated in one day. For any shorter frequencies or larger datasets, more EMR instances need to be provisioned.

Table 2: The Elapsed Time of Movie Recommendation Jobs

Scenario	Data file size	Records	Elapsed time
1-day	428MB	17,280,000	56 minutes
3-day	1.2GB	51,840,000	177 minutes
6-day	2.4GB	103,680,000	407 minutes

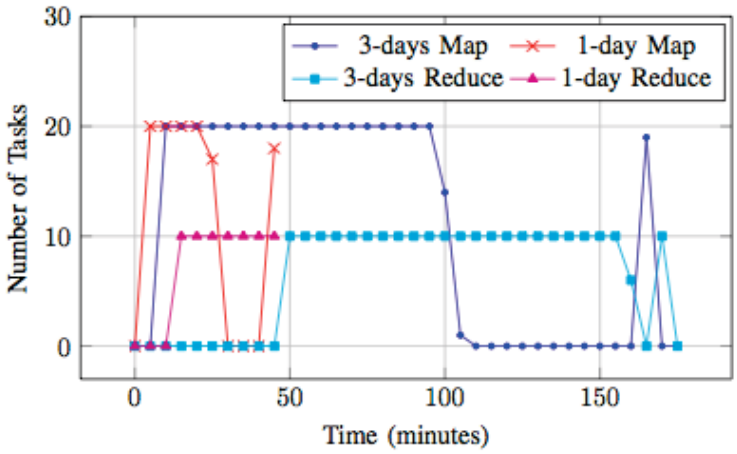


Figure 10: Number of Running MapReduce Tasks and Time Spent

From the platform status, Figure 10 shows the number of running tasks comparing the results of the 1-day input and the 3-day input. The evaluation environment has 10 core instances. Each instance has 2 mappers and 1 reducer. The running map and reduce tasks are at their maximum most of the time. In addition, all the mappers and reducers finish their job approximately at the same time, which implies that there is no data skew problem here.

Figure 11 shows the system status comparing 1-day data input and 3-days data input. As Figure 11(c) shows, in both experiments, there is no data written to HDFS and the data read from HDFS is less than 5KB/s all the time. This is mainly due to the fact that EMR uses S3 for input and output instead of HDFS. In the EMR services, the mappers read from S3 and hold intermediate results in local disk of EC2 instances. Then in shuffling stage, these intermediate results are sent through network to corresponding reducers that may sit on other instances. In the end, the reducers write the final result to S3.

In Figure 11(a), the experiment with 3-day input has the CPU utilization at 100% most of the time, except at three low points at 100 minutes, 125 minutes and 155 minutes, which is the time when the shuffling or writing results of reduce tasks occur. The average network I/O measurement in Figure 11(b) shows spikes at around the same time points, 100 minutes, 125 minutes and 155 minutes respectively while other times the average network I/O stays quite low. The highest network I/O rate is still below the bandwidth of Amazon's network in the same region, which is 1G bps to our best knowledge. The experiment with 1-day input has the same pattern with different time frames. From these observations we can infer that CPU utilization would be the bottleneck for processing higher frequency of data .

Table 3 shows the cost of the EMR service under different terms of data sizes and updating frequencies. For example, with a once-per-three hour updating frequency and 1.2G input files, we

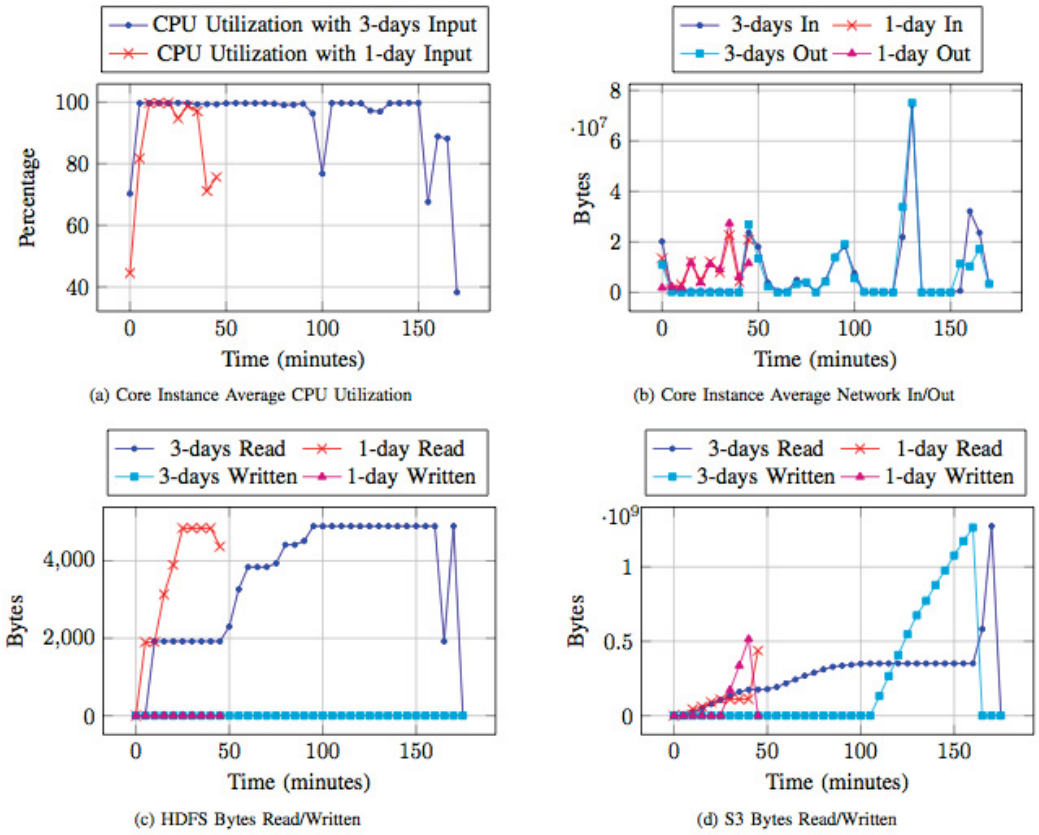


Figure 11: System Status of Hadoop Cluster

need to run the Hadoop recommendation jobs 8 times a day. Each time, it takes 2 hours and 57 minutes. The EMR price on AWS is 0.08 USD per instance hour. For a sub-hours usage, the billing system rounds it to an hour. As we have used 11 instances in total, it costs \$0.08*11*3*8 = \$21.12 per day. The NA (not applicable) in the table means that the processing time for data at this size is longer than the updating interval so that more instances need to be provisioned.

Table 3: Daily Cost of AWS Services

Update Frequency	Once per day			Once every 3 hours			Once per hour		
Input Data Size	428M	1.2G	2.4G	428M	1.2G	2.4G	428M	1.2G	2.4G
Daily Cost	\$0.88	\$2.64	\$6.16	\$7.04	\$21.12	NA	\$21.12	NA	NA

5.3 Processing Netflix Movie Ratings Data in S4

S4 shares the same MapReduce concept with Hadoop. The difference is that S4 can save intermediate result in its processing elements (PEs), which are the computation units of S4. Developers can customize PEs by inheriting class `ProcessingElement` and implementing the functions with their business logics. The core function for a PE is `OnEvent`, which processes new events and executes the business logic implemented by the developer. Listing 5.1 shows an example of the *OnEvent* function.

Listing 5.1: OnEvent in HistoryPE

```
1  public void onEvent(Event event) {
2      String strMovieId = event.get("mId", String.class);
3      logger.trace("userid [{}] movie id [{}]", getId(), strMovieId);
4      List<Event> newEvents = new ArrayList<Event>();
5      boolean bExist = false;
6      for (String favMovie : favMovies) {
7          if (favMovie.equals(strMovieId)) {
8              logger.trace("This is an existing movie id.");
9              bExist = true;
10         } else {
11             newEvents.addAll(createEventsForMutualFan(strMovieId, favMovie));
12         }
13     }
14     if (bExist) {
15         return;
16     }
17     favMovies.add(strMovieId);
18     for (Event e : newEvents) {
19         downStream.put(e);
20     }
21 }
```

For this movie item-based recommendation algorithm, three processing elements are designed in a pipeline, namely `HistoryPE`, `SimilarMoviesPE` and `RecoOutputPE`. Figure 12 shows the data flow graph of the stream processing application with these PEs.

The input and output stream of each PE consist of a series of events. Each event is a key-value tuple. For example, a `MutualFans` event (1, 3) indicates that movie 1 and movie 3 have a mutual fan. The key in this event is 1 and the value is 3. An event in S4 is a Java object, and it is serialized when transferred between PEs. The details of each PE are as follow:

`HistoryPE` filters the original input stream and stores the movies that a user watches or favors. Figure 13 shows an example that an instance of the `HistoryPE` stores that Jack is a fan of movie 1 and 2. When the `HistoryPE` receives a new event (Jack, 3) that indicates Jack is a fan of movie 3, the MovieID 3 is first stored in this instance, then the `HistoryPE` generates all new movie pairs (1, 3), (3,1), (2, 3) (3,2) that have a mutual fan as Jack. Since movie pair (1,2) and (2,1) are historic that means they have been processed already. Therefore, S4 allows only processing incremental and newly updated events, instead of processing the whole datasets every time a new event arrives. These generated events are sent to the next PE, `SimilarMoviesPE`.

The `SimilarMoviesPE` accumulates the numbers of mutual fans of movies and stores the result in its instances as the `SimilarMovie` list. Figure 14 shows an instance of `SimilarMoviesPE`. Movie 1 has two similar movies, which are movie 3 with 15 mutual fans and movie 5 with 10 mutual fans. When the `SimilarMoviesPE` receives a new MutualFans event (1, 3) from the `HistoryPE`, the `SimilarMovie` list is updated and re-ordered. Finally, the `SimilarMoviesPE` generates a `TocRecos` event in the pair of (MovieID, SimilarMovie list) and send it to the `RecoOutputPE`.

The `RecoOutputPE` stores the top-N similar movie lists for all the movies and outputs the movie recommendation list to a text file. The `RecoOutputPE` is the last PE so it does not generate any events.

5.3.1 The Experiments

Our objective is to study to what extend a stream processing platform can utilize computing resources to handle increasing data sizes or data rates before the platform scales out on more computing resources. Therefore we design evaluation scenarios to emulate the incremental updates of data sets, in terms of data sizes and stream rates.

We set up the the movie recommendation application on eleven Amazon EC2 instances, with one master instance and ten worker instances. Figure 15 shows the master instance runs ZooKeeper [HKJR10] and the stream generator. ZooKeeper is the coordinator of S4 cluster. It keeps track

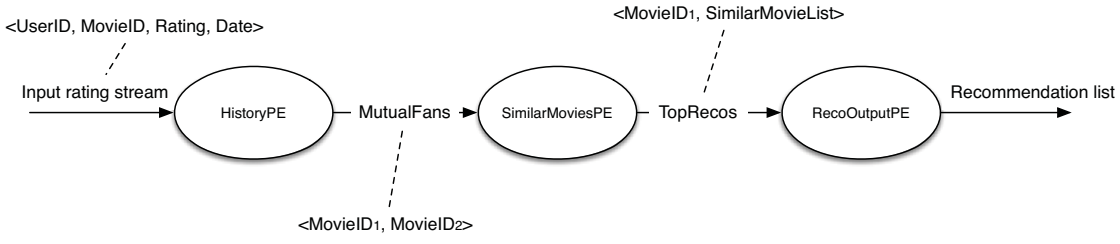


Figure 12: Data Flow Graph of Stream Processing Application

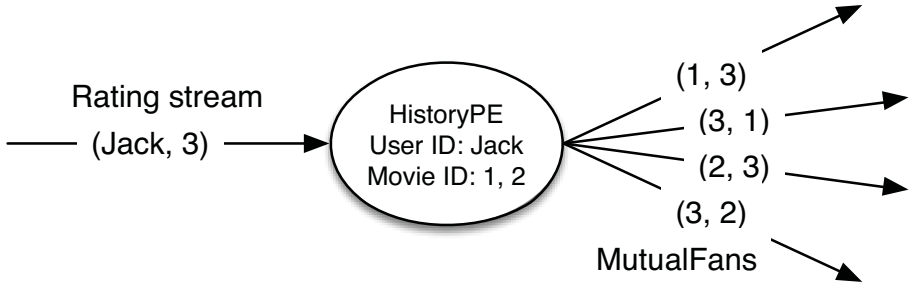


Figure 13: An Example of HistoryPE

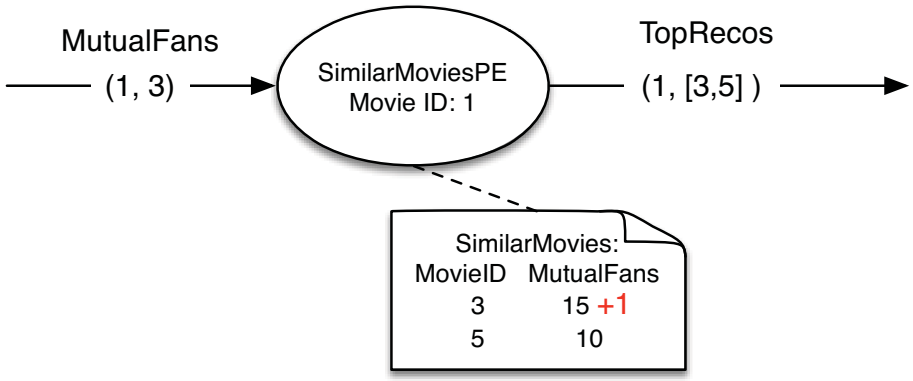


Figure 14: An Example of SimilarMoviesPE

of the changes of the worker instances (such as adding or removing a worker instance) and notifies master instance when changes occur. The stream generator reads rating records from the Netflix dataset stored on the master instance, and sends rating events to worker instances. Each worker instance has a S4 node running on it and each S4 node contains a set of PE instances. The master and worker instances are with the type of *m1.small*.

Given the fix size of EC2 instances, we collect the metrics in Table 4 to observe the scalability of the movie recommendation application as the stream rate increases . The status of EC2 instances are automatically collected and reported to CloudWatch.These system level metrics helps to identify any bottleneck that limits the performance. We then explore the metrics of the platform status to observe factors at the application level that link to a bottleneck and affect performance most. For the monitoring of Apache S4, we report the metrics from S4 to Graphite [gra], a real-time monitoring tool installed on the master instance.

