Workload Management for Data-Intensive Services

by

Harold Vinson Chao Lim

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Shivnath Babu, Co-supervisor

_____
Jeffrey S. Chase, Co-supervisor

_____
Landon P. Cox

_____
Anirban Mandal

_____
Sujay S. Parekh

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2013

Abstract

Workload Management for Data-Intensive Services

by

Harold Vinson Chao Lim

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Shivnath Babu, Co-supervisor

_____
Jeffrey S. Chase, Co-supervisor

_____
Landon P. Cox

_____
Anirban Mandal

_____
Sujay S. Parekh

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2013

Copyright © 2013 by Harold Vinson Chao Lim
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

Data-intensive web services are typically composed of three tiers: i) a display tier that interacts with users and serves rich content to them, ii) a storage tier that stores the user-generated or machine-generated data used to create this content, and iii) an analytics tier that runs data analysis tasks in order to create and optimize new content. Each tier has different workloads and requirements that result in a diverse set of systems being used in modern data-intensive web services.

Servers are provisioned dynamically in the display tier to ensure that interactive client requests are served as per the latency and throughput requirements. The challenge is not only deciding automatically how many servers to provision but also when to provision them, while ensuring stable system performance and high resource utilization. To address these challenges, we have developed a new control policy for provisioning resources dynamically in coarse-grained units (e.g., adding or removing servers or virtual machines in cloud platforms). Our new policy, called proportional thresholding, converts a user-specified performance target value into a target range in order to account for the relative effect of provisioning a server on the overall workload performance.

The storage tier is similar to the display tier in some respects, but poses the additional challenge of needing redistribution of stored data when new storage nodes are added or removed. Thus, there will be some delay before the effects of changing a resource allocation will appear. Moreover, redistributing data can cause some inter-

ference to the current workload because it uses resources that can otherwise be used for processing requests. We have developed a system, called Elastore, that addresses the new challenges found in the storage tier. Elastore not only coordinates resource allocation and data redistribution to preserve stability during dynamic resource provisioning, but it also finds the best tradeoff between workload interference and data redistribution time.

The workload in the analytics tier consists of data-parallel workflows that can either be run in a batch fashion or continuously as new data becomes available. Each workflow is composed of smaller units that have producer-consumer relationships based on data. These workflows are often generated from declarative specifications in languages like SQL, so there is a need for a cost-based optimizer that can generate an efficient execution plan for a given workflow. There are a number of challenges when building a cost-based optimizer for data-parallel workflows, which includes characterizing the large execution plan space, developing cost models to estimate the execution costs, and efficiently searching for the best execution plan. We have built two cost-based optimizers: Stubby for batch data-parallel workflows running on MapReduce systems, and Cyclops for continuous data-parallel workflows where the choice of execution system is made a part of the execution plan space.

We have conducted a comprehensive evaluation that shows the effectiveness of each tier's automated workload management solution.

To my wife, Jacqueline, and family.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I thank God for all the blessings I have received in life.

I am extremely thankful to my beloved wife, Jacqueline. Her unwavering support, patience, understanding, and encouragement made this whole journey much more meaningful.

I owe a lot to my co-advisors, Jeffrey S. Chase and Shivnath Babu. I am really grateful to Jeff for taking me under his wings and introduce me to systems research, at a time when I was lost and had grown disinterested in doing research in computer vision. I appreciate the regular walks we had across campus to the Duke Gardens, while discussing a wide range of things and is always reminded to look at the bigger picture. I am really thankful to Shivnath for guiding me throughout my PhD life. Specifically, I thank him for allowing me to be part of the Starfish project and encouraging me to work on the wonderful field of data management. His unending source of energy has really pushed me to succeed.

I would also like to extend my gratitude to the rest of my committee members: Landon P. Cox for opening my eyes to mobile systems and privacy research by allowing me to be part of the Vis-à-Vis project, Sujay S. Parekh for sharing his expertise on control systems, and Anirban Mandal for the valuable discussions we had.

I would like to especially thank Aman Kansal, Ramakrishna Kotla, Jie Liu, Venugopalan Ramasubramanian for giving me the opportunity to work at Microsoft Re-

search as a summer intern for two summers. Aman and Jie introduced me to the problem of power management of data centers. Our solution resulted in a nice publication and a pick of the month selection by the IEEE STC on Sustainable Computing. Rama and Rama introduced me to experimental analysis research on interference in the cloud. The two summer internships really helped me understand the other exciting research problems out there and gave me two meaningful breaks from my doctoral research work.

I would like to thank all of the other people I have interacted and collaborated with: Amre, Hero, Ned, Rishi, and Vamsi. I am sure I am missing many more people in this list.

Last, but definitely not the least, I want to thank my family. Thanks to my mother Vicky, my late father Harry, my twin sister Debbie, my oldest sister Candice, my auntie Tessie, and my uncle Victor. Thank you for being very supportive of me throughout my doctoral studies.

1

Introduction

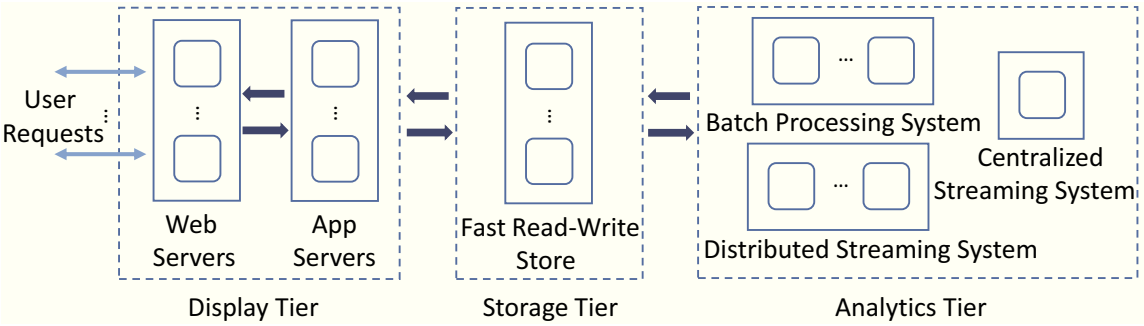


FIGURE 1.1: A general high-level system architecture for data-intensive services.

It is now common for modern data-intensive services to handle heterogeneous workloads [32, 41, 67, 83]. For instance, to support their Web services, Yahoo! [107] has both online workloads for serving web pages and tracking user activities and batch workloads for analysis of user activity and generating models and pages. In a recent paper, HP Labs describes workloads for next generation business intelligence services [109]. An example workload is performing sentiment analysis on Twitter [121] tweet feeds, which involves combining structured and unstructured historical data with streaming data. Similarly, LeFevre et al. [71] describe data-intensive services

that involve both relational and non-relational processing workloads, in the context of restaurants (e.g., identifying prior active restaurant patrons).

As has been reported by a number of related work [73, 112], using a single system to support all these workloads is a bad design practice because it can cause significant performance problems. Rather, a careful mix and match of appropriate systems is needed to support diverse workloads as required by the data-intensive service, which is evident by the complex systems used in production by large enterprises [4, 41, 84]. Figure 1.1 shows a general high-level system architecture that captures most data-intensive services. The architecture can be described as having a tiered architecture, with each tier focused on specific types of workloads and functionalities. We categorize the systems that comprise data-intensive services into the following tiers: *display tier*, *storage tier*, and *analytics tier*.

Example Data-Intensive Service: To better understand the system architecture and the diverse workloads handled by data-intensive services, let us consider a targeted advertisement service of a social networking website as a concrete example. The goal of this service is to give advertisement providers the ability to customize the specific subset of consumers that has access to a particular advertisement.

The display tier is the main entry point for interacting with the social networking website. In this context, the display tier has two types of users: advertisement providers, and regular users (consumers) of the social network website. Advertisement providers have access to a dashboard interface where they can set up their advertisement campaigns, such as configuring the behavioral traits of their target consumer and setting up the content of the advertisement, and also monitor the progress and statistics of their campaigns, such as showing real-time click-through rates of specific advertisements. Through the display tier, regular users socialize with other users, such as sharing/uploading contents, posting messages, and viewing other user's activities. Moreover, targeted advertisements are displayed to the users, based

on analysis and workloads found in the other tiers. Regardless of the type of users, the workload in this tier can be generalized into simple get and put requests. As Figure 1.1 shows, stateless systems, such as clustered Web and Application servers, are typically used in this tier. Data, such as user activities and complex models used for determining the advertisement to display to each user, are routed to and from the other tiers. Management of the workload in this tier entails ensuring that user requests are processed at an acceptable latency (e.g., the response time of the dashboard interface should be less than 500ms). Performance problems usually result from the inability to handle the amount of workload (e.g., the number of concurrent requests) due to over-utilization of resources allocated to systems. Moreover, these services frequently experience rapid load surges and drops. Nevertheless, there is a growing trend of deploying data-intensive services in the cloud [101], which gives an opportunity for managing the workload of the display tier by dynamically controlling the amount of cloud resources provisioned for the underlying systems. At the same time, the cloud also presents new challenges for designing policies for dynamic resource provisioning, such as limited access to physical-level measurements and exposure to only coarse-grained resources (virtual machine instances).

As the name suggests, the storage tier stores and serves the data generated and requested by the other tiers. In the simplest case, when a regular user interacts with the display tier, the contents (data) they generate are stored in a fast read-write store, such as a distributed key-value storage system, found in the storage tier. When another user wants to view these contents, the display tier requests them from the storage tier. In a more complex case, the storage tier acts as an intermediary between the display and analytics tier (details of the analytics tier will be discussed in the succeeding paragraph). For instance, displaying the correct set of advertisements to users entails first continuously collecting and storing user activities in the storage tier. These activities are then periodically processed in the analytics tier to create

behavioral models of users, which are also stored in the storage tier. These models are matched with advertisement campaigns with similar configured behavioral traits. Using this information, the display tier can then request from the storage tier the appropriate advertisements to display to each user. Similarly, an advertisement provider may want to view or monitor the status of her advertisement campaigns, such as click-through rate. As part of the user activities collected and stored in the storage tier, information regarding clicked advertisements can also be collected. The analytics tier can then perform temporal windowed aggregation (e.g., aggregate the data every second grouped by advertisement provider) on this data and store the results back in the storage tier, which can then be served to the display tier. Since the storage tier is only responsible for storing and serving data, its workload can be characterized as create, read, update and delete (CRUD) operations. Like in the display tier, there is an opportunity for dynamically provisioning resources of storage system, such that the latency and/or throughput for processing CRUD operations are within an acceptable level. However, the statefulness of systems found in the storage tier presents new challenges, such as storage nodes requiring data before it can improve read performance (i.e., newly instantiated storage nodes do not have any data to serve).

In contrast to the other tiers, the analytics tier handles more complex analytical workloads that can be described as data-parallel workflows composed of units with producer-consumer relationships based on data that can be run in batch fashion or continuous fashion. When run in batch fashion, the units are specified as a data-parallel MapReduce computation. On the other hand, when run in continuous fashion, the units are specified as a data parallel *windowed* MapReduce computation. Consider the two already mentioned workload examples for this tier: building user behavioral models and generating campaign statistics. Building user behavioral models involve multiple MapReduce computations that can range from performing

extract-transform-load (ETL) operations to performing statistical machine-learning algorithms [24]. Likewise, generating campaign statistics can also involve multiple units of computations, such as filtering, grouping, joining, and aggregation of data. However, this workload can be categorized as a continuous data-parallel workflow, where each computation is performed on a temporal window of data, such that the campaign statistics are continuously updated ranging from short (e.g., seconds) to long (e.g., days) time intervals. In this tier, there is an opportunity for management through automatic optimization for finding the best strategy to run the workload. This includes not only selecting the most suitable system configuration settings, but also transforming the units of the workflow into a logically equivalent (i.e., generates the same results), but more efficient workflow. For example, generating user behavioral models may require joining multiple datasets, which normally takes two units of MapReduce computations in a batch processing system, such as Hadoop [47]. It is possible to perform this operation in a single MapReduce computation by controlling how the dataset is partitioned (e.g., performing merge-join). However, the space of possible transformations can be large and high-dimensional. Moreover given that a number of the systems found in the analytics tier allows developers to create the workload using different interfaces, there is a challenge for identifying and enumerating valid strategies with minimal (and possibly missing) information. For instance, schema and partitioning information may not be known.

In addition, notice that in Figure 1.1, different types of systems are found in the analytics tier: batch processing system, distributed streaming system, and centralized streaming system. This setting is not uncommon and is in fact, widely used in practice by enterprises [41, 84]. Different specialized systems are used to handle the different behaviors of analytical workloads processed in the analytics tier. For example, batch processing systems can be used to build user behavioral models because it requires processing large scale data (e.g., there can be millions of users). On

the other hand, a centralized streaming system can be used for generating campaign statistics with short time intervals, which requires a system specifically designed for low latency requirements. However, choosing the most suitable system for a particular workload require considerable effort and the choice is not as clear-cut as we have described in our examples. Thus, there is an opportunity for a multi-system workload management solution that automatically selects the strategy to run a workload, which includes selecting the most suitable system, that minimizes the completion time.

As the example of a targeted advertisement service has shown, although the complexity and the number of systems that comprise data-intensive services requires careful management of the workloads, there is an opportunity for an automated approach to managing workloads. This dissertation presents automated workload management solutions for data-intensive services by developing policies and mechanisms for various systems of data-intensive services that leverage technologies from cloud computing and principles from control theory, and cost-based optimization.

1.1 Contributions

The focus of this dissertation is to study the systems and workloads of modern data-intensive services. The hypothesis is that with the right sets of policies and mechanisms, complex systems can be managed automatically to ensure data-intensive services are running at an acceptable level of performance. In validating this hypothesis, this dissertation makes the following contributions:

1. **Introduce a new cluster provisioning policy called *Proportional Thresholding*, which is a policy enhancement for feedback controllers that enables stable control across a wide range of cluster sizes using the coarse-grained control offered by popular virtual compute cloud ser-**

vices. Instead of using a single performance target value, this policy uses a dynamic range of performance targets to deal with the discrete nature of actuators. We show how proportional thresholding results in less oscillations compared to traditional control policies, such as integral control.

2. **Design and implement a new control architecture for dynamic provisioning of stateful systems, such as key-value storage.** The architecture coordinates a *Horizontal Scale Controller*, responsible for scaling the cluster, and a *Data Rebalance Controller*, responsible for moving data between nodes, through the use of a state machine. It is able to preserve stability during system adjustments, such as adding new nodes, or moving data between nodes, by taking into account actuator delays, interference, and mutual dependence of the controllers.

3. **Introduce the large optimization plan space for executing batch data-parallel workflows processed in the analytics tier.** The optimization plan space can be defined by the transformations that can be applied to a data-parallel workflow of jobs. A *transformation* is defined by a set of preconditions and postconditions: If the preconditions hold on a plan P^-, then we can generate a plan P^+ on which the postconditions hold such that P^- and P^+ will produce the same result. We categorize the transformations into: (i) intra-job vertical packing transformation, (ii) inter-job vertical packing transformation, (iii) horizontal packing transformation, (iv) partition function transformation, and (v) configuration transformation.

4. **Introduce optimization opportunities that leverage on the information exposed by structured interfaces for expressing batch data-parallel workflows.** This information can be extracted automatically and

do not require knowledge of the functionality of the units of computations of the data-parallel workflow. The lineage of data is used as conditions for consolidating units and eliminating the need for some phases of the workflow, such as sorting, and shuffling of data.

5. **Design and implement techniques for automatically optimizing batch data-parallel workflows.** To efficiently optimize a workflow, we divide the workflow into optimization units generated around workflow jobs with dependent input and derived datasets. Moreover, we use a cost-based approach and a recursive random search strategy to efficiently find the optimal settings. We show that our optimizer is able to achieve 5X performance speedup, while only incurring at most 12.8% overhead.

6. **Characterize and optimize the execution plan space of continuous data-parallel workflows.** As mentioned previously, a majority of analysis over activity and operational data involves continuous data-parallel workflows. A common, but nontrivial, class of this workload that is at the heart of many data-intensive services is the windowed aggregation query, which performs a continuous time-based windowing operation on a data stream, and then performs grouping and aggregation on each window. A central aspect of our characterization is the interplay between different logical query processing algorithms and the properties of different systems that are capable of continuous query execution (refer to the systems in the analytics tier of Figure 1.1). We bring out the tradeoffs among different execution plans through a detailed empirical evaluation and use this information to develop a cost-based optimizer that can pick a good plan from this space.

1.2 Thesis Outline

This thesis is structured in such a way that each chapter focuses on a specific tier and subset of systems shown in Figure 1.1. In Chapters 2 and 3, we focus on workload management of systems that are responsible for storing, displaying, and returning final contents to users. The approach we take is to first design and build a control system for provisioning resources from the cloud (i.e., virtual machine instances) for stateless systems, such as Web servers found in the display tier, that dynamically adapts to change in workload (Chapter 2). Then in Chapter 3, we not only describe in detail the new challenges of provisioning resources for stateful applications, such as key-value storage systems, that are typically found in the storage tier of data-intensive services, but also describe the design and implementation of *Elastore*, an automated control system for elastic storage systems. Chapters 4 and 5 describes the management of two types of workloads in the analytics tier. In Chapter 4, we focus on distributed batch processing systems that are used in the analytics tier. We present the design and implementation of *Stubby*, which is an automated cost-based optimizer for execution of batch data-parallel workflows. On the other hand, Chapter 5 focuses on continuous data-parallel workflows. In this chapter, by using windowed aggregation queries as an example, we characterize the execution plan space of continuous data-parallel workflows, which includes the choice of systems: i) centralized streaming systems, ii) distributed streaming systems, and iii) distributed batch processing systems. Then, we describe *Cyclops*, an optimizer that can select a good plan from the execution plan space for a given a query. Finally, Chapter 6 summarizes the key contributions of this dissertation and discusses future research directions.

2

Management of Workloads in the Display Tier

In this chapter, we first present the context to the typical deployment setting for systems that comprise data-intensive services. Specifically, we give a background overview of cloud computing infrastructure services, which is now typically used by data-intensive services to obtain resources and deploy systems on demand. We present the challenges for managing workloads of stateless systems through dynamically provisioning of cloud resources. We introduce a new control policy, called *proportional thresholding*, that specifically addresses these challenges. Finally, we present our control system that uses *proportional thresholding* for dynamically provisioning resources of stateless systems, which are commonly found in the display tier of data-intensive services.

2.1 Background Overview of Cloud Computing Infrastructure Services

Cloud computing derives from a long history of research and development on various approaches to IT outsourcing, in which customers draw from a utility provider's pool of capacity on a pay-as-you-go basis as an alternative to operating their own

infrastructure. Some approaches target specific classes of applications. For example, the market for turnkey Application Service Providers (ASPs) is active in some sectors (e.g., *salesforce.com*), and MapReduce/Hadoop middleware is widely used for data-parallel cluster computing clouds.

We focus on cloud computing infrastructure services. Typically the customer or *guest* selects or controls the software for virtual server instances obtained from one or more utility resource *providers*. The resource providers own hosting substrate resources (e.g., servers and storage), and offer them for lease to the guests. The guests may in turn use their private "slices" of leased resources to run software that provides a service to a dynamic community of clients. Virtual computing cloud infrastructure is a common and flexible example of the *utility computing* paradigm: recent offerings include Amazon Elastic Compute Cloud (EC2) [6], Aptana Cloud [12], and Joyent [61]. Advances in virtualization technologies, such as platform support (e.g., Intel's VT extensions) and hypervisor software (e.g., Xen [17] and VMware [126]), have made it easier for resource providers to adopt this paradigm and offer shared pools of hosting server resources as a service to guests.

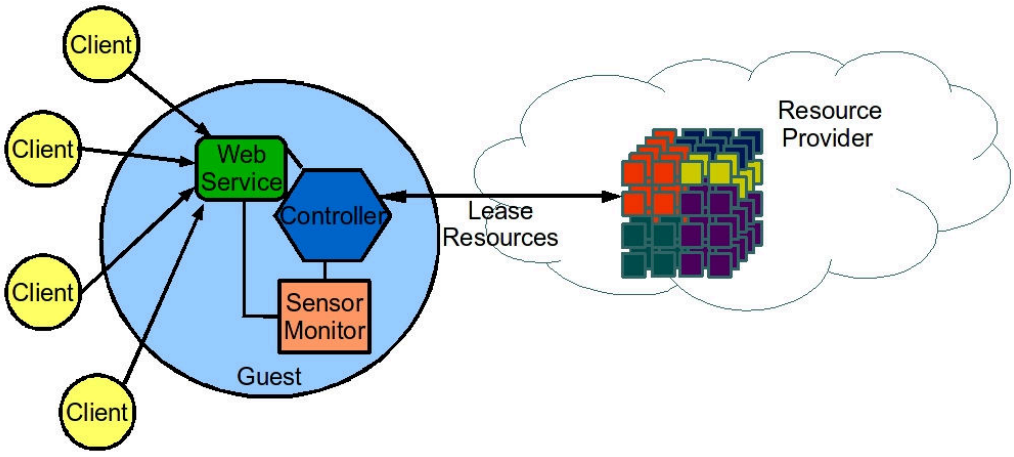


FIGURE 2.1: A guest, with a control system, running a web service on a dynamic slice of leased server resources.

Consider a simple motivating scenario in which a small startup company runs a

multi-tiered web application service, e.g., a Tomcat [116] server cluster that serves dynamic content to clients (see Figure 2.1). Rather than purchasing its own infrastructure to run its service, the company leases a slice of resources from a cloud hosting provider to reduce capital and operating costs. The application is horizontally scalable: it can grow to serve higher request loads by adding more servers. The hosting provider is modeled on Amazon's EC2 service [6]. It bills its guests on a per-instance hour basis for active virtual machines, and offers an API with a fixed set of sizing choices for each virtual machine instance: small, large, and extra large. The cloud API also offers support for zones to guide the placement of VM instances in the network.

The guest company wishes to manage the resources in its slice so as to maximize the return on its investment, while handling the stream of client requests with acceptable service quality. One option is for the guest to simply lease a static set of virtual servers. This approach works only if there is a single "ideal" size for the server set: service quality degrades if the slice is under-provisioned, and the guest pays excess rent if the slice is over-provisioned.

Cloud hosting services offer an opportunity for the customer to monitor the guest application and modulate the slice resources dynamically to balance service quality and cost. The challenge is to provide a general platform and off-the-shelf feedback control policies to automate this dynamic adaptation and take advantage of the natural elasticity of shared resources in cloud computing systems. The control policies must be stable and effective under the wide range of conditions that might be encountered in practice. Adaptive resource provisioning is just one example of the need for feedback-driven application control.

One premise of this dissertation is that cloud customers should be empowered to operate their own dynamic controllers, outside of the cloud platform itself, or perhaps as extensions to the cloud platform. Starting from this premise, we focus

on the problem of building external controllers for dynamic applications hosted on the cloud. We refer to the guest application controllers as *slice controllers*. Our perspective presents challenges for the design of both the cloud platform and the guest slice controllers. From the perspective of the cloud platform, the challenge is to export a sufficient set of well-behaved sensors and actuators to enable control policies to function effectively: Karamanolis et al. [63] suggest that system builders should focus on designing systems that are amenable to feedback control using standard off-the-shelf control policies. On the other hand, the slice controllers must make the best of the sensors and actuators built into the APIs that are actually available, and these APIs may be constrained in various ways to simplify the platform. For example, the APIs tend to provide a coarse granularity of control rather than the continuous or approximately-continuous actuators.

2.2 Feedback Control in a Cloud Computing Infrastructure

There have been a number of works on using feedback control to meet service requirements (e.g., [93, 96, 111, 124]). However, extending their approaches to the context of cloud computing presents new challenges. This section expands on the issues for effective slice controllers in a cloud computing infrastructure.

2.2.1 Decoupled Control

Cloud hosting platforms export a defined service interface to their customers (guests). The interface provides a useful separation of concerns: the guest is insulated from the details of the underlying physical resources, and the provider is insulated from the details of the application.

Our position is that the controller structure should also reflect this separation of concerns in the functionalities of the controllers. A cloud hosting provider runs its own control system (a *cloud controller*) to arbitrate resource requests, select guest

VM placements, and operate its infrastructure to meets its own business objectives. But *application control* should be factored out of the cloud platform and left to the guest. A clean decoupling of application control policy from the cloud platform mechanism is a necessary architectural choice to prevent cloud platforms from becoming brittle as guest demands change.

One way to facilitate this separation is for guests to select or implement their own (optional) slice controllers, outside of the cloud hosting platform. A principled layering offers the best potential for guests to innovate in their control policies and customize their controllers to the needs of their applications, and for the control architecture to scale to large numbers of diverse guests. For example, this structure is common to Amazon EC2 and Eucalyptus [38]. Both of these providers have their own control policy for arbitration, which is encapsulated from guests.

The layered approach requires a cloud hosting API that is sufficiently rich to support these interacting controllers. This separation implies that the guest slice controllers function independently of one another. Moreover, the cloud controller functions without direct knowledge of the application performance metrics, or the impact of allocation and placement choices on the service quality of the guests: it must obtain any knowledge it requires from the slice controller through the cloud hosting API. It is an open question how advanced control policies should interoperate and cooperate across the platform boundary. Note that these controllers are self-interested and are not mutually trusting: the interacting control loops have the structure of an economic negotiation.

Many of the previous works on feedback-controlled adaptive resource provisioning assume a central controller that combines application control and arbitration policy (e.g., [96, 124, 111]). Urgaonkar et al. [124] use queueing theory to model a multi-tier application and to determine the amount of physical resources needed by the application. Soundararajan et al. [111] present control policies for dynamic

provisioning of a database server. These approaches do not transfer directly to cloud environments with decoupled control. Padala et al. [93] is suitable for decoupled control, but requires adjustment for coarse-grained actuators, as discussed below.

2.2.2 Control Granularity

The control API for the cloud hosting platform is a "tussle boundary". There may be a gap between the sensors and actuators desired by slice controllers and those exposed by the resource providers. Providers may hide useful information to preserve their flexibility, or hide power to simplify the operation of the cloud controller policy and the underlying hosting mechanisms. The slice controller must make the best of whatever sensors and actuators are available, and whatever control intervals and granularity the provider allows.

For example, in a virtualized environment, resource providers may choose not to export the access to hypervisor-level actuators of the cloud computing infrastructure, such as controlling the CPU and memory allocations of a virtual machine and the location of the physical host of a virtual machine. EC2 and Eucalyptus discretize into a small range of predefined sizes, and do not expose these fine-grained actuators.

It is an open question what granularity is required for effective control. Many previous works on feedback control for Internet services have focused on an integrated control loop with fine-grained access to the sensors and actuators of the underlying virtualization platform on a single server. For horizontally scalable clusters, it is necessary to dampen the control loop at small cluster sizes, when the control granularity is coarse relative to the allocated resource (accumulated actuator value). The next section discusses a *proportional thresholding* technique for slice controllers to function with the coarse-grained control that is typical of current cloud platforms.

2.3 Proportional Thresholding

This section presents proportional thresholding to illustrate the challenges of coarse-grained control and the means to address it in a slice controller for a cloud hosting platform. Our control approach is similar to Padala et al. [93], which dynamically adjusts the CPU entitlement of a virtual machine to meet Quality of Service (QoS) goals by empirically modeling the relationship of CPU entitlement and utilization to tune the parameters of an integral control. The question is how to adapt the control policy for the case when fine-grained actuators for adjusting CPU entitlements are not available, e.g., the slice controller can only request changes to the number of virtual machines in a cluster. Resizing the number of virtual machines changes the capacity of the slice in coarse discrete increments. At small cluster sizes this may cause the control system to oscillate around a target CPU utilization.

Other works have considered the oscillation problem and instead, use static thresholding for their control policy (e.g., [124, 111]). In this policy, rather than having only one target goal, the goal is turned into a *target range*, defined by a high and low threshold. Thus, the system is considered on target when the sensor measurement falls inside the target range. Urgaonkar et al. [124] use this idea to allocate physical servers for a multi-tier Internet application. Only when the observed rate differs from the predicted rate by an application-defined range does the server resize. Their policy releases resources only when they are needed by other applications, which is difficult to achieve when the application controllers and cloud controller are decoupled. Similarly, Soundararajan et al. [111] present control policies for dynamic replication of database servers based on static thresholding with a target range. Their control policies are in steady state only when the average latency of their database servers is inside the low and high threshold.

Static thresholding is simple to use, however, it may not be robust to scale.

Consider the motivating scenario, mentioned in Section 2.1, of a guest running as a web service host. Since the Tomcat cluster is request-balanced, going from 1 to 2 machines can increase capacity by 100% but going from 100 to 101 machines increases capacity by not more than 1%. The relative effect of adding a fixed-sized resource is not constant, so using static threshold values may not be appropriate.

2.3.1 Design of Proportional Thresholding

Proportional thresholding addresses the problems outlined in the previous section. This control policy modifies an integral control by using a dynamic target range, instead of a single target value. Moreover, the dynamic target range decreases as the accumulated actuator values increases. *Proportional thresholding* is particularly important for sensors with large dynamic range and fixed coarse-grained actuators, such as using horizontal scaling to handle flash crowds, while maintaining a certain performance target. Specifically in our motivating scenario, the impact of a constant change in the actuator value is dependent on the current server set. For example, in a request balanced cluster, this behavior is illustrated in Figure 2.2, where we plot the effect of increasing the size of a Tomcat cluster to the CPU utilization while maintaining a fixed workload. Note that in our example, we use CPU utilization as our performance target metric. Other metrics can also be used, depending on the given service requirements. Moreover, proportional thresholding can handle multi-dimensional metrics (e.g., CPU and Memory utilizations) by first transforming the multi-dimensional target metrics into a uni-dimensional metric.