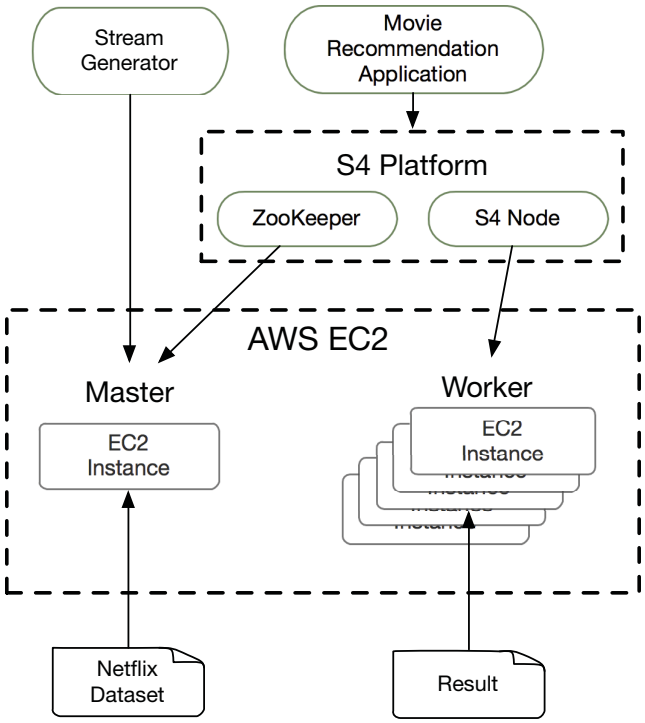


Figure 15: The Deployment Architecture of Stream Processing Application

Without Scaling

The input streams are configured by tuning the interval of i ms between events generated to control the stream rates. With $i = 1, 5, 10$, the generated stream has the input rate of approximately 850 events/s, 192 events/s, and 98 events/s respectively. With input rate of 850 events/s, the application can process each day's rating data approximately in size of 1.7GB (850 ratings per second * 24 (hours) * 3600 (seconds) = 73,440,000 records).

We measure the throughput of a PE as the number of events processed by the PE. Table 5 shows

Table 4: Collected Metrics

Platform	Apache S4
System Status	CPU Utilization
	Disk I/O
	Network I/O
Platform Status	Number of PE instances
	PE processing time

the throughputs (rounded) are almost the same as the input stream rate in the first two experiments. In the third experiment, when the events arriving at an intensive rate (i.e. 1ms interval between events and 850 events/s), the throughput drops to 276 events/s. Ideally, a scalable stream processing platform can produce the throughputs comparable to the increasing input rate. Once the runtime platform encounters a bottleneck, the throughput plateaus and no longer catches up with the input rates.

Table 5: Throughputs of the HisotryPE

Experiment Number	Input Stream Rate (events/s)	Throughput (events/s)
1	98	98
2	192	192
3	850	276

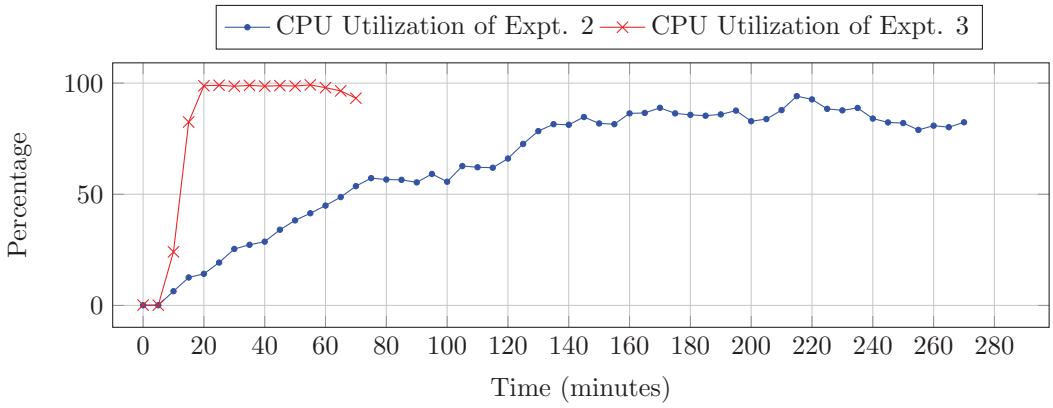
Further exploring the system and platform metrics collected helps to identify the factors contribute to this throughput degradation. Figure 16 shows the system status of movie recommendation application comparing experiment 2 (of 192 events/s input rate) and experiment 3 (of 850 events/s input rate).

In Figure 16(a), the CPU utilization of experiment 3 quickly rises to 100% in the first 20 minutes, while the CPU utilization of experiment 2 increases from 0% to 82% in 160 minutes and becomes steady around 80%. Figure 16(b) shows that the network I/O of experiment 3 climbs to approximately 13 MB/s and then drops down drastically. This is because of that the CPU utilization reaches 100% and many unprocessed events are stuck in the buffer. As Figure 16(c) shows, the disk I/O is quite low because S4 holds the intermediate data in memory.

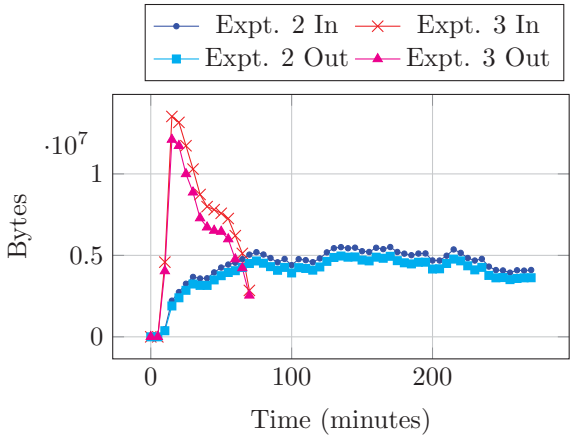
Figure 17 shows the number and processing time of `HistoryPE` and `SimilarMoviesPE` in the movie recommendation application. From these platform status, we observe the following phenomena.

Data characters : Figure 17(a) and Figure 17(c) indicate that `HistoryPE` has significantly more instances than `SimilarMoviesPE`. This is due to the nature of the Netflix dataset, which contains 480,189 users and only 17,770 movies. S4 creates a new `HistoryPE` instance every time it receives a rating from a new user ID.

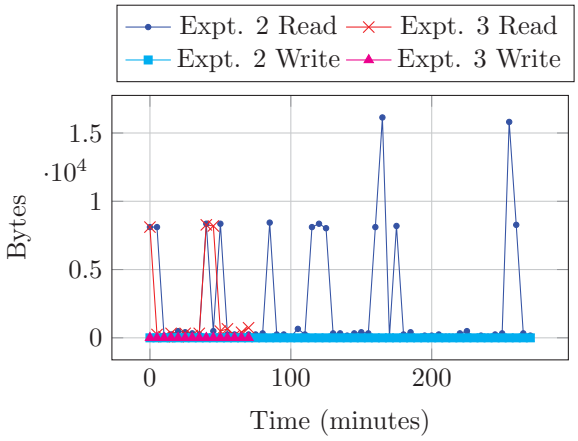
Number of PEs vs Input rate : As experiment 3 has a higher input stream rate, the number of instances of both the `HistoryPE` and the `SimilarMoviesPE` increases much faster in experiment 3 than experiment 2 as Figure 17(a) and (c) show. This leads to the full-utilized CPU usage in Figure 16(a).



(a) Average CPU Utilization



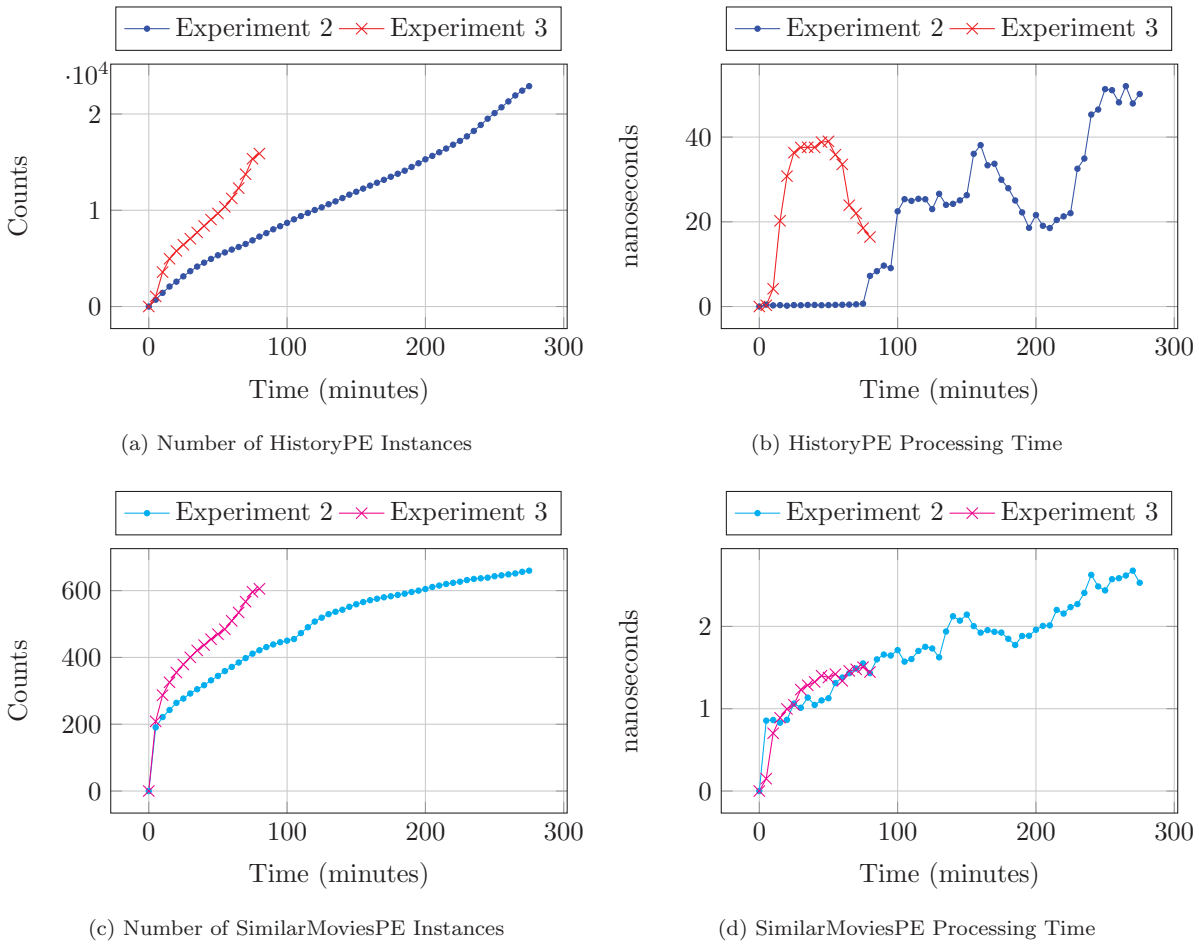
(b) Average Network In/Out



(c) Average Disk Bytes Read/Written

Figure 16: AWS EC2 System Status

Processing time vs Input rate : Figure 17(b) and Figure 17(d) show the average processing time of the `HistoryPE` and the `SimilarMoviesPE`. The average processing time measures the time taken for a PE to process a single event respectively. The processing time of the `HistoryPE` in experiment 3 increases, but the growth is unstable compared to the `SimilarMoviesPE`. In experiment 2, the processing time even begins to drop after 50 minutes. The fluctuation of the average processing time here is caused by a few 'lagging' instances. The 'lagging' occurs when some `HistoryPE` instances receive rating events from users who have rated thousands of movies, in which case the `HistoryPE` instances have to generate thousands of `MutualFans` events. The processing time of the `HistoryPE` on 'lagging' instances is usually hundreds or thousands of nanoseconds while the normal ones are only less than ten nanoseconds. The processing time of the `SimilarMoviesPE` keeps growing because all of its instances do ordering of ever increasing number of movies as data accumulates over time.



(a) Number of HistoryPE Instances

(b) HistoryPE Processing Time

(c) Number of SimilarMoviesPE Instances

(d) SimilarMoviesPE Processing Time

Figure 17: The S4 Platform Status

Manual Scaling

The experiments above indicate that the movie recommendation application needs to scale out as input stream rate increases. By configuring the AWS AutoScaling Group (ASG) and specifying the rule-based threshold of CPU utilizations, EC2 instances can be automatically provisioned (and de-provisioned). However current S4 only supports static configuration of EC2 instances through ZooKeeper as we presented in Figure 15 . When a new EC2 instance is available, S4 does not automatically run a S4 node on the instance. In addition, since PEs are running on a S4 nodes, scaling the S4 platform also requires transferring PEs and their associated states to a new S4 node on a EC2 instance, otherwise, the intermediate data accumulated in PEs are lost.

We manually provision EC2 instances to the movie recommendation application and try to find the minimum number of EC2 instances required to keep the throughput of the `HistoryPE` equal to the input rate since the `HistoryPE` is the most demanding PEs in this application. Table 5 shows

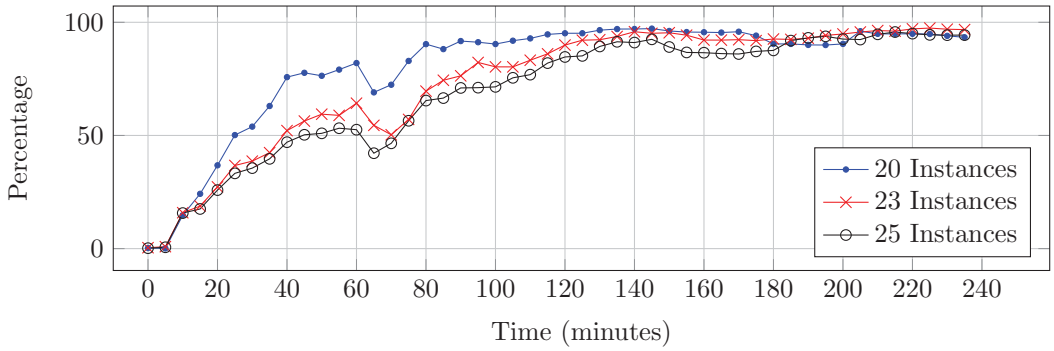


Figure 18: Average CPU Utilization of the S4 Cluster

that with input stream as 850 events/s, the 10-instance S4 cluster has a throughput of 276 events/s, which is approximately 27.6 events/s per instance. Intuitively, we estimate the number of instances needed is between the range of [10, $850/27.6 \approx 30$]. Then we use the bisection method to determine the cluster size. First, we choose 20 EC2 instances, if the throughput matches the input rate and the CPUs are not completely utilized, we cut down the number. Likewise we launch more instances if the throughput drops. In the end, we find the minimum size of S4 cluster to handle input stream rate of 850 events/s is 23 EC2 instances.

Figure 18 shows the average CPU utilization with 20, 23 and 25 nodes in the S4 cluster respectively. With 20 instances, the CPU utilizations of some instances reach 100%, which leads to the degradation of throughput. The average CPU utilization of S4 cluster with 25 instances is lower than the 20-instance one, and not fully utilized. Since AWS charges instance hours, under utilized instance incur unnecessary cost. In the end, we scale the S4 cluster to 23 instances, in which case the throughput keeps steady at 850 events/s and the CPUs are nearly 100%. Figure

We summarize the daily cost of handing input streams at different rates in Table 6. The cost is determined by the type and number of virtual machine instances and hours run by the application.

Table 6: Daily Cost of AWS Services

Input Stream Rate (events/s)		98	192	850
Daily Cost	Standard Linux	$17.16	$17.16	$35.88
	RHEL 6.4	$31.68	$31.68	$66.24

The experiment results show scaling techniques can help stream processing applications to meet the workload demands and save the cost. For a stream processing platform such as S4 that doesn't support scaling on the fly, the obstacle of autoscaling is to transfer PEs and their states from one instance to another. We further address this issue in Chapter 7.

5.4 Comparison of S4 and Hadoop

Using AWS CloudWatch and billing service, we are able to observe the compare measurements for both performance and cost metrics. We present how the measurements can be collected through AWS. These metrics allow us to discuss and explore the difference of the two programming models and platforms used to analyze the same datasets.

Programmability : Both Hadoop and S4 are open-source projects, and they provide a simple programming interface for developers. Hadoop is a little more programmer-friendly since Java is the only allowed programming language for S4 while Hadoop supports other programming languages through the Hadoop Streaming interface.

Deployment complexity : Hadoop is more complicated to deploy than S4 because it includes MapReduce module, HDFS, and many complex configuration files while S4 only needs the ZooKeeper and the S4 nodes. However, Hadoop is a widely-used platform and has a mature community so that it is easy to find documents or choose an integrated Hadoop service like Amazon EMR.

Integration with other tools : Hadoop can output metrics to data files or real-time monitoring tools, all you need to do is editing the configuration file. But for S4, you have to modify the source code to output the metrics to other tools, such as Graphite that we used in the experiments.

Autoscaling options : Amazon EMR allows users to increase or decrease the number of task instances. For core instances that contain HDFS, users can increase them, but not decrease. All these scaling operations can be made when the Hadoop cluster is running, which enables the auto-scaling ability for EMR if you have set up some proper rules based on the Hadoop metrics. In contrast, S4 does not support scaling up or down at runtime. So it remains future work for S4 to take adavantage of the auto-scaling feature of Amzaon EC2 service.

Ability to process incremental data : As the experiments of movie recommendation app shows, to maintain a frequent update on the movie recommendation, Hadoop costs much more as the data accumulates. In the real-time big data analytics scenarios, stream processing systems performs better and costs less.

Chapter 6

Dynamic Load Balancing

6.1 Overview

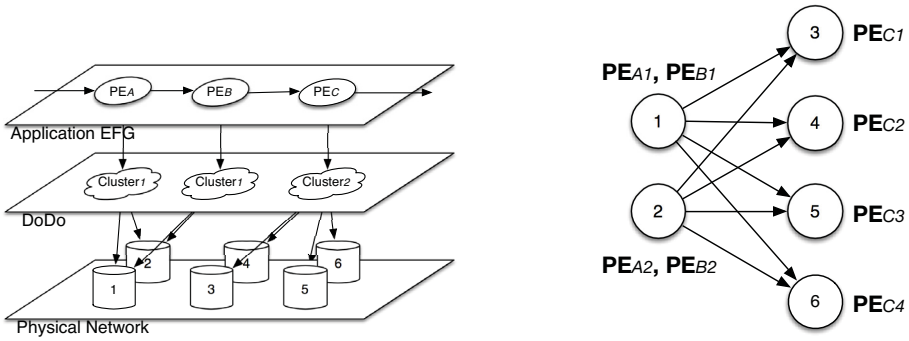
Distributed stream processing systems aim to process, analyze and make decisions on-the-fly based on immense quantities of data streams being dynamically generated at high rates. Stream services often exhibit multi-modal and spike workloads. For example, Mickulicz et at [MNG13] presents a cloud-based sports service, called YinzCam that has various modes of workload (e.g., pre-game, in-game, post-game, game-day, non-gameday, in-season, off-season) and exhibits that the traffic during the actual hours of a game is twenty-fold of that on non-game days. To handle these multi-modal workloads, YinzCam applies and tunes Amazon Web Services autoscaling configuration to automatically scale up and down the virtual machine instances on-demand. However, autoscaling considers a group of virtual machine instances as a whole, and does not necessarily guarantee that the workload is balanced on all instances.

Most stream processing software frameworks such as Borealis [AAB+05], SPADE [GAW+08] and S4 [NRNK10a] are operator-based, which means applications are organized as data flow graphs consisting of operators at the nodes connected by directed edges representing the data streams. The operators, which are also called PEs (Processing Elements) in some systems, are allocated to physical nodes within networked clusters.

Allocating a PE to a physical node has a direct effect on scalability. For a stream processing service in where PEs are connected to each other, these dynamic changes of inputs may significantly impact the loads of certain PEs due to the different selectivities (i.e., the ratio of the input and output data rates) of PEs. Even a small raise of the input stream of a low-selectivity PE may lead to a dramatic increase of its output rate. Thus the destination PEs, the PEs that takes its output as input, have to handle a much higher input rate. As a result, the cluster nodes running destination

PEs could be overloaded while the other nodes are not fully utilized. The selectivity of a PE depends on not only the business logic of the service, but also the data in the input stream. Normally, the selectivities of PEs fluctuate as the input stream varies over time, which makes the change of load distribution unpredictable. Therefore, the ability to balance the loads on different nodes at runtime is essential for a scalable and reliable stream processing service. By changing the allocation of PEs we can adjust the loads on nodes and optimize the utilization of computing resources. Meanwhile, an inefficient allocation may result in unnecessary data transfers between operators through the network that can cause extra latency.

6.2 The Optimization Method



(a) EFG Mapping Progress

(b) DFG

Figure 19: DoDo assigns physical nodes for an application with 3 types of PEs (PE_A, PE_B, PE_C). Both PE_A and PE_B are allocated on $Cluster_1$ that has 2 nodes. PE_C is allocated on $Cluster_2$ that has 4 nodes.

The goal of optimization is to minimize the average load variance and maximize the throughputs of stream processing services. We design an optimization method applied in the mapping process to assign PEs to a cluster node as depicted in Figure 19. At the bottom layer, the physical nodes are connected by a high bandwidth network in a cloud environment. At the top layers, as an example, both operators PE_A and PE_B are allocated on $Cluster_1$. Hence when $Cluster_1$ is overloaded, moving PE_B or PE_A to $Cluster_2$ could potentially solve the problem. To make this moving decision load adaptive, we consider the following aspects in the optimization method development:

1. We assume that CPU utilization data are collected on nodes to measure the load of PEs periodically over fixed-length time series. The input stream for a PE consists of a series of events. In each period, the load of a PE is defined as the CPU time needed by the event

processing. If the average events rate in period i is λ and the average processing time of an event is p, then the load of the PE s_P is λp.

2. The optimization works on the level of clusters. It means that if a PE is assigned to a cluster, then its instances are allocated equivalently on each node of the cluster.

3. The method optimizes the current PE's allocation based on the current number of nodes in clusters. It does not make decisions on adding or removing nodes.

Given the definition of the load of a PE as $s_P = \lambda p$, we further define the load of a cluster. Assume a stream processing application running on k clusters $(C_1, C_2, ..., C_k)$. The application contains n PEs $(P_1, P_2, ..., P_n)$ with loads $(s_{P_1}, s_{P_2}, ..., s_{P_n})$. Let a cluster C_i have j PEs $(P_1, P_2, ..., P_j)$ running on it, $(j \leq n)$. N_i denotes the number of nodes in cluster C_i. We assume that the average load of a cluster C_i can be expressed as s_{C_i}:

$$s_{C_i} = \frac{1}{N_i} \sum_{l=1}^{j} s_{P_l} \tag{1}$$

6.2.1 Static Mapping Optimization

First, we discuss the static mapping problem. The assumption is that the load on each PE is steady and the input stream rate does not change over time. In this case, the optimization objective is to find a mapping plan so that the difference between the average loads on different clusters is minimized, i.e. $s_{C_1} \approx s_{C_2} \approx ... \approx s_{C_k}$.

If we further assume that each cluster has only one node, the problem becomes allocating the set of PEs with loads $(s_{P_1}, s_{P_2}, ..., s_{P_n})$ to k clusters with balance. This is a k-way *Number Partitioning Problem* [Kor09], which is to divide a given set of numbers into a collection of subsets so the sum of the numbers in each subset is as nearly equal as possible. This problem is NP-hard. Most existing approaches focus on suboptimal solutions. Among them, a greedy heuristic algorithm is effective and widely used. The principle of the algorithm is to sort the numbers in a decreasing order, and then assign each number in turn to the subset with the smaller sum so far [ZMP11].

Now we consider a cluster has multiple nodes, and the number of nodes is different on clusters. We apply a similar strategy as the greedy heuristic algorithm above to solve this partitioning problem. Instead of having the subsets being ordered by the sum, we consider the sum should be divided by the capacity of the cluster (i.e. the number of nodes in that cluster) to present the average load of the cluster. For an instance, when a PE P_i is assigned to a cluster C_j, the additional cluster load is s_{P_i}/N_j, where N_j is the number of nodes in cluster C_j.

The static mapping optimization above can only help to balance the load at a specific period of time when the load is of each PE is steady and the variation of input stream rate is negligible.

6.2.2 Dynamic Mapping Optimization

In the problem of dynamic mapping optimization, the load of each PE can vary over time as the input stream rate fluctuates. In this case, we need to consider not only the current load, but also how to keep the loads balanced over time. For an instance of two clusters C_1 and C_2, the average load of C_1 is s_{C_1} and the average load of C_2 is s_{C_2}. Assume the input stream rate is increasing over time and let Δs_{C_1} and Δs_{C_2} denote their incremental loads after time t. The PE and cluster mapping plan should make $s_{C_1} \approx s_{C_2}$ and $\Delta s_{C_1} \approx \Delta s_{C_2}$.

An algorithm presented in [XZH05] uses the correlation of load series of PEs over fixed-length time periods to determine the allocation of PEs. In this algorithm, the authors define the load variance of an operator/node as the variance of the load time series of an operator/node. Their research observes [XZH05] that if the correlation coefficient of the load time series of two PEs is small, putting them together on the same node helps minimize the load variance. In other words, the smaller correlation coefficient indicates the bigger difference of loads of the two PEs in a given period of time. So the algorithm allocates operators to the node that has the largest correlation coefficient. Since this algorithm directly allocates PEs to physical nodes, further improvement on this algorithm is necessary to consider allocating PEs to clusters of various size of physical nodes.

In our case, each PE is assigned to a cluster that contains a sets of nodes. Our calculation is on the level of clusters and PEs. We assume that a load series S (such as a load series of a PE or a load series of a cluster) contains the load array $(s_1, s_2, ..., s_k)$ of the most recent k periods. For the load series of a PE, the load in the array is the load of the PE at a specific time period. For the load series of a cluster, the load in the array is the average load of the cluster at a specific time period. Then the mean and the variance of the load series S are defined as follows:

$$ES = \frac{1}{k}\sum_{i=1}^{k} s_i \tag{2}$$

$$varS = \frac{1}{k}\sum_{i=1}^{k} s_i^2 - \left(\frac{1}{k}\sum_{i=1}^{k} s_i\right)^2 \tag{3}$$

Given two load series $S_1 = (s_{11}, s_{12}, ..., s_{1k})$ and $S_2 = (s_{21}, s_{22}, ..., s_{2k})$, their covariance $\mathrm{cov}(S_1, S_2)$ and correlation coefficient ρ are defined as follows:

$$cov(S_1, S_2) = \frac{1}{k}\sum_{i=1}^{k} s_{1i}s_{2i} - \left(\frac{1}{k}\sum_{i=1}^{k} s_{1i}\right)\left(\frac{1}{k}\sum_{i=1}^{k} s_{2i}\right) \tag{4}$$

$$\rho = \frac{cov(S_1, S_2)}{\sqrt{varS_1}\sqrt{varS_2}} \tag{5}$$

Now we formalize the optimization problem as follows:

Assume a stream processing application running on k clusters $(C_1, C_2, ..., C_k)$. The application contains n PEs $(P_1, P_2, ..., P_n)$ with loads $(s_{P_1}, s_{P_2}, ..., s_{P_n})$. Let ρ_{ij} denotes the correlation coefficient

of C_i and C_j for $1 \leq i, j \leq n$. The objective is to find a PE mapping plan with the following properties:

1. $s_{C_1} \approx s_{C_2} \approx ... \approx s_{C_k}$
2. $\sum_{1 \leq i \leq j \leq k} \rho_{ij}$ is maximized

Now we present a greedy algorithm in Algorithm 1 that maps PEs with a load series measured to clusters of different size. The input of this algorithm is a list of paired clusters. First, clusters are ordered by their average loads according to Eq. 1. The i^{th} cluster in the ordered list is paired with the $(n-i+1)^{th}$ cluster in the list, which means the cluster of the largest average load is paired with the cluster of the smallest load.

In this algorithm, we do pairwise PE transmission between clusters so that each cluster will do at most one PE transmission at a time. This limits the number of transmissions required. As Algorithm 1 indicates, for each cluster pair, we first determine whether a PE transmission should be made between the pair. Line 2 shows the conditions if a PE transmission is necessary: 1) the load differences between the two clusters must be larger then a threshold; and 2) the correlation coefficient of the two clusters must be less than a threshold. These two conditions make the PE transmission only occurs between unbalanced clusters. Then the algorithm selects PEs from the cluster with larger load, i.e. cluster C_1 where $s_{C_1} > s_{C_2}$.

We then need to find the PE that is the most correlated to cluster C_1 or the least correlated to cluster C_2 so that after the PE transmission, the correlation of C_1 and C_2 is the largest. Let $\rho(S_P, S_1)$ denotes the correlation coefficient between the load series of PE P and the load series of all the other PEs in C_1 except P. Then the PE selection from C_1 should follow the decreasing order of $(\rho(S_P, S_1) - \rho(S_P, S_2))/2$ as Line 3 indicates. Let N_1 and N_2 denote the capacities of cluster C_1 and C_2. If a PE is moved from C_1 to C_2, the load of C_2 increases as the amount of $ES_{PE} * N_1/N_2$. To avoid a reverse transmission, a threshold for the load difference is as the total load of the selected PEs must be less than $(ES_{C_1} - ES_{C_2})/2$.