Similar to Padala et al. [93], the control policy uses an integral control because it eliminates steady state errors. An integral control is defined by

$$u_{k+1} = u_k + K_i \times (y_{ref} - y_k), \tag{2.1}$$

where u_{k+1} is the new actuator value, u_k is the current actuator value, K_i is the inte-

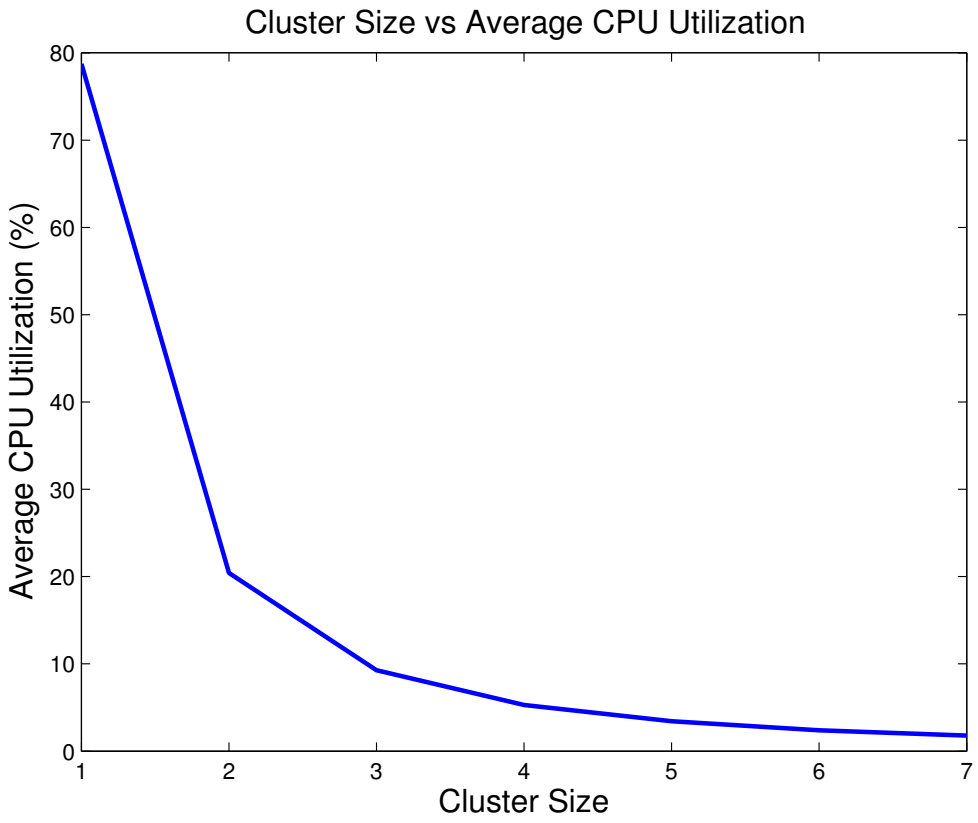


FIGURE 2.2: The effect of increasing the size of a Tomcat cluster to the CPU Utilization while maintaining a fixed workload.

gral gain parameter, y_{ref} is the target sensor measurement, and y_k is the current sensor measurement. The integral gain parameter K_i can be estimated empirically [96]. To avoid the problem of having oscillations due to the coarse-grained actuator, we then define y_h and y_l as the high and low target sensor measurements, which defines the target range. The modified integral control is as follows:

$$u_{k+1} = \begin{cases} u_k + K_i \times (y_h - y_k) & \text{if } y_h < y_k \\ u_k + K_i \times (y_l - y_k) & \text{if } y_l > y_k \\ u_k & \text{otherwise} \end{cases} . \tag{2.2}$$

This way, similar to static thresholding, a change in the actuator value only occurs when the sensor measurement is outside the target range. More specifically, the

actuator increases value only when it is above the high target and decreases in value when it goes below the low target.

In *proportional thresholding*, the target range used by the controller should be able to change dynamically depending on the accumulated actuator values. This addresses the problem with static thresholds, which does not give an effective control, in terms of ensuring high resource utilization and meeting client demands. With static thresholds, the behavior in Figure 2.2 can potentially lead to poor resource utilization. It should be noted that this behavior is not restricted to horizontal scaling.

The dynamic target range should capture the property of being resource-efficient. Since the relative effect of the increment becomes finer as the number of allocated resources, the target range should narrow for more precise control as the number of allocated resources increases. This means that our modified integral control (Equation 2) should have the property of converging to the standard integral control (Equation 1) as the accumulated actuator values goes to infinity. In order to achieve this behavior and assuming we set $y_h = y_{ref}$, proportional thresholding adjusts y_l depending on the number of actuator values accumulated and at the same time have the following behavior: $\lim_{x\to\infty} y_l = y_{ref}$, where x is the accumulated actuator values. In the next section, we describe our prototype control system and how we formulate the control parameters, specifically K_i and the equation for y_l for a specific actuator by empirically modeling the behavior of an actuator and sensor measurements.

2.4 Implementation

2.4.1 Prototype Control System

Like our motivating scenario, our guest serves as a web service host by using leased virtual machines to form a Tomcat cluster. Our prototype control system controls the size of the Tomcat cluster. Moreover, as mentioned in Section 2.2, our prototype

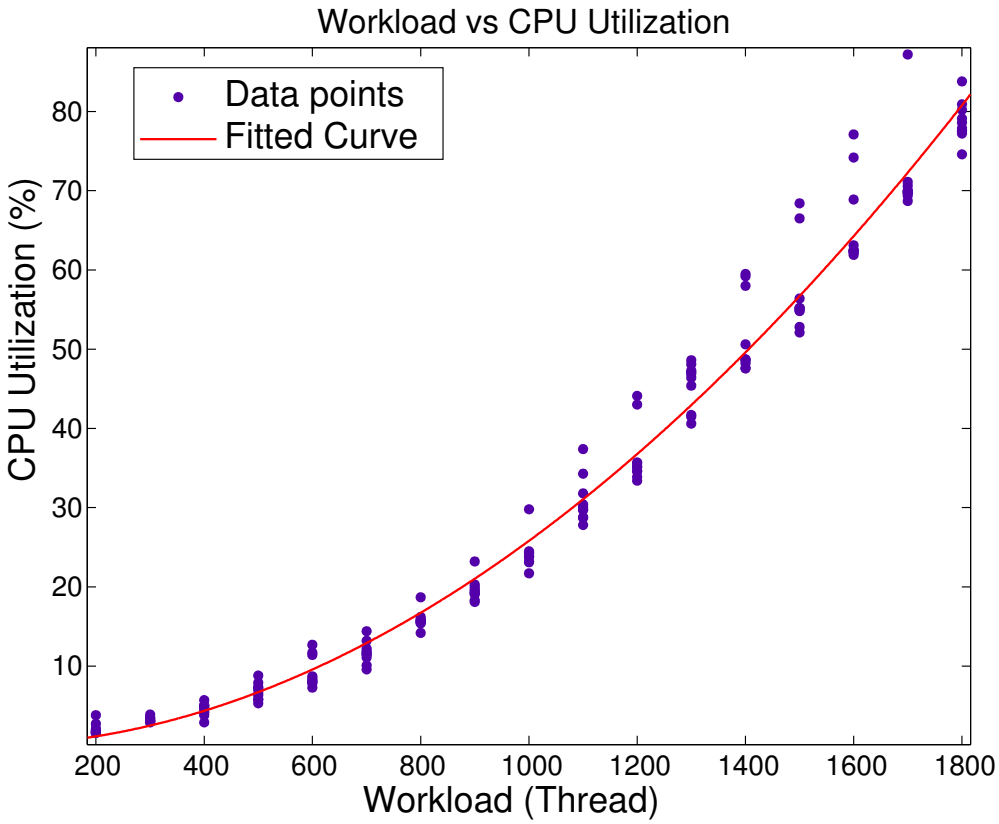


FIGURE 2.3: The CPU utilization of a Tomcat server under various workload.

assumes that there is a front-end Apache web server that balances the distribution of requests across all cluster nodes and a static back-end PostgreSQL server [99].

We use ORCA as the underlying architecture and resource leasing mechanism. It is a software toolkit developed at Duke University that allows guests to lease resources from a resource substrate. It is also a service-oriented infrastructure and the architecture provides resource leasing abstractions for the guests. Using virtualization, ORCA enables guests to share a common pool of resources [91]. Furthermore, to emulate the API provided by Amazon EC2, we only expose similar API functionalities to the control system. We use Automat [135] for our control interface. It is a programmable hosting center toolkit integrated with ORCA that supports external slice controllers. Similar to the motivating scenario, our slice controller performs

horizontal scaling.

Our prototype control system contains 3 key components: instrumentation, feedback controller, and ORCA guest controller plug-in module. This subsection describes these components in detail.

Instrumentation: We use Hyperic HQ [55] to gather the CPU utilization of all leased virtual machines. Hyperic HQ follows a server-agent model, where the server gathers the data from its agents. We deployed a Groovy script [43] plug-in to the HQ server that allows our control system to get the CPU utilization of all leased virtual machines by sending an HTTP request to the HQ server.

When our prototype control system is initialized, it requests for a virtual machine and installs the Hyperic HQ server on that machine. We then use a handler, which deploys an instance of an HQ agent on a specified virtual machine, that gets upcalled each time a succeeding new instance of virtual machine is leased. Our control system has a separate thread that at regular intervals sends an HTTP request to the HQ server and waits for an XML response containing the CPU utilization of all virtual machines. The XML is then parsed and stored in an internal Java object. The sensor measurement used by the control system's feedback controller is then the average CPU utilization of all allocated virtual machines, filtered by an exponential moving average filter.

Feedback Controller: Our prototype control system's feedback controller runs on a separate thread. The feedback controller has 3 control policies: *proportional thresholding*, integral control and static thresholding. Whenever there is a new sensor measurement, it uses the selected control policy to calculate the number of virtual machines to add or remove. The feedback controller also waits for the overall control system to be ready before performing new calculations. There are reasons that makes this necessary. First, ORCA uses a lease-based mechanism and does not allow cancellations of ongoing leases. If the control system wants to deallocate a leased

virtual machine, it has to wait for the lease to expire. Also, instantiating a new virtual machine takes roughly 2 minutes to complete. Thus, the feedback controller needs to make sure that the instantiation or destruction of virtual machine has finished before recalculating the amount of resources needed.

Controller Plug-in Module: ORCA allows guests to use the resource leasing mechanisms through a controller plug-in module. Guest software developers are provided with Java classes and interfaces. Specifically, ORCA provides guest controllers with an IController class that has a tick method, which gets called at regular intervals. Together with the information from the instrumentation and the feedback controller components, our control system uses this method to request for resources.

The controller plug-in module also comes with event handlers that get triggered at specific points of a lease's lifecycle. We use the onBeforeExtendTicket and onLeaseComplete handlers that get triggered just before a lease expires and after a lease reservation is processed, respectively. Since the feedback controller and the leasing mechanism runs asynchronously on separate threads, we use these two handlers to perform synchronization. These handlers modify a common state variable, which the feedback controller uses to determine the status of the system, such as whether ORCA has finished processing a slice reservation request.

As mentioned before, each new lease request is attached with a handler that installs and configures the necessary application. Our handler installs and configures Hyperic HQ agent and Tomcat server when a new virtual machine is instantiated and also performs the necessary shutdown sequence, such as shutting down the Tomcat server, when a virtual machine is about to be removed.

2.4.2 Control Parameters

For the parameter K_i of our control policy, we use the value $K_i = -.07$, which is estimated offline. This value is derived by using linear regression to model the

relationship between the CPU utilization and the cluster size under a synthetic heavy workload: $y_{k+1} = .8819y_k - .5892u_k$. Using the Z-transform of this model, we estimate the settling time and maximum overshoot corresponding to a range of K_i values. We then use a K_i that gives a settling time of 15.12 sample intervals and maximum overshoot of 0.0002544%.

Since the Tomcat cluster is request-balanced, we empirically measured the CPU utilization of a single machine under various workload to formulate the equation for y_l (see Figure 2.3). From these data, we find the best-fit curve, which in this case is

$$\texttt{CPU} = (3.869 \times 10^{-5}) \times \texttt{workload}^{1.947}. \tag{2.3}$$

Given a target CPU utilization, y_h, we can then use the equation to get the estimated average workload for each machine applied to the system:

$$\texttt{workload}_{est} = (\frac{y_h}{3.869 \times 10^{-5}})^{\frac{1}{1.947}}. \tag{2.4}$$

The idea is that we are interested in finding the lowest threshold value, such that the minimum amount of resources is used while still satisfying client demands. $\texttt{workload}_{est}$ tells us that any average workload greater than $\texttt{workload}_{est}$ will result in a CPU utilization of greater than y_h. This means that we only want to reduce the number of virtual machines if the resulting average workload is less than or equal to $\texttt{workload}_{est}$. The total workload from $\texttt{workload}_{est}$ when the number of machines is reduced by 1 is given by

$$\texttt{workload}_{tot} = \texttt{workload}_{est} \times (\texttt{currVM} - 1). \tag{2.5}$$

Using $\texttt{workload}_{tot}$ to solve for the average workload of the current number of machines gives the lowest average workload that is also greater than or equal to the average workload of the number of machines reduced by 1:

$$\texttt{workload}_{low} = \texttt{workload}_{est} \times \frac{\texttt{currVM} - 1}{\texttt{currVM}}. \tag{2.6}$$

We then calculate the y_l by applying $\texttt{workload}_{low}$ to Equation 3.

$$y_l = y_h \times (\frac{\texttt{currVM} - 1}{\texttt{currVM}})^{1.947} \tag{2.7}$$

Equation 7 shows that y_l converges to y_h as the number of virtual machines becomes very large. Moreover, *proportional thresholding* converges to the standard integral control (Equation 1).

In our implementation, we formulate the control parameter K_i and the equation for y_l offline, which depending on the type of target systems may already be enough. In our case, even though the model may turn out to be not very accurate due to changes in workload, our control system is still able to have an acceptable and stable behavior. One of the benefits of feedback control is that an accurate estimate of the models is not necessary because it can be robust to errors in the parameter estimates. For example, using the $K_i = -.07$ under a lighter workload gives a settling time of 39.96 sample intervals and maximum overshoot of $-.004345\%$, which has a longer convergence time but still results in a stable controller. Furthermore, another reason is that the spectrum of workloads used to formulate y_l encompasses a wide spectrum. As mentioned by Hellerstein et al., a good strategy for modeling systems is to start simple, which means that there is no need to develop a complex model, if a simpler model is already sufficient [50]. Nevertheless, there are target systems that need a complex and accurate model, which requires tuning the control parameters online. Although not specifically for automated control in cloud computing, there have been works that applies adaptive controllers for computing systems [82, 65]. We leave the research problem of integrating the modeling process with the control system as a future work, specifically for a cloud computing infrastructure.

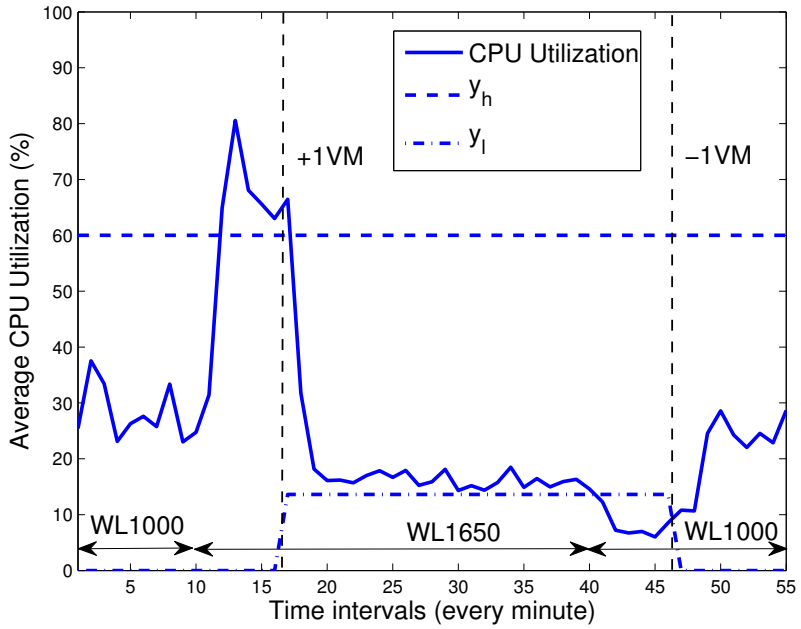
2.5 Evaluation

2.5.1 Synthetic Workloads

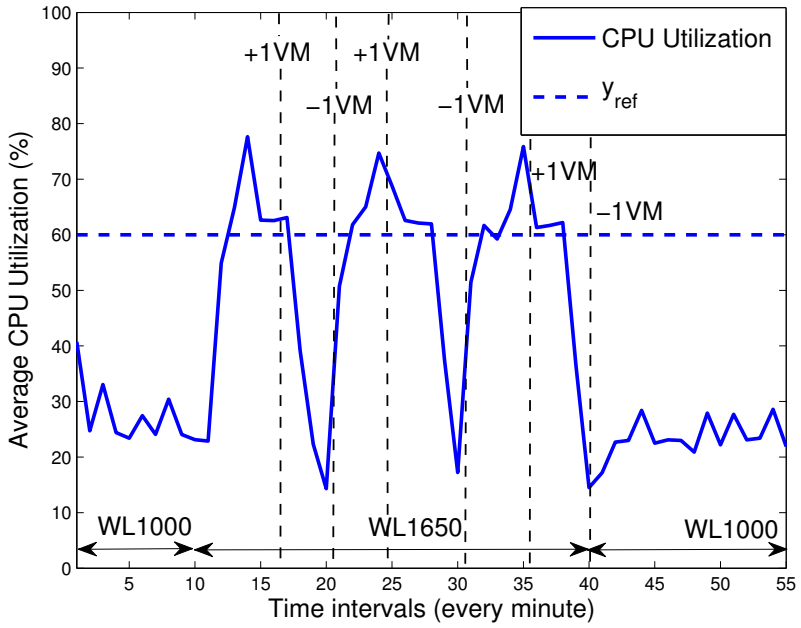
To evaluate our prototype control system, we compared the performance of our prototype using 3 control policies under various synthetic workloads: *proportional thresholding*, static thresholding, and integral control.

In Figure 2.4, we conducted an experiment, where we first applied a workload of 1000 threads, then at the 10^{th} time interval, we increased the workload to 1650 threads, and finally at the 40^{th} time interval the workload goes back to 1000 threads. Figures 2.4(a), and 2.4(b) show the behavior of the controller under *proportional thresholding* and integral control, respectively. Note that both figures start with 1 allocated virtual machine. Under *proportional thresholding*, the system does not oscillate, such that the controller allocates 1 virtual machine when the workload is 1000 threads and 2 virtual machines when the workload goes up to 1650 threads. In contrast, under integral control, the system oscillates between 1 and 2 virtual machines when the workload goes up to 1650 threads, which may not be desirable since it leads to short-term unpredictability.

In Figure 2.5, we conducted an experiment where we slowly ramp up the number of threads from 1000 to 1650 to 3200 and then finally decreasing the workload to 2450 threads. Figures 2.5(a), and 2.5(b) show the behavior of the controller under *proportional thresholding* and static thresholding, respectively. Like the previous experiment, both figures also start with 1 allocated virtual machine. Figure 2.5(a) also shows how y_l changes with cluster size, specifically, with 3 virtual machines, y_l has gone up to 23.84%. When the workload drops to 2450 threads, the controller is able to reduce the cluster size to 2 virtual machines and hence, conserve resources. This is in contrast to static thresholding, where the cluster size remains at 3, even though 2 virtual machines are enough to handle the workload.

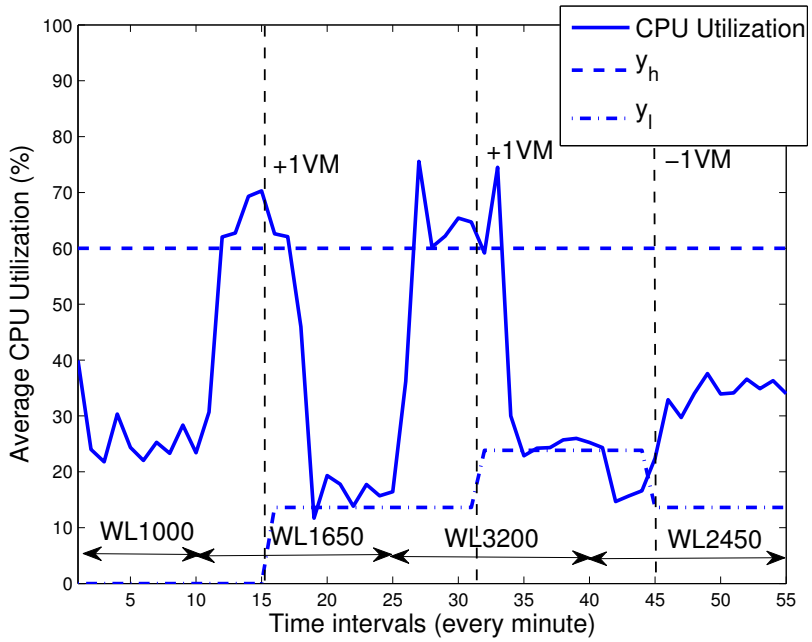


(a) Proportional Thresholding

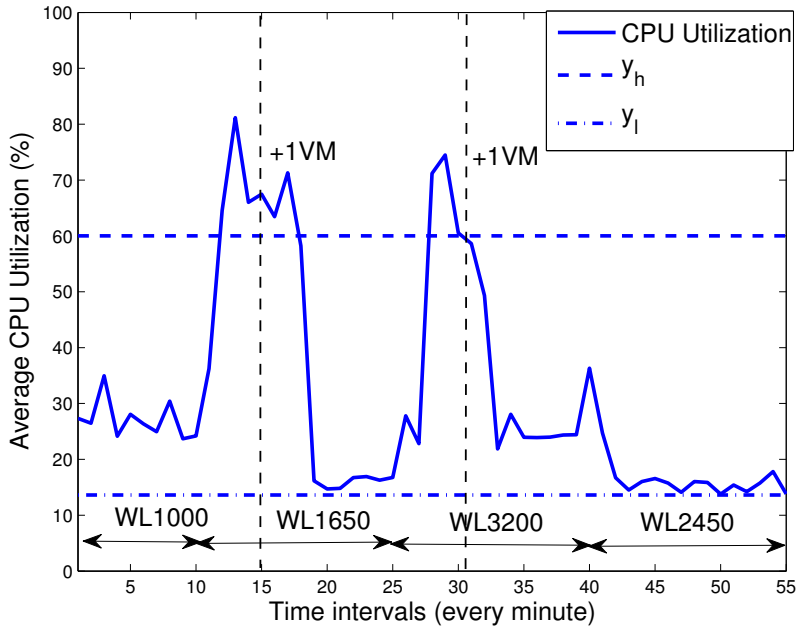


(b) Integral Control

FIGURE 2.4: Comparison between proportional thresholding and integral control under synthetic workloads.



(a) Proportional Thresholding



(b) Static Thresholding

FIGURE 2.5: Comparison between proportional thresholding and static thresholding under synthetic workloads.

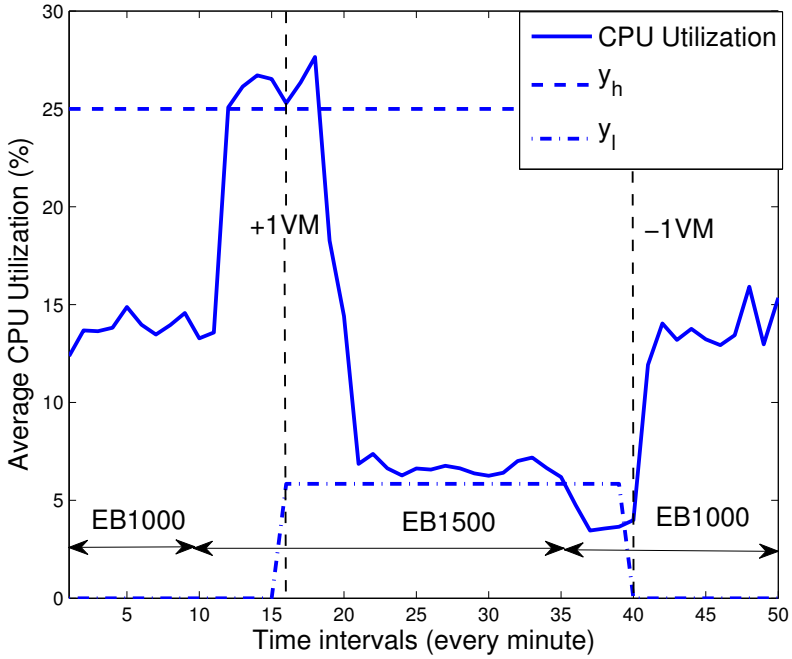
2.5.2 TPC-W Workloads

We also evaluated our prototype control system under TPC-W benchmark workloads [118]. TPC-W workloads simulate the activities of clients of an online bookstore. Moreover, we use a Java Servlet implementation of the TPC-W online bookstore application [119]. To ensure that the bottleneck is in the Tomcat tier, we used the TPC-W browsing mix workload, which simulates clients browsing an online bookstore. The workload's simulated concurrent clients, called Emulated Browsers(EB), are also configurable.

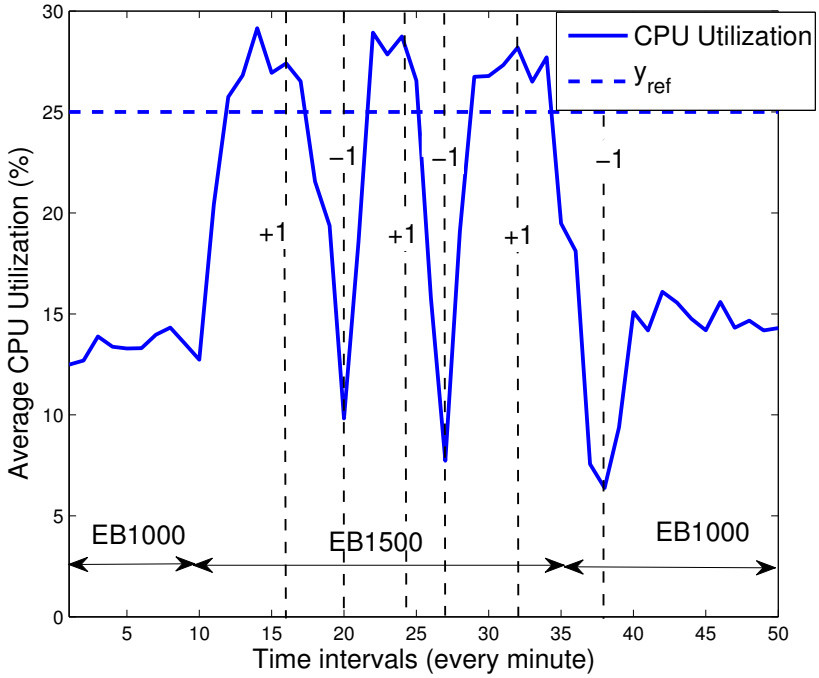
In Figure 2.6, we conducted an experiment that initially had 1000 Emulated Browsers interacting with the online bookstore. After the 10^{th} time interval, 500 additional Emulated Browsers were added for a duration of 25 time intervals. Figures 2.6(a), and 2.6(b) show the behavior of the controller under *proportional thresholding* and integral control, respectively. Note that in this experiment we set $y_{ref} = 25\%$. Similar to the controller's behavior for our experiment in Section 2.5.1, under *proportional thresholding*, the system does not oscillate. The controller increases the size of the cluster by 1 when the workload goes up to 1500 EBs and then decreases back to the original cluster size when the workload goes back to 1000 EBs. Conversely, under integral control, the cluster size oscillates between 1 and 2 virtual machines.

2.6 Conclusions

We have presented issues that make feedback control in a cloud computing infrastructure different from feedback control of other computer systems: decoupled control, and control granularity. Moreover, we have shown why prior works related to automated control may not work, when used in a cloud computing infrastructure. We have introduced *proportional thresholding*, a new control policy that takes into account the coarse-grained actuators provided by resource providers. Using the ac-



(a) Proportional Thresholding



(b) Integral Control

FIGURE 2.6: Comparison between proportional thresholding and integral control under TPC-W workloads.

tuator constraints similar to Amazon EC2, we have presented a prototype control system that performs better than traditional integral control and static thresholding for dynamically provisioning stateless applications.

3

Management of Workloads in the Storage Tier

While the previous chapter focuses on stateless systems that are commonly found in the display tier of Figure 1.1, this chapter focuses on managing workloads for the storage tier. As is the case for managing the display tier, we again manage the workload of this tier by dynamically provisioning resources. We describe the design and implementation of *Elastore*, which is a control system for elastic storage systems. At its core, *proportional thresholding* is also used for the control policy but new challenges have to be addressed due to the need for migration of data to fully achieve the benefits of dynamic provisioning.

3.1 Introduction

Web-based services frequently experience rapid load surges and drops. Web 2.0 workloads, often driven by social networking, provide many recent examples of the well-known flash crowd phenomenon. One recent Facebook application that "went viral" saw an increase from 25,000 to 250,000 users in just three days, with up to 20,000 new users signing up per hour during peak times [11].

There is growing commercial interest and opportunity in automating the man-

agement of such applications and services. Automated surge protection and adaptive resource provisioning for dynamic service loads has been an active research topic for at least a decade. Today, the key elements for wide deployment are in place. Most importantly, a market for cloud computing software and services has emerged and is developing rapidly, offering powerful new platforms for *elastic* services that grow and shrink their service capacity dynamically as their request load changes.

Cloud computing services manage a shared "cloud" of servers as a unified hosting substrate for guest applications, using various technologies to virtualize servers and orchestrate their operation. A key property of this cloud hosting model is that the cloud substrate provider incurs the cost to own and operate the resources, and each customer pays only for the resources it demands over each interval of time. This model offers economies of scale for the cloud provider and a promise of lower net cost for the customer, especially when their request traffic shows peaks that are much higher than their average demand. Such advantageous demand profiles occur in a wide range of settings. In one academic computing setting, it was observed that computing resources had less than 20% average utilization [66], with demand spikes around project deadlines. This chapter focuses on another driving example: data-intensive multi-tier Web services, which often show common dynamic request demand profiles (e.g., [27]). Figure 3.1 depicts this target environment.