6.3 Architecture

Figure 20 shows the architecture overview of a distributed stream processing system (DSPS) with DoDo, which is a software layer constructed on top of networked cluster nodes. DoDo is designed to provide fundamentals to support a flexible combination between PEs and cluster nodes in a DSPS. DoDo arranges a cluster for each of the PEs and allows different PEs using the same cluster. Moreover, DoDo can move PEs to another cluster and adjust the number of nodes in a cluster. All

Algorithm 1: The Mapping Algorithm

input : Paired clusters

output: PE transmission list

```
foreach cluster pair (C1, C2) in paired cluster list do
    if ES_C1 − ES_C2 > Threshold_AverageLoad and ρ(S_C1, S_C2) < Threshold_Correlation then
        Order PEs by (ρ(S_P, S_C1) − ρ(S_P, S_C2))/2 desc;
        // TotalLoad is the sum of the loads of PEs in the transmission list;
        TotalLoad = 0;
        foreach PE in Desc Ordered PE list do
            if (ES_PE + TotalLoad) * N1/N2 < (ES_C1 − ES_C2)/2 then
                add (PE, C1, C2) in transmission list;
                TotalLoad+ = ES_PE;
```

of these operations can be made on-the-fly. When an application is deployed to a DSPS, (1) the platform informs the *Cluster Manager* to provide a cluster for each type of the PEs; (2) the *Cluster Manager* registers the combination of PEs and clusters as well as the communication information of the physical nodes in each cluster synchronized by ZooKeeper; (3) then PEs are instantiated on physical nodes and the DSP platform starts to process events; (4) new events generated from a PE are sent to the cluster that contains the next PE through the *Events Dispatcher*.

DoDo consists of six components as shown in Figure 21:

1. The *Data Flow Graph (DFG) Optimizer* makes optimization decisions based on the information collected by the *Health Monitor* from physical nodes.

2. The *Health Monitor* collects both system utilization data and PE health data such as input event rates, processing time and the number of PE instances.

3. The *PE Transmission Controller* notifies the *Events Dispatcher* to transmit PE states by creating PE transmission tasks in ZooKeeper.

4. The *Cluster Manager* registers the combination of PEs and clusters as well as the communication information of the physical nodes in each cluster synchronized by ZooKeeper.

5. The *Health Client* reports health data to the *Health Monitor* on the master node.

6. The *Events Dispatcher* is in charge of sending events of data stream to the destination node and transmitting PE states whenever necessary. The *Events Dispatcher* is connected with ZooKeeper so that it can send events to the destination node.

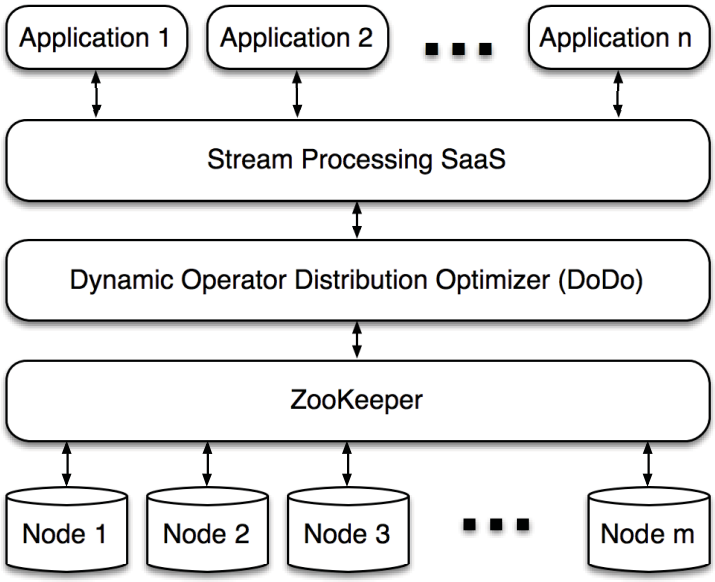


Figure 20: Architecture Overview

The *Health Client* and the *Events Dispatcher* are running on the worker nodes along with PEs. All the other four components of DoDo are running on a master node. The system is uncentralized as the master node is only dedicated to the optimization. In other words, the stream processing application can keep working even if the master node is down.

6.3.1 The Allocation Decision Making Process

The *DFG Optimizer* makes the mapping optimization decisions using Algorithm 1. We divide the decision making process into three major steps.

First step : collecting information. The information contains 1) the event flow graph (EFG) of the stream processing application, in which the vertices represent different types of PEs and the edges represent data streams; 2) the PE-cluster mapping and the capacities of clusters; and 3) the health data collected by the *Health Monitor*. 1) and 2) are written to the ZNodes on ZooKeeper by the *Cluster Manager* when the application is deployed. For the health data, each node has a *Health Client* running on it. The *Health Client* collects the data and sends to the master node, where the data is stored and monitored.

Second step : making mapping decision. Based on the information gathered in the first step, the *DFG Optimizer* decides whether a PE transmission should be made. The *DFG Optimizer* reads the health data from the storage of the *Health Monitor* and the data from ZooKeeper for a given time

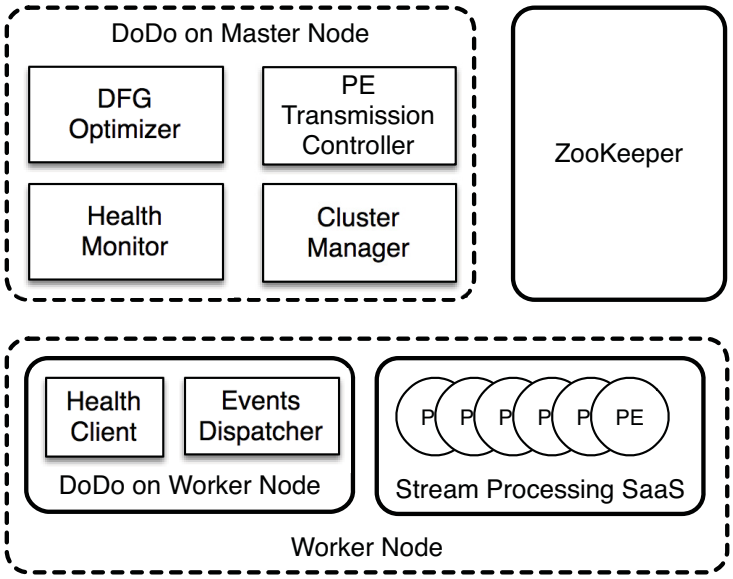


Figure 21: Structure of DoDo

interval. Then it applies the optimization algorithm to get the result as a PE transmission list.

The last step : executing the PE transmission. If the result from the second step is an empty PE transmission list, it implies that either the system is well balanced or it cannot find a PE-cluster mapping plan to solve the unbalance problem. If the result from the second step is not empty, the *DFG Optimizer* notifies the *Cluster Manager* to change the mapping on ZooKeeper so that the *Events Dispatcher* can send the events to the new destination. Moreover, if some PEs in the list are stateful, the *PE Transmission Controller* creates a PE transmission task on ZooKeeper. As the *Events Dispatcher* has setup watches on the ZNodes of ZooKeeper, it is notified to execute the PE transmission task. Then the *Events Dispatcher* creates some special events that contain the states, and sends them to the new cluster. When the *Events Dispatcher* on the new cluster receives the special type of events, it asks the DSPS to generate the corresponding PE instances and assign the states to the PE instances.

6.3.2 Implementation

Our implementation is based on *S4*, which is a widely-used open-source stream processing framework first developed by Yahoo!. Yahoo! has implemented an online parameter optimization (OPO) system using S4 to automate the tuning of one or more parameters of a search advertising system with live traffic. In a two-week experiment the optimal parameters generated by the OPO system demonstrated reasonable improvements in the primary measures of system performance: revenue by

0.25% and click yield by 1.4% [NRNK10a]. In another paper, researchers built a high-throughput system on S4 for reasoning over RDF streams [HK11]. We choose S4 because of its pluggable architecture, with which we can easily make DoDo work as the dynamic operator distribution layer. Moreover, S4 is suitable for comparison experiments as itself support neither adding nodes to a cluster nor PE transmission at runtime.

In our previous work, we implemented a prototype of DoDo [WL14a]. The prototype included the components of DoDo except the *DFG Optimizer*. In this paper, we have implemented *DFG Optimizer* along with two algorithms. One algorithm is Algorithm 1, the mapping optimization algorithm proposed in Section 6.2. The other algorithm is only focused on maximizing the correlation without considering the capacity of clusters.

6.4 Evaluation

The overall architecture is demonstrated by a real-time top-N recommendation application in social streams. The main objective is to evaluate the efficiency of the PE allocation optimization method under different scenarios to balance the load of PEs on clusters. We use Twitter's stream data and develop a stream processing service that reads tweets stream from the Twitter Sample Stream API [MPLC13], extracts topics from the tweets, counts occurrence of each topic and outputs the top-10 topic list every 10 seconds. The implementation in terms of PEs contains one adaptor (for generating stream) and three PEs in the pipeline, namely *ExtractorPE*, *TopicCountAndReportPE* and *TopNTopicPE*. The adaptor reads tweets from Twitter sample stream API by default.

As the sample stream from Twitter is only approximately 50 tweets per second and only a few of them have topics, it can hardly make stress test to our clusters. So we modify the adaptor to generate sampling tweets with topics that follow the Zipfian distribution (with 5000 elements and the exponent as 0.5). Figure 22 shows the occurrences of the topics when we generate 100,000 sample tweets. We also tune the generating speed of tweets, making the speed start from 20 tweets per second and increase 20 tweets every second. By this setting, we can simulate incremental input streams and measure the platform's throughput.

The testbed is a 11-node cluster provided by Utah's Emulab [WLS+02]. Each node has a 3GHz 1-core Xeon processor, 2GB of RAM and 210,000 RPM 146GB SCSI disks. All nodes are interconnected by a 100Mb Ethernet. The operating system uses 64bit Fedora 15 with Linux kernel version 2.6.40. We deploy ZooKeeper and DoDo components on a master node. We divide the rest 10 nodes into 3 clusters: Cluster_1 with 3 nodes, Cluster_2 with 6 nodes, Cluster_3 with only one node.

We test the system performance under two different deployments of the application. In the first case, which is called extreme case as Table 7 implies, we deploy the stream generator on Cluster_1,

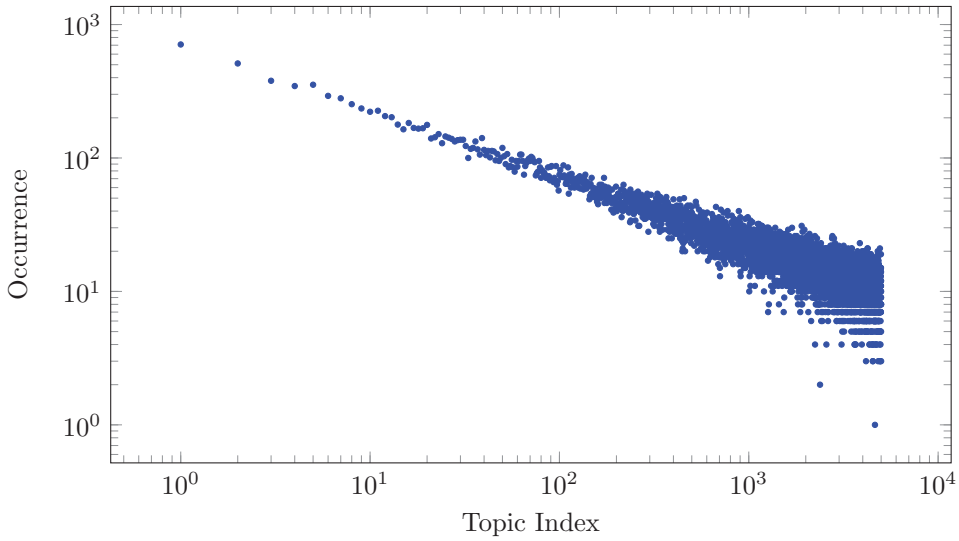


Figure 22: Topics follow Zipfian Distribution

and *ExtractorPE* on Cluster_2. Other two PEs are deployed on Cluster_3 at the beginning. Cluster_3 is more likely to get overloaded because it contains only one node but has 2 PEs running on it. In another case, which is called optimal case as Table 8 implies, the Cluster_1 remains the same. We put only the *TopNTopicPE* on Cluster_3, and the other 2 PEs on Cluster_2 that contains the most nodes among all the clusters. For both cases, we make a comparison test between Algorithm 1 and its variation that does not consider the capacities of clusters. The purpose is to observe under each deployment to what extent the optimization method can help allocate loads of PEs. We run the application until the rate of input stream stops increasing, which means the platform reaches its maximum arriving requests, and thus the maximum throughput.

Table 7: Deployment of Extreme Case

	Capacity	Allocated PEs
Cluster_1	3	*Stream Generator*
Cluster_2	6	*ExtractorPE*
Cluster_3	1	*TopicCountAndReportPE, TopNTopicPE*

Figure 23 shows the result of our evaluation. The legend entry *Opt with capacity* stands for the optimization algorithm proposed in Algorithm 1. The legend entry *Opt w/o capacity* stands for a similar algorithm without considering a cluster's capacity in the correlation estimation. The legend entry *No-opt* means the original S4 without any optimization algorithm.

In the extreme case shown in Figure 23 (a) and (b), the optimization algorithm with capacity

Table 8: Deployment of Optimal Case

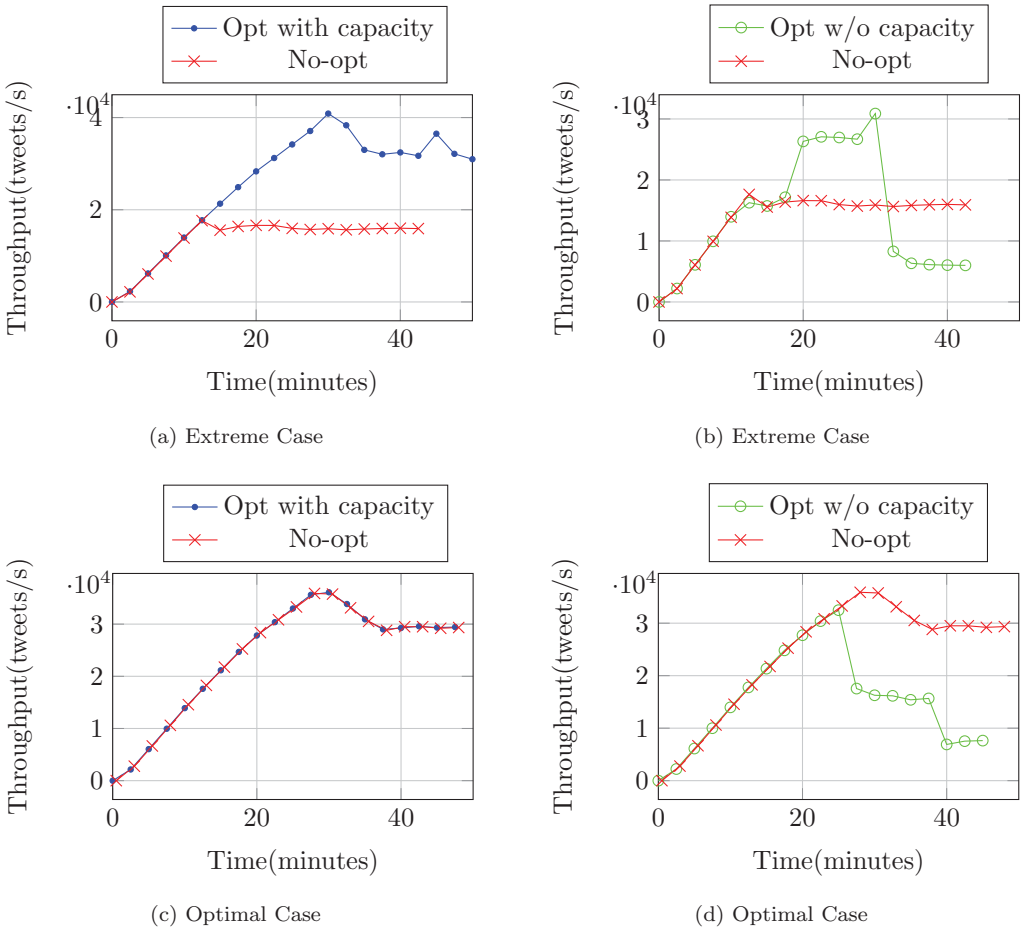
	Capacity	Allocated PEs
Cluster_1	3	*Stream Generator*
Cluster_2	6	*ExtractorPE*, *TopicCountAndReportPE*
Cluster_3	1	*TopNTopicPE*

considered has the best performance. At the beginning, the throughput increases along the input rate. Both algorithms of optimization perform well. As the input rate reaches 19K tweets/s, the throughput of original S4 starts dropping and finally becomes steady at approximately 16K tweets/s. This is caused by the unbalance between Cluster_3 and Cluster_2. Cluster_3 has only one node and runs two PEs and it becomes the bottleneck of the whole platform when the input rate reaches 19K tweets/s, while Cluster_2 is still relatively free. Our optimization algorithm (i.e.*Opt with capacity*) recognizes the unbalance and moves *TopicCountAndReportPE* from Cluster_3 to Cluster_2. This decision utilizes the cluster resource better and improves the throughput by 87.73% (as shown in Figure 23 (a)). The other algorithm, *Opt w/o capacity*, has made the same decision as *Opt with capacity* when Cluster_3 is overloaded. However, as the input stream rate keeps growing, Cluster_2 becomes overloaded, too. *Opt w/o capacity* decides to move *ExtractorPE* from Cluster_2 to Cluster_3 because it does not consider the capacity of Cluster_3. Then the throughput drops dramatically from 30K tweets/s to 6K tweets/s, which is 62.5% less than the original S4 (as shown in Figure 23 (b)).

In the optimal case, both the original S4 and *Opt with capacity* do well as shown in Figure 23 (c). Their throughputs are almost the same because the optimization algorithm does not need to change the mapping of PEs to clusters. This also indicates the the overhead of our optimization algorithm and the associated architecture is ignorable since it produces almost identical throughputs as the original streaming service of S4.

In the contrary, *Opt w/o capacity* shown in Figure 23 (d) has made two mapping decisions unnecessarily and thus degraded the performance. First the throughput drops to 16K tweets/s when *TopicCountAndReportPE* is transmitted to Cluster_3. Then after the second decision of PE transmission, the throughput drops again, down to 7K tweets/s.

Through the experiments we can see the decision of PE allocation is significant to the overall performance and scalability of a stream processing service. Without adding any new resources, DoDo can improve the throughput by changing the PE allocation dynamically to utilize the cluster resources. Figure 24 shows the average CPU utilization of the clusters in the extreme case with the algorithm *Opt with capacity*. The figure clearly shows that the CPU utilization of Cluster_3 goes down dramatically at the minute of 12, meanwhile the CPU utilization of Cluster_2 increases for



(a) Extreme Case

(b) Extreme Case

(c) Optimal Case

(d) Optimal Case

Figure 23: Twitter Top-N Topic List Application

approximately 150%. This is caused by the PE transmission from Cluster_3 to Cluster_2. As a result, for the ten nodes of the whole system, the *DFG Optimizer* has utilized them 3 times as before.

Although an experienced developer can tune the allocation of PEs at the beginning of the deployment to avoid the extreme case, however, such a static allocation still cannot handle incremental data stream loads that may cause unbalance of operators at runtime. This evaluation demonstrates our optimization method can help better utilize computing resources by means of dynamically balancing the load of PEs on clusters, and thus improve service throughputs.

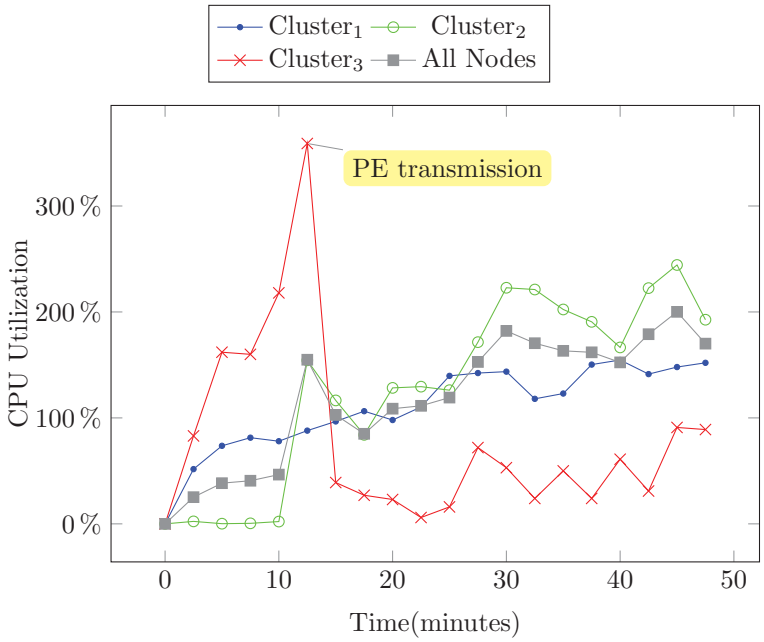


Figure 24: Average CPU Utilization of Clusters in Extreme Case

Chapter 7

Autoscaling

Dynamic load balancing improves the utilization by balancing the workload of PEs on existing clusters. In the case that a streaming service requires scaling out the clusters, dynamic load balancing should be further expanded on clusters provisioned on demand. By nature, this is the process of autoscaling. Autoscaling in distributed stream processing systems has two major problems: (1) how to synchronize the new node information to all the existing PEs at runtime; and (2) stateful PE instances must be re-shuffled based on the events dispatcher scheme and transferred to another node when the number of nodes in this cluster has changed. With the architecture presented in Chapter 6, the first problem can be solved by keeping tracking of the status of cluster nodes and synchronizing the state to other related nodes by means of ZooKeeper. As for the second problem, we should optimize the relocation of PEs by moving as few PEs as possible. The PE transmission can be handled by the *PE Transmission Controller* in DoDo with techniques when we deal with load balancing.

In this chapter, we present the essential algorithm of autoscaling and extension to the DoDo architecture that further enables autoscaling PE workloads upon the provisioning of cluster nodes. The contribution is three folds. First, we demonstrate the mechanism to add or remove nodes in a cluster under the DoDo architecture. Second, we integrate the autoscaling algorithm into the *DFG Optimizer* of DoDo so that the DoDo architecture supports both autoscaling and dynamic load balancing. Third, the result of our evaluation shows that autoscaling can achieve higher throughput and use less instance time of the cloud resources in some scenarios.

7.1 Mechanism

The workflow of autoscaling is devided into four phases shown in Figure 25, including Overload Detection (step (1) (3)), Instance Provisioning (step (4)), Cluster Information Synchronization (step

(5)(6)), PE Relocation (step (7) (9)).

The steps of the workflow are described as follows: (1) the *Health Client* on work nodes collects both system utilization data and the PE health data. The *Health Client* sends these data to the *Health Monitor* on the master nodes; (2) In the master node, the *DFG Optimizer* analyzes the data from the clusters and detects if any of the clusters are overloaded; (3) if overloading occurs, the *Cluster Manager* is notified; (4) In the case of overloading, or in the other words, all the nodes of the cluster are currently saturated, the *Cluster Manager* launches a new node and assigns it to the cluster until the number of nodes has reached the limitation of capacity; (5) the new node registers itself to *ZooKeeper* and all other nodes connected to *ZooKeeper* are notified by *ZooKeeper* about the registration of this new node; (6) As the number of nodes in the cluster has changed, the workload of PE needs to be rebalanced (please refer to Chapter 6). Hence, some PE instances should be migrated to other nodes. The *PE Transmission Controller* calculates the new location of PE instances using a partitioning algorithm based on hashed keys (details are covered in section 7.1.3) and (7) arranges the PE transmissions; (8) PE states are transferred from previous node the PE originally residents to the new node. The transmission is accomplished by creating a special type of event and sending them via the *Events Dispatcher*; (9) when the *Events Dispatcher* on the new node receives these PE transmission events that contain PE states in the body, it generates new PE instances and restores the states from those events.

7.1.1 Overload Detection and Instance Provisioning

As we have collected both system utilization data and PE health data on each work node, the overload detection can be achieved by analyzing these data. Much work has been done to dynamic resource provisioning in a cloud [DADCB09, LBCP09, PSZ+07, USCG05, RDG11]. There are two types of overload detection methods. One type is predicting workload by observing past workload patterns, such as time-of-day or seasonal patterns. The other type is adjusting numbers of instances based on short-term fluctuations.

Both methods must take the following characters into consideration: (1) Instance Provisioning Time. It may take up to 10 minutes to launch an instance and start the services. If the overload detection reacts too frequently to load changes, it may cause unnecessary scaling in and out when facing workload spikes [MLH10]. (2) Number of instances to provision. A resource estimation based on the current or predicted load is needed before perform the instance provisioning. (3) Economic issues. The billing model of most commercial cloud infrastructure is usally based on instance hours, and both 1-minute and 59-minute usage cost a whole hour's fee. In addition, the price of using one instance for an hour can fluctuate over time in some commercial cloud. As a result, the provision plan should adopt to the specific cost model of the cloud infrastructure.

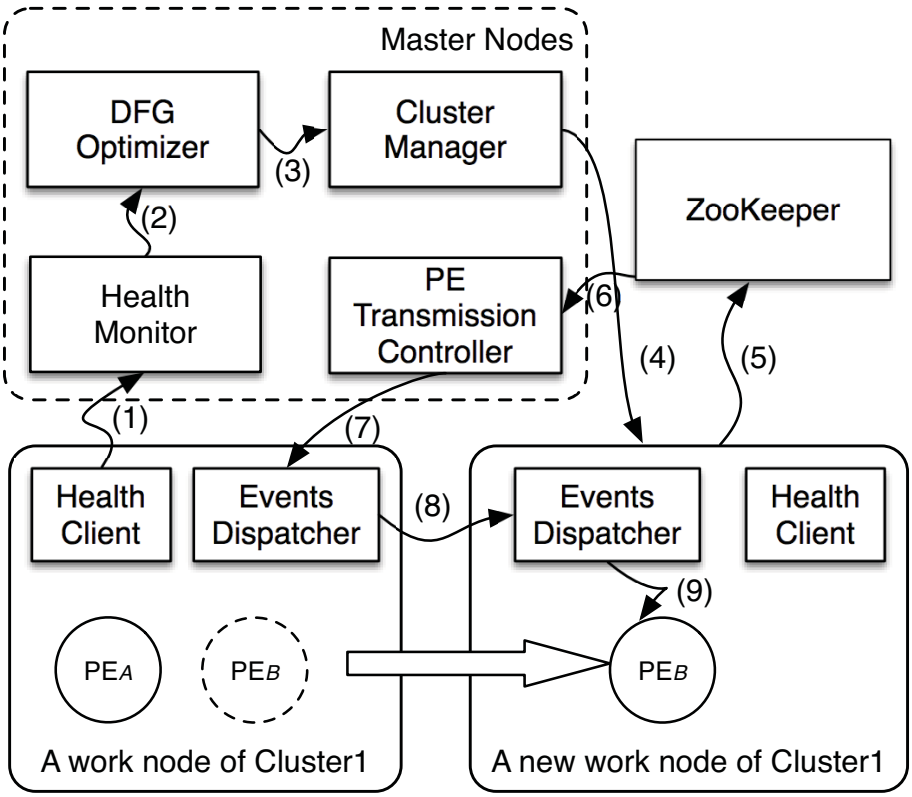


Figure 25: Workflow of Autoscaling

In this chapter, a resource provision algorithm, Algorithm 2, is designed to add new nodes into the cluster and automatically balance the workload of PEs to the new nodes. The average CPU time that PEs occupies on an instance, denoted by ES_C, is measured to detect the overloading. Once ES_C is larger than the threshold $Threshold_{MaxAverageLoad}$, the algorithm asserts that this cluster needs to add more instances. We also set up a threshold $Threshold_{Capacity}$ to limit the overall numbers of instances in all clusters. After new instances are launched, the algorithm waits T_launch minutes for the instances to fully booting up and starting to share the load. In the same way, when ES_C is smaller than the threshold $Threshold_{MinAverageLoad}$, the algorithm remove I_L instances so that the average load of each node in the cluster is approximately $k * Threshold_{MaxAverageLoad}$, where k is the ratio of the ideal load on a node to the $Threshold_{MaxAverageLoad}$.