Mechanisms for elastic scaling are present in a wide range of applications. For example, many components of modern Web service software infrastructure can run in clusters at a range of scales; and can handle addition and removal of servers with negligible interruption of service. This chapter deals with *policies* for elastic scaling of the *storage tier*, which is an integral part of modern data-intensive multi-tier Web services, based on automated control, building on the foundations of previous works [96, 129, 93, 92, 62] discussed in Section 3.7. We first focus on challenges that are common for a general form of virtual cloud hosting, often called *infrastructure*

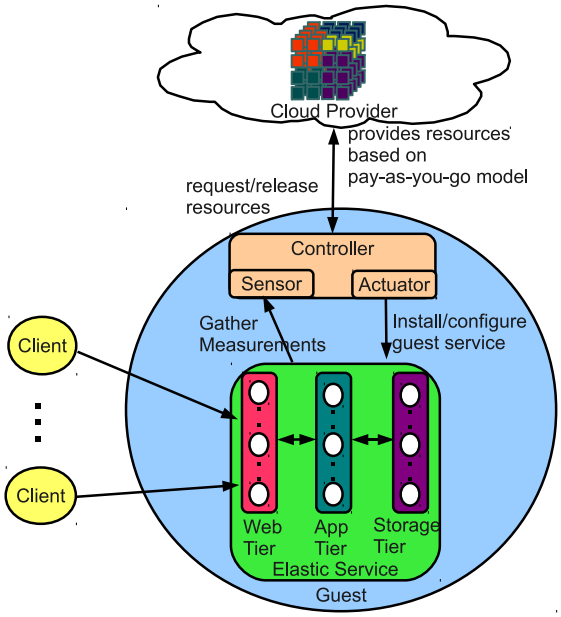


FIGURE 3.1: A multi-tier application service (guest) hosted on virtual server instances rented from an elastic cloud provider. An automated controller uses cloud APIs to acquire and release instances for the guest as needed to serve a dynamic workload.

as a service, in which the customer acquires virtual server instances from a cloud substrate provider, and selects or controls the software for each server instance. Amazon's Elastic Compute Cloud (EC2) is one popular example: the EC2 API allows customers to request, control, and release virtual server instances on demand, with pay-as-you-go pricing based on a per-hour charge for each instance. A recent study [101] reported that the number of Web-sites using Amazon EC2 grew 9% from July to August 2009, and has an annual growth rate of 181%.

We then address new challenges associated with scaling the storage tier in a data-intensive cluster-based multi-tier service in this setting. We employ an integral control technique called *proportional thresholding* to modulate the number of discrete virtual server instances in a cluster. Many previous works modulate a continuous resource share allotted to a single instance [129, 93, 92]; cloud systems with per-instance pricing like EC2 do not expose this actuator. We also address new challenges

of *actuator lag* and *interference* stemming from the delay and cost of redistributing stored data on each change to the set of active instances in the storage tier.

While the discussion and experiments focus on cloud infrastructure services with per-instance pricing, our work is also applicable to multiplexing workloads in an enterprise data center. Some emerging cloud services offer packaged storage APIs as a service under the control of the cloud provider, instead of or in addition to raw virtual server instances for each customer to deploy a storage tier of their choice. In that case, our work applies to the problem faced by the cloud provider of controlling the elastic cloud storage tier shared by multiple customers.

We have implemented a prototype controller, called *Elastore*, for an elastic storage system. We use the Cloudstone [110] generator for dynamic Web 2.0 workloads to show that the controller is effective and efficient in responding to workload changes.

3.2 System Overview

Figure 3.1 gives an overview of the target environment: an elastic *guest* service hosted on server instances obtained on a pay-as-you-go basis from a cloud substrate provider. In this example, the guest is a three-tier Web service that serves request traffic from a dynamic set of clients.

Since Web users are sensitive to performance, the guest (service provider) is presumed to have a Service Level Objective (SLO) to characterize a target level of acceptable performance for the service. An SLO is a predicate based on one or more performance metrics, typically response time quantiles measured at the service edge. For any given service implementation, performance is some function of the workload and servers that it is deployed on; in this case, the resources granted by the cloud provider.

The purpose of controlled elasticity is to grow and shrink the active server instance set as needed to meet the SLO efficiently under the observed or predicted workload.

Our work targets guest services that can take advantage of this elasticity. When load grows, they can serve the load effectively by obtaining more server instances and adding them to the service. When load shrinks, they can use resources more efficiently and save money by releasing instances.

This chapter focuses on elastic control of the storage tier, which presents challenges common to the other tiers, and additional challenges as well: state rebalancing, actuator lag, interference, and coordination of multiple interacting control elements. Storage scaling is increasingly important in part because recent Web 2.0 workloads have more user-created content, so the footprint of the stored data and the spread of accesses across the stored data both grow with the user community. Our experimental evaluation uses the Cloudstone [110] application service as a target guest. Cloudstone mimics a Web 2.0 events calendar application that allows users to browse, create, and join calender events.

3.2.1 Controller

We implement a *controller* process that runs on behalf of the guest and automates elasticity. The controller drives actuators (e.g., request/release instances) based on sensor measures (e.g., request volume, utilization, response time) from the guest and/or cloud provider. Our approach views the controller as combining multiple control elements, e.g., one to resize each tier and one for rebalancing in the storage tier, with additional rules to coordinate those elements. Ideally, the control policy is able to handle unanticipated changes in the workload (e.g., flash crowds), while assuring that the guest pays the minimum necessary to meet its SLO at the offered load.

For clouds with per-instance pricing, the controller runs outside of the cloud provider and is distinct from the guest application itself. This makes it possible to implement application-specific control policies that generalize across multiple cloud

providers. (RightScale takes this approach.)

In general, these clouds present a problem of *discrete actuators*. As Figure 3.1 shows, the controller is limited to elasticity actuators exposed by the cloud provider's API. Cloud infrastructure providers such as Amazon EC2 allocate resources in discrete units as virtual server instances of predetermined sizes (e.g., small, medium, and large). Most previous work on provisioning elastic resources assume continuous actuators such as a fine-grained resource entitlement or share on each instance [96, 129]. In Chapter 2.3, we have described and developed a *proportional thresholding* technique for stable integral control with coarse-grained discrete actuators. We apply this technique to elastic control of the storage tier in a cloud with per-instance pricing.

Our approach to integrated elastic control assumes that each tier exports a control API that the controller may invoke to add a newly acquired storage server to the group (*join*) and remove an arbitrary server from the group (*leave*). These operations may configure the server instances, install software, and perform other tasks necessary to attach new server instances to the guest application, or detach them from the application. We also assume a mechanism to balance load across the servers within each tier, so that request capacity scales roughly with the number of active server instances.

3.2.2 Controlling Elastic Storage

The storage tier is a distributed service that runs on a group of server instances provisioned for storage and allocated from the cloud provider. It exports a storage API that is suitable for use by the middle tier to store and retrieve data objects. We make the following additional assumptions about the architecture and capabilities of the storage tier.

- It distributes stored data across its servers in a way that balances load effectively for reasonable access patterns, and redistributes (*rebalances*) data in

response to *join* and *leave* events.

- It replicates data internally for robust availability; the replication is sufficient to avoid service interruptions across a sequence of *leave* events, even if a departing server is released back to the cloud before *leave* rebalancing is complete.

- The storage capacity and I/O capacity of the system scales roughly linearly with the size of the active server set. The tiers cooperate to route requests to a suitable storage server.

The design of robust, incrementally-scalable cluster storage services with similar goals has been an active research topic since the early 1990s. Many prototypes have been constructed including block stores [69, 102] and file systems [115, 40], key-value stores [36, 7], database systems [34], and other "brick-based" architectures. For our experiments, we chose the Hadoop Distributed File System (HDFS), which is based on the Google File System [40] design and is widely used in production systems.

As we have framed the problem, elastic control for a cloud infrastructure service presents a number of distinct new challenges.

Data Rebalancing: Elastic storage systems store and serve persistent data which imposes additional constraints on the controller. On adding a new node, a clustered Web server, such as the target systems addressed in Chapter 2, gives immediate performance improvements because the new node can quickly start serving client requests. In contrast, adding a new storage node does not give immediate performance improvements to an elastic storage system because the node does not have any persistent data to serve client requests. The new node must wait until data has been copied into it. Thus, rebalancing data across storage nodes is a necessary procedure, especially if the elastic storage system has to adapt and handle changes in client workloads.

Interference to Guest Service: Data rebalancing consumes resources that can otherwise be used to serve client requests. The amount of resources (bandwidth) to allocate to the rebalancing process affects its completion time as well as the degree of adverse impact on the guest application's performance during rebalancing. Note that overall improvement to system performance can be achieved only through data rebalancing. It may not be advisable to allocate a small bandwidth for rebalancing since it can take hours to complete, causing a prolonged period of performance problems due to suboptimal data placement. It may be better to allocate more bandwidth to complete rebalancing quickly while suffering a bigger intermediate performance hit. Finding the right balance automatically is nontrivial.

Actuator Delays: Regardless of the bandwidth allocated for rebalancing, there will always be a delay before performance improvements can be observed. The controller must account for this delay, or else it may respond too late or (worse) become unstable.

3.3 Components of the Controller

Our automated controller for the elastic storage tier, which we call *Elastore*, has three components:

- *Horizontal Scale Controller (HSC)*, responsible for growing and shrinking the number of storage nodes.
- *Data Rebalance Controller (DRC)*, responsible for controlling the data transfers to rebalance the storage tier after it grows or shrinks.
- *State machine*, responsible for coordinating the actions of the HSC and the DRC.

We present each of these components in turn and discuss how they address the challenges listed in Section 3.2. In this chapter, we use Hadoop Distributed File System (HDFS) [48] as an example storage system that is controlled by Elastore.

3.3.1 Horizontal Scale Controller (HSC)

Actuator: The HSC uses cloud APIs to change the number of active server instances. Each storage node in the system runs on a separate virtual server instance.

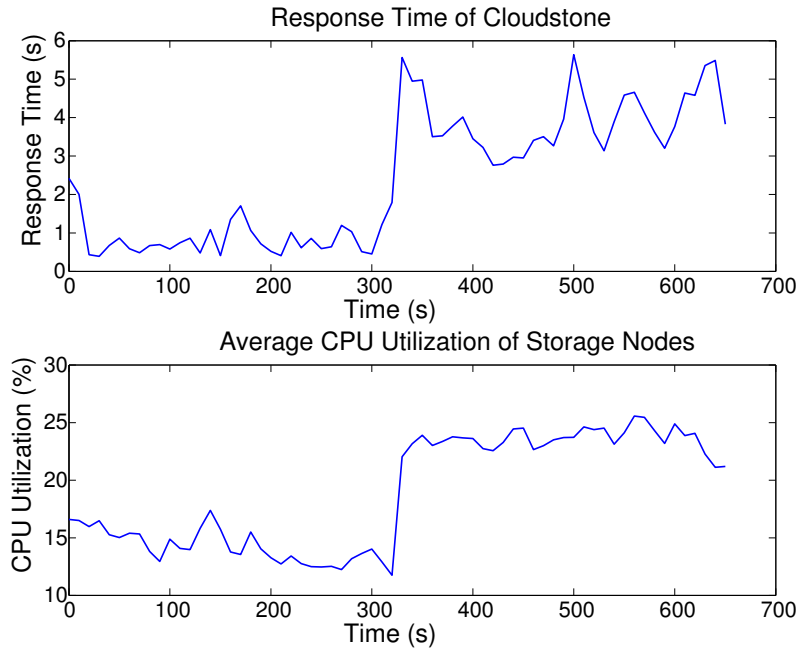


FIGURE 3.2: Cloudstone response time and average CPU utilization of the storage nodes, under a light load and a heavy load that is bottlenecked in the storage tier. CPU utilization in the storage tier correlates strongly with overall response time (the coefficient is .88), and is a more stable feedback signal.

Sensor: The HSC bases its elastic sizing choices on a feedback signal incorporating one or more system metrics. A good choice of metric for the target environment satisfies the following properties: (i) the metric should be easy to measure accurately without intrusive instrumentation because the HSC is external to the guest application, (ii) the metric should expose the tier-level behavior or performance, (iii)

the metric should be reasonably stable, and (iv) the metric should correlate to the measure of level of service (e.g, the service's average response time) as specified in the client's service level objective (SLO).

Our experiments use CPU utilization on the storage nodes as the sensor feedback metric because it satisfies these properties. The CPU utilization can be obtained from the operating system or the virtual machine without instrumenting application code. Moreover, tier-level metrics, such as CPU utilization, allow the controller to pinpoint the location of the performance bottleneck. Figure 3.2 shows that CPU utilization in the storage tier is strongly correlated to overall response time when the bottleneck is in the storage tier, even if the bottleneck is on the disk arms rather than the CPU. Figure 3.2 also shows that CPU utilization is a more stable signal than response time. We chose this metric for convenience: other metrics could be used instead of or in addition to CPU utilization.

Control Policy: We use *proportional thresholding* for the control policy of the HSC (refer to Section 2.3). We fit a function to empirical measurements of the CPU utilization of HDFS datanodes (storage nodes) at various load levels to determine the parameter values to use for proportional thresholding.

3.3.2 Data Rebalance Controller (DRC)

When the number of storage nodes grows or shrinks, the storage tier must rebalance the layout of data in the system to spread load and meet replication targets to guard against service interruption or data loss. The DRC uses a rebalancer utility that comes with HDFS to rebalance data across the storage nodes. Rebalancing is a cause of actuator delay and interference. For example, a new storage node added to the system cannot start serving client requests until some of the data to be served has been copied into it; and the performance of the storage tier as a whole is degraded while rebalancing is in progress.

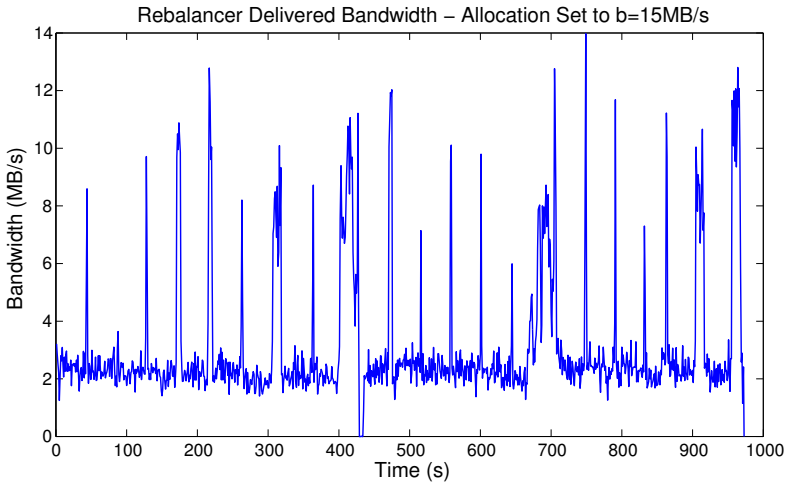


FIGURE 3.3: Delivered bandwidth of the HDFS rebalancer (version 0.21) for b=15MB/s. Although the bandwidth peaks at the configured setting b, the average bandwidth is only 3.08MB/s. We tuned the control system for the measured behavior of this actuator.

Actuator: The tuning knob of the HDFS rebalancer—i.e., the actuator of the DRC—is the *bandwidth* b allocated to the rebalancer. The bandwidth allocation is the maximum amount of outgoing and incoming bandwidth that each storage node can devote to rebalancing. The DRC can select b to control the tradeoff between lag—i.e., the time to completion of the rebalancing process—and interference—i.e., performance impact on the foreground application—for each rebalancing action. Nominally, interference is proportional to b and lag is given by s/b where s is the amount of data to be copied.

We discovered empirically that the time to completion of rebalancing given by the current version of the HDFS rebalancer is insensitive to b settings above about 3 MB/s. The reason is that the rebalancer does not adequately pipeline data transfers, as illustrated in Figure 3.3. However, since HDFS and its tools are used in production deployments, and unreliable actuators are a fact of life in real computer systems, we decided to use the HDFS rebalancer "as is" for now and adapt to its behavior in the control policy.

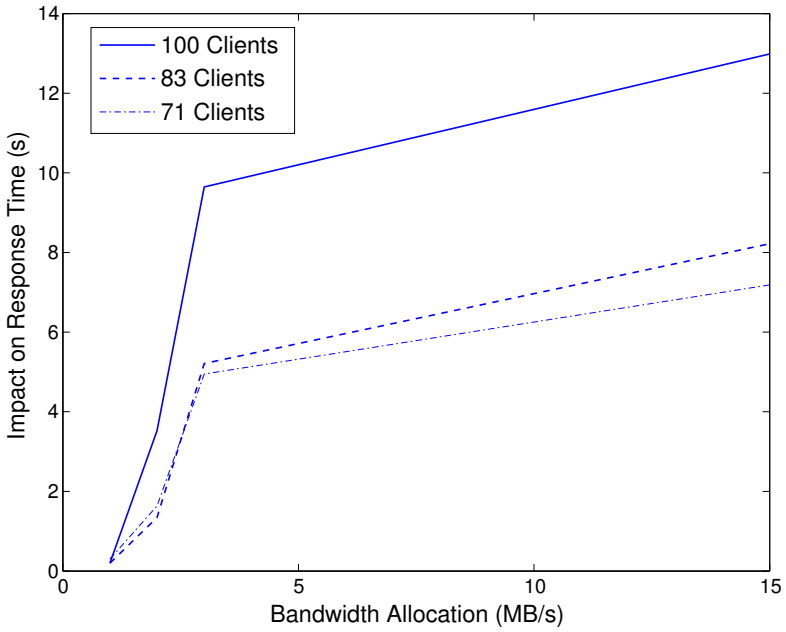


FIGURE 3.4: The impact of HDFS rebalancing activity on Cloudstone response time, as a function of the rebalancer's bandwidth cap and the client load level. The effect does not depend on the cluster size N because the cap b is on bandwidth consumed at each storage node.

Figure 3.4 shows the interference or *performance impact (Impact)* of rebalancing on Cloudstone response time, as a function of the bandwidth throttle (b) and the per-node load level (l). *Impact* is defined as the difference between the average response time with and without the rebalancer running. As expected, *Impact* increases as b and l increase. Running the rebalancer with b=1MB/s gives negligible impact on average response time.

Sensor and Control Policy: We conducted a set of experiments to model the following relationships using multi-variate regression:

- The time to completion of rebalancing (*Time*) as a function of the bandwidth throttle (b) and size of data to be moved (s): $Time = f_t(b, s)$.
- The impact of rebalancing on service response time (*Impact*) as a function of

the bandwidth throttle (b) and per-node workload (l): $Impact = f_i(b, l)$.

The goodness of fit is high ($R^2 \geq 0.995$) for both models. Values of s and l are used as sensor measurements by the DRC, and b is the tuning knob whose value has to be determined. The choice of b represents a tradeoff between *Time* and *Impact*. As previously stated, the controller must consider the lag (*Time*) to complete an adjustment and restore a stable service level, and the magnitude of the degradation in service performance (*Impact*) during the lag period.

To strike the right balance between actuator lag and interference, the DRC poses the choice of b as a cost-based optimization problem. Given a cost function $Cost = f_c(Time, Impact) = f_c(f_t(b, s), f_i(b, l))$, DRC chooses b to optimize *Cost* given the observed values of s and l. The cost function is a weighted sum: $Cost = \alpha Time + \beta Impact$. The ratio of $\frac{\alpha}{\beta}$ can be specified by the guest based on the relative preference towards *Time* over *Impact*. Another alternative is to choose b such that *Time* is minimized subject to an upper bound on *Impact*. These choices are useful in adjusting to significant load swings. The controller may also use the *Impact* estimate to drive "just in time" responses to more gradual load changes without violating SLO, but we do not evaluate that alternative in this work.

3.3.3 State Machine

To preserve stability during adjustments, the HSC and DRC must coordinate to manage their mutual dependencies. The first dependency arises from the DRC's actuator lag. After a storage node has been added by the HSC, the service obtains the full benefit of the node only after rebalancing completes. The second dependency is due to noise introduced into the sensor measurements that a controller relies on, while the actions of the other controller are being applied. For example, the data copying and additional computations done during rebalancing impact the CPU utilization measurements seen by the HSC.

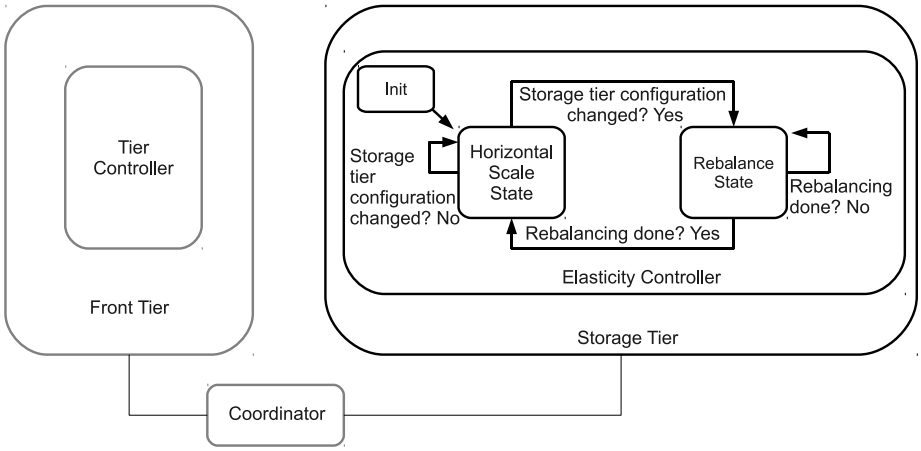


FIGURE 3.5: Block diagram of the control elements of a multi-tier application. This diagram shows the internal state machine of the elasticity controller (Elastore) of the storage tier, but depicts the application tier as a black box.

Ignoring these dependencies can lead to poor control decisions, or much worse, unstable behavior due to oscillation. Consider a scenario where the HSC does not take the DRC's actuator lag into account. After adding a new storage node, the HSC may not see any changes in its sensor measurements, or the sensor measurements may show a decline in performance. This observation will cause the HSC to allocate more storage nodes unnecessarily to compensate for the lack of improvement in system performance. In turn, the completion time and impact of rebalancing could deteriorate further.

The elasticity controller (Elastore) uses the state machine shown in Figure 3.5 to coordinate the actions of the HSC and DRC. Figure 3.5 also illustrates how Elastore fits as an element of an integrated control solution for a multi-tiered services. In this chapter, we focus on the storage tier and treat the control elements for other tiers as a black box. Section 3.6.3 provides further discussion on the problem of coordinating multiple per-tier control elements.

When Elastore starts up, it goes from the *Init State* to the *Steady State*. In this state, only the HSC is active. It remains in this state until the HSC triggers an adjustment to the active server set size. When nodes are added or removed, the state

machine transitions to the *Rebalance State*. The HSC is dormant in the *Rebalance State* to allow the previous change to stabilize and to ensure that it does not react to interference in its sensor measurements caused by data rebalancing.

The DRC, as described in Section 3.3.2, enters the *Rebalance State* after a change to the active server set size. It remains in this state until data rebalancing completes, after which the state machine returns to the *Steady State*. A form of rebalancing, called decommissioning, occurs on removal of a storage node to maintain configured replication degrees. HDFS stores n (a configurable parameter) replicas per file block, one of which may be on a node identified for removal. The replica of a block on a decommissioned node can be replaced by reading from any of the $n - 1$ remaining copies. HDFS has an efficient internal replication mechanism that triggers when the replica count of any block goes below its threshold. Currently, the DRC does not regulate this process because HDFS does not expose external tuning knobs for it. In any case, we observed that this process has minimal impact on the foreground application.

3.4 Implementation

3.4.1 Cloudstone Guest Application

CloudStone: We modified and configured Cloudstone to run with GlassFish application servers for displaying contents to users, PostgreSQL database servers for storing and serving structured data, and HDFS for storing and serving content objects such as PDF documents and image files. This required adding an HDFS class abstraction to Cloudstone to enable it to use HDFS storage APIs. We also added new parameter types to Cloudstone's configuration file so that users can easily configure and switch between different file systems without having to recompile the source code. In all, this involved adding 200 lines of code to Cloudstone. The experiments use a block size in the storage tier of 800KB, which is the maximum size of binary

files generated by Cloudstone. The HDFS replica count is set to three following best practices from production deployments.

HDFS: HDFS distributes the content objects (files) across an elastic set of N storage nodes, called datanodes. A namenode tracks metadata including replica counts and locations for each file.

With its current implementation, HDFS does not ensure that storage nodes are request-balanced, since its internal policy is based on disk usage capacity. However, Cloudstone's workload generator is designed such that structured data and content objects are accessed in a uniform distribution, which naturally balances requests across all HDFS datanodes.

Finally, we modified HDFS to expose the rebalancer's bandwidth throttle b as an actuator to the external controller. We created an RPC interface in the HDFS namenode that notifies all HDFS datanodes of changes to the bandwidth limit.

3.4.2 Cloud Provider

We use a local ORCA [56, 26] cluster as our cloud infrastructure provider. ORCA is a resource control framework developed at Duke University. It provides a resource leasing service which allows guests to lease resources from a resource substrate provider, such as a cloud computing provider. The test cluster exports an interface to instantiate Xen virtual machine instances [17] on a shared pool of 30 host servers.

3.4.3 Elastore

The controller is written in Java and contains 1500 lines of code. ORCA allows guests to use the resource leasing mechanisms through a controller plug-in module written to various toolkit APIs [135]. The control policy is clocked by periodic upcalls from the ORCA leasing core to a tick method in the controller. The controller plug-in module also installs event handlers that trigger notifications from the leasing core at specific

points of a lease's life cycle. We use the onBeforeExtendTicket and onLeaseComplete handlers that are triggered just before a lease expires and after a new lease reservation is complete (e.g., a new datanode is instantiated).

Each new lease request is attached with a guest application control handler that installs, configures, and launches the guest software (Cloudstone and/or HDFS) on the leased server instances after they start. Our handler installs and configures the HDFS datanode software package when a new storage node is instantiated and also performs the necessary shutdown sequence, such as shutting down the HDFS datanode, when the controller decides to decommission a storage node. The control system includes two other important components, described next.

Instrumentation: To get the sensor measurements mentioned in Section 3.3, we modified the HDFS datanode to gather system-level metrics such as CPU utilization. We included the Hyperic SIGAR library with each HDFS datanode. At periodic intervals, the HDFS datanode uses SIGAR to gather the system-level metrics and piggybacks this information on the regular heartbeat messages of the HDFS datanode to the HDFS namenode. We also modified the HDFS namenode and implemented a remote procedure call (RPC) that allows the controller to get the sensor measurements of all HDFS datanodes in a single call. With this implementation, the controller only needs to contact the HDFS namenode to get the sensor measurements for all storage nodes.

The controller has a separate thread that periodically obtains these measures: the sensor interval is set to 10 seconds. The controller then processes the sensor measures and applies the control policy as described. It computes the average CPU utilization of the HDFS datanodes, and applies an exponential moving average filter of six time periods to the average CPU utilization.

Subcontroller Modules: The controller has two subcontroller modules, corre-

sponding to HSC and DRC, as described in Section 3.3. Each of these modules runs on a separate thread. As mentioned in Section 3.3, the coordination between these two subcontroller modules is guided by a finite state machine interlock. Since the feedback subcontrollers and the leasing mechanism run asynchronously on separate threads, they synchronize through a common state variable accessed by the upcall handlers. This state variable activates and deactivates the subcontroller modules according to the state of the controller's finite state machine.

3.5 Evaluation

3.5.1 Experimental Testbed

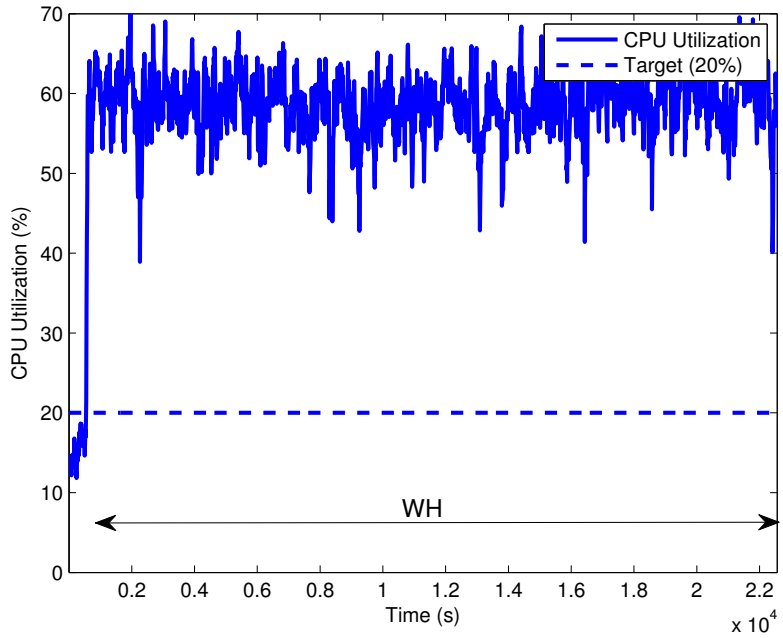
Our experimental service cluster consists of a group of servers running on a local network. To focus on the storage tier, the front-end application tier and database tier of Cloudstone are statically over-provisioned: the database server (PostgreSQL) runs on a powerful server with 8GB of memory and 3.16 GHz dual-core CPU, while the forward tier (GlassFish) runs in a fixed six-node subcluster, where each node has 1GB of memory and a 2.8GHz CPU. The storage tier nodes are dynamically allocated virtual machine instances, with fixed settings of resource configuration, based on the control policy discussed in Section 3.3. We used weak virtual machine instances for the storage nodes to trigger responses from the controller at a smaller scale of workloads. The virtual machine instances have 30GB disk space, 512MB of memory, a single disk arm, and a 2.8GHz CPU, with a CPU cap set at 20%. Before each experiment, the HDFS tier is preloaded with at least 36GB worth of data (i.e., images and binary files used by Cloudstone). The Cloudstone workload generator is running on separate well-provisioned machines, and is never bottlenecked.