7.1.2 Cluster State Synchronization

Cluster information consists of the number of nodes in the cluster, the IP addresses of each node, the partition ID of each nodes. Before the *Events Dispatchers* send out an event, they need to decide which node of the cluster is the destination of the event. A partitioning algorithm takes the

Algorithm 2: Instance Provisioning Algorithm

input : metrics of clusters

output: void

```
1  // I_L is the number of instances to add/remove
2  int I_L = 0;
3  foreach cluster C in cluster list do
4      // I_C is the current number of instances in cluster C
5      // Add instances
6      if ES_C > Threshold_MaxAverageLoad and I_C < Threshold_Capacity then
7          I_L = I_C * (ES_C - Threshold_MaxAverageLoad)/Threshold_MaxAverageLoad;
8      // Remove instances
9      if ES_C < Threshold_MinAverageLoad and I_C > 2 then
10         // k is the ratio of the ideal load on a node to the Threshold_MaxAverageLoad
11         I_L = I_C * (ES_C - Threshold_MaxAverageLoad)/(k * Threshold_MaxAverageLoad)
12 // T_launch and T_shutdown are the launching and shutting down time for an instance
13 if I_L > 0 then
14     add I_L instance for cluster C;
15     sleep T_launch minutes;
16 if I_L < 0 then
17     remove abs(I_L) instance for cluster C;
18     sleep T_shutdown minutes;
```

cluster information and the key of the event as input parameters. The output is the destination node associated with a unique ID that this event should be sent to. Then the *Events Dispatchers* are able to retrieve the destination IP address of the event. When autoscaling occurs, the changes of cluster information need to be synchronized to all clusters, otherwise events cannot be routed to the new instances.

In our architecture, *ZooKeeper* works as a coordinator and holds the information of all the clusters in a distributed stream processing system. Once *ZooKeeper* loses the connection with a node, the *Cluster Manager* deletes this node in related clusters and notifies the *Events Dispatchers* of all the clusters. Likewise, when a new instance is provisioned, *ZooKeeper* notifies all the connected nodes with the information of the new instance. Therefore, the *Events Dispatchers* updates the cluster information and sends events to the new instances based on the partitioning algorithm.

7.1.3 PE Relocation

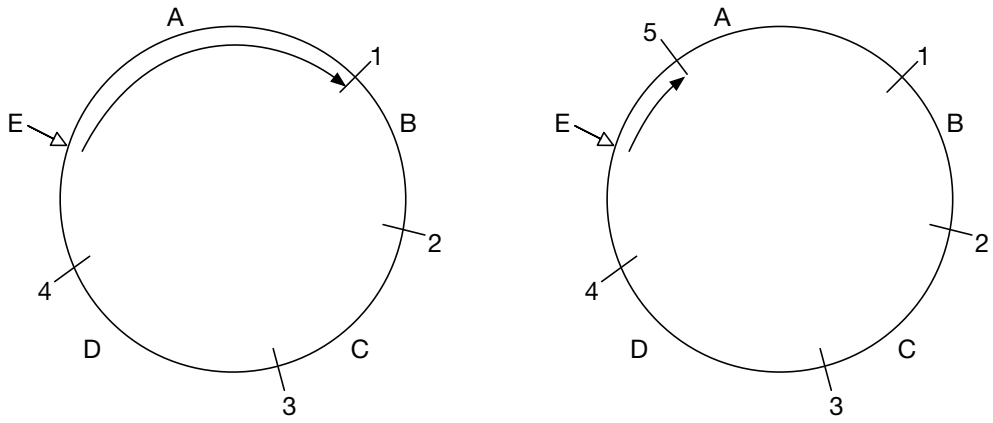
In a distributed stream processing system, PE instances contain and aggregate the states of events. Therefore, the events with the same event key are routed to the same work node. When the size of a cluster changes, some events are routed to the new destination node without the previous states of those events. To retain the previous states on the new node, a partitioning algorithm is designed to map events to nodes of clusters based on their keys. The goal of this partition algorithm is to evenly distribute events to all the nodes of a cluster so that the load is balanced among these nodes. Assume the events have K different keys and the cluster contains N nodes with parition IDs $0, 1, 2, ..., N$. One partitioning solution is to map an event with a *key* to a partition ID notated as PID is as follows:

$$PID = hash(key) \quad mod \quad N \tag{6}$$

This algorithm assumes that the events with the same key are distributed to the same node when N is constant. In this case, each node stores approximately K/N PE states. However, when nodes are added to or removed from a cluster, i.e. N is dynamic, a new partition ID can be generated to almost every key, which causes relocating all the corresponding PE states to the nodes with the new partition IDs. This may incur significant overhead of transferring PEs over the network and thus degrade any performance gain of autoscaling.

The PE relocation problem in a distributed stream processing system shares characters with the data migration problem in distributed storage systems whereby clusters are scaled out horizontally [Cat11]. *Consistent Hashing* [KLL+97, KSB+99] is widely used in distributed storage systems [LM10] for partitioning data evenly among storage nodes that are dynamically added to or removed from the cluster.

The *Consistent Hashing* algorithm hashes both keys and nodes using the same hash function, then marks the positions of their hashed values on a ring. The source code of class *HashRing* is attached in Appendix B. Each point on the ring stands for a hashed value. If the hash function returns an integer, the points on the ring will range from -2^{31} to $2^{31} - 1$. For example, keys A, B, C, D, E and nodes $1, 2, 3, 4$ are mapped on a ring as Figure 26a shows. To map a key to a node, the *Consistent Hashing* algorithm just locates the key on the ring as a starting point, then find the first node point in clockwise. Consider key E in Figure 26a as an example, the first node point is node 1, thus key E and A are mapped to node 1. Based on this logic, it can be inferred that only adjacent node is affected when nodes are added to or removed from a cluster. In Figure 26b, when a new node 5 is added to the cluster, the next node point of key E is node 5. According to the Consistent Hashing algorithm, PE relocation can be achieved by moving key E from node 1 to node 5. All other key-node mappings remain the same.



(a) Hash Ring with 4 Nodes

(b) Hash Ring with 5 Nodes

Figure 26: An Example of Consistent Hashing

To assure the number of keys on every node is evenly distributed so that the load are balanced in the cluster, the distance between the node points on the ring should be equal. An approximation approach is to hash each node multiple times to generate a series of points on the ring. Next, approximately K/N keys are mapped to each node. When a new node is added to the cluster, we can estimate $K/(N+1)$ keys need to be relocated. Thus the *Consistent Hashing* help reduce the number of PE states we need to transfer between work nodes when we add or remove nodes in a cluster.

7.2 Experiment and Evaluation

7.2.1 Environment Setup

The real-time top-N recommendation application in social streams is applied to evaluate the performance of stream processing architecture with autoscaling enabled. The objective is to compare the throughput in three options, namely original stream processing engine (SPE), DoDo with dynamic load balancing (DoDo+LB) and DoDo with autoscalingn (DoDo+AS).

The autoscaling mechanism in section 7.1 is implemented in DoDo. Algorithm 2 is integrated in *DFG Optimizer*, it is called by the DoDo daemon periodically. The top-N recommendation application contains one event generator and three PEs in the pipeline, namely *ExtractorPE*, *TopicCountAndReportPE* and *TopNTopicPE*.

Each event generator creates 50 tweets every second at the beginning, and increasingly generates 5 more tweets every 3 seconds. Thus each event generator is able to gradually increase the tweets

generation speed to 4,050 tweets per second in 40 minutes. The hashtag topics of tweets are randomly selected from a topic set, in which the occurrences of topics follow Zipfian distribution (with 5,000 elements and the exponent as 0.5) as Figure 22 shows.

The testbed is set up on Amazon Web Services (AWS) using EC2 instances with the type of *m1.small*. Each instance has 1 virtual CPU, 1.7GB memory, and 160GB storage. The operating system uses 64bit RHEL 6 with Linux kernel version 2.6.32.

The deployment of DoDo with autoscaling involves three types of nodes based on their usage, namely

1. Master node. It has the *DoDo Manager*, Graphite (a real-time metrics aggregation and visualization tool), and ZooKeeper installed;
2. Event generator node. It is used for generating tweets and send events to work nodes;
3. Worker node, where the PE instances are located.

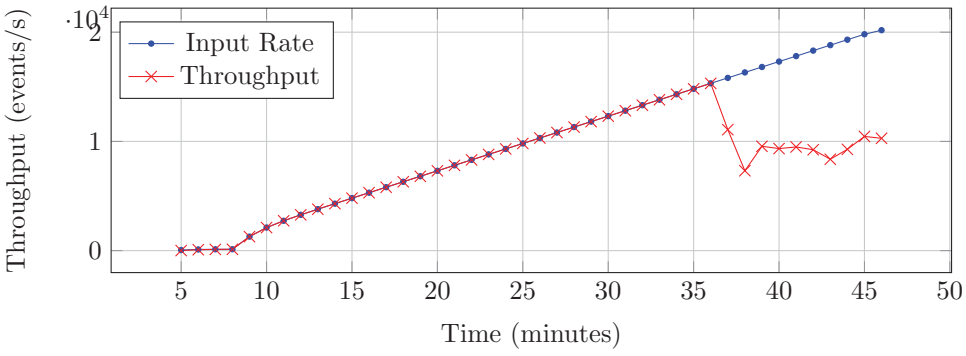
We test the system performance under three deployments of the application, including SPE, DoDo+LB and DoDo+AS. In each deployment, we have 4 clusters:

1. Cluster_1: event generator nodes.
2. Cluster_2: work nodes with *ExtractorPE*.
3. Cluster_3: work nodes with *TopicCountAndReportPE* and *TopNTopicPE*.
4. Cluster_4: master nodes.

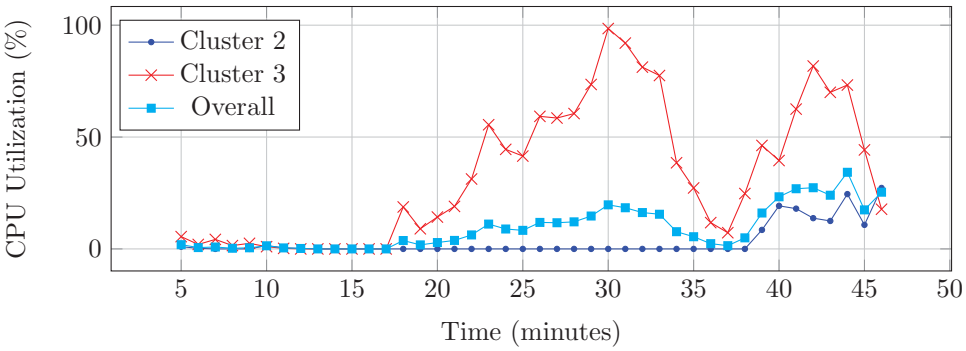
The settings of the experiments are as Table 9 shows. In Experiment 1, we use the original S4 platform, which has neither dynamic load balancing nor autoscaling. The testbed consists of 12 nodes in Cluster_2 and 3 nodes in Cluster_3.In Experiment 2, we use DoDo with dynamic load balancing only. Autoscaling is disabled. The settings of clusters are the same as Experiment 1. In Experiment 3, DoDo with autoscaling is enabled. The node setting contains 1 node in Cluster_2 and 1 node in Cluster_3 and the capacity is set to 15 for scaling out the clusters.

Table 9: Setting of Experiments

	Cluster_1	Cluster_2	Cluster_3	Cluster_4	Dynamic Balancing	Autoscaling
Experiment 1 (SPE)	5 nodes	12 nodes	3 nodes	1 node	No	No
Experiment 2 (DoDo+BL)	5 nodes	12 nodes	3 nodes	1 node	Yes	No
Experiment 3 (DoDo+AS)	5 nodes	1 node	1 node	1 node	Yes	Yes



(a) Throughput of S4



(b) Average CPU Utilization

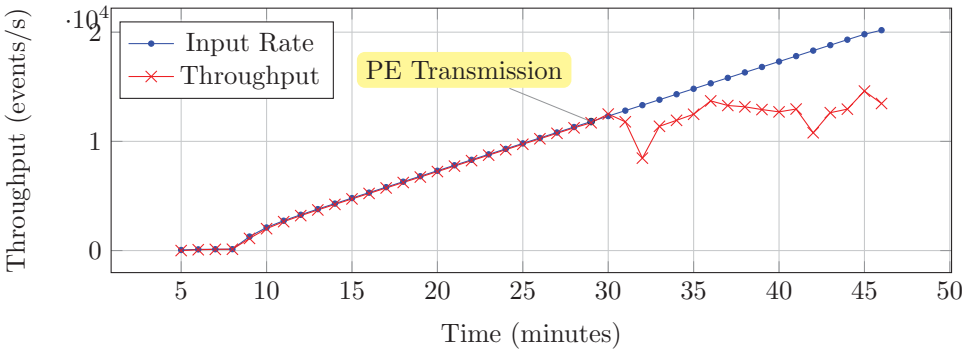
Figure 27: Experiment 1

7.3 Evaluation

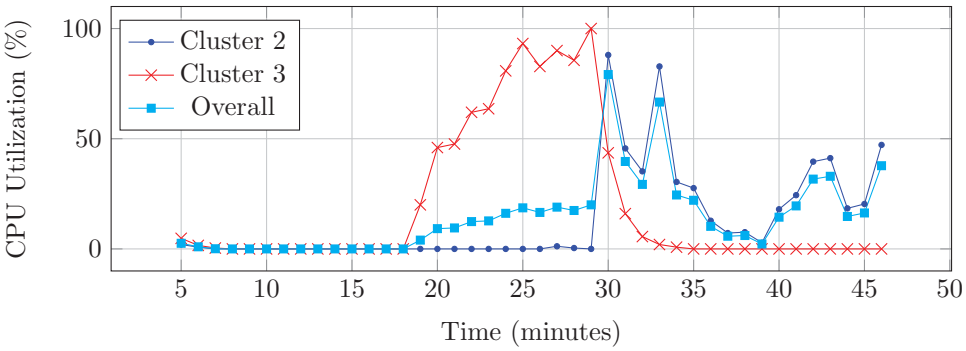
Figure 27(a) shows the throughput of Experiment 1. S4 performs well at the beginning. As the input rate reaches 15,315 events/s, the throughput of S4 starts decreasing and becomes steady at approximately 9,500 events/s. Figure 27(b) further shows that the CPU utilization of Cluster_3 reaches 100% at 30 minutes, when the input rate is at 12,313 events/s. This indicates that Cluster_3 is not capable of processing events at such a rate. Arriving events accumulated at the queue of the S4 stream processing engine. These events remain in the queue before they are processed, therefore the throughput of the whole system decreases.

Experiment 2 has the same setting as Experiment 1 except that DoDo provides dynamic load balancing, which enables transferring PE instances among clusters at runtime. As Figure 28(a) shows, the throughput fluctuates slightly after the PE transferring and eventually becomes steady at approximately 14,400 events/s. The throughput has been increased 51.58% comparing to Experiment 1.