3.5.2 Controller Effectiveness

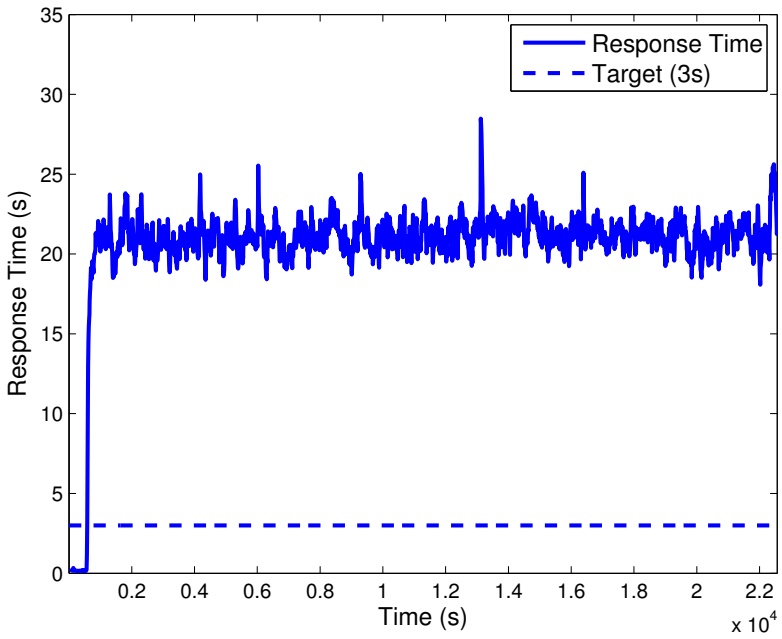
Internet workloads are known to show predictable long-term variations and highly unpredictable short-term fluctuations [124]. Long-term variations, usually predictable through models of past observations, can be handled by pre-provisioning resources in anticipation of the changes in workload behavior. However as mentioned in Section 3.1, unpredictable changes to the workload, such as flash crowds, happen often in practice. These changes cannot be anticipated by simply observing past observations, and are hard to deal with. We are interested in evaluating the effectiveness and adaptability of our controller under such unanticipated workload behavior. We first use Cloudstone to subject HDFS to dynamic workloads that represent sudden increases in load. We want to evaluate whether our controller is able to dynamically provision more resources to handle the client workload and to fix the SLO violations that arise.

In our first experiment, we programmed the load signal to first generate a small workload (load factor of 1.0). At around 600 seconds, the load factor is increased by a factor of 10. We set the target response time to be three seconds, which corresponds to 20% CPU utilization of the storage nodes. The storage system is set-up running with minimum number (three) of HDFS datanodes to handle the initial workload.

Figures 3.6 and 3.7 show the performance of Cloudstone with static resource provisioning and our control policy, respectively. With static provisioning, the system becomes under-provisioned for the increase in workload (see Figures 3.6(a) and (b)). Since resources are statically provisioned, the performance will continue to have SLO violations indefinitely until the workload goes back down. With our control policy, the controller detects the impact on performance of the increase in workload and decides to increase the storage cluster size by nine (see Figures 3.7(a) and (b)). Figure 3.7(a) also shows the period, marked with an arrow and labeled as "Rebalance",

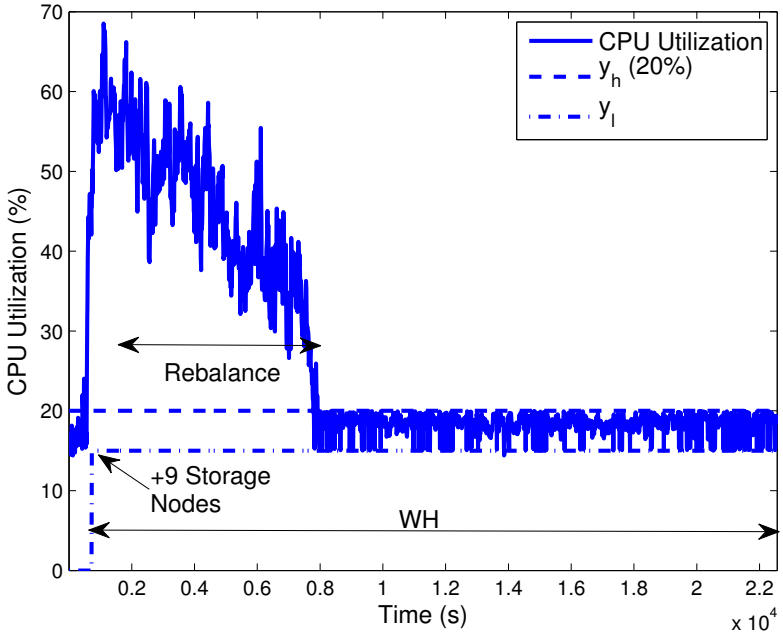


(a) Average CPU utilization of the HDFS datanodes with static provisioning

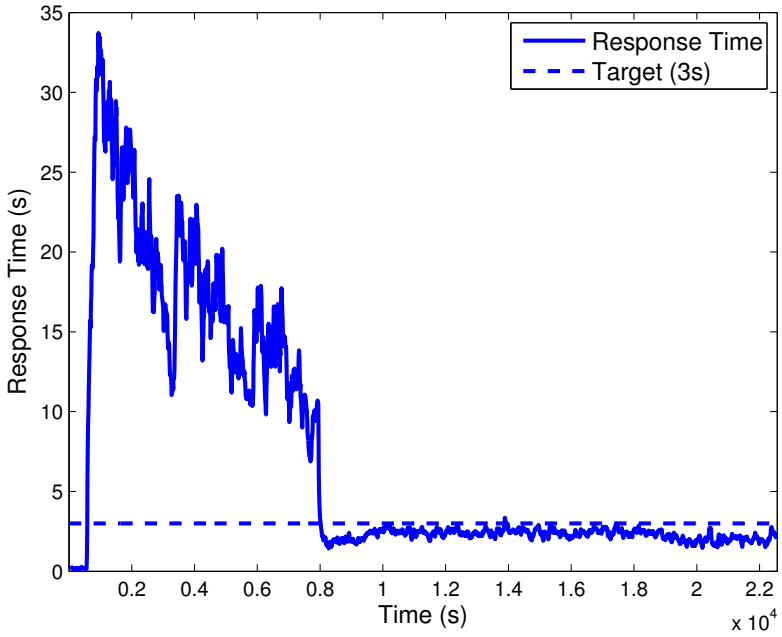


(b) Response time of the Cloudstone application with static provisioning

FIGURE 3.6: The performance of Cloudstone with static allocation under a 10-fold increase in workload volume. The time periods with high volume of workload is labeled as "WH".



(a) Average CPU utilization of the HDFS datanodes with dynamic provisioning



(b) Response time of the Cloudstone application with dynamic provisioning

FIGURE 3.7: The performance of Cloudstone with our control policy under a 10-fold increase in workload volume. The time periods with high volume of workload is labeled as "WH".

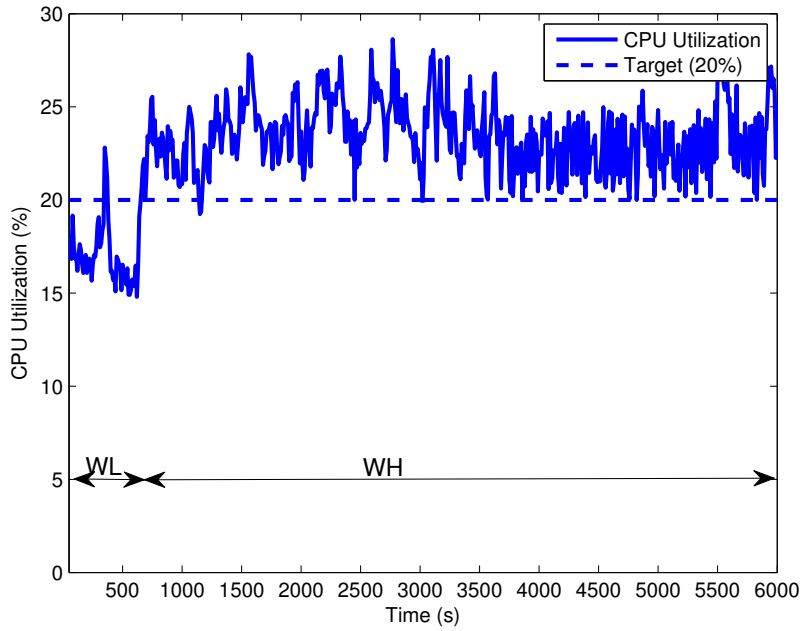
when the rebalancing process is taking place. By the $t = 7800$ seconds, the average response time and CPU utilization have dropped back below the target limit due to the successful addition and integration of new HDFS datanodes.

With our controller, although there is an impact of up to ten seconds in response time due to the rebalancing process, the system is able to adapt to the new workload and fix the SLO violations (Figure 3.7(b)). As discussed in Section 3.3.2, the noisy behavior of the response time is unavoidable due to the current implementation of the HDFS rebalancer. Furthermore, Figure 3.7(b) shows what happens when the cost of data rebalancing is paid under bursty workloads.

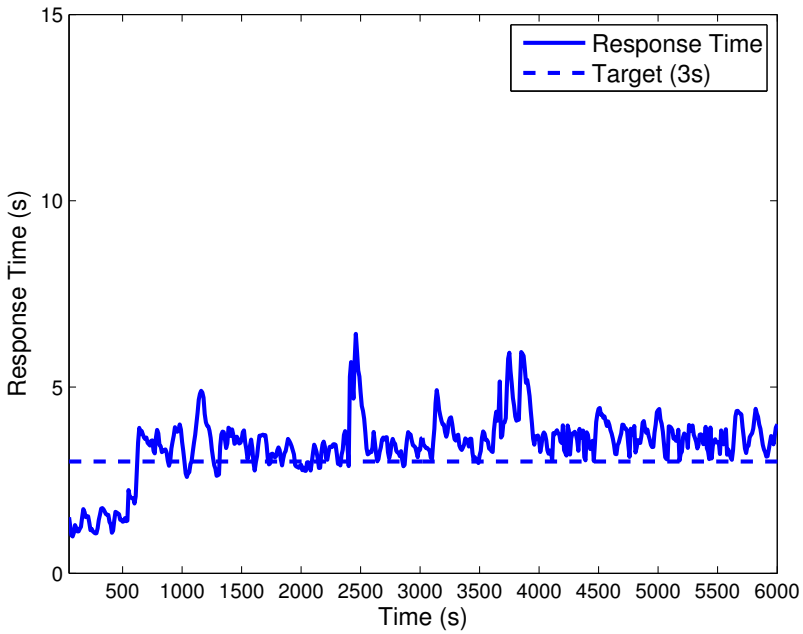
In our next experiment, we programmed the load signal to generate a workload factor of 7.0. The storage system is initially provisioned with ten HDFS datanodes to handle the running workload. At around 600 seconds, a small increase (35%) in the workload volume is introduced.

Similarly, Figures 3.8 and 3.9 show the performance of Cloudstone with static resource provisioning (Figures 3.8(a) and (b)) and with the elastic control policy (Figures 3.9(c) and (d)). At around the 700^{th} second, the controller decides to add one more storage node. The rebalancing process incurs an average impact of four seconds. By the $t = 2450$ seconds, the rebalancing process has completed and the SLO violation has been eliminated.

In both experiments, we picked a rebalance policy that has a balanced tradeoff between the data rebalancer's completion time and impact. In section 3.5.4, we discuss how the α and β parameters of the cost function can be tuned by the guest to get the desired ratio of impact to completion time. The tuning will be done based on how much rebalance cost a guest is willing to absorb to fix SLO violations rapidly.

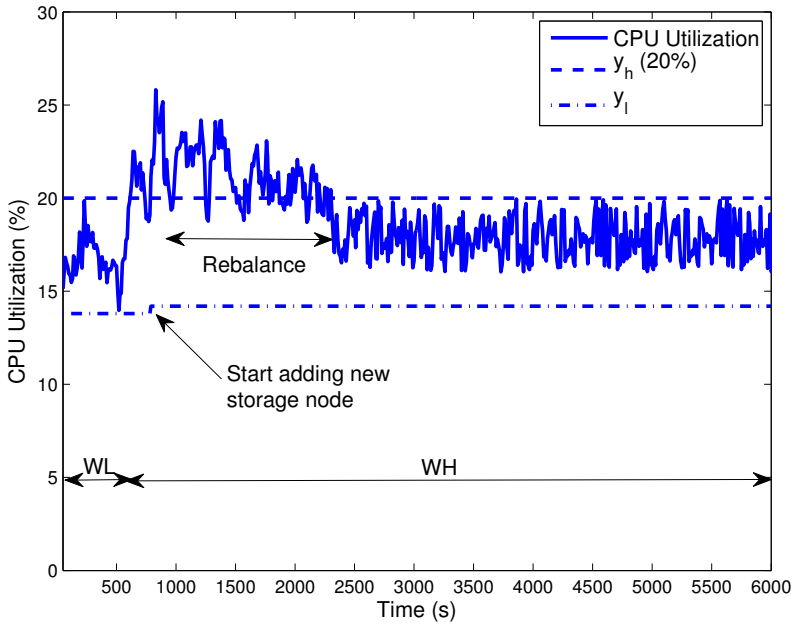


(a) Average CPU utilization of the HDFS datanodes with static provisioning

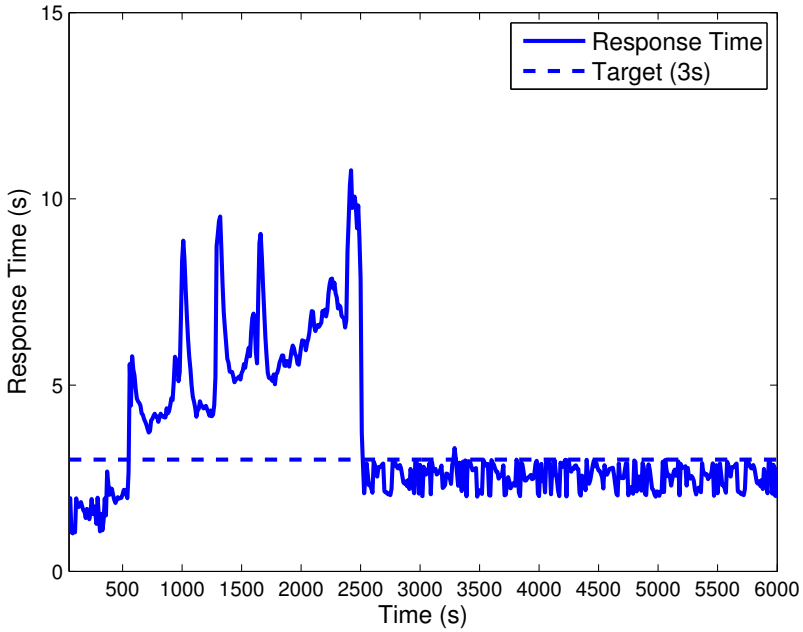


(b) Response time of the Cloudstone application with static provisioning

FIGURE 3.8: The performance of Cloudstone with static allocation under a small increase in workload volume. The time periods with low and high volume of workload are labeled as "WL" and "WH", respectively.



(a) Average CPU utilization of the HDFS datanodes with dynamic provisioning



(b) Response time of the Cloudstone application with dynamic provisioning

FIGURE 3.9: The performance of Cloudstone with our control policy under a small increase in workload volume. The time periods with low and high volume of workload are labeled as "WL" and "WH", respectively.

3.5.3 Resource Efficiency

In the next experiment, we subject Cloudstone to a sudden decrease in workload from an initial load factor of 5.0 to a load factor of 3.5. The system is initially provisioned to handle the initial workload without any SLO violations. We are interested to see whether our controllable elastic storage system meets our resource efficiency goal mentioned in Section 3.2. Figure 3.11 shows the behavior of our controller.

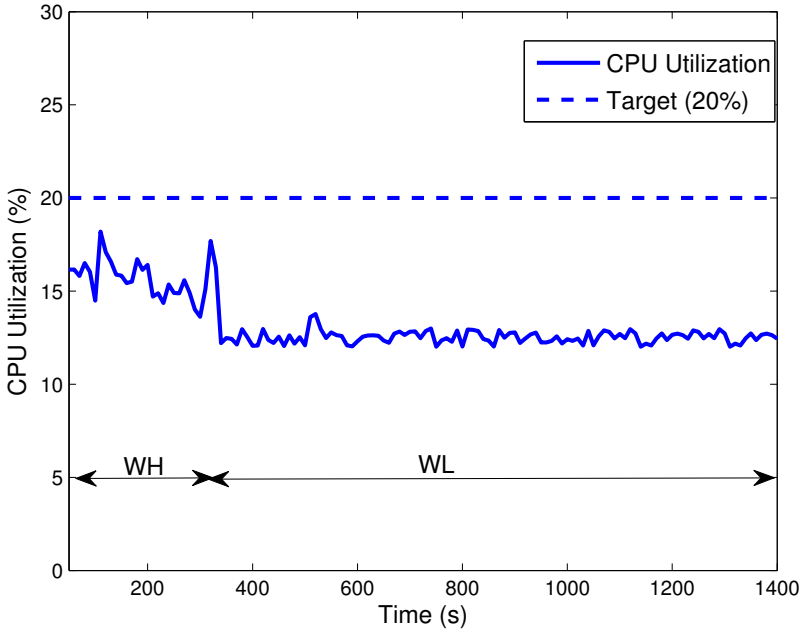
Similar to the previous experiment, we compare the performance of static thresholding with our control policy. In this experiment, the workload is decreased after 370 seconds. Figures 3.10(a) and (b) show the CPU utilization and response time graph of the system with static provisioning. Since the resource configuration does not change in static provisioning, we also see a decrease in response time that is two seconds below the threshold for SLO violation. However under a prolonged decrease in workload, static provisioning will incur unnecessary resource costs because it is over-provisioned for the current workload, with utilization well below the target.

With our control policy, on the 420^{th} second, our controller is able to detect and determine that the system is over-provisioned. The controller then releases the excess HDFS datanode and returns the resources to the cloud provider (see Figure 3.11(a)). As shown in Figure 3.11(b), even with a decrease in the size of the storage cluster, there still are no SLO violations.

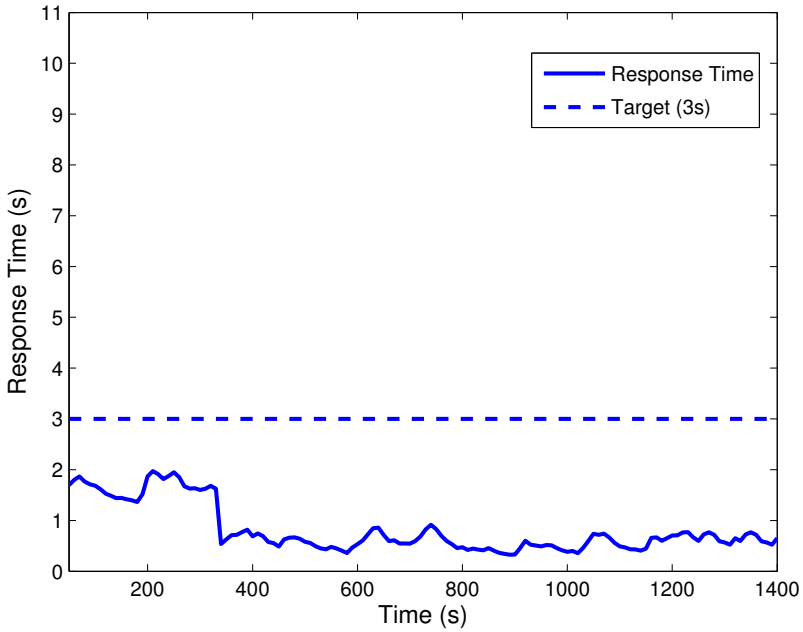
3.5.4 Comparison of Rebalance Policies

For illustrative purposes, we compared our rebalance policy with two other policies: aggressive and conservative. An aggressive policy allocates as much bandwidth as possible to the data rebalancer. On the other hand, a conservative policy allocates minimal bandwidth so that there is minimal impact on the response time of Cloudstone during rebalancing.

In this experiment, we drive a heavy workload to Cloudstone and then let the

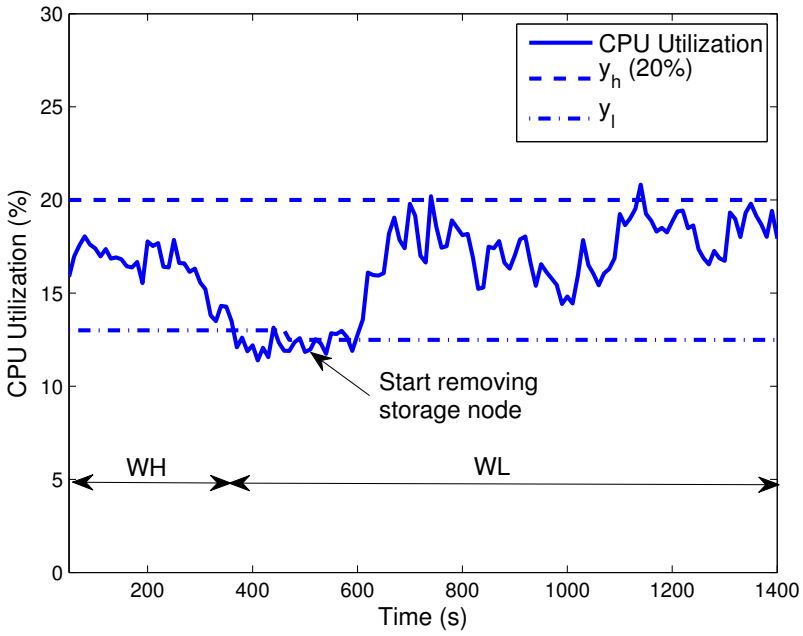


(a) Average CPU utilization of the HDFS datanodes with static provisioning

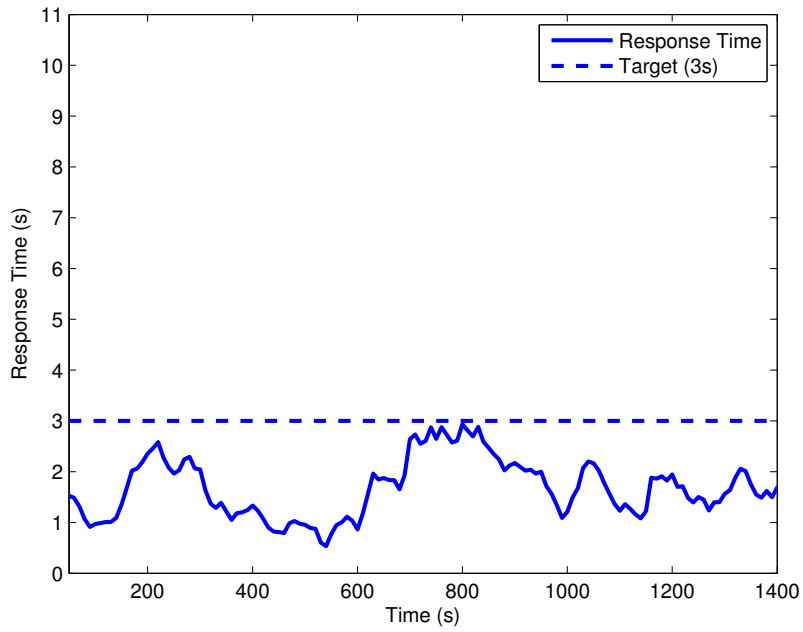


(b) Response time of the Cloudstone application with static provisioning

FIGURE 3.10: The performance of Cloudstone with static provisioning under a decrease in workload volume. The time periods with low and high volume of workload are labeled as "WL" and "WH", respectively.



(a) Average CPU utilization of the HDFS datanodes with dynamic provisioning



(b) Response time of the Cloudstone application with dynamic provisioning

FIGURE 3.11: The performance of Cloudstone with our control policy under a decrease in workload volume. The time periods with low and high volume of workload are labeled as "WL" and "WH", respectively.

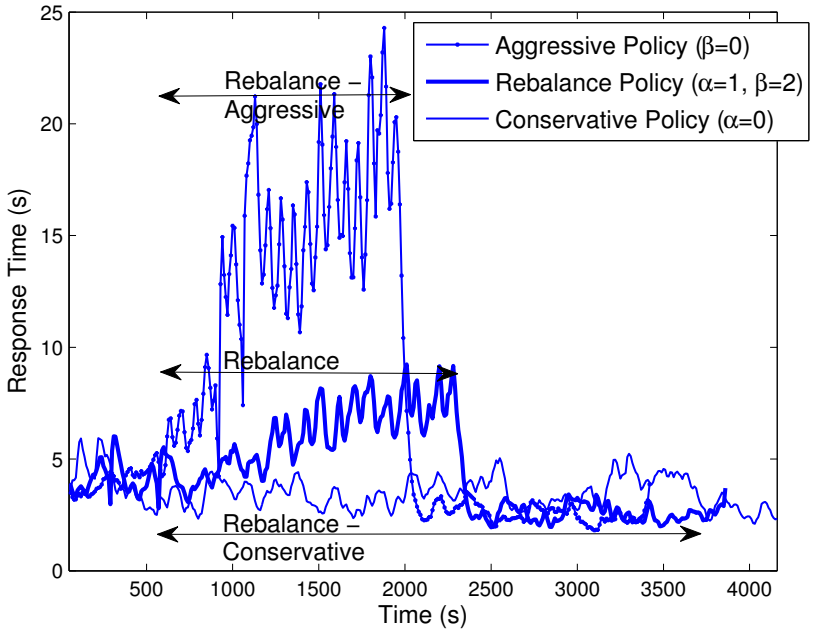


FIGURE 3.12: The response time of Cloudstone under different rebalance policies: Aggressive policy, our controller's rebalance policy, and conservative policy.

controller allocate a new storage node and start the rebalancing process. Figure 3.12 shows the response time of Cloudstone when the rebalancer is triggered. For each policy in the figure, the period reflecting the running time of the rebalancer is marked with an arrow and labeled as "Rebalance". An aggressive rebalance policy gives the shortest time to completion but also severely impacts the response time of Cloudstone. However, compared to our policy, it only gives around five minutes of improvement to the rebalancer's time to completion. Moreover, our policy gives 15 seconds less impact on the response time of Cloudstone than an aggressive rebalance policy. A conservative policy gives minimal impact on response time but takes more than twice as long to complete; which is also not good because it prolongs the period of SLO violations. Our controller's rebalance policy shows a balance between time to completion and the impact on Cloudstone. It should be noted that a conservative and aggressive rebalance policy can be attained by setting the α and β parameters

respectively to zero.

3.6 Discussion

3.6.1 Other Cloud Computing Models

In this chapter, we focused on the infrastructure-as-a-service model of cloud computing (like Amazon EC2) where each guest runs a private storage service on virtual server instances leased from the cloud provider. Software-as-a-service is another popular model on the cloud (like Amazon S3) where the cloud provider offers a software service using a pay-as-you-go pricing model; rather than leasing out virtual resources. In this case, the control problem of storage elasticity does not arise for the guests because they don't control the storage infrastructure. However, the control problem has simply moved to the cloud provider. Our overall approach can be used by the cloud provider, but an additional challenge arises. The storage service will now be used and shared among multiple guests, each with its own performance objectives and data. The controller needs to make sure that there is performance isolation and differentiation across guests. It is worth noting a recent paper that discusses the problem of massive resource inefficiencies in emerging parallel systems [9]. Someone has to pay for this inefficiency—the cloud provider will have to pay in the software-as-a-service model unless they leverage elastic storage.

3.6.2 Data Rebalancing

Automated data rebalancing is a critical ingredient of a controllable elastic storage system. The kinds of rebalancing needed is specific to the storage system and application needs. In our target guest, for example, data rebalancing entails moving files to new storage nodes, replicating files for availability, and ensuring that the load is balanced across all nodes. On the other hand, collocating multiple data items is a crucial need during data rebalancing in a database system, e.g., collocating indexes

along with the indexed data records. An ideal rebalancer should hide system-specific details, and expose appropriate tuning knobs so that the elasticity controller can invoke the rebalancer to meet service-specific needs on completion time and performance impact. The best way to implement a rebalancer is a nontrivial question.

HDFS Rebalancer: The current implementation of the HDFS rebalancer limits the performance of the elasticity controller. The bursty data transfer rates observed in Figure 3.3 and the noisy response time values in Figure 3.7(b) are unfortunate side-effects of this implementation. The implementation also causes high computational overhead. For example, the HDFS rebalancer creates separate socket connections between HDFS datanodes for each scheduled block transfer. Small block sizes can cause many open socket connections between datanodes. It should be noted that the issues with the HDFS rebalancer are tangential to the main point of this chapter, which is addressing the challenges of automated control of an elastic storage tier.

3.6.3 Dealing with Multiple Actuators

One issue with having multiple actuators is that there will be sensor data interference and dependency. For example, we have shown that the data rebalancing process has an impact on sensor measurements. Thus, there is a need for coordination and synchronization among multiple actuators. In this work, we used a *hierarchical* coordination scheme to coordinate between two actuators: cluster resizing and data rebalancing. This policy treats the two actuators as mutually exclusive, and data rebalancing always gets triggered after cluster resizing. One limitation of this policy is that if the workload changes while in the *Rebalance State*, the elasticity controller waits until the rebalancing process finishes before making another decision regarding the size of the cluster. In this situation, the controller can potentially be less responsive, i.e., longer time to adapt to workload changes.

A possible future work is to look into alternative policies for coordination between

the two actuators. One possible approach is to run both actuators concurrently. We could develop techniques to filter out the impact of the rebalancing process on the sensor measurements. While the data rebalancer is running, the horizontal scale controller can then use the filtered measurements to determine whether further changes to the cluster configuration are necessary. These enhancements may make the controller more agile, which may be useful in rapid dynamic change of workloads, but we must balance stability and agility. Our current solution is simple and provably stable in that the controllers can never work at cross purposes.

Multi-tier Services: We focused on controlling the storage tier of a multi-tier service, which is active only when the controller has determined that the bottleneck is in the storage tier. We can consider controlling a multi-tier service as dealing with multiple actuators. In this case, each tier can have an elastic number of resources (e.g., virtual server machines).

We are interested in finding the minimum amount of coordination among multiple actuators that still results in an effective and efficient control system. Treating each tier as an independent actuator with its own control policy can cause shifting of the performance bottleneck between tiers. Our proposed solution involves using an interlock to coordinate between tiers. A tier can only release resources when the interlock is not being held by the other tiers. The interlock is acquired by a tier when it detects that its resources are being over-utilized. In Chapter 2, we have designed a controller for the display tier. However we leave as a future work, the evaluation of the coordination policy between the display tier controller and our storage tier controller.

3.6.4 Adapting to Expected Load Changes

Currently, we only addresses the case of unpredictable workloads, in which the cost of rebalancing has to be paid by guests in order to fix SLO violations. As mentioned in

Section 3.5.2, for the other type of workloads that exhibit reasonable load changes, the HSC controller can perform pre-provisioning of resources. With a predictable workload signal, we can use our models of time to completion of rebalancing (*Time*) and the impact of rebalancing (*Impact*) (refer to Section 3.3) to find when to trigger the actuators so that no SLO violations happen. The control policy then turns into a constrained optimization problem that minimizes the chosen bandwidth allocated to the rebalancer, while ensuring *Time* is earlier than the projected time of SLO violation and the sum of the projected growth in workload and *Impact* is smaller than the SLO threshold.

3.7 Related Work

To our knowledge, we are the first to address the problem of automated control for elastic storage in the context of cloud computing. Specifically, no other work has focused on the combination of issues regarding discrete actuators, interference of the data rebalancing process on guest services, and actuator delays when designing a controller for elastic storage systems. SCADS [14] is a closely related work that deals with the problem of dynamically scaling a storage system. Its design uses machine learning to determine and predict resource requirements. Our controller employs a reactive policy. Moreover, we automate the data rebalancing process which is necessary for scaling the storage system.

Control of Computing Systems: There has been work on feedback control of computing systems [96, 129, 93, 92]. These works assume the availability of continuous actuators. Moreover, these works address the issue of control for non-clustered systems. For example, Wang et al. [129] dynamically adjust the CPU capacity of a guest virtual machine. We extend their work by designing a controller for clustered services. Specifically, our work uses the cluster mechanism of incrementally-scalable systems to dynamically provision cluster nodes.

In terms of automated control of computing systems in the context of cloud computing, Padala et al. [93] have also considered a decoupled architecture (between the guest and cloud provider) in the design of their control system. However, our work differs in that our control system also takes into account actuator constraints, which are inherent in all commercial cloud providers. For example, rather than adjusting CPU allocation, which commercial providers do not provide, our work regulates the number of instantiated virtual machines.

There has also been work addressing the problem of control of Web applications [123, 124, 62]. Urgaonkar et al. [123] use queuing theory to analytically model a multi-tier Internet application, and use this model to determine how many servers are needed in each tier. One difference with our work is that they use a centralized control architecture, which may not be feasible in the cloud context when the provider and guests are separate business entities. Yaksha [62] does not perform resource provisioning, rather it performs admission control to improve a Web application's performance.

Data Rebalancing: Previous works have addressed the data rebalancing problem in a storage system [81, 8, 105]. In these works, rebalancing data is performed in a way that it does not cause any violations to the foreground application's SLOs. Aqueduct [81] uses a feedback controller that throttles its bandwidth usage to ensure that the quality of service of the foreground application is not affected while data transfers (e.g., backups) are performed in the storage system; and only unused bandwidth is used. If there is only limited unused bandwidth, then this approach can take a long time to complete; which may not be acceptable in the context of controllable elastic storage systems. For our controller, rebalancing data is not treated as an optional procedure, but as a required procedure to fix SLO violations.

Actuator Delays: Soundararajan et al. [111] address the issue of actuator delays for a different control problem. They present a control policy for database replication

on a static-sized cluster. Their controller waits for the replication process to complete before making a new decision. Our work addresses the problem of automated control of elastic storage systems while accounting for the delays brought by data rebalancing. Aside from waiting for the data rebalancing process to complete, our controller also finds the right balance between the time to completion and impact of data rebalancing. The use of proportional thresholding further distinguishes our work from [111].

Performance Differentiation for Storage Systems: There has been a long line of work that uses scheduling policies and admission control schemes to ensure performance guarantees and differentiation in storage systems [125, 44, 128]. For example, Jin et al. [60] present a share-based scheduling algorithm that enforces desired shares of resources for a storage service. Triage [64] uses a feedback controller to perform admission control by throttling client request rates to the storage system. These control schemes complement our work because we deal with allocating the right amount of resources to handle client workloads, while the aforementioned control schemes ensure that there is performance isolation between different sets of clients.

3.8 Conclusions

In this chapter, we presented *Elastore*, an automated controller for elastic storage systems in the context of cloud computing. The design addresses several challenges that exist due to the nature of the storage system and the cloud infrastructure. To address the issue of discrete actuators, our controller uses proportional thresholding to determine the size of the storage cluster. Moreover, the controller must treat data rebalancing as a first-class actuator. The controller uses a cost-optimization-based approach to determine the amount of bandwidth to use for rebalancing data as the cluster size grows or shrinks. A cost function is used to find the best tradeoff between the impact on the guest service and the time to completion of the rebalancing process.

Finally, the controller uses a state machine to coordinate between multiple actuators and to be robust to actuator delays.

We evaluated our controller using a Web 2.0 benchmark application running on an experimental testbed that consists of a variable number of Xen virtual machines instantiated from an inventory of physical servers. The experimental results confirmed that our controller is able to dynamically provision coarse-grained resources (i.e., virtual machines) under unanticipated changes in the client workload. Our rebalance policy balances the performance impact and time to fix SLO violations. Furthermore, our controller maintains client SLOs while being very resource efficient.

4

Optimization of Batch Data-Parallel Workflows

We now shift our focus to the management of one type of workloads found in the analytics tier: batch data-parallel workflows. Batch data-parallel workflows are composed of data-parallel MapReduce computations with producer-consumer relationships based on data that are run in batch fashion. While in the previous chapters, due to the simplicity of the workloads (i.e., get and put requests), workload management involves dynamic provisioning of resources, this chapter introduces management techniques for complex analytical workloads by optimizing the workload itself and controlling how they are processed by the underlying system. For the rest of this chapter, we describe the design and implementation of *Stubby*, which is a transformation-based optimizer for MapReduce workflows.

4.1 Introduction

Web clicks, social media, scientific experiments, and datacenter monitoring are among sources that generate large quantities of data every day. Rapid innovation and improvements in productivity necessitate timely and cost-effective analysis of this data. This trend is fueling a massive increase in workloads composed of workflows of data-

parallel jobs. The jobs are connected among each other through producer-consumer relationships specified by the workflow. MapReduce systems like *Hadoop* [47] and Google MapReduce [35] are now popular choices to run these workflows.

Automatic optimization of these MapReduce workflows is important as well as challenging. The use of data-intensive workflows is growing beyond large Web companies to those with few MapReduce tuning experts. Furthermore, with MapReduce systems being relatively young and evolving rapidly, it is hard to find experienced programmers and administrators to develop and run efficient MapReduce workflows. Recent studies show the order of magnitude performance gap that exists between optimized and unoptimized versions of MapReduce workflows [51, 130].

As an example, consider the MapReduce workflow shown in Figure 4.1 which is derived from a realistic business report generation application. (We will use this workflow as a running example.) It was convenient for the developer to express the report generation application as a workflow of seven MapReduce jobs. Optimization techniques that we introduce in this chapter can automatically convert this seven-job workflow into an equivalent, but highly optimized, two-job workflow. The performance gains are quite dramatic.

The central contribution of this chapter is an automatic cost-based optimizer, called *Stubby*[1], for MapReduce workflows. Stubby considers multiple optimization types that can be composed together, generating a large plan space for a MapReduce workflow W. One optimization type is called *vertical packing* where map and reduce functions from jobs in producer-consumer relationships in W are combined. Vertical packing produces new jobs that avoid the local and network I/O due to shuffling of data between the map and reduce phases of MapReduce execution [35]. For example, vertical packing can be applied to the jobs J5 and J7 in Figure 4.1, replacing these

[1] The name Stubby (meaning short and stocky) comes from the fact that our workflow optimizer makes workflows shorter (packing multiple jobs into one job to reduce workflow height) and fatter (packing multiple parallel function pipelines into one job).

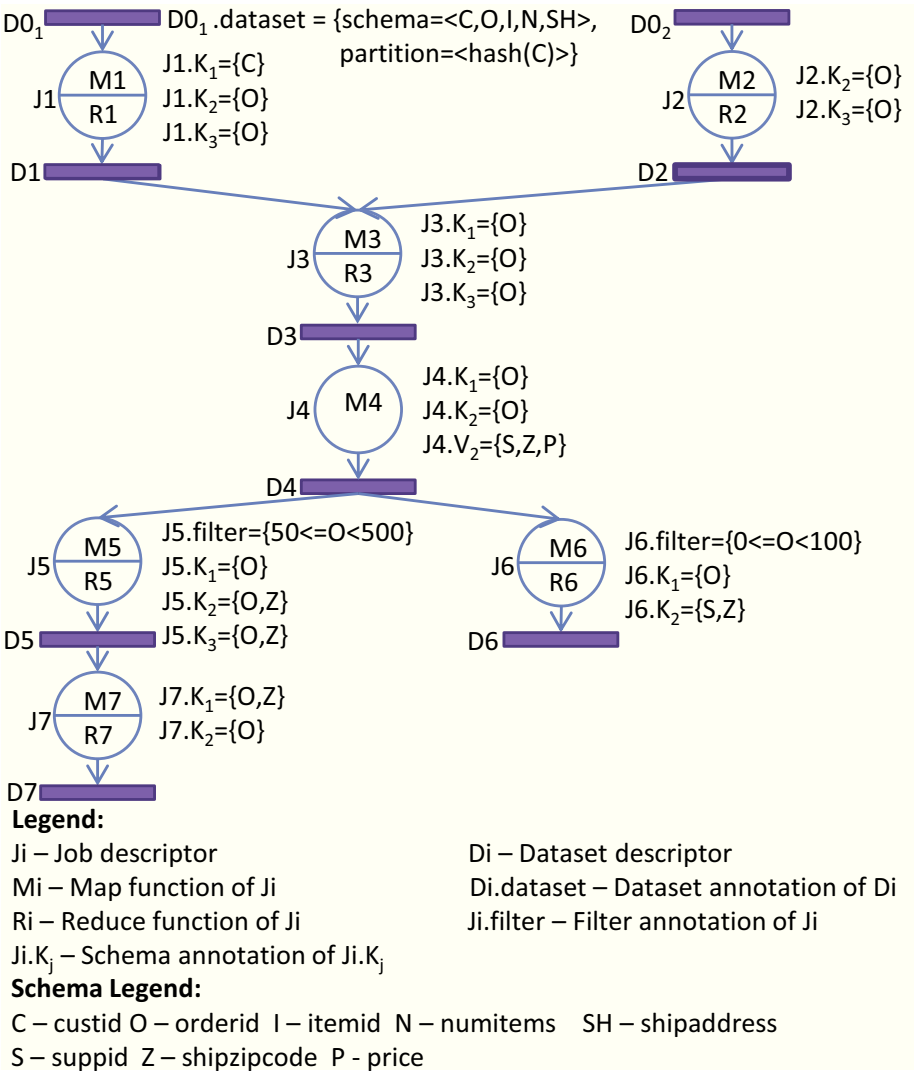


FIGURE 4.1: An example MapReduce job workflow and its annotations (known information) given to Stubby for optimization.

two jobs with a single job whose reduce function is a combination of J5's reduce function R5, J7's map function M7, and J7's reduce function R7.

Another optimization type is called *horizontal packing* which combines map and reduce functions so that jobs processing the same (large) dataset d can share the read I/O incurred for d [39, 87, 127]. Other optimization types include choices for the partition function of MapReduce jobs, data layouts (e.g., partitioning and compression) of intermediate data read and written by MapReduce jobs, the degree of

parallelism to use while running map (reduce) functions as concurrent map (reduce) *tasks*, and many others.

Developing a cost-based optimizer for practical MapReduce workflows poses three nontrivial challenges which we respectively refer to as the *plan spectrum*, *interface spectrum*, and *information spectrum*. The plan spectrum refers to the large and high-dimensional space of possible plans to run a given workflow.

The interface spectrum refers to the many possible ways in which a MapReduce workflow W can be generated in practice. A user could have generated W by writing the map and reduce functions in some programming language for each job in W. W could have been generated by query-based interfaces like *Pig* or *Hive* that convert queries specified in some higher-level language to a MapReduce workflow. W could have been generated by program-based interfaces like *Cascading* or *FlumeJava* that integrate workflow definitions into popular programming languages [23].

Furthermore, W could have been generated by composing multiple smaller workflows developed independently [89]. For example, it is natural to generate the workflow in Figure 4.1 by composing two individual workflows. One component workflow comprises jobs J1, J2, and J3, possibly written in Java, for cleaning and transforming data snapshots taken periodically from OLTP databases. The second component workflow comprises jobs J4, J5, J6, and J7 that are generated from a Pig query that computes various aggregates for report generation. Tools like *Oozie* and *Amazon Elastic MapReduce Job Flow* provide interfaces for such flexible development of MapReduce workflows [89].

The information spectrum refers to a problem endemic to MapReduce systems: the information needed to enumerate or to cost alternate plans considered by an optimization type may not always be available. For example, it is common in MapReduce systems to interpret data (lazily) at processing time, rather than (eagerly) at loading time. Hence, properties of the workflow's input data (e.g., schema, partitioning) may

not be known. Lack of such information can make some vertical packing optimizations inapplicable because their correctness cannot be guaranteed. It is also common to have MapReduce programs or user-defined functions written in languages like Java, Python, and Ruby; effectively requiring the optimizer to deal with black-box jobs in workflows. Statistics such as selectivity estimates or processing costs could also be unavailable.

4.1.1 Contributions and Roadmap

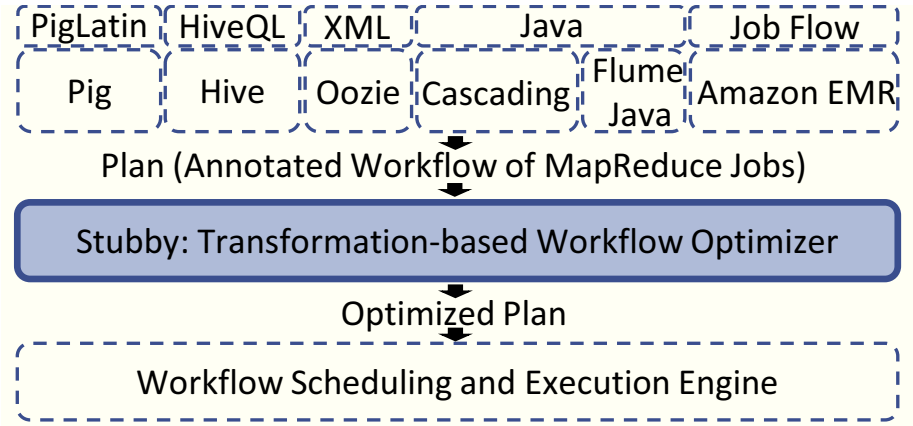


FIGURE 4.2: Stubby in the MapReduce execution stack.

Stubby has been designed to address the challenges posed by the plan, interface, and information spectrums. Figure 4.2 shows how Stubby fits in a MapReduce system. Different interfaces can be used to generate the MapReduce workflow given to Stubby for optimization.

Stubby accepts input in the form of an *annotated MapReduce workflow*—which we call a *plan*—and returns an equivalent, but optimized, plan. Annotations are a generic mechanism for workflow generators to convey useful information found during workflow generation. Stubby will find the best plan subject to the given annotations, while working correctly (but not optimally) when zero to few annotations are given. Stubby is also compatible with optimizations that the workflow generator may do,

e.g., projection pushdown or join ordering [39, 88, 130].

We designed Stubby as a transformation-based optimizer. A *transformation* is defined by a set of preconditions and postconditions: If the preconditions hold on a plan P^-, then Stubby can generate a plan P^+ on which the postconditions hold such that P^- and P^+ will produce the same result. However, P^- and P^+ may have different estimated costs and actual performance. The set of conditions—where a condition may refer to one or more annotations—is a succinct way of capturing the information needed for each optimization type. The combination of transformations and annotations gives Stubby some attractive features to deal with the information and interface spectrums:

- Stubby can search selectively through the subspace of the full plan space that can be enumerated correctly and costed based on the information available in any given setting.

- Stubby's core optimizer-level components for plan enumeration, search, and costing are reusable across different interfaces used to generate MapReduce workflows. Adding a new interface mainly requires writing a component to generate the respective annotations for workflows coming from that interface.

- Similar to extensible optimizers like *EXODUS* [42] developed for database systems, Stubby allows new transformations to be added to extend the optimizer's functionality easily.

The current set of transformations supported by Stubby is described in Section 4.3. Section 4.4 will then discuss how Stubby addresses the plan spectrum challenge through a novel enumeration and search algorithm. Section 4.5 describes how plan costs are estimated. Stubby has been prototyped fully and Section 4.7 describes a

comprehensive evaluation. Notably, we compare Stubby with a baseline that represents how an industrial-strength system (Pig) is used in production today. Stubby consistently outperforms the baseline by 2-4.5X.

4.2 Overview

4.2.1 MapReduce Workflows

A MapReduce workflow W is a *Directed Acyclic Graph (DAG)* G_W that represents a set of MapReduce jobs and their producer-consumer relationships. Each vertex in G_W is either a MapReduce job J or a dataset D. Each edge in G_W is between a job (vertex) J and a dataset (vertex) D, and denotes whether D is an input or an output dataset of J.

Each MapReduce job J in G_W is of the form $J = \langle p, c, a \rangle$. Here, p represents the MapReduce program that is run as part of J. *Configuration* c controls how the program p will be executed as tasks during J's execution [51]. Details of the configuration are given in Section 4.3.5. Annotations a give any available information about the operation and execution of the program that is relevant for workflow optimization. Annotations are discussed in Section 4.2.2.

Each dataset D in G_W is of the form $D = \langle d, l, a \rangle$. Here, d represents the dataset's descriptor in the distributed file-system that forms the persistent storage layer of a MapReduce system. *Layout* l controls how D is laid out in the distributed file-system, including how the dataset is partitioned and/or compressed. Stubby currently has support for horizontal partitioning only. The annotations a in this case give any available information about D.

MapReduce Program: For the purposes of this chapter, a MapReduce program is specified by the following four functions [35].[2] All functions except map are optional.

[2] MapReduce implementations like Hadoop allow other functions to be specified, e.g., for parsing/splitting map inputs and secondary sorting of map outputs. Our implementation of Stubby for

K_1-K_3 and V_1-V_3 are the respective key and value types.

- *map* function: $map(K_1, V_1) \Rightarrow list(K_2, V_2)$. A map function invocation is made for every key-value pair $\langle K_1{=}k_1, V_1{=}v_1 \rangle$ in the input dataset. During job execution, the key-value pairs in the input are processed in parallel by a set of *map tasks*. The number of map tasks is determined by the job configuration [51].

- *reduce* function: $reduce(K_2, list(V_2)) \Rightarrow list(K_3, V_3)$. For each unique key $K_2{=}k$ in the map output key-value pairs, a reduce function invocation is made with the group of all values that have key $K_2{=}k$. The number of reduce tasks is determined by the job configuration [51].

- *combine* function: $combine(K_2, list(V_2)) \Rightarrow list(K_2, V_2)$. For any key $K_2{=}k$ in the map output key-value pairs, a combine function may optionally be invoked with two or more values associated with k. This function is used by map tasks to preaggregate map outputs to reduce I/O and network costs at the expense of additional compute cost. The invocation of this function can be turned on or off, and its granularity of invocation adjusted, by the job configuration [51].

- *partition* function: $partition(K_2) \Rightarrow$*partition descriptor*. This function is used to partition the map output key-value pairs among the reduce tasks. The default is hash partitioning on key K_2 along with sorting the map output key-value pairs on K_2 per partition so that pairs with the same value of K_2 are grouped together for each $reduce(K_2, list(V_2))$ function invocation. Range partitioning is an alternative to hash partitioning.

Hadoop supports these additional complexities to a fair extent. We omit the details in order to focus on the research contributions. For ease of exposition and without loss of generality, for any producer job J_p whose output is read by a consumer job J_c, we will assume that the key-value pairs output by J_p are input as is to J_c's map function.

4.2.2 Annotations

Annotation is the medium used in Stubby to represent and communicate information needed for the different optimization types applicable to a workflow W. Broadly speaking, annotations can be categorized based on whether they represent information about the (i) datasets in W, (ii) operations performed by the MapReduce programs in W, or (iii) the run-time execution of the programs in W. We will next describe the specific annotation types supported currently by Stubby under these three categories. Section 4.6 will describe how the annotations are generated.

Annotations for datasets: *Dataset annotations* expose information known about the datasets in a workflow. Physical design information is the most relevant and includes any known partitioning, ordering, compression, and file-level information for the data as stored on the distributed file-system. For example, the dataset annotation for the base dataset $D0_1$ in Figure 4.1 conveys to Stubby that $D0_1$ is hash partitioned on an attribute named "custid".

Annotations for programs: Stubby currently supports two types of annotations—*schema* and *filter*—to expose known properties of MapReduce programs that are otherwise black-boxes to Stubby. Schema annotations expose the composition of the key and value types—K_1-K_3 and V_1-V_3—in a MapReduce program. For example, a schema annotation in Figure 4.1 specifies key K_2 in job J5 as consisting of two fields: "orderid" and "shipzipcode". In addition, key K_2 in job J7 is the single field "orderid". Identical field names are used in schema annotations to indicate data that flows unchanged through different functions in MapReduce programs. This concept is defined formally in Section 4.3.1. Schema annotations can be accompanied by filter annotations to convey that a program uses as input only a subset of the dataset generated by its producer job in the workflow (e.g., see jobs J5 and J6 in Figure 4.1).

Annotations for program execution: *Profile annotations* expose statistical in-

formation about the run-time execution of a program. This information is useful to estimate the cost of running a program under different data layouts and job configurations. Based on our previous work on the *Starfish* system, we chose to expose two categories of information through profile annotations [51]: (i) Dataflow statistics capture the distribution of key-value pairs and bytes flowing through different phases of a MapReduce program execution; (ii) Cost statistics capture the distribution of execution time spent in different phases of a MapReduce program execution.

4.2.3 Problem Definition and Solution Approach

Given an initial plan P for a MapReduce workflow W—namely, the workflow DAG G_W and a set of annotations associated with W—the goal of Stubby is to automatically find a plan P_{opt} for W with minimum overall estimated execution cost. The space of possible plans for W is defined by transformations that can be applied to a plan. We categorize these transformations into: (i) intra-job vertical packing transformation, (ii) inter-job vertical packing transformation, (iii) horizontal packing transformation, (iv) partition function transformation, and (v) configuration transformation. Section 4.3 describes each transformation in terms of its preconditions, postconditions, and required annotations. Sections 4.4 and 4.5 describe Stubby's enumeration and search as well as plan costing techniques respectively.

For describing the transformations, we identify five subgraphs that characterize different types of producer-consumer relationships arising among jobs in the workflow DAG. These producer-consumer subgraphs are shown in Figure 4.3: *one-to-one*, *one-to-many*, *many-to-one*, *none-to-one*, and *one-to-none*.

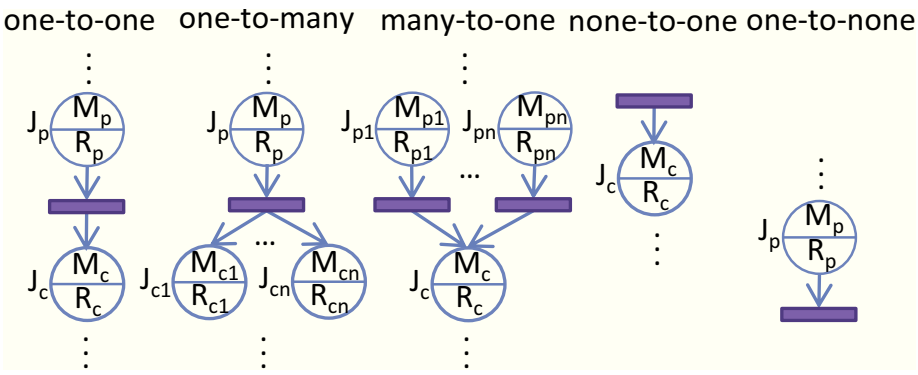


FIGURE 4.3: Five types of producer-consumer subgraphs that can arise in a workflow DAG (some combinations of these subgraphs can also arise).

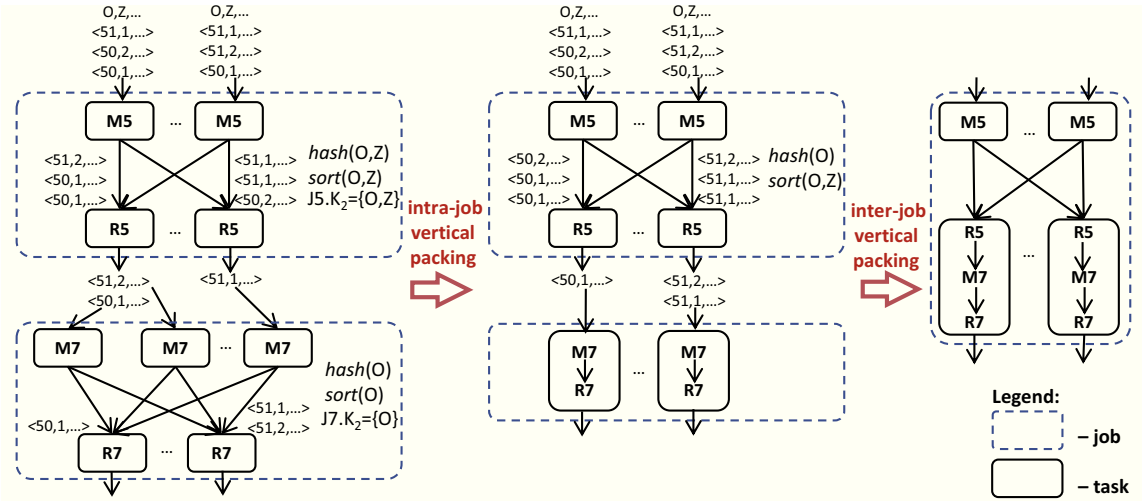


FIGURE 4.4: A task-level illustration of vertical packing transformations applied to the example workflow from Figure 4.1.

4.3 Transformations that Define the Plan Space

4.3.1 Intra-job Vertical Packing Transformation

An intra-job vertical packing transformation converts a MapReduce job into a Map-only job. Suppose M and R respectively denote the map and reduce functions of the job. Without the vertical packing transformation, M will be invoked in the job's map tasks, and R will be invoked in the job's reduce tasks. After the transformation, the

M and R functions will be pipelined together and invoked in the new job's map tasks. The data output by M will now be provided directly to R without going through the partition, sort, and shuffle phases of MapReduce job execution.

We will begin with an example of the transformation applied to our example MapReduce workflow from Figure 4.1. We will then specify formally the preconditions and postconditions for a common case where the transformation applies. This specification will be followed by a discussion of extended scenarios where Stubby will apply the transformation as well as a discussion of the performance implications. A similar presentation style will be used for all other transformations.

Figure 4.4 shows a task-level view of the one-to-one producer-consumer subgraph comprising jobs J5 and J7 from Figure 4.1. The plans shown respectively on the left hand side (denoted P^-) and the middle (denoted P^+) of Figure 4.4 are the plans before and after applying an intra-job vertical packing transformation to Job J7. Job J5's reduce function R5 needs its input key-value pairs grouped on J5.K_2=$\{O, Z\}$, and Job J7's reduce function R7 needs its input grouped on J7.K_2=$\{O\}$. As shown on the left side of Figure 4.4, plan P^- generates both groupings using MapReduce's default strategy: do hash partitioning of the respective map-output key-value pairs on K_2, and sort the pairs within each partition on K_2.

Plan P^+, on the other hand, generates the grouping needed in the producer job J5 differently: a hash partitioning is done on $\{O\}$, and a per-partition sort is done on the $\{O, Z\}$ combination. The nice property of this grouping technique is that it satisfies the grouping needs of both the producer job J5 and the consumer job J7. Consequently, there is no need to have the partition, sort, and shuffle phases in J7. J7's reduce function R7 can be moved to the map-side and invoked in the map tasks; as shown in plan P^+ in Figure 4.4. Effectively, P^+ is pipelining key-values pairs from M7 to R7.

Preconditions and Postconditions: Let us build on the intuition from the above

example to formalize the preconditions and postconditions for the intra-job vertical packing transformation. [3] Recall that if the preconditions hold on a plan P^-, then we can generate a plan P^+ on which the postconditions will hold such that P^- and P^+ will produce the same result. However, P^- and P^+ may have different performance. We will first consider one-to-one subgraphs and then present extensions.

Preconditions on plan P^- in intra-job vertical packing:

1. There is a one-to-one producer-consumer subgraph with producer job J_p and consumer job J_c.

2. The output key-value pairs of the map function M_c of J_c satisfy the following invariant: M_c can output a key-value pair with $J_c.K_2{=}k$ only from one or more key-value pairs with $J_c.K_2{=}k$ given as input to the reduce function R_p of J_p. These functions could, in turn, be pipelines of map, reduce, and combine functions due to previous applications of transformations.

Intuitively, the above conditions state that the data in the $J_c.K_2$ fields flows unchanged—allowing for filtering as well as addition or removal of duplicates—from the input of the producer job J_p's reduce function to the output of the consumer job J_c's map function. Stubby checks these conditions based on the schema annotations given in the workflow.

Postconditions on plan P^+ in intra-job vertical packing:

1. The partition function of J_p in the new P^+ will partition on $\{J_p.K_2 \cap J_c.K_2\}$ and sorts per partition on the combined sort key $\{J_p.K_2 \cap J_c.K_2, (J_p.K_2 \cup J_c.K_2) - (J_p.K_2 \cap J_c.K_2)\}$; which allows the partition function of J_p to satisfy the reduce-side grouping requirements of both J_p and J_c.