Experiment 3 further enables the autoscaling feature. In Experiment 3, Cluster_2 and Cluster_3



(a) Throughput of S4 with DoDo

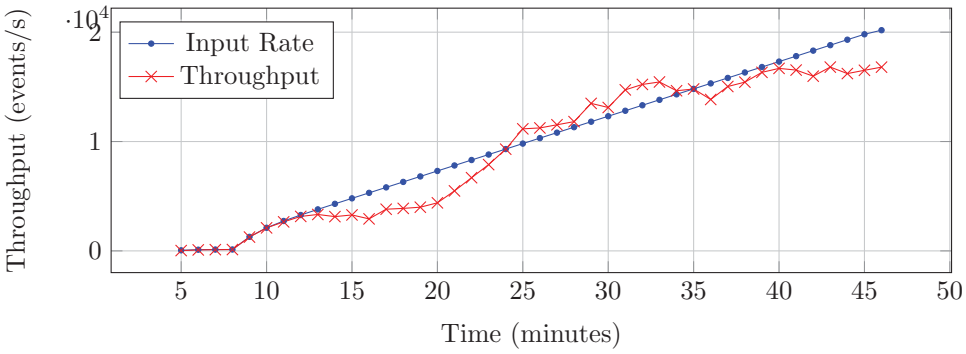


(b) Average CPU Utilization

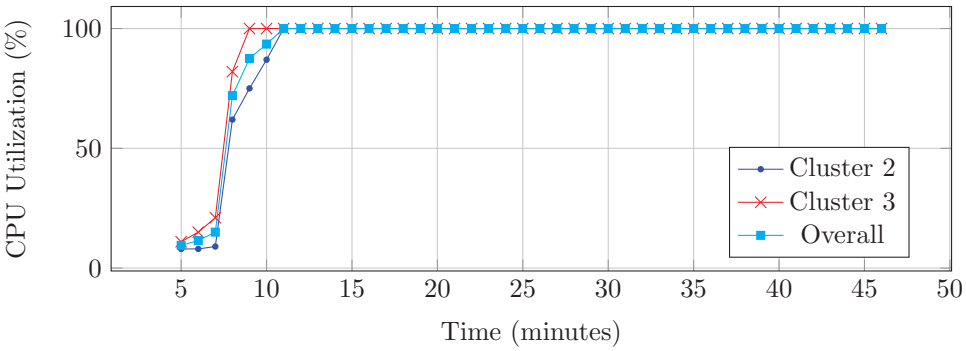
Figure 28: Experiment 2

are initiated with only one node. As Figure 29(b) shows, the CPU utilization increases to 100% rapidly on both the clusters. Meanwhile, Figure 30 shows that the clusters keep scaling out until they reach the capacity, which is set to 15. In the end, Cluster_2 has 6 nodes and Cluster_3 has 9 nodes. The throughput, as Figure 29(a) indicates, keeps increasing and becomes steady at approximately 16,700 events/s. The throughput increases approximately 75.79% compared to the throughput of Experiment 1. In addition, the load distribution among clusters is improved than other experiments because the overall CPU utilization is the highest.

Note that the throughput between 24 minutes and 34 minutes is even higher than the input rate. The reason is that each work node of S4 has an event queue and all received events are first put into the queue. Another thread keeps popping and processing events from the queue. When the work node is overloading, the speed of processing events can not catch up with the input stream rate so that more and more events are stuck in the queue and block the stream. When the cluster scales out, as some load are shed to the newly added nodes, the input rate on each work node drops. As a result, the work nodes are able to process not only the leftover events in the queue, but also the new generated events, thus achieves a throughput even higher than the input rate.



(a) Throughput of S4 with Autoscaling-DoDo



(b) Average CPU Utilization

Figure 29: Experiment 3

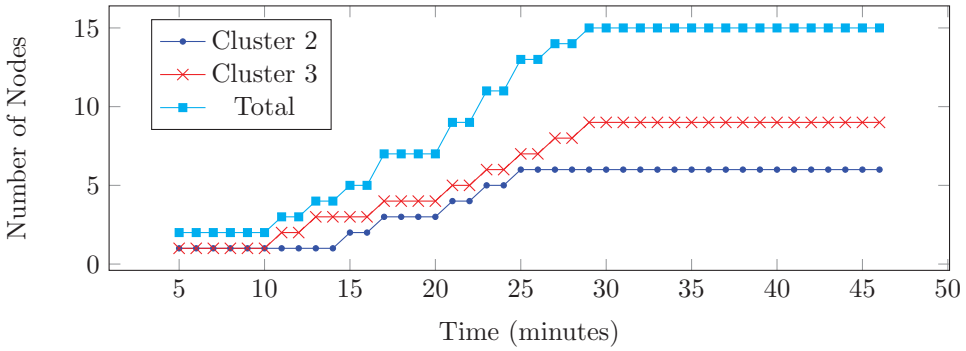
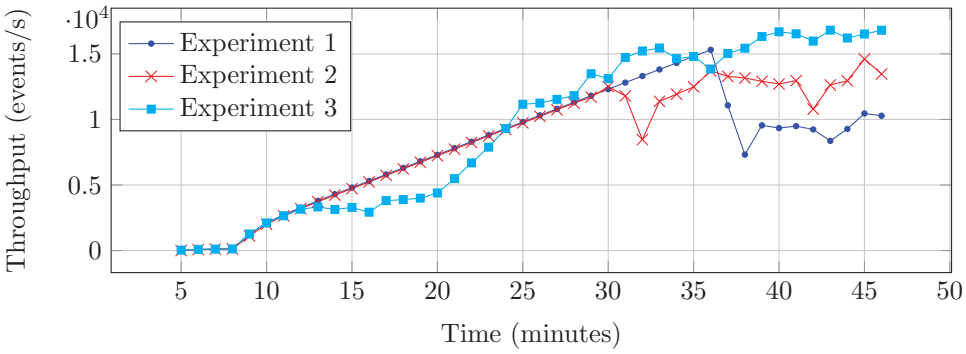
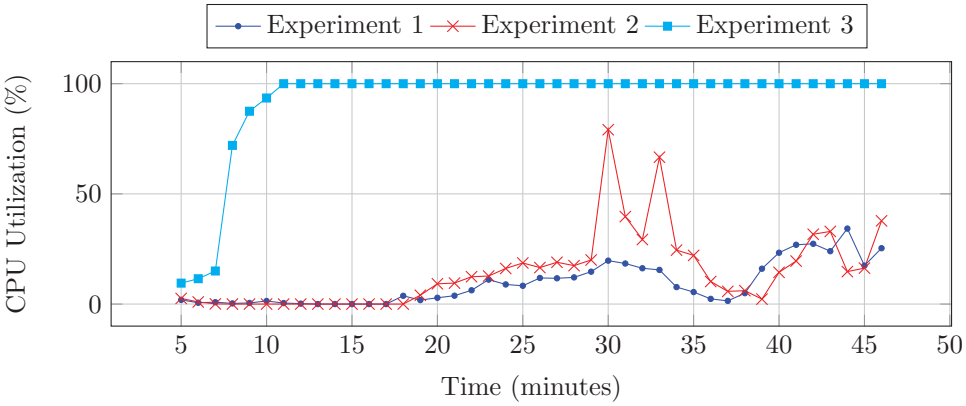


Figure 30: Autoscaling in Experiment 3

To further differentiate the scalability of these three experiment, the throughput and CPU utilization of the three experiments are plotted in In Figure 31. As Figure 31 shows, during 13 minutes to 24 minutes, Experiment 3's throughput is lower than the other two experiments. The main reason is that it takes time to launching instances, and the speed of provisioning instances is behind the



(a) Throughputs of Experiments



(b) Overall Average CPU Utilization of Experiments

Figure 31: Comparison of 3 Experiments

speed of the input rate. At the end, Experiment 3 has the highest throughput as well as highest utilization. The experiments demonstrate both dynamic load balancing and autoscaling architectures significantly expand the ability of the stream processing platform to scale under increasing workload and available computing resources.

Chapter 8

Conclusion

In this paper we discuss the difference between distributed batch-processing systems and DSP systems, compare their performance and cost on cloud infrastrucutre through a case study of a movie recommendation application implemented on Hadoop and S4. Our experiments demonstrates the advantages of DSP systems in analyzing data that is frequently updated. The experiments also shows that DSP systems require dynamic load balancing and autoscaling to deal with the unpredicted rate of the input stream.

To solve the problem of fluctuating input stream, we propose DoDo, a software layer that is load-adaptive between a DSP platform and physical cluster nodes managed by Zookeeper. It enables dynamic operator distribution (dynamic load balancing) with software components that are pluggable to existing DSP platforms. Dynamic load balancing is helpful to improve the throughput of DSP systems and the utilization of cluster resources. In some cases DSP systems may face overloads that need to autoscale cluster nodes on demand. So we also present an autoscaling mechanism under the DoDo architecture. We use ZooKeeper to synchronize the information of new nodes to all the existing PEs at runtime. To optimize the relocation of PEs when performing autoscaling, we use consistent hashing as the partitioning algorithm so that we can move as few PEs as possible.

With the DoDo architecture, we then design an optimization method for dynamic operator distribution of stream processing services and an overload dectection algorithm for autoscaling. The optimization algorithm takes both the correlations between the load series of clusters and the capacities of clusters into consideration. Through a case study we articulate the benefits of the optimization method on better utilizing available resources, which is an useful feature for cloud computing services, as it helps to prevent unnecessary autoscaling to clusters that are overloaded while other clusters still have capacities to handle streams. The overload detection algorithm can detect overload using the average load on nodes of each cluster and calculate how many instances

to provision. Our experiments demonstrates improved throughput and utilization of cloud resources using S4 with DoDo.

Our future work lies on more evaluations in different scenarios comparing with other algorithms, and further improve our algorithms when the data transferring overhead needs to be considered.

Bibliography

[AAB+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[ABPA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[Agh85] Gul Abdulnabi Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[Ama14] EC Amazon. Cloudwatch, 2014.

[apa14] Apache hadoop, 12 2014.

[ASST05] Gediminas Adomavicius, Ramesh Sankaranarayanan, Shahana Sen, and Alexander Tuzhilin. Incorporating contextual information in recommender systems using a multidimensional approach. *ACM Transactions on Information Systems (TOIS)*, 23(1):103–145, 2005.

[BHK98] John S Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pages 43–52. Morgan Kaufmann Publishers Inc., 1998.

[BM02] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1175–1220, 2002.

[Bre00] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[Bre12] Eric Brewer. Pushing the cap: Strategies for consistency and availability. *Computer*, 45(2):23–29, 2012.

[BTO13] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 15–26, New York, NY, USA, 2013. ACM.

[BWR+11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 7. ACM, 2011.

[Cat11] Rick Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[CC09] Rebecca L Collins and Luca P Carloni. Flexible filters: load balancing through backpressure for stream programs. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 205–214. ACM, 2009.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM, 1998.

[CKL+07] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.

[Cor13] Corvil. Nasdaq omx to implement corvil operational performance monitoring across u.s. trading platforms, June 2013.

[DADCB09] Marcos Dias De Assunçao, Alexandre Di Costanzo, and Rajkumar Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 141–150. ACM, 2009.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DK04] Mukund Deshpande and George Karypis. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems (TOIS)*, 22(1):143–177, 2004.

[DQRJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment*, 3(1-2):515–529, 2010.

[DRK06] Yannis Drougas, Thomas Repantis, and Vana Kalogeraki. Load balancing techniques for distributed stream processing applications in overlay environments. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 8–pp. IEEE, 2006.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[GA12] Buğra Gedik and Henrique Andrade. A model-based framework for building extensible, high performance stream processing middleware and programming language for ibm infosphere streams. *Software: Practice and Experience*, 42(11):1363–1391, 2012.

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

[GB99] Deepak Gupta and Pradip Bepari. Load sharing in distributed systems. In *Proceedings of the National Workshop on Distributed Computing*, 1999.

[GJPPM+12] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[GL12] Katharina Gorlach and Frank Leymann. Dynamic service provisioning for the cloud. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 555–561. IEEE, 2012.

[gra] Graphite - scalable realtime graphing.

[Gra03] Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.

[Gra08] Jim Gray. Distributed computing economics. *Queue*, 6(3):63–68, 2008.

[H+07] James R Hamilton et al. On designing and deploying internet-scale services. In *LISA*, volume 18, pages 1–18, 2007.

[HB11] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[Her11] Herodotos Herodotou. Hadoop performance models. *arXiv preprint arXiv:1106.0940*, 2011.

[HK11] Jesper Hoeksema and Spyros Kotoulas. High-performance distributed stream reasoning using s4. In *Ordring Workshop at ISWC*, 2011.

[HKBR99] Jonathan L Herlocker, Joseph A Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 230–237. ACM, 1999.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[HLL+10] Suhel Hammoud, Maozhen Li, Yang Liu, Nasullah Khalid Alham, and Zelong Liu. Mrsim: A discrete event based mapreduce simulator. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on*, volume 6, pages 2993–2997. IEEE, 2010.

[HM98] Fred Howell and Ross McNab. Simjava: A discrete event simulation library for java. *Simulation Series*, 30:51–56, 1998.

[HX98] Kai Hwang and Zhiwei Xu. *Scalable parallel computing: technology, architecture, programming*. McGraw-Hill, Inc., 1998.

[HY79] Alan R Hevner and S Bing Yao. Query processing in distributed database system. *Software Engineering, IEEE Transactions on*, (3):177–187, 1979.

[JCR11] Eaman Jahani, Michael J Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011.

[JK84] Matthias Jarke and Jurgen Koch. Query optimization in database systems. *ACM Computing surveys (CsUR)*, 16(2):111–152, 1984.

[Jos07] Nicolai M Josuttis. *SOA in practice: the art of distributed system design. "* O'Reilly Media, Inc.", 2007.

[KHP+09] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Buğra Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Middleware 2009*, pages 308–327. Springer, 2009.

[KK14] Matthias Keller and Holger Karl. Response time-optimized distributed cloud resource allocation. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, DCC '14, pages 47–52, New York, NY, USA, 2014. ACM.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

[Kor09] Richard E Korf. Multi-way number partitioning. In *IJCAI*, pages 538–543, 2009.

[KSB+99] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Computer Networks*, 31(11):1203–1213, 1999.

[LBCP09] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. Automated control in cloud computing: challenges and opportunities. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM, 2009.

[LHKK12] Simon Loesing, Martin Hentschel, Tim Kraska, and Donald Kossmann. Stormy: an elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pages 55–60. ACM, 2012.