[3] A proof of the correctness of these conditions is given in the appendix of this chapter.

2. For any reduce task in job J_p, all key-value pairs output by that reduce task should be input in the same order to a single map task in job J_c. This requirement can be enforced by specifying a condition on the configuration (recall Section 4.2.1) of job J_c. Note that the map tasks in plan P^- are free to process subsets of key-value pairs output by one or more reduce tasks in job J_p.

Extensions: With some adjustments, the preconditions and postconditions given earlier for one-to-one subgraphs become applicable to none-to-one and many-to-one subgraphs and their hybrid combinations. For a none-to-one producer-consumer subgraph (e.g., at job J2 in Figure 4.1), the first postcondition effectively becomes a precondition that should hold on the job's input dataset. Recall that dataset annotations give the partitioning and ordering information required to check whether such conditions hold.

For a many-to-one subgraph (e.g., at job J3 in Figure 4.1), the second precondition should hold for each producer-consumer pair. The postconditions also need to be adjusted to have the same partitioning on $J_{p_i}.K_2$ for all producer jobs J_{p_i} so that all key-value pairs with $J_c.K_2{=}k$ can be input to a single map task in the consumer job J_c.

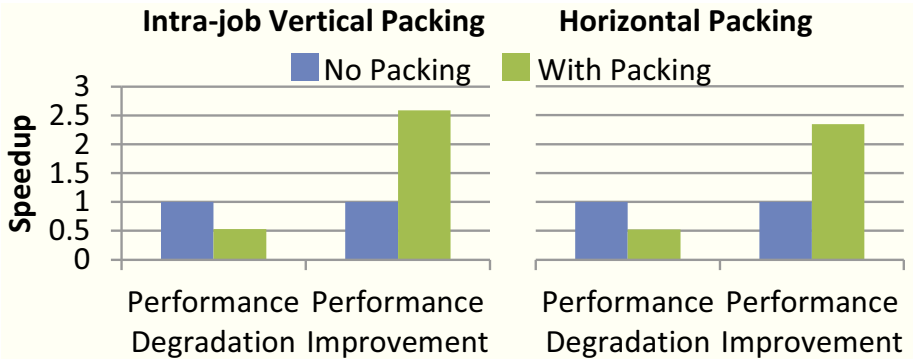


FIGURE 4.5: Performance degradation and improvement caused by vertical packing and horizontal packing transformations.

Performance Implications: The new plan P^+ produced by an intra-job vertical

packing transformation can perform better or worse than the old plan P^-; motivating a cost-based approach to decide whether to apply the transformation or not. For illustration, Figure 4.5 shows the actual performance with and without vertical packing for a none-to-one subgraph when we vary the properties of the input dataset. A 10-node Hadoop cluster on Amazon EC2 is used. (Further details of the experimental setup are given in Section 4.7.)

Figure 4.5 shows the speedup over the case of not applying the transformation. Note that, in one case, vertical packing leads to a 2.5X speedup. As expected, the performance gains from applying intra-job vertical packing come from eliminating the large overhead of moving the map output data to the reduce tasks: CPU cost for partitioning and sorting the data, I/O from writing and reading to local disk, as well as network transfer costs.

However, in the other case, vertical packing leads to a 0.5X degradation in performance. Interestingly, there are a number negative performance effects of vertical packing:

- A vertical packing creates a dependence between the configuration choices for the producer job J_p and consumer job J_c, reducing the degrees of freedom in choosing the best plan. The degree of map-side parallelism in J_c is now dependent on the reduce-side parallelism in J_p due to the second postcondition.

- Note that, for job J5 in Figure 4.4, the application of intra-job vertical packing led to a choice of partitioning on $\{O\}$ in P^+, whereas P^- partitions on the $\{O, Z\}$ combination. It is possible that attribute $\{O\}$ has few unique values in the data—one in the worst case—but the $\{O, Z\}$ combination has many unique values. In this case, vertical packing can lead to significant performance degradation by limiting the parallelism in P^+.

- In popular MapReduce implementations like Hadoop, map and reduce tasks are

run in *task slots* that usually have preconfigured resources (e.g., heap memory). Thus, packing more functions to run in the same task has the potential to cause suboptimal resource usage in one of two ways: (i) resource contention from executing more functions per task slot, and (ii) resource under-utilization from using fewer task slots than what is available.

These issues have to be taken into account during plan costing in order to ensure that vertical packing is considered in a comprehensive cost-based fashion.

4.3.2 Inter-job Vertical Packing Transformation

An inter-job vertical packing transformation moves functions from a job J into another job, completely eliminating the need for J. The example workflow in Figure 4.1 shows multiple opportunities for this transformation. For example, since J4's map function M4 is invoked for every key-value pair output by job J3, and does not require any grouping, M4 can be pipelined after J3's reduce function; eliminating reads and writes for the dataset D3. Moreover, a previously-transformed job can be further transformed as shown on the right side of Figure 4.4.

Preconditions and Postconditions: Under the following preconditions, the functions in a Map-only job can be moved to another job as part of an inter-job vertical packing transformation:

1. There is a one-to-one producer-consumer subgraph with (only) one producer job J_p and (only) one consumer job J_c.

2. One of J_p or J_c is a Map-only job.

Extensions: Multiple choices exist to apply this transformation to a one-to-many producer-consumer subgraph. For example, consider a Map-only producer job J_p: (i) The functions of J_p can be replicated and packed with the functions in the map

task of each consumer job; or (ii) J_p and one of the consumer jobs can be packed into a single job, while ensuring that J_p's original output dataset is still generated (materialized to disk) for the other consumer jobs.

Performance Implications: Similar to intra-job vertical packing, this transformation can have positive or negative performance implications. The performance gains from applying inter-job vertical packing come from eliminating disk and network I/O as well as the overhead of setting up and cleaning up additional map tasks. However, most negative performance effects of intra-job vertical packing apply here as well. If one of the MapReduce jobs has to be run as a single task (e.g., a top-K computation), then an inter-job vertical packing transformation can cause the entire computation to run as a single task; giving extremely poor performance.

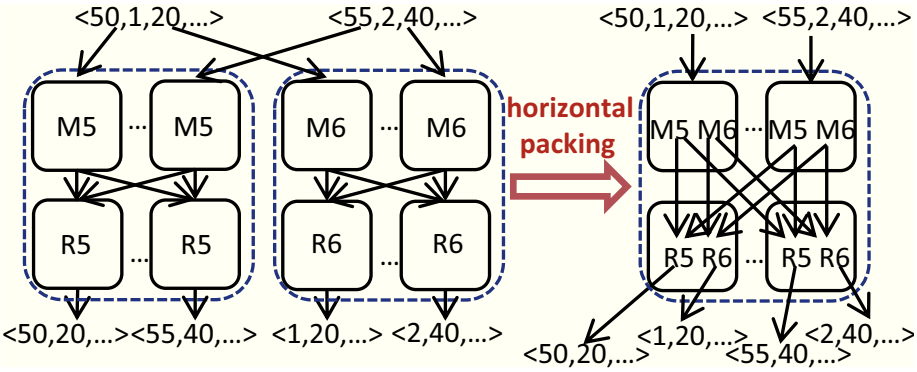


FIGURE 4.6: A task-level illustration of horizontal packing applied on jobs J5 and J6 of the example workflow (refer to Figure 4.1).

4.3.3 Horizontal Packing Transformation

A horizontal packing transformation packs the map (reduce) functions of multiple jobs that read the same dataset into the same map (reduce) task of a transformed job. Jobs J5 and J6 of the example workflow in Figure 4.1 read the same dataset D4. Figure 4.6 shows a task-level view of packing J5 and J6 into a single job.

While vertical packing transformations pipeline functions sequentially, a horizontal packing transformation puts multiple map (reduce) functions from separate parallel pipelines into a single job's map (reduce) task. An input key-value pair $\langle K_1,V_1\rangle$ will go through all pipelines in the map task, and each pipeline will generate its own $\langle K_2, list(V_2)\rangle$ outputs. In the reduce task, each $\langle K_2, V_2\rangle$ pair will only go through the pipeline that corresponds to the map function that generated the pair.

Preconditions and Postconditions: The easy precondition for applying a horizontal packing transformation is that two or more jobs should have the same input dataset, e.g., in a one-to-many producer-consumer subgraph [39, 87].

Extensions: The precondition of reading the same input dataset can be relaxed so that a horizontal packing transformation can be applied to any set of concurrently-runnable jobs, e.g., jobs J1 and J2 in our example workflow. The only additional requirement is to ensure that the map functions in separate parallel pipelines only process key-value pairs from the respective input datasets of these functions. In conjunction with the vertical packing transformations, such an extended horizontal packing transformation can transform jobs J1, J2, and J3 of our example workflow into a single job.

Performance Implications: Figure 4.5 shows that horizontal packing transformations can lead either to performance gain or to performance degradation. Both experimental results are from a 10-node Hadoop cluster on Amazon EC2. The workflow used has two consumer jobs that perform filtering, grouping, and aggregation on an input dataset. A very large input dataset is used in one case and a smaller dataset in the other.

On the positive side, horizontal packing transformations can improve performance by eliminating local-disk and network I/O from reading the input dataset multiple times. On the negative side:

- A horizontally-packed job essentially runs all individual jobs with the same configuration. This dependence can cause performance issues. For example, the performance degradation for the smaller dataset in Figure 4.5 was because the cluster had enough resources to run all consumer jobs concurrently and most efficiently; resulting in better performance than when running a single horizontally-packed job. Furthermore, packing multiple functions in parallel per task can cause issues such as excessive spilling of key-value pairs to local disk due to the concurrent memory overheads [39].

- Depending on the selectivity of the map functions, the extra overhead in the packed job from partitioning and sorting the combined map-output data from all individual jobs may outweigh the performance gains from read sharing [87].

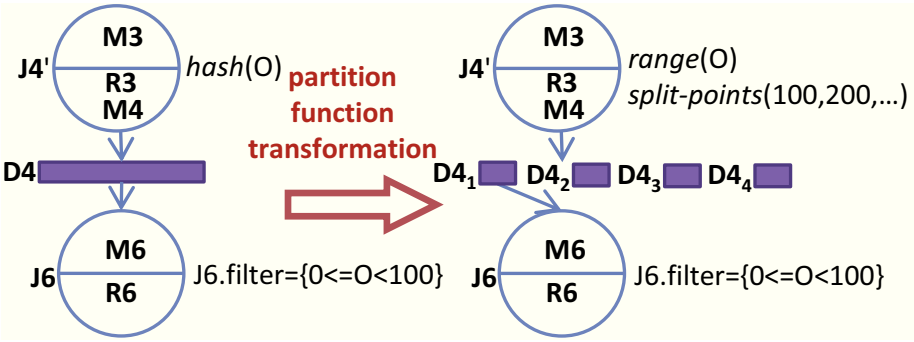


FIGURE 4.7: An illustration of partition function transformation applied on job J4' that transforms the partition function to range partitioning, which enables partition pruning on job J6.

4.3.4 Partition Function Transformation

Partition function transformation changes how the map output key-value pairs are partitioned and sorted during the execution of a job. This transformation includes, but is not limited to: (i) changing the partitioning type (default is hash), (ii) changing the splitting points for range partitioning, and (iii) changing the fields on which

per-partition sorting happens (default is K_2). For example, in Figure 4.7, this transformation changes the partition function of job J4' from using hash partitioning to range partitioning. (Note that J4' is itself a transformed job that was generated by an inter-job vertical packing of jobs J3 and J4 of the example workflow.)

Preconditions and Postconditions: There are no preconditions for a partition function transformation on a job J. The new partition function for J in plan P^+ should satisfy all current conditions on the partition function for J in P^-. For example, note that the application of an intra-job vertical packing transformation will place a postcondition on the partition function of the producer job. Furthermore, the MapReduce workflow given to Stubby could have some initial conditions already imposed on a job's partition function. For example, a MapReduce job for sorting an input dataset will need to use range partitioning.

Performance Implications: Partition function transformation can improve the performance of a job. First, the correct choice of partition function can decrease data skew in the reduce tasks within a single job. When the profile annotation for a job provides the data distribution of map-output key-value pairs, range partitioning with good splitting points can be chosen instead of hash partitioning to ensure that data is distributed evenly across all reduce tasks.

Second, the partition function of a producer job J_p affects the layout of its output dataset. Thus, adjusting the partition function's splitting points based on any filter annotations provided for a consumer job J_c will enable *partition pruning* in J_c. With partition pruning, J_c will only read the partitions of J_p's output dataset that are relevant to J_c; saving on local and network I/O.

For example, consider job J6 in our example workflow (see Figure 4.7). J6 discards all input key-value pairs with orderid ≥ 100 (exposed through the filter annotation). Thus, the partition function of J4' can be transformed to range partitioning (e.g.,

in ranges of 100) so that J6's input data descriptor can be set to be the partition(s) containing the output of J4' with $0 \leq$ orderid < 100.

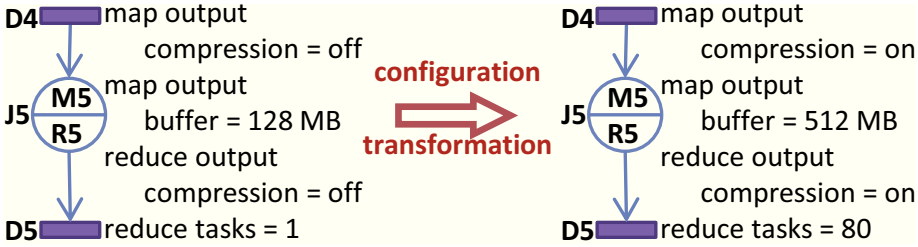


FIGURE 4.8: An illustration of configuration transformation applied on job J5 of the example workflow.

4.3.5 Configuration Transformation

A configuration transformation changes the configuration of a MapReduce job in a workflow. Figure 4.8 shows an example of this transformation applied on job J5. Here, J5 is transformed to use 80 reduce tasks, a map output buffer size (for two-phase sorting) of 512 MB, and compression is turned on for the map and reduce output key-value pairs (in turn, affecting dataset D5's layout).

Preconditions and Postconditions: There are no preconditions for a configuration transformation on a job J. The new configuration for J in plan P^+ should satisfy all current conditions on the configuration for J in P^-. For example, recall from Section 4.3.1 that the application of an intra-job vertical packing transformation will place a condition on the configuration of the consumer job.

Performance Implications: As observed in [51], the configuration space for a MapReduce job is large and high-dimensional. In Hadoop, for example, a job's performance is controlled by the settings of dozens of parameters such as those shown in Figure 4.8. The respective performance impacts of these parameters are correlated and vary based on the properties the MapReduce program, input datasets, and cluster resources. Furthermore, the configuration transformation applied on a

producer job J not only affects J's performance, but also the performance of the consumer jobs that read J's output. Thus, nontrivial cost-based decisions have to be made in order to pick the best configurations for jobs in a workflow.

4.4 Search Strategy

Given a plan P (i.e., an annotated MapReduce workflow W), Stubby's goal is to find the sequence of valid transformations to apply to P in order to generate an equivalent plan P' that minimizes the overall execution time of W. Different sequences of transformations can generate very different plans. For example, consider the MapReduce workflow in Figure 4.1. One option is to apply the intra-job vertical packing transformation on job J7, followed by the inter-job vertical packing transformation, in order to pack jobs J5 and J7 into a single job (as shown in Figure 4.4). Alternatively, we can apply the horizontal packing transformation on jobs J5 and J6 to generate a different packed job, as shown in Figure 4.6. The *Plan Space* S_P for plan P consists of all valid alternative plans for P generated by applying combinations of transformations to P.

Workflow Optimization Process: One approach to optimize a plan P is to apply enumeration and search techniques to the full plan space S_P. However, the large size of S_P renders this approach impractical. More efficient search techniques can be developed based on two key insights. The first insight comes from how transformations interact with each other. In theory, a decision to apply any transformation on a particular job in P can influence the choice of a transformation on any other job in the same plan. However, in practice—primarily due to the semantics and implementation of the MapReduce programming model—arbitrary interactions among transformations across multiple jobs are uncommon. Consider again the example workflow from Figure 4.1. The decision to apply an inter-job vertical packing transformation on jobs J3 and J4 does not affect the transformations that are applicable

to job J7; therefore, these decisions can be made independently.

Thus, we follow a *divide-and-conquer* approach: P is divided into (possibly overlapping) subplans, denoted $P^{(i)}$, with smaller plan subspaces $S_P^{(i)}$ such that the globally-optimal choice in S_P can be found by composing the optimal choices found for each $S_P^{(i)}$. Each $P^{(i)}$, along with the corresponding $S_P^{(i)}$, defines an *Optimization Unit* $U^{(i)}$. The idea behind an optimization unit is to bring together a set of related decisions that affect each other, but are independent of the decisions made at other optimization units. In other words, the goal is to break the large plan space S_P into independent subspaces $S_P^{(i)}$ such that $S_P = \cup S_P^{(i)}$. Within each $U^{(i)}$, Stubby is responsible for enumerating and evaluating the different transformations applicable to the jobs in $U^{(i)}$.

The second key insight is that the order of applying transformations is important if we prefer to avoid expensive backtracking techniques. Applying a transformation may enable the use of another transformation (e.g., an intra-job vertical packing transformation on job J7 enables an inter-job vertical packing between J5 and the new J7' to eliminate one entire job) or it may prevent it (e.g., a horizontal packing transformation on jobs J5 and J6 prevents an intra-job vertical packing transformation on job J7). Therefore, it is essential to guide the search efficiently towards a sequence of transformations that can lead to near-optimal execution plans.

We organize transformations in two (overlapping) groups. The first group, termed *Vertical*, focuses on applying intra- and inter-job vertical packing transformations. The second group, termed *Horizontal*, focuses on applying the horizontal packing transformation. The aforementioned transformations are unique in the sense that, once applied, they change the structure of the workflow graph. On the other hand, the partition function and configuration transformations do not change the graph structure. These two transformations are included in both the Vertical and the

Horizontal groups.

The Vertical transformations are applied within all optimization units before the Horizontal transformations are considered. This ordering stems from two observations. First, for the new horizontally-packed job, the horizontal packing transformation creates a map-output key K_2 that combines the K_2 keys from the original jobs. This new, and possibly complex, key can prevent the application of vertical packing transformations on succeeding jobs. Following our running example from Figure 4.1, applying horizontal packing to jobs J5 and J6 will prevent using intra-job vertical packing on job J7 because the preconditions can no longer be met. Second, intra- and inter-job vertical packing transformations can potentially bring higher benefits as they eliminate entire shuffle steps as well as writing and reading intermediate data between jobs. On the other hand, horizontal packing transformations can only reduce the amount of data read through scan sharing.

Overall, Stubby's optimization process is as follows:

1. Generate the first optimization unit consisting of one or more jobs in the MapReduce workflow graph G_W (described in Section 4.4.1).

2. Enumerate and search within an optimization unit U using the Vertical transformations in order to find the (near) optimal subplan for U (described in Section 4.4.2). These transformations may alter the structure of the subgraph in U.

3. Dynamically generate the next optimization unit in G_W in topological sort order, and apply Step 2.

4. Repeat Step 3 until the entire graph G_W is covered.

5. Repeat Steps 1-4 using the Horizontal transformations to find the overall (near) optimal execution plan for W.

4.4.1 Dynamic Generation of Optimization Units

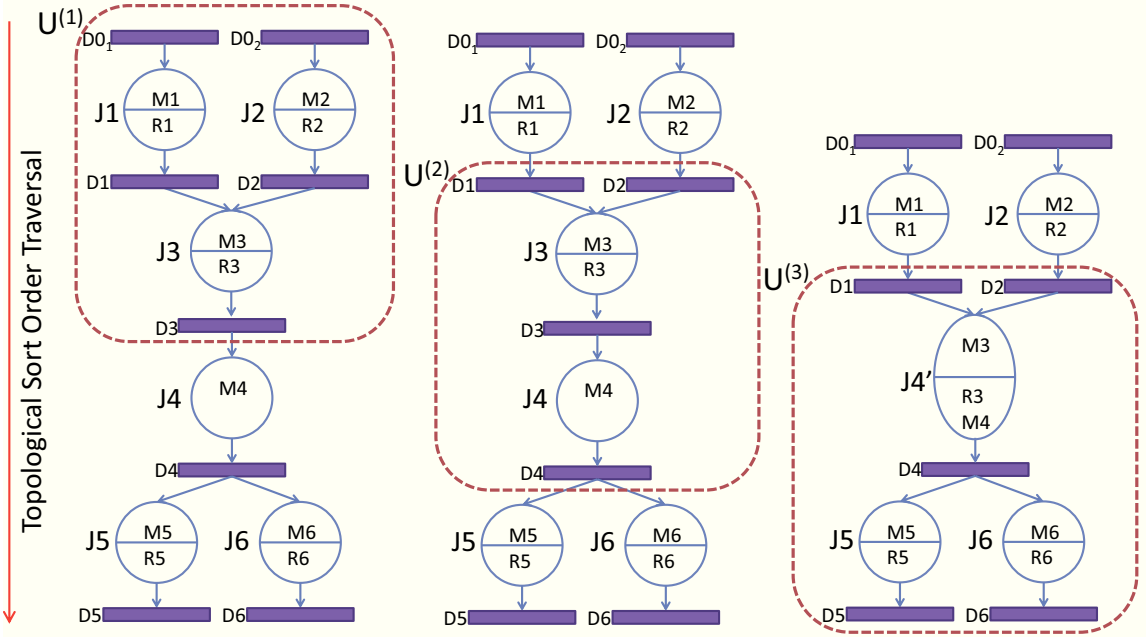


FIGURE 4.9: An illustration of Stubby's dynamic generation of optimization units as it traverses the example workflow graph.

Stubby builds the optimization units dynamically based on the following observation: when two jobs J_i and J_k are separated by one or more jobs in the workflow graph (i.e., the dependency path between J_i and J_k contains at least one other job), then the effect of J_i on the execution of J_k diminishes rapidly in practical settings. Hence, decisions for J_i can be made independently from decisions made for J_k. For example, the choice for applying inter-job vertical transformation on jobs J3 and J4 in our example workflow from Figure 4.1, will not affect the choice for using an intra-job vertical transformation on job J7.

Each optimization unit $U^{(i)}$ consists of a set of concurrently-runnable producer jobs and the corresponding set of consumer jobs. Figure 4.9 offers a pictorial representation of the optimization units. The first optimization unit $U^{(1)}$ (denoted by a dotted box in Figure 4.9) consists of the producer jobs J1 and J2 as well as the

consumer job J3. The plan space $S_W^{(1)}$ contains the subplans formed by all valid combinations of transformations that can be applied on jobs J1, J2, and J3.

Applying transformations within an optimization unit may alter the structure of the graph. As an example, suppose only configuration transformations are beneficial to reduce the total running time of the jobs in $U^{(1)}$. In this case, the structure of the graph remains unchanged. Since Stubby traverses the graph in topological sort order, the next optimization unit $U^{(2)}$ will be generated with J3 as the producer job and J4 as the consumer job (see Figure 4.9). Now suppose that the best transformation to apply is inter-job vertical packing to job J4. This transformation will replace jobs J3 and J4 with a new job J4'. The next optimization unit $U^{(3)}$ will consist of the new producer job J4' and the consumer jobs J5 and J6.

4.4.2 Search Within an Optimization Unit

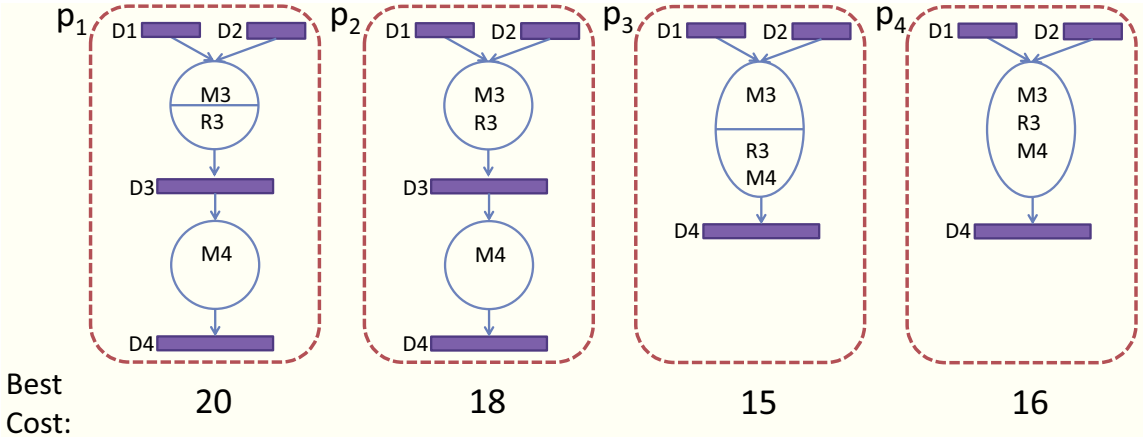


FIGURE 4.10: Enumeration of all valid transformations for optimization unit $U^{(2)}$ from Figure 4.9. The corresponding best estimated cost (running time) from RRS invocation is also shown.

For each optimization unit $U^{(i)}$, Stubby must find the subplan from $S_W^{(i)}$ that minimizes the total running time of the MapReduce jobs in $U^{(i)}$. Stubby addresses this problem by generating alternative valid subplans using transformations selected through an enumeration and search over $S_W^{(i)}$.

The number of jobs within any individual optimization unit $U^{(i)}$ is typically small. We observed that applying all combinations of transformations apart from the configuration transformation within $U^{(i)}$ usually results in a small number of unique subplans. Therefore, Stubby exhaustively applies all transformations, except the configuration transformation, in order to generate all possible subplans p_1–p_n for $U^{(i)}$. For example, as illustrated in Figure 4.10, this exhaustive enumeration for optimization unit $U^{(2)}$ from Figure 4.9, generates only four alternative subplans p_1–p_4.

Configuration transformations are applied on the jobs in each generated subplan p_i. These transformations can change any of the numerous MapReduce job configuration parameter settings including the number of map and reduce tasks, memory allocation settings, controls for I/O and network usage, and others [51]. In order to search the large and high-dimensional space of configuration transformations efficiently, Stubby uses *Recursive Random Search (RRS)*. RRS is a fairly recent technique developed to solve black-box optimization problems [132].

RRS first samples the configuration space randomly in order to identify promising regions that contain the optimal configuration setting with high probability. It then samples recursively in these regions which either move or shrink gradually to locally-optimal settings based on the samples collected. RRS then restarts random sampling in order to find a more promising region to repeat the recursive search. Each transformed subplan generated for p_i through RRS is associated with an estimated execution cost (see Section 4.5). The output of RRS for p_i is the configuration transformation that leads to the subplan $p_i^{(opt)}$ with the lowest estimated cost for p_i. After RRS has been invoked for all the subplans p_1–p_n in the optimization unit $U^{(i)}$, Stubby will select the $p_i^{(opt)}$ with the overall lowest estimated cost as the best subplan for $U^{(i)}$.

Consider the example in Figure 4.10 which shows the four subplans p_1–p_4 for the optimization unit $U^{(2)}$ from Figure 4.9. RRS will be invoked four times for $U^{(2)}$ in order to find the best configuration transformation and estimated cost for each p_i. When the RRS invocations complete, Stubby will choose to retain subplan p_3 from Figure 4.10 which has the lowest estimated cost among p_1–p_4. Note that p_3 was generated by applying the inter-job vertical packing transformation on job J4.

Overall Optimization Process: In summary, Stubby uses a *two-phase greedy* enumeration and search strategy. In each phase, Stubby generates optimization units dynamically while traversing the workflow graph in topological sort order. In the first phase, the producer jobs in each optimization unit $U^{(i)}$ are optimized by applying transformations from the Vertical group. At the end of the optimization process within $U^{(i)}$, (only) the best subplan for $U^{(i)}$ is retained by applying the corresponding transformations to the jobs in $U^{(i)}$. After the entire graph is traversed once, the above process is repeated once more. However, in this second phase, transformations from the Horizontal group are applied. The fully-optimized workflow graph is ready when the second traversal completes.

4.5 Plan Costing

For each annotated MapReduce workflow W that is generated during the enumeration and search strategy described in Section 4.4, Stubby must estimate the execution cost of W. Stubby uses Starfish's *What-if Engine* for this purpose [51]. The Starfish What-if Engine is given four inputs:

1. The dataflow and cost statistics of each job in W (recall the profile annotations discussed in Section 4.2.2).

2. The configuration to run each job in W with (chosen by RRS).

3. The size and layout information for W's input datasets (recall the dataflow annotations discussed in Section 4.2.2).

4. The cluster setup and resource allocation that will be used to run W. This information includes the number of nodes and network topology of the cluster, the number of map and reduce task slots per node, and the memory available for each task execution.

The Starfish What-if Engine uses these inputs and a mix of analytical, black-box, and simulation models to reason about the impact of configuration settings, data properties, and cluster resource properties on the execution of each MapReduce job J in W. The What-if Engine will then output cost estimates for each job as well as the entire workflow. Because of space constraints, we refer the reader to [51] for a detailed description of the Starfish What-if Engine. If any of the inputs required to use the What-if Engine are unavailable—e.g., profile or dataset annotations are not provided in the workflow—then the cost estimation will have to fall back to a simpler cost model such as the number of jobs as used in [70].

One challenge while using the Starfish What-if Engine is that Stubby's vertical and horizontal packing transformations change the jobs in W. For example, the intra-job vertical packing transformation will change a MapReduce job into a Map-only job. Thus, the packing transformations have to generate new annotations—in particular, the dataflow and cost statistics—for the new jobs that they generate. This process is called *adjustment* in Stubby since the new annotations are generated by modifying the old ones.

Space constraints preclude the discussion of all adjustments. The adjustments that Stubby uses for profile annotations are motivated by cardinality estimation techniques used in database systems. For instance, during an intra-job vertical packing transformation, the reduce function is moved into the map task and is executed after

the map function. The new map-task record selectivity[4] is calculated as the product of the record selectivities of the old map and reduce functions. On the other hand, the CPU cost of the new map task is calculated as the sum of the CPU costs of the old functions.