[LLAH13] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. Hsim: a mapreduce simulator in enabling cloud computing. *Future Generation Computer Systems*, 29(1):300–308, 2013.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[LM13] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of sql and nosql databases. In *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, pages 15–19. IEEE, 2013.

[LSY03] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.

[MLH10] Ming Mao, Jie Li, and Marty Humphrey. Cloud auto-scaling with deadline and budget constraints. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*, pages 41–48. IEEE, 2010.

[MNG13] Nathan D. Mickulicz, Priya Narasimhan, and Rajeev Gandhi. To auto scale or not to auto scale. In *Presented as part of the 10th International Conference on Autonomic Computing*, pages 145–151, Berkeley, CA, 2013. USENIX.

[mon15] Mongodb, 3 2015.

[MPLC13] Fred Morstatter, Jurgen Pfeffer, Huan Liu, and Kathleen M Carley. Is the sample good enough? comparing data from twitter's streaming api with twitter's firehose. *Proceedings of ICWSM*, 2013.

[Mur09] AC Murthy. Mumak: Map-reduce simulator. *MAPREDUCE-728, Apache JIRA*, 2009.

[MyS95] AB MySQL. *MySQL: the world's most popular open source database*. MySQL AB, 1995.

[net] Netflix prize.

[NRNK10a] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, 2010.

[NRNK10b] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, volume 10, pages 1–15, 2010.

[PSZ+07] Pradeep Padala, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 289–302. ACM, 2007.

[R+11] Philip Russom et al. Big data analytics. *TDWI Best Practices Report, Fourth Quarter*, 2011.

[RDG11] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507. IEEE, 2011.

[RIS+94] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994.

[s4] S4: Distributed stream computing platform.

[SC11] Michael Stonebraker and Rick Cattell. 10 rules for scalable performance in'simple operation'datastores. *Communications of the ACM*, 54(6):72–80, 2011.

[SGR00] Trevor Schroeder, Steve Goddard, and Byrov Ramamurthy. Scalable web server clustering technologies. *Network, IEEE*, 14(3):38–45, 2000.

[SHLD11] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 348–355. IEEE, 2011.

[SKH95] Behrooz A Shirazi, Krishna M Kavi, and Ali R Hurson. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Society Press, 1995.

[SKKR01] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[SMH+13] Ge Song, Zide Meng, Fabrice Huet, Frederic Magoules, Lei Yu, and Xuelian Lin. A hadoop mapreduce performance prediction method. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 820–825. IEEE, 2013.

[SZT12] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High performance MySQL: Optimization, backups, and replication*. " O'Reilly Media, Inc.", 2012.

[UKOVH09] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank Van Harmelen. *Scalable distributed reasoning using mapreduce*. Springer, 2009.

[USCG05] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, and Pawan Goyal. Dynamic provisioning of multi-tier internet applications. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 217–228. IEEE, 2005.

[VCC11] Abhishek Verma, Ludmila Cherkasova, and Roy H Campbell. Play it again, simmr! In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 253–261. IEEE, 2011.

[vdVvdWM12] Jan Sipke van der Veen, Bram van der Waaij, and Robert J Meijer. Sensor data storage performance: Sql or nosql, physical or virtual. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 431–438. IEEE, 2012.

[WBPG09] Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[WL14a] Xing Wu and Yan Liu. Enabling a load adaptive distributed stream processing platform on synchronized clusters. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, 2014.

[WL14b] Xing Wu and Yan Liu. Optimization of load adaptive distributed stream processing services. In *Services Computing (SCC), 2014 IEEE International Conference on*, pages 504–511. IEEE, 2014.

[WLG15a] Xing Wu, Yan Liu, and Ian Gorton. Scalability and cost evaluation of incremental data processing using amazon's hadoop service. In Kuan-Ching Li, Hai Jiang, Laurence T. Yang, and Alfredo Cuzzocrea, editors, *Big Data: Algorithms, Analytics, and Applications*, chapter 2, pages 21–38. Chapman and Hall/CRC, 2015.

[WLG15b] Xing Wu, Yan Liu, and Ian Gorton. Software qos enhancement through self-adaptation and formal models. In *Proceedings of the 11th international ACM Sigsoft conference on Quality of software architectures*. ACM, 2015.

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[XCL14] Zhen Xiao, Qi Chen, and Haipeng Luo. Automatic scaling of internet applications for cloud computing services. *Computers, IEEE Transactions on*, 63(5):1111–1123, 2014.

[XZH05] Ying Xing, Stan Zdonik, and J-H Hwang. Dynamic load distribution in the borealis stream processor. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 791–802. IEEE, 2005.

[YCY+13] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Performance Evaluation Review*, 40(4):23–32, 2013.

[ZMP11] Jilian Zhang, Kyriakos Mouratidis, and HweeHwa Pang. Heuristic algorithms for balanced multi-way number partitioning. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 693–698. AAAI Press, 2011.

Appendix A

MapReduce Source Code for Movie Recommendation

The python source code of the mapper in Round 1.

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)

for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # split the line into words
    words = line.split()

    # write the results to STDOUT (standard output);
    if len(words) < 3:
        continue
    elif int(words[2]) > 3:
        print '%s\t%s' % (words[0], words[1])
```

The python source code of the reducer in Round 1.

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    print line.strip()
```

The python source code of the mapper in Round 2.

```
#!/usr/bin/env python

import sys

movie_ids = list()
user_id = 0

def print_id_pairs(id_list):
    for i in range(0, len(id_list)-1):
        for j in range(i+1, len(id_list)):
            print '%s-%s\t1' % (id_list[i], id_list[j])

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split('\t')
    # write the results to STDOUT (standard output);
    if len(words) < 2:
        continue
    if user_id != words[0] and user_id != 0:
        print_id_pairs(movie_ids)
        del movie_ids[:]
```

```
    user_id = words[0]
    movie_ids.append(words[1])

print_id_pairs(movie_ids)
```

The python source code of the reducer in Round 2.

```
#!/usr/bin/env python

import sys

movie_pair_cnt = 0
movie_pair = 0

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split('\t')
    # write the results to STDOUT (standard output);
    if len(words) < 2:
        continue
    if movie_pair != words[0] and movie_pair != 0:
        print '%s\t%d' % (movie_pair, movie_pair_cnt)
        movie_pair_cnt = 0
    movie_pair = words[0]
    movie_pair_cnt += int(words[1])


print '%s\t%d' % (movie_pair, movie_pair_cnt)
```

The python source code of the mapper in Round 3.

```
#!/usr/bin/env python
```

```
import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # write the results to STDOUT (standard output);
    if len(words) < 2:
        continue
    ids = words[0].split('-')
    if len(ids) == 2:
        print '%s\t%s\t%s' % (ids[0], ids[1], words[1])
        print '%s\t%s\t%s' % (ids[1], ids[0], words[1])
```

The python source code of the reducer in Round 3.

```
#!/usr/bin/env python

import sys

top_lists = []
movie_id = 0

def print_top_lists(m_id, top_list):
    out_str = '%s\t' % m_id
    for i, data in enumerate(sorted(top_list, key=lambda m:m[1], reverse=True)[:10]):
        if i != 0:
            out_str += ','
        out_str += '%s-%d' % (data[0], data[1])
    print out_str
```

```
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split('\t')
    # write the results to STDOUT (standard output);
    if len(words) < 3:
        continue
    if movie_id != words[0] and movie_id != 0:
        print_top_lists(movie_id, top_lists)
        del top_lists[:]
    movie_id = words[0]
    top_lists.append((words[1], int(words[2])))

print_top_lists(movie_id, top_lists)
```

Appendix B

Consistent Hashing

The java source code of the class `HashRing` is as below. Please visit https://github.com/xing5/DynamicS4/ to access the source code of this research project.

```
/**
 * A hash ring is used to map resources to a to a set of nodes.
 * This hash ring implements
 * <a href="http://www8.org/w8-papers/2a-webserver/caching/paper2.html#chash1">
 * Consistent Hashing</a> and therefore adding a removing nodes minimally
 * changes how resources are map to the nodes.
 * <p/>
 * This implementation also allows you to apply non-uniform node weighting.  This
 * feature is usefull when you want to allocate more resources to some nodes and
 * fewer to others.
 * <p/>
 * The default weight of node is 200.  The weight of a node determins how many
 * points on the hash ring the node is alocated. Higher node weights increases
 * the uniform distribution of resources.
 * <p/>
 * Note that the order that nodes are added to the ring impact how resources
 * map to the nodes due to node hash collisions.
 *
 */
public class HashRing<Node, Resource> {
```

```
private static final Logger logger = LoggerFactory.getLogger(HashRing.class);

public static int DEFAULT_WEIGHT = 200;

private static class Wrapper<N> {
    private N node;
    private int weight;

    public Wrapper(N node, int weight) {
        this.node = node;
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "Wrapper{" + "node=" + node + ", weight=" + weight + '}';
    }
}

public long hash(String hashKey) {
  byte[] digest = null;
  try {
  MessageDigest md = MessageDigest.getInstance("SHA-256");
  md.update(hashKey.getBytes("UTF-8"));
  digest = md.digest();
} catch (Exception e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
}

    int b = 378551;
    int a = 63689;
    long hash = 0;

    for (int i = 0; i < digest.length; i++) {
```

```
        hash = hash * a + (int)digest[i];
        a = a * b;
    }

    return Math.abs(hash);
}

//@Inject
//private Hasher hasher;

private final TreeMap<Integer, Wrapper<Node>> ring = new TreeMap<Integer, Wrapper<Node>>();
private final LinkedHashMap<Node, Wrapper<Node>> nodes = new LinkedHashMap<Node, Wrapper<Node

/**
 * Constructs a <tt>HashRing</tt> which uses the OBJECT_HASHER to hash the nodes and values.
 *
 */
public HashRing() {
}



/**
 * Adds all the specified nodes to the <tt>HashRing</tt> using the default
 * weight of 200 for each node.
 *
 * @param nodes the nodes to add
 */
public void addAll(Iterable<Node> nodes) {
    for (Node node : nodes) {
        add(node);
    }
}

/**
 * Adds all the specified nodes to the <tt>HashRing</tt> using the default
```

```
 * weight of 200 for each node.
 *
 * @param nodes the nodes to add
 */
public void add(Node... nodes) {
    addAll(Arrays.asList(nodes));
}

/**
 * Adds a node to the <tt>HashRing</tt> using the default
 * weight of 200 for the node.
 *
 * @param node the node to add
 */
public void add(Node node) {
    add(node, DEFAULT_WEIGHT);
}

/**
 * Adds a node to the <tt>HashRing</tt> using the specified weight.
 *
 * @param node the node to add
 * @param weight the number of hash replicas to create the node in the <tt>HashRing</tt>
 * @throws IllegalArgumentException if the weight is less than 1
 */
public void add(Node node, int weight) {
    if( weight < 1 ) {
        throw new IllegalArgumentException("weight must be 1 or greater");
    }

    Wrapper<Node> wrapper = new Wrapper<Node>(node, weight);
    logger.error("HashRing add node: " + node.toString());
    nodes.put(node, wrapper);
    for (int i = 0; i < wrapper.weight; i++) {
      int index = (int) hash(node.toString() + i);
```

```
      //logger.debug("lvl[" + i +"] index["+index+"] + str(" + node.toString() +")");
        ring.put((int)hash(node.toString() + i), wrapper);
    }
}

/**
 * Removes a previously added node from the <tt>HashRing</tt>
 *
 * @param node the node to remove
 * @return true if the node was previously added
 */
public boolean remove(Node node) {
    Wrapper<Node> wrapper  = nodes.remove(node);
    if( wrapper == null ) {
        return false;
    }

    // We HAVE to re-hash the ring to keep it consistent since
    // nodes hashes may collide and last node added takes over the
    // the previously added node.  Order matters.
    ring.clear();
    for (Wrapper<Node> w : nodes.values()) {
        for (int i = 0; i < w.weight; i++) {
            ring.put((int)hash(w.node.toString() + i), w);
        }
    }
    return true;
}

/**
 * Removes all previously added nodes.
 */
public void clear() {
    ring.clear();
    nodes.clear();
```

```
}

/**
 * @return all the previously added nodes.
 */
public List<Node> getNodes() {
    return new ArrayList(nodes.keySet());
}

/**
 * Maps a resource value to a node.
 *
 * @param resource the resource to map
 * @return the Node that the resource maps to or null if the <tt>HashRing</tt> is empty.
 */
public Node get(Resource resource) {
    Map.Entry<Integer, Wrapper<Node>> entry = getFirstEntry(resource);
    if (entry==null) {
        return null;
    }
    return entry.getValue().node;
}

/**
 * Maps a resource value to an interator to the nodes in the <tt>HashRing</tt>
 * starting at the Node which resource maps to.
 *
 * Note that duplicate node objects may be returned.  This is because
 *
 *
 *
 * @param resource the resource to map
 * @return a Iterator
 */
public Iterator<Node> iterator(Resource resource) {
```

```
final Map.Entry<Integer, Wrapper<Node>> first = getFirstEntry(resource);

return new Iterator<Node>() {
    Map.Entry<Integer, Wrapper<Node>> removealCandidate;
    Map.Entry<Integer, Wrapper<Node>> last;
    Map.Entry<Integer, Wrapper<Node>> next = first;

    public boolean hasNext() {
        // We might allready know the next entry..
        if( next != null )
            return true;

        // Since we use last to figure out the next..
        if( last==null )
            return false;

        // Figure out the enxt entry...
        Map.Entry<Integer, Wrapper<Node>> next = ring.higherEntry(last.getKey());
        if( next == null ) {
            next = ring.firstEntry();
        }

        // We don't need last anymore..
        last = null;

        // But the next entry might circle back to the first...
        if( next.getKey()==first.getKey() ) {
            next = null;
        }
        return next!=null;
    }

    public Node next() {
        if( !hasNext() ) {
            throw new NoSuchElementException();
```

```
                }
                removealCandidate = last = next;
                next = null;
                return last.getValue().node;
            }

            public void remove() {
                if( removealCandidate ==null ) {
                    throw new IllegalStateException();
                }
                HashRing.this.remove(last.getValue().node);
                removealCandidate =null;
            }
        };
    }

    private Map.Entry<Integer, Wrapper<Node>> getFirstEntry(Resource resource) {
        if (ring.isEmpty()) {
            return null;
        }
        int hash = (int) hash(resource.toString());
        logger.debug(String.format("Key[%s] hash[%d]", resource.toString(), hash));
        Map.Entry<Integer, Wrapper<Node>> entry = ring.ceilingEntry(hash);
        if( entry == null ) {
            entry = ring.firstEntry();
        }
        return entry;
    }


}
```