4.6 Implementation

We have implemented Stubby as a standalone system that can be employed by the many interfaces used to generate MapReduce workflows, as shown in Figure 4.2. To this extent, we have added a new feature in Apache Pig [98] for exporting and importing annotated MapReduce workflows used by Stubby. Pig was only a choice of convenience; our work applies to arbitrary MapReduce workflows.

Annotations: As described in Section 4.3, some transformations in Stubby require additional information which is expressed as annotations. We have made some minor modifications to the compilation process in Pig—which translates a Pig Latin query to a MapReduce workflow—to automatically extract any available schema, filter, and dataset annotations. The details are given in the Appendix. For example, the composition of the key and value types in a MapReduce job are extracted based on any schema information included in the Pig Latin query. Filter annotations are generated based on any filter statements contained in the query. We generate profile annotations using Starfish's *Profiler* which collects profiles through dynamic instrumentation of unmodified MapReduce workflows [51].

Transformations and Execution: Recall from Section 4.3 that vertical packing transformations chain multiple functions together for execution within the same map or reduce task. Similarly, horizontal packing transformations bring multiple independent functions into the same task. These transformations require the use of wrapper

[4] Record selectivity is defined as the ratio of the number of output key-value pairs over the number of input key-value pairs.

MapReduce classes to execute multiple functions inside a map or a reduce task. In addition, horizontal packing needs a tagging mechanism for guiding the data correctly through the different function pipelines. The Pig execution engine already offered support for wrapper classes and tagging, so only minor modifications had to be made in order to execute Stubby-generated plans correctly.

4.7 Experimental Evaluation

In our experimental evaluation, we used a Hadoop cluster running on 51 Amazon EC2 nodes of the m1.large type. Each node has 7.5 GB memory, 2 virtual cores, 850 GB local storage, and is set to run at most 3 map tasks and 2 reduce tasks concurrently. Thus, the cluster can run at most 150 map tasks in a concurrent map wave, and at most 100 reduce tasks in a concurrent reduce wave.

For evaluation, we selected representative MapReduce workflows from several application domains. These MapReduce workflows are listed in Table 4.1 and described in detail in Section 4.7.1. All workflows are expressed in Pig Latin and executed using the Pig execution engine running on Hadoop.

For comparison purposes, we established a *Baseline* that represents how an industrial-strength system (Pig) is used in production today. In particular, we enabled all (rule-based) optimizations supported by Pig and manually-tuned the configuration parameter settings using rules-of-thumb found in [30].

Our evaluation methodology is as follows:

1. We present the overall performance improvements achieved by Stubby, as well as the performance improvements observed when only a subset of the plan space is considered (Section 4.7.2).

2. We compare the performance benefits from Stubby against other state-of-the-art techniques (Section 4.7.3).

Table 4.1: MapReduce workflows and corresponding data sizes.

Abbr.	Workflow	Dataset Size
IR	Information Retrieval	264 GB
SN	Social Network Analysis	267 GB
LA	Log Analysis	500 GB
WG	Web Graph Analysis	255 GB
BA	Business Analytics Query	550 GB
BR	Business Report Generation	530 GB
PJ	Post-processing Jobs	10 GB
US	User-defined Logical Splits	530 GB

3. We evaluate the efficiency of Stubby in terms of its overheads while optimizing MapReduce workflows (Section 4.7.4).

4. We provide a closer look at how Stubby works within an optimization unit to enumerate and find the best transformations (Section 4.7.5).

4.7.1 MapReduce Workflows

Information Retrieval: Term Frequency-Inverse Document Frequency (TF-IDF) is a representative workflow from the information retrieval domain. TF-IDF calculates weights representing the importance of each word to a document in a collection. The TF-IDF weight is a function of the normalized frequency of a word in a document and the number of documents that contain the word. The default TF-IDF workflow consists of three jobs that calculate: (a) the frequency of a word in a document, (b) the total number of words in each document, and (c) the number of documents containing each word as well as the TF-IDF weight of each ⟨word,document⟩ pair. The input dataset is a randomly generated corpus that is partitioned on the document name.

Social Network Analysis: A workflow from the social network analysis domain is used to find the top 20 coauthor pairs who have collaborated most frequently with each other. The input dataset is a list of randomly generated ⟨paperID, authorID⟩

pairs from a power-law distribution, partitioned on {paperID}. The workflow consists of four jobs J_1–J_4: J_1 combines all authors for each paper; J_2 creates and counts the coauthor pairs; J_3 samples the data and creates partition split points for J_4; and J_4 finds the top 20 coauthor pairs in decreasing order.

Log Analysis: Pavlo et. al. [97] describe a complex join task from the log analysis domain. The workflow uses two input datasets: uservisits (partitioned on {date}) and pageranks. We use the data generator provided in [97] to generate the two datasets. This workflow consists of four jobs. The first job filters uservisits by a specified date range and joins it with pageranks on page url. The second job performs an aggregation to find the average pagerank and total ad revenue, grouped by user. The third job samples and creates partition split points for the last job. The last job finds the user with the highest total ad revenue.

Web Graph Analysis: PageRank [94] is an example of a web graph analysis algorithm that finds the ranking of web pages based on the hyperlinks pointing to each page. This algorithm can be implemented as an iterative workflow where each iteration is composed of two jobs. The first job joins on the {pageID} key of the two datasets: (a) the adjacency list with each web page and its outgoing hyperlinks, and (b) the current pagerank of each web page. The second job calculates the new pagerank of each web page. We generated an adjacency list of web pages from a power-law distribution.

Business Analytics Query: Query 17 from the TPC-H benchmark is a representative example of a complex business analytics (SQL) query [117]. This query determines how much yearly revenue would be lost on average if orders were no longer filled for small quantities of certain parts. Query 17 generates a four-job workflow. Job J_1 scans and processes the lineitem table. Job J_2 applies a filter condition on the part table, joins the output of J_1 and the filtered part table, and

finds the average quantity of each part. Job J_3 performs another filtered join on the outputs of J_1 and J_2. The final job J_4 calculates the total price of all parts. We use the TPC-H data generator to generate the input datasets for this workflow. The tables lineitem and part are both partitioned on {partID}.

Business Report Generation: Business report generation often involves multiple queries (e.g., that perform different groupby aggregates) on a single source dataset [28]. We emulate this scenario by creating a seven-job workflow that processes the lineitem table from the TPC-H Benchmark. The first job scans and performs an initial processing of the data. Two jobs read, filter, and find the sum and maximum of the prices for the {orderID, partID} and {orderID, supplierID} groupings respectively. The results of these two jobs are further processed by separate jobs to find the overall sum and maximum prices for each {orderID}. Finally, the results are used separately to find the number of distinct aggregated prices.

Post-processing Jobs: It is common in MapReduce deployments to have workflows that only operate on small datasets (e.g., in the order of GBs). These workflows would only use a small portion of the resources available in the cluster. For example, small datasets can result from filtering or aggregation operations. To capture this scenario, we created a three-job workflow that operates on a small dataset. The first job scans and performs an initial processing of the data. The other two jobs are groupby-aggregates that compute covariance and correlation respectively on the output of the first job. We use the TPC-H data generator to generate the input dataset for this workflow.

User-defined Logical Splits: It is common for users to specify logical splits for a set of jobs in a workflow in order to analyze different subsets of data records differently. For example, a Web portal log analysis workflow may want to perform different types of analysis based on specific age groups of users. We created a three-

job workflow to emulate this scenario. The workflow consists of a preprocessing (producer) job that outputs the data needed by two consumer jobs. Each consumer job processes only a subset of this data by filtering records in the map function.

4.7.2 Breakdown of Performance Improvements

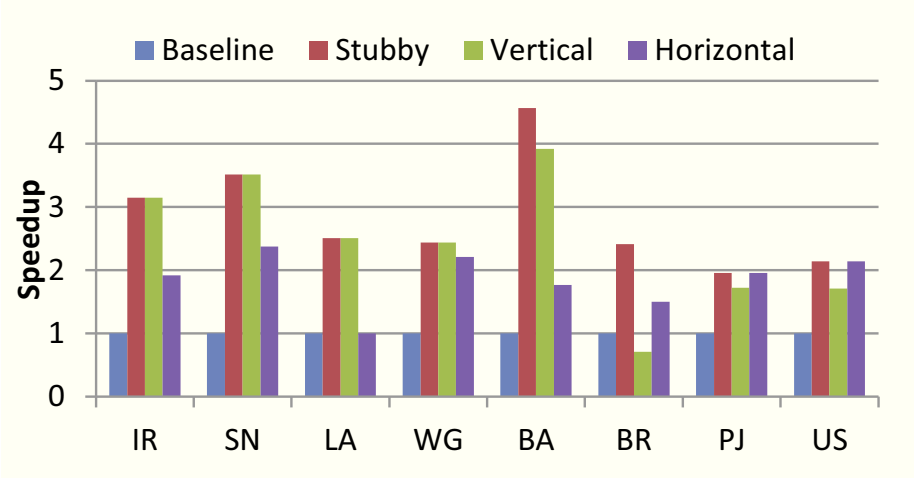


FIGURE 4.11: Speedup over the Baseline achieved by Stubby, Vertical, and Horizontal.

First, we evaluate the overall improvement given by Stubby on workflow performance. We also evaluate the improvements offered by our two groups of transformations (Vertical and Horizontal) when used in isolation. This breakdown allows us to study the source of improvements obtained from using Stubby. Figure 4.11 shows the speedup over the Baseline performance achieved by (i) Stubby with all transformations enabled, (ii) Stubby while using only the Vertical group transformations (denoted Vertical), and (iii) Stubby while using only the Horizontal group transformations (denoted Horizontal). Note that in both Vertical and Horizontal group transformations, Stubby also considers partition function and configuration transformations. Overall, Stubby is able to achieve between 2X and 4.5X speedup over the Baseline. As seen in the figure, the improvements vary depending on the workflow.

For the Information Retrieval (IR), Social Network Analysis (SN), Log Analysis (LA), and Web Graph Analysis (WG) workflows, the performance gains are predominantly due to the vertical packing transformations. These workflows do not present any opportunity for horizontal packing. The speedup achieved by Horizontal is primarily due to the cost-based selection of configuration transformations. The results for these workflows also reflect the spectrum of performance gains we can get from the different packing transformations. For example, Vertical achieves a 2.5X speedup over Horizontal for Log Analysis, whereas the speedup is only 0.2X for Web Graph Analysis. The computation in job J_2 of PageRank dominates the overall running time of the workflow, so vertically packing it with job J_1 offers limited benefit.

The Business Analytics Query (BA) shows a scenario where both vertical and horizontal packing contribute to the overall performance gains from Stubby. Specifically, the intra-job vertical packing transformation is applicable to the two join jobs in BA (jobs J_2 and J_3). Since both J_2 and J_3 process the dataset produced by the first job J_1, horizontal packing is also applicable. Stubby applies both transformations to obtain higher benefits compared to using Vertical or Horizontal alone.

The Business Report Generation (BR) workflow is a notable case. Vertical transforms the seven-job workflow into a five-job workflow. However, Vertical performs worse than Horizontal because the nature of BR makes it well suited for benefiting from horizontal packing transformations. Vertical also performs worse than Baseline because we have enabled Pig to use its rule-based optimizations (one of which is horizontal packing). By applying transformations from both the Vertical and Horizontal groups, Stubby generates a three-job workflow that gives a 2.4X speedup.

The Post-processing Jobs (PJ) workflow offers an example where horizontal packing is a wrong decision. Since Baseline performs horizontal packing whenever possible, it generates a suboptimal plan for this workflow. Stubby and Horizontal, being

cost-based, correctly decide not to perform horizontal packing for PJ in this case. Furthermore, unlike Baseline, the three other approaches apply the configuration transformation in a cost-based fashion, leading to the performance benefits seen for PJ in Figure 4.11.

The User-defined Logical Splits (US) workflow is one case where the partition function transformation applies. Specifically, the partition function in the producer job can be changed from the default hash partitioning to range partitioning; thereby enabling partition pruning to be applied to the data read by each consumer job in US.

Overall, it is apparent that different workflows present different transformation opportunities. Stubby is able to recognize and take advantage of these opportunities appropriately to offer speedups ranging from 2X to 4.5X over the Baseline.

4.7.3 Comparison against State-of-the-Art

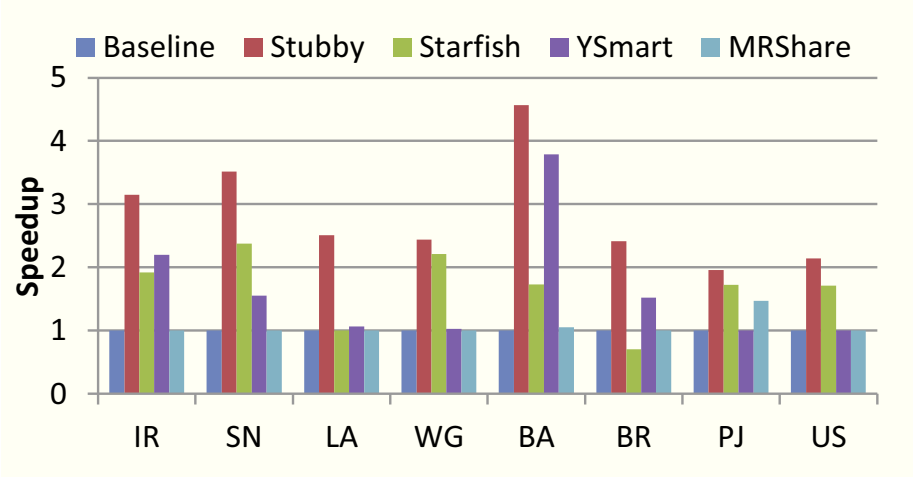


FIGURE 4.12: Speedup over the Baseline achieved by Stubby, Starfish, YSmart, and MRShare.

In this section, we compare Stubby against the following three state-of-the-art approaches for optimizing MapReduce workflows:

1. *Starfish*, based on a cost-based approach proposed in [51], to find good configuration parameter settings for each MapReduce job in the workflow.

2. *YSmart*, based on a rule-based approach proposed in [70], to perform vertical and horizontal packing transformations aggressively in order to minimize the number of jobs in the workflow. We have enhanced YSmart with a rule-based approach for selecting configuration parameter settings.

3. *MRShare*, based on a cost-based approach proposed in [87], to perform horizontal packing transformations. A rule-based approach is used for selecting configuration parameter settings.

Figure 4.12 shows the speedup achieved over the Baseline after optimizing our eight workflows using Stubby, Starfish, YSmart, and MRShare. Overall, the other approaches are all able to achieve good speedups over the Baseline, with the speedup value depending on the workflow. Stubby is able to outperform all other approaches for all workflows since Stubby considers a strict superset of the optimization opportunities that the others consider, and in a cost-based fashion. For example, Stubby is the only optimizer that considers the opportunity to prune partitions through partition function selection for the Log Analysis and User-defined Logical Splits workflows.

From the speedups that Starfish achieves in Figure 4.12 (ranging between 1.5X and 2.4X), we observe that finding good configuration parameter settings in a cost-based fashion can give significant performance improvements. However, Starfish misses out on all vertical and horizontal packing opportunities that can provide significantly higher speedups, like in the case of the Business Analytics Query (BA).

YSmart and MRShare do not automatically find good configuration settings to use. For example, YSmart is able to achieve a 1.5X speedup for the Social Network

Analysis (SN) workflow from performing vertical packing. With better configuration settings, Stubby is able to increase the speedup to 3.5X. Similarly, MRShare is able to achieve a 1.4X speedup for the Post-processing Jobs (PJ) workflow, whereas Stubby can achieve close to 2X speedup from selecting better configuration settings.

With a rule-based approach that tries to minimize the number of MapReduce jobs, YSmart can sometimes make suboptimal decisions. This case was evident in the Post-processing Jobs (PJ) workflow where YSmart performed horizontal packing on the two consumer jobs. Stubby and MRShare, on the other hand, used their cost-based approach to determine that horizontal packing was not a good choice, and chose to have the two jobs run independently.

Finally, as MRShare only considers the horizontal packing transformation, it does not provide any performance improvements for many of the MapReduce workflows considered in our evaluation.

4.7.4 Optimization Efficiency

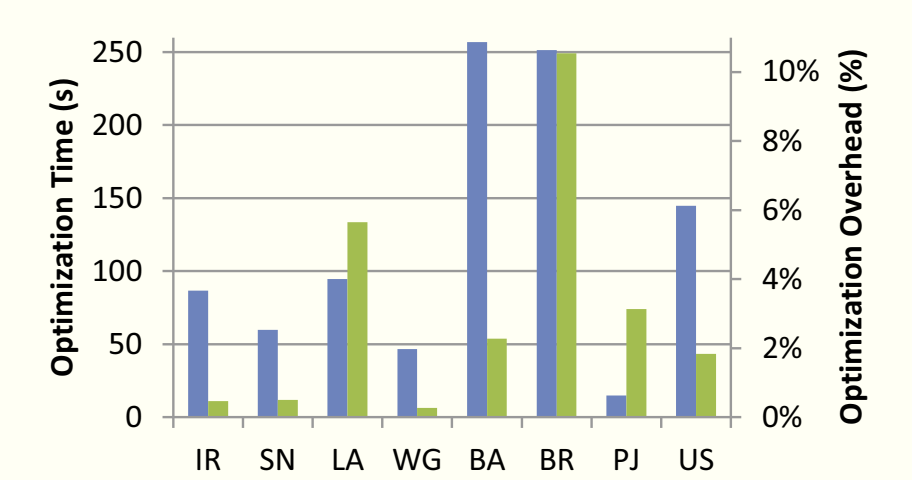


FIGURE 4.13: Optimization overhead for all workflows in terms of (a) absolute time (blue bars), and (b) a percentage over the total running time of each workflow (green bars).

In this section, we evaluate the efficiency of Stubby in finding near-optimal trans-

formations to apply to a given MapReduce workflow. Figure 4.13 shows the optimization time of Stubby in seconds as well as a percentage over the Baseline running time. Stubby spent on average less than 2 minutes to optimize each workflow. In the worse case, Stubby took around 5 minutes for optimizing the Business Analytics Query (BA) and Business Report Generation (BR) workflows, which contain 4 and 7 jobs respectively.

Percentage-wise, the optimization overhead for seven out of the eight workflows is less than 6%. At worst, Stubby introduced an overhead of 10.5% for the BR workflow which is our largest workflow with 7 jobs. Overall, Stubby's optimization overhead is small compared to the 2X to 4.5X speedup that Stubby gives for these workflows (recall Section 4.7.2). Since many analytical workflows are run periodically, the optimization overhead of Stubby can be amortized over multiple workflow runs.

4.7.5 Deep Dive into an Optimization Unit

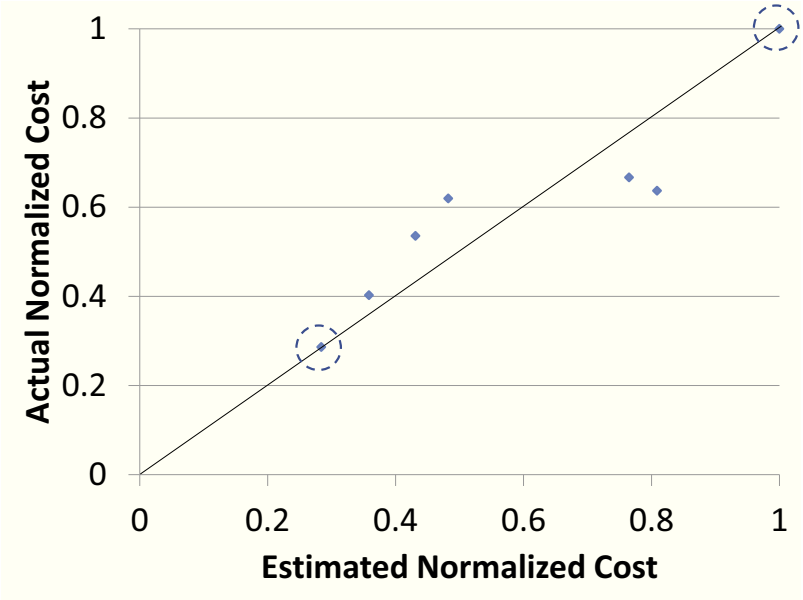


FIGURE 4.14: Actual vs. estimated normalized cost for all combinations of valid transformations in the first optimization unit of the Information Retrieval workflow.

As discussed in Section 4.4, Stubby (i) enumerates all combinations of valid trans-

formations within an optimization unit in order to generate all valid subplans, (ii) finds the best job configurations for each subplan, and (iii) selects the subplan with the lowest estimated cost. In this experiment, we drill down into the first optimization unit $U^{(1)}$ of the Information Retrieval (IR) workflow. In $U^{(1)}$, seven distinct combinations of transformations can be applied to yield seven subplans p_1–p_7.

We captured the best configuration settings generated by Stubby for each subplan p_i and used them to run each p_i separately. Figure 4.14 shows the scatter plot of the actual and estimated normalized costs for the seven subplans. Ideally, the points in the scatter plot should fall on the solid line. The inaccuracies are due to measurement errors during profiling and estimation errors when calculating plan costs [51]. We observe that the cost estimates are good enough for Stubby to identify the subplans that will lead to the best and worst performance (indicated by dotted circles in Figure 4.14).

4.8 Related Work

A number of recent projects provide users with various interfaces for generating data-parallel workflows [23, 54, 98, 134]. DryadLINQ and FlumeJava provide libraries and classes for specifying workflows using popular programming languages such as C# and Java respectively [23, 134]. On the other hand, systems like Hive, Pig, Jaql, and SCOPE provide their own high-level declarative languages for creating MapReduce workflows [54, 98, 138]. Our work on Stubby is complementary to these projects in that Stubby is designed to support different interfaces by sitting directly above the workflow scheduling and execution engine (refer to Figure 4.2). The optimization techniques that we introduce in this chapter can be applied to any MapReduce workflow regardless of the interface used to generate the workflow.

There is a large body of work on automatically optimizing worklows of data-parallel jobs [51, 59, 70, 87, 88, 127, 133]. The techniques used can be categorized as

either rule-based, such as FlumeJava [23], Manimal [59], and YSmart [70], or cost-based, such as MRShare [87] and Starfish [51]. This category of work differs from Stubby in one or more ways such as: (a) considering a much smaller plan space, (b) focusing on some specific interface, or (c) relying on the guaranteed availability of specific types of information.

YSmart translates SQL-like queries into a set of MapReduce jobs based on four primitive job types: selection-projection, aggregation, join, and sort. YSmart's rule-based optimizer then uses the knowledge of the job primitives used in the queries in order to merge MapReduce jobs. YSmart's goal is to minimize the total number of jobs, which can occasionally lead to suboptimal plans in terms of actual performance. Also, YSmart does not consider optimization opportunities available from partition function transformations and configuration transformations. Similarly, FlumeJava uses information regarding the provided Java class abstractions to pack higher-level operations into the minimum number of MapReduce jobs. MRShare focuses on optimization of multiple MapReduce jobs by applying cost-based decisions for horizontal packing transformations on the jobs. MRShare does not consider workflows or vertical packing. Starfish proposes a cost-based approach for applying (only) configuration transformations.

In contrast, Stubby considers a much larger plan space for workflow optimization that subsumes the plan spaces covered by each of the previously mentioned works. Furthermore, Stubby is designed to be a general-purpose system for workflow optimization where workflows can be optimized regardless of the interfaces used and availability of information. Stubby is able to consider the correct subspace of the full plan space based on the information available. For example, if schema annotations are not available, then Stubby will not consider intra-job vertical packing transformations.

While Stubby considers a large plan space, there are transformations that are

not supported by Stubby currently. For example, Wu et al. [130] develop cost-based query optimization techniques for multi-way join queries in MapReduce systems. Their approach automatically translates a user-submitted query into a final plan of MapReduce jobs by optimizing operator selection and ordering for joins. A transformation-based optimizer has been developed for the SCOPE system [138]. The focus of this optimizer is on how partitioning, sorting, and grouping properties can be exploited to avoid unnecessary operations during parallel processing of relational operators. FTOpt [122] introduces the space of fault-tolerance plans for workflows and then uses a cost-based approach to select the best fault-tolerance strategy for each job of a workflow.

The vertical packing transformations in Stubby are related to work on optimizing the computation of multiple aggregates over the same or similar sets of grouping attributes (e.g., [28]). Stubby is also related to work done on optimizing workflows of extract-transform-load (ETL) processes and business processes. For example, Simitsis et al. converted the problem of optimizing ETL workflows into a state space search problem where each state is a graph representation of the workflow [108]. The authors introduced rules for generating equivalent states and used a greedy heuristic search algorithm to find the optimal state.

4.9 Conclusions

As the popularity of MapReduce for big data analytics grows, the software ecosystem around MapReduce is also growing rapidly to provide users with different interfaces for generating MapReduce workflows. However, automatic cost-based optimization of these workflows remains a challenge due to the multitude of interfaces, large size of the execution plan space, and the frequent unavailability of all types of information needed for optimization. We introduced Stubby as a comprehensive solution to this problem.

Stubby is an extensible, cost-based, and transformation-based workflow optimizer that works across different interfaces for generating MapReduce workflows. Stubby is designed to sit above the MapReduce system, but below and external to any software system that submits workflows to the MapReduce system. Depending on the information available, Stubby considers all valid transformations from the full plan space (which we described in detail) to cost and pick the near-optimal set of transformations to apply on an input workflow. A comprehensive experimental evaluation showed the effectiveness of Stubby which generated optimized workflows with speedups of up to 4.5X over the baseline.

5

Characterization and Optimization of Continuous Data-Parallel Workflows

While in the previous chapter, we described a cost-based optimizer for batch data-parallel workflows, this chapter aims to optimize continuous data-parallel workflows. Continuous data-parallel workflows are composed of data-parallel *windowed* MapReduce computations with producer-consumer relationships based on data that are run in continuously. We focus on an important class of continuous workflows that is at the heart of many data-intensive services: continuous windowed aggregation queries. In this chapter, we characterize the plan space of these queries. We also describe *Cyclops*, which is an optimizer that can select a good plan, including the choice of system, from the plan space for a given query.

5.1 Introduction

Timely analysis of activity and operational data is critical for companies to stay competitive. Activity data from a company's website contains page and content views, searches, and advertisements shown as well as clicked. This data is analyzed for purposes like behavioral targeting, where personalized content is shown based

on a user's past activity, and showing advertisements or recommendations based on the activity of her social friends [24]. Operational data includes monitoring data collected from web applications (e.g., request latency) and cluster resources (e.g., CPU usage). Proactive analysis of operational data is used to ensure that web applications continue to meet all service-level requirements.

The vast majority of analysis over activity and operational data involves *continuous data-parallel workflows*. A continuous data-parallel workflow Q is a query that is composed of *windowed* MapReduce computations with producer-consumer relationships based on data. The workflow is issued only once over data D that is constantly updated. Q runs continuously over D and lets users get new results as D changes, without having to issue the same query repeatedly. Q has a *window* property that specifies temporally when to generate new results and how much updates to D to process. Continuous queries arise naturally over activity and operational data because of two reasons: (i) the data is generated continuously in the form of append-only streams; (ii) the data has a time component such that recent data is usually more relevant than older data.

The growing interest in continuous queries is reflected by the engineering resources that companies have been investing recently in building continuous query execution platforms. Yahoo! released *S4* in 2010, Twitter released *Storm* in 2011, and Walmart Labs released *Muppet* in 2012 [86, 113, 68]. Also prominent are recent efforts to add continuous querying capabilities to the popular *Hadoop* platform for batch analytics. Examples include the *Oozie* workflow manager, *MapReduce Online*, and Facebook's real-time analytics system [89, 31, 18]. These platforms add to older research projects like *Aurora*, *Borealis*, *NiagaraCQ*, *STREAM*, and *TelegraphCQ*, as well as commercial systems like *Esper*, *Infosphere Streams*, *StreamBase*, and *Truviso* [1, 37, 29, 114, 15, 25].

A common, but nontrivial, class of query that is at the heart of many continuous

analytics applications is the *windowed aggregation* query, which performs a continuous time-based windowing operation on a data stream, and then performs grouping and aggregation on each window. The range of applications captured by this query can range from simple filters over data streams to more complex statistical queries that look for anomalies and correlations.

For example, a behavioral targeting application uses windowed aggregation to aggregate each user's activity over a recent one hour window, updated every five minutes. The scale of these queries can be huge. The grouping may be over 100s of million web-site users that a company has. The aggregation functions used range from standard SQL aggregates like sum to complex user-defined aggregates (UDAs) that apply machine-learning techniques (e.g., to generate recommendation models for each user).

The importance of continuous data-parallel workflows is further reinforced by the number of work that introduce different ways of executing windowed aggregations (e.g., [1, 10, 15, 16, 49, 89, 95]), which form a rich and complex execution plan space. However, no attempt has been done yet to comprehensively study and characterize the plan space. Such a study will provide great value to application developers, database administrators, and system architects.

Contributions: While using windowed aggregation query as an example, this chapter focuses on the execution and optimization continuous data-parallel workflows, in the context of three diverse, but representative, systems that are popularly used in the industry: Esper [37], Storm [113], and Hadoop [47]. As we will show in this chapter, the execution plans for windowed aggregation queries differ along a number of dimensions:

1. Incremental vs. non-incremental processing

2. Using a hierarchical approach to windowing and aggregation

3. Types of parallelism used within and across windowing and aggregation, including pipelining, partitioning, and shuffling

4. Centralized vs. distributed processing

5. Streaming execution vs. repeated-batch execution

6. Use of partial aggregation is dictated by the type of aggregation in the query, which we categorize as *holistic*, *semi-holistic*, or *non-holistic* (defined in Section 5.3)

We systematically characterize the resulting rich execution plan space for windowed aggregation queries (Sections 5.2-5.4). We not only provide a detailed empirical evaluation that shows the tradeoffs among different plans (Section 5.6), but also developed a cost-based optimizer, called *Cyclops* that picks a good execution plan, including the choice of system, for a given query (Section 5.5).

5.2 Query Semantics and Properties

In this section, we define the exact semantics of windowed aggregation queries that should be provided by all implementations of plans across different systems. The template for a windowed aggregation query Q is as follows:

Q: Select $S.K$, $aggr(S.V)$
From S [Range r seconds, Slide s seconds]
(optional) Where *filter_condition_on_tuples_of_S*
Group By $S.K$
(optional) Having *filter_condition_on_groups*

S is a stream of timestamped tuples. Without loss of generality, we assume that three attributes in S are relevant to Q: timestamp t, grouping attribute K, and attribute V for aggregation per group.

Processing query Q involves generating new results whenever the temporal window relevant to the query changes, as defined by the **Range** and **Slide** properties in the From statement. The Range property r specifies the interval over which the tuples in the stream are part of the window: tuples with timestamp in the interval $(t-r, t]$ are part of the window at time t. The Slide property s specifies how the window advances over the stream, which dictates when the query is computed over the stream. Specifically, if the current interval for tuples part of the window is $(t-r, t]$, then the next interval is $(t-r+s, t+s]$. The Range (Slide) and the **arrival rate** property (tuples/second or bytes/second) of the input stream together specify the amount of data in each window (slide).

Query Q performs grouping on key $S.K$ and applies the aggregation function *aggr*() on values $S.V$ per group. We introduce two more properties to ease the discussion of the efficiency of this operation in later sections. The **skew factor** property captures how skewed the distribution of the grouping key is among tuples in the stream. High skew can cause load imbalances in distributed execution. The **domain size** property captures the number of unique groups in the input stream, which affects the size of query results and intermediate state.

The pluggable aggregation function *aggr*() makes window aggregation queries very expressive and powerful because *aggr*() can range from standard SQL aggregates to complex user-defined black box functions. The optional Where and Having statements allow input stream tuples and output groups to be filtered respectively.

Running example: The following query is our running example:

$Q1$: Select $S.K$, $sum(S.V)$

From S [Range 4 seconds, Slide 2 seconds]

Group By $S.K$

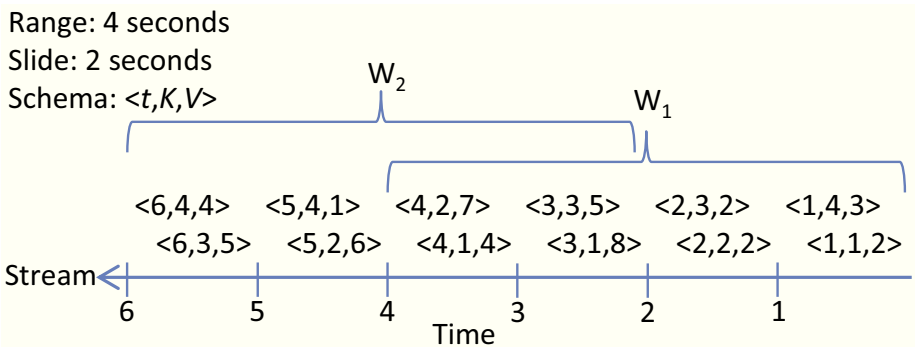


FIGURE 5.1: Two successive windows for a windowed aggregation query with Range = 4 seconds and Slide = 2 seconds. The two $\langle t,K,V \rangle$ tuples arriving every second in the stream are shown.

Each window in $Q1$ is composed of tuples in an interval of 4 seconds, advancing every 2 seconds on the input stream. Figure 5.1 shows the window operation of $Q1$ applied on an example stream S. The first window W_1 contains the tuples in the interval $(0, 4]$ (refer to the t field of the tuples). Since the Slide is 2 seconds, the second window W_2 then contains the tuples in the interval $(2, 6]$, namely, $\langle$3,1,8$\rangle$, $\langle$3,3,5$\rangle$, $\langle$4,1,4$\rangle$, $\langle$4,2,7$\rangle$, $\langle$5,2,6$\rangle$, $\langle$5, 4,1$\rangle$, $\langle$6,3,5$\rangle$, $\langle$6,4,4$\rangle$. The tuples in W_2 will be grouped by K and the V fields summed to give the result for W_2 as: $\langle$6,1,12$\rangle$, $\langle$6,2,13$\rangle$, $\langle$6,3,10$\rangle$, and $\langle$6,4,5$\rangle$. Note that all the result tuples for a window with interval $(t - r, t]$ have the timestamp t. For example, t=6 for W_2.

5.3 Logical Plan Space

This section describes three independent dimensions of choice that are available for the execution of a windowed aggregation query. These dimensions are independent of the system used.

5.3.1 Incremental Vs. Non-Incremental

Incremental processing shares computation across consecutive windows and processes only the differences (delta) between consecutive windows [15, 46, 85]. The differences

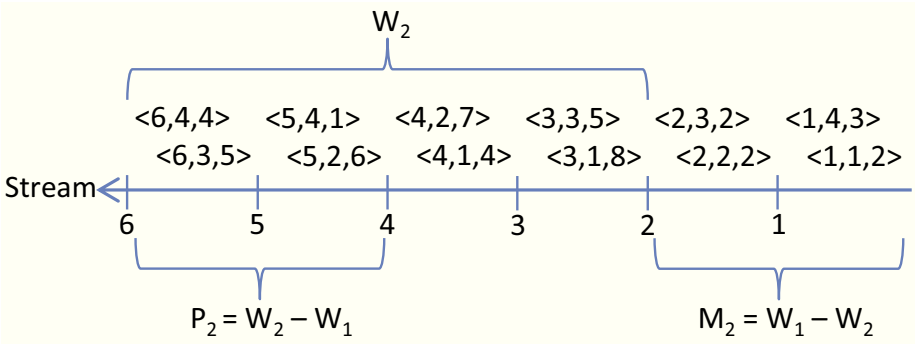


FIGURE 5.2: An illustration of the plus (P_2) and minus (M_2) tuples of window W_2 of the example from Figure 5.1.

between windows are categorized as *plus tuples*, which are the tuples in the current window that are not in the previous window, and *minus tuples*, which are the tuples in the previous window that are not in the current window. Figure 5.2 shows the plus tuples (labeled P_2=W_2-W_1) and minus tuples (labeled M_2=W_1-W_2) between the current window W_2 and previous window W_1 for our running example (Figure 5.1).

To process the plus and minus tuples, Incremental processing has to maintain a *synopsis*, namely, the intermediate state that gets updated with the application of these tuples. For example, the synopsis needed during Incremental processing of the running example query $Q1$ comprises the query results for the previous window. When processing W_2, the synopsis will be the query results (grouping and sum aggregation) for the previous window W_1: ⟨4,1,14⟩, ⟨4,2,9⟩, ⟨4,3,7⟩, and ⟨4,4,3⟩. The V field of minus tuples in M_2 are subtracted from, and the V field plus of tuples in P_2 are added to, the corresponding results with the same K field. For example, for the K=1 group, there is one minus tuple (⟨1,1,2⟩) in M_2 with the same K, and no tuples in P_2. Thus, the V field (2) of this tuple is subtracted from the corresponding V field of the synopsis (14). A similar exercise is done for all other groups, producing the same results for W_2 as described in Section 5.2.

In distributed databases, aggregation functions are categorized as *distributive*,

algebraic, or *holistic*. For a distributive function, the new aggregation result can be generated by applying new tuples to a synopsis composed only of the existing aggregation result. Sum, count, and max are distributive aggregation functions. The result of an algebraic aggregation function can be computed from the results of some number of distributive functions. For example, the algebraic aggregation function average is $\frac{sum}{count}$. The new aggregation result for an algebraic function can be generated from new tuples and a constant-sized synopsis that contains existing aggregation results for the corresponding distributive functions. Holistic aggregation functions do not admit a synopsis of bounded size.

We build on the above definitions to create the following categories for aggregation functions based on how each function behaves under plus tuples and minus tuples:

- Non-holistic: Functions that are distributive or algebraic for both plus and minus tuples (e.g., sum, count, variance).
- Semi-holistic: Functions that are distributive or algebraic for plus tuples, but not for minus tuples (e.g., max, min, top-k).
- Holistic: Functions that are not distributive or algebraic for plus tuples (e.g., median, black-box user-defined aggregates).

Incremental processing is resource efficient for non-holistic functions only. The alternative, Non-incremental processing, processes each window independently and as a batch containing all the tuples in the window's $(t - r, t]$ interval. Non-incremental processing can be applied to all categories of aggregation functions and have been utilized by a number of recent work and systems (e.g., [2, 37, 47, 103, 114]).

Pros and cons of Incremental processing: The advantage of Incremental processing is that it can reuse work from one window to the next. Recall the query

properties like Range, Slide, and arrival rate from Section 5.2. If there is a large overlap between windows—e.g., a large Range with a small Slide and fast stream arrival rate—then the data processed for each window during Incremental processing will be small relative to the total amount of data in the window. At the same time, depending on the query properties, Incremental processing can also cause wasted work. For example, Incremental processing for a query with non-overlapping windows—e.g., when Range = Slide—will cause all tuples in a window to be processed twice. In contrast, Non-incremental processing will process all tuples in a window only once in this scenario. Also, Non-incremental processing does not need any memory resources to maintain a synopsis across windows.

Incremental processing has more subtle disadvantages. Processing the differences based on keeping a synopsis introduces dependencies across the processing of successive windows. These dependencies reduce opportunities for parallel processing and also complicate fault tolerance. Making query execution fault tolerant is easier for Non-incremental processing: a window can be simply processed again after a failure, without having to worry about how the synopsis is rebuilt.

5.3.2 Hierarchical Processing

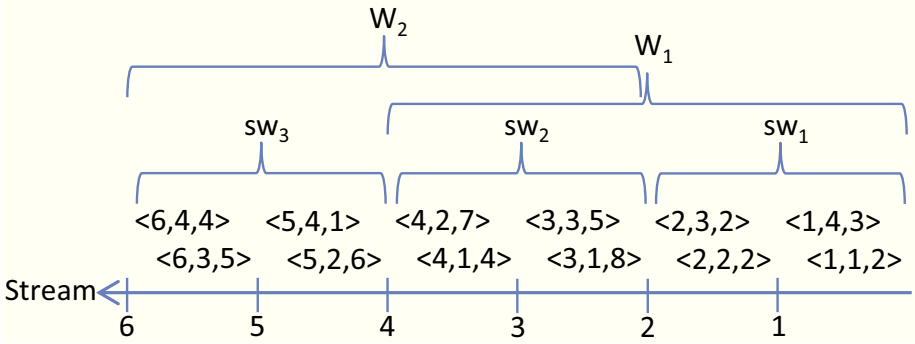


FIGURE 5.3: An illustration of the subwindows (sw_1, sw_2, sw_3) that can be created from the example stream from Figure 5.1.

In Hierarchical processing, the aggregation results over a window are computed in multiple passes [13, 33, 49, 72, 79, 89]. The aggregation function is first applied on smaller non-overlapping subwindows to generate partial aggregation results. Another aggregation pass is applied on the partial results to generate the final results for a window. Figure 5.3 shows the same example stream as before but with 2-second non-overlapping subwindows: sw_1, sw_2, and sw_3. In Hierarchical processing, each subwindow uses Non-incremental processing to generate partial aggregation results. Note that although it is possible to perform Incremental processing on the subwindows, we disregard this option because the subwindows are non-overlapping.

When executing the running example query $Q1$ on the example stream, sw_1 will have the following results $\langle 2,1,2\rangle$, $\langle 2,2,2\rangle$, $\langle 2,3,2\rangle$, and $\langle 2,4,3\rangle$, sw_2 will have the following results $\langle 4,1,12\rangle$, $\langle 4,2,7\rangle$, $\langle 4,3,5\rangle$, and sw_3 will have the following results $\langle 6,2,6\rangle$, $\langle 6,3,5\rangle$, $\langle 6,4,5\rangle$. Note that the results of the subwindows are the synopses needed to compute the final aggregation results. Since $Q1$ uses the non-holistic sum aggregation function, the results are simply the sum of each group for each subwindow. Other non-holistic and semi-holistic aggregation functions, e.g., average, may need more information in the synopses.

The results from the subwindows are used to compute the final results of windows W_1 and W_2. This second aggregation pass can use either Non-incremental or Incremental processing, and the plans are termed Non-incremental Hierarchical and Incremental Hierarchical respectively. For Non-incremental processing of W_2, the results of sw_1 and sw_2 are further grouped and summed. For Incremental processing of W_2, the results of sw_1 form the minus tuples (M_2) and the results of sw_3 form the plus tuples (P_2). Holistic aggregation functions do not work well with any type of Hierarchical processing because the aggregation results need all tuples of the window and cannot be generated efficiently by combining results of the subwindows.

Pros and cons of Hierarchical processing: At first glance, it may seem that Hierarchical processing is always inefficient with respect to, say, single-pass Incremental processing, because multiple passes of computation are needed per window. However, Hierarchical processing has several interesting features:

- Efficient processing of semi-holistic functions is enabled because, by using Non-incremental processing in both passes, minus tuples will never have to be processed.

- Rich opportunities for parallel processing are created. Pipelined parallelism within the grouping and aggregation operations enables the two aggregation passes (subwindow processing Vs. window processing) to run concurrently. Different (sub)windows can be processed in parallel. Partitioned parallelism can be applied within (sub)windows.

- Efficient hybrid combinations of Incremental and Non-incremental processing become possible. For example, Incremental processing can now be used by default even for semi-holistic functions like max. If the current max is part of the minus tuples, then the query can temporarily switch to Non-incremental processing over the appropriate subwindows' results—not the data in the complete window—to find the new max value efficiently.

Depending on query properties, the above features can give large gains. However, there is a tradeoff in Hierarchical processing between the performance improvement due to reusing subwindow results and the extra computational overhead for merging partial results. The first aggregation pass may not reduce data sizes significantly for input streams with a large domain size (recall Section 5.2), and could hurt performance overall.

5.3.3 Types of Parallelism

We describe three different ways to use parallelism in the execution of a windowed aggregation query.

Pipelined: There are opportunities for inter-operator pipelining [1, 10, 31, 45]: a window operation can run independently in a separate thread or task from the grouping and aggregation operations. Similar to pipelined database query execution, output tuples of the window operation are fed into the grouping and aggregation operations that are executing in parallel. Pipelined parallelism within a logical operation is also possible. For example, different passes of the grouping and aggregation operation can be pipelined in Hierarchical processing.

Partitioned: Partitioned parallelism can be applied to window as well as grouping and aggregation operations [5, 16, 25, 47, 80, 90]. Different parts of the input stream that constitute a (sub)window can be read and processed in parallel by a set of window operation tasks. The output tuples of all window tasks are then partitioned on the grouping key K—e.g., using hash partitioning—and processed in parallel by a set of grouping and aggregation tasks (which could use either Incremental or Non-incremental processing). Note that such partitioning is possible because there is no dependence among tuples belonging to different grouping keys.

Shuffled: Shuffled parallelism is a special case of partitioned parallelism where the output tuples of the window tasks are first partitioned randomly (shuffled) across a set of grouping and aggregation tasks [54, 113]. The processing done by these grouping and aggregation tasks generates intermediate results (partial aggregates) that are then partitioned on the grouping key K to a second set of grouping and aggregation tasks that compute the final result. Similar to Hierarchical processing, holistic aggregation functions do not work well with Shuffled parallelism because the intermediate results cannot be combined efficiently to generate the final aggregation

Table 5.1: A categorization of systems based on their properties.

Properties	Systems
Centralized, streaming	Aurora [2], DBToaster [3], Esper [37], Gigascope [33], STREAM [15]
Distributed, streaming	Aeolus [104], Borealis [1], Infosphere Streams [10], Microsoft CEP [5], MapReduce Online [31], Muppet [68], NiagaraCQ [29], Oracle CEP [90], S4 [86], SAP Sybase Event Stream Processor [103], StreamBase [114], StreamCloud [45], Storm [113], TelegraphCQ [25], Truviso [120]
Distributed, repeated-batch	Comet [49], Dryad [57], Facebook's real-time analytics system [18], Hadoop [47], Hive [54], MapReduce [35], Nephele [80], Oozie [89], Spark [136], Spark Streaming [137]

result.

Pros and cons: Pipelined parallelism can lower the latency to produce query results. It also avoids writing tuples to disk. However, Pipelined parallelism cannot be scaled to handle large window sizes and stream arrival rates as easily as Partitioned parallelism. At the same time, Partitioned parallelism incurs overheads in partitioning, network communication, and executing parallel tasks; which can result in suboptimal performance on smaller windows.

While Shuffled parallelism needs multiple passes, it enables efficient parallel processing in two (fairly common) cases where regular Partitioned parallelism can be highly inefficient. First, highly skewed streams can lead to load imbalance when partitioned directly on the grouping key. Second, a windowed aggregation query may not have a Group By clause (e.g., a global aggregation on all tuples). Here, Shuffled parallelism can alleviate the bottleneck by generating partial aggregation results across parallel tasks before merging them in a single task to generate the final aggregation results for a window.

5.4 Execution Plan Space

The logical plans from Section 5.3 can be implemented in different systems. Moreover recent work has claimed that the time has come and gone for the idea of "one size fit all" to system design [73, 112], which is further reinforced by the number of specialized continuous query systems currently available. Thus, the choice of system is part of the execution plan space. We categorize the systems in two dimensions (see Table 5.1 for examples).

- *Centralized Vs. Distributed*: Centralized systems are designed for ultra-fast performance on a single node. These systems can only be scaled up (adding more resources to the node) instead of being scaled out (adding more nodes to the cluster). They usually have coarse-grained fault tolerance only, e.g., hot standby pairs or full system restart on failure. In contrast, distributed systems are designed with easy scale out as the primary goal. They have built-in mechanisms for partitioned processing, resource elasticity, and fine-grained fault tolerance (e.g., automatically restarting failed tasks or replay processing of tuples).

- *Streaming Vs. Repeated-Batch*: Streaming systems cater to low-latency requirements by having continuously-running tasks that process tuples as they arrive. Output tuples produced by an operator are pushed quickly to consumer operators. The ability of streaming systems to respond quickly to tuple arrival comes at the cost of per-tuple processing overheads. In contrast, batch systems like parallel databases and MapReduce systems can be made to process continuous queries by running subqueries or jobs repeatedly on different subsets of the input data stream. These subqueries/jobs are scheduled appropriately using internal or external mechanisms. While batch processing avoids high

per-tuple overheads and makes efficient use of CPU and I/O resources, the added overhead of starting, stopping, and scheduling tasks is unsuitable for low-latency requirements.

In this chapter, we focus on three continuous query systems: (i) *Esper*, a centralized streaming system [37]; (ii) *Storm*, a distributed streaming system [113]; and (iii) *Hadoop*, a distributed repeated-batch system [47]. We found that these three systems—which are all open-source and have been gaining significant traction in the industry recently—together capture the broad spectrum of continuous query systems.

Next, we describe the execution plans supported by each system. The implementations of these plans adhere to the exact windowed aggregation query semantics mentioned in Section 5.2. Moreover, we assume that streams come in "as is" (e.g., not grouped), such that grouping must be applied after creating windows from streams, and tuples in the input streams are already timestamped outside of the system (e.g., by the source), such that systems need not worry about time synchronization. We continue to use our running example (see Figure 5.1) to illustrate the plans.

5.4.1 Esper: Centralized and Streaming

Esper is a centralized streaming system that supports four execution plans: Incremental, Non-incremental, Incremental Hierarchical, and Non-incremental Hierarchical. In particular, plans with Partitioned parallelism are not supported because of Esper's centralized nature. Only inter-operator Pipelined parallelism is used.

Figures 5.4(a) and (b) show the respective implementations of Non-incremental processing and Incremental processing of window W_2 of the example stream. In both figures, Esper logically has two separate pipelined operators: window operator (labeled W) and grouping-aggregation operator (labeled GA). The input stream is processed continuously. The W operator is responsible for extracting the correspond-

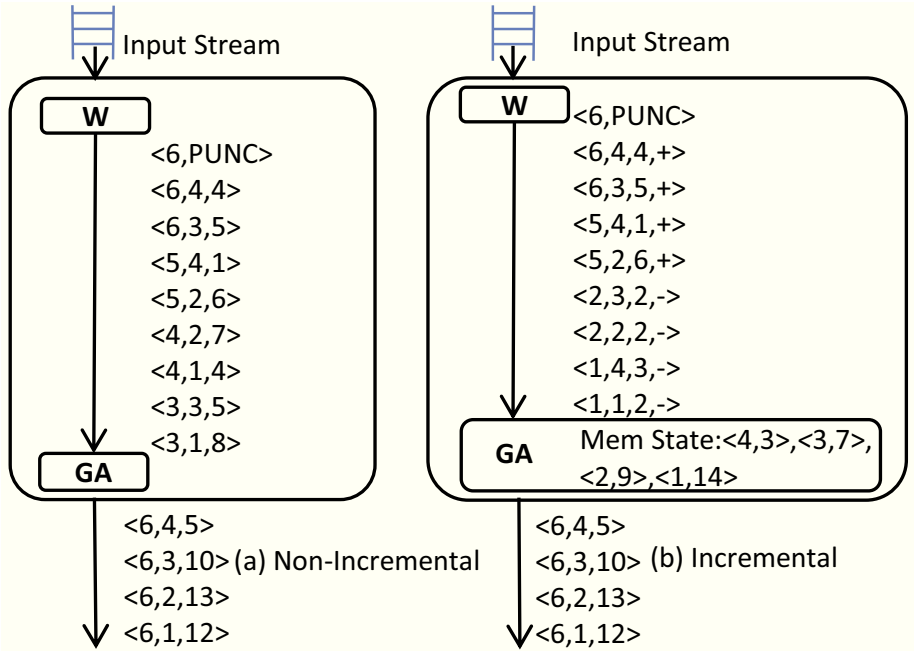


FIGURE 5.4: Esper's plans for (a) Non-incremental and (b) Incremental processing for window W_2 of the example stream.

ing tuples from the stream that are part of the current window and pushing these tuples to the GA operator.

Because of their continuous execution of queries, streaming systems (both centralized and distributed) need a mechanism to distinguish tuples of one window from those of another window. Our implementation uses *punctuation tuples* (labeled <PUNC> in all plan figures) to communicate window boundaries. For Non-incremental processing, the GA operator performs grouping and aggregation as it receives new tuples. When a punctuation tuple arrives, GA outputs the results for the current window. The timestamp of the output tuples is set to the timestamp of the punctuation tuple.

As described in Section 5.3, Incremental processing only processes plus (P_2) and minus (M_2) tuples, rather than all tuples of a window. Thus, the GA operator needs to identify these tuples. In our implementation, the W operator includes additional

meta information to specify the type of a tuple (marked by "+" and "-" in all plan figures involving Incremental processing). The synopsis needed in Incremental processing is maintained in memory by the GA operator. For example, the GA operator in Figure 5.4(b) shows the synopsis represented by the results of W_1.

Figures 5.5(a) and (b) show the respective implementations of Non-incremental Hierarchical and Incremental Hierarchical processing for window W_2. The main difference brought by Hierarchical processing is the addition of two more pipelined operators (W and GA operators). The first set of W and GA operators are processing the subwindows. (Due to space constraints, only the tuples for subwindows sw_2 and sw_3 are shown.) The second set of W and GA operators use the output of the subwindows to generate the final aggregation results for W_2. Note that the second W operator is responsible for performing the window operation on the results of the subwindows. Although the processing for sw_1 is not shown in the first set of W and GA operators of the figure, for Incremental Hierarchical processing in Figure 5.5(b), the second W operator collects the results of sw_1, and then outputs these results as minus tuples for W_2.

5.4.2 Storm: Distributed and Streaming

Storm is a distributed streaming system that continuously runs *topologies* in a cluster. Each topology is a graph of operators, where operators can be run using parallel tasks. Storm also includes multiple mechanisms (e.g., hash partitioning and shuffling) to distribute tuples across parallel tasks.

Storm's implementation of Incremental and Non-incremental processing has three differences from Esper: (i) use of multiple parallel tasks for the W and GA operators, and (ii) use of Partitioned parallelism; and (iii) replication of punctuation tuples to all tasks. Figure 5.6 shows the task-level illustration of Incremental processing with Partitioned parallelism for window W_2. Disjoint subsets of tuples from the input

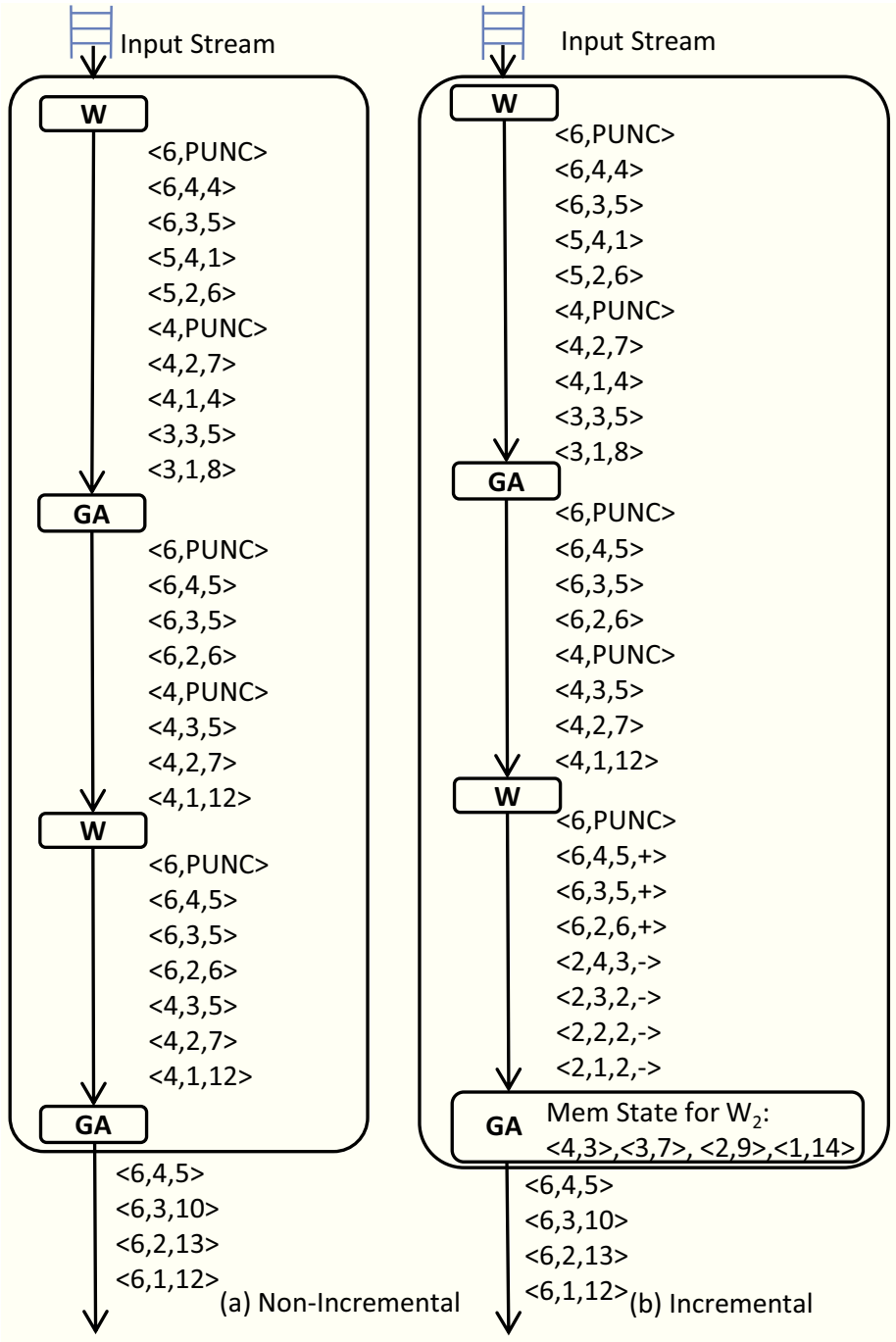


FIGURE 5.5: Esper's plans for (a) Non-incremental and (b) Incremental Hierarchical processing for window W_2 of the example stream.

stream are processed by the parallel tasks of W. The output tuples are then pushed to one of the parallel GA tasks. Partitioned parallelism is achieved by implementing

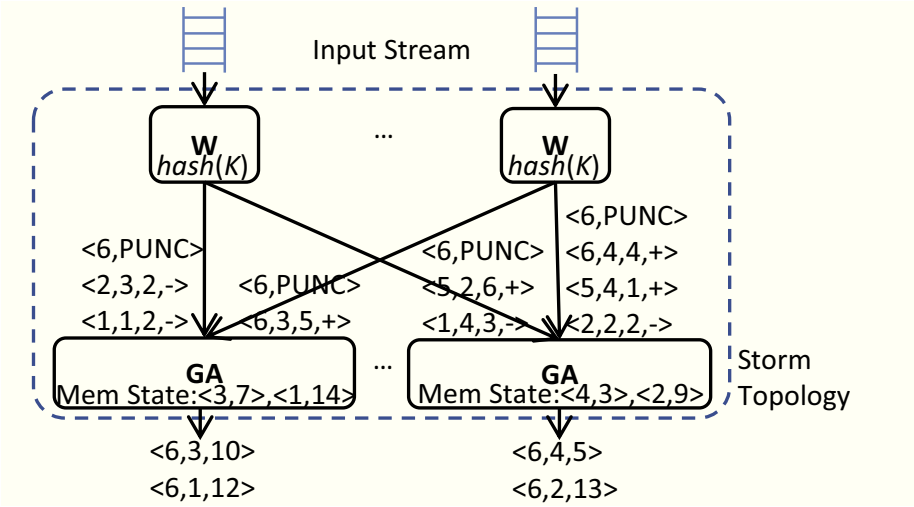


FIGURE 5.6: Task-level illustration of a Storm topology performing Incremental processing with Partitioned parallelism for window W_2 of the example stream.

a function in each W task that performs hashing on the grouping key K to determine the corresponding GA task to send the output tuple. Each GA task is responsible for ensuring that tuples from different W tasks are grouped only for the same window. It has to carefully merge and synchronize internally the input tuples it receives. Thus, it maintains separate queues of input tuples for each W task and only generates aggregation results when the same timestamped punctuation tuples (labeled <PUNC> in the figure) from all W tasks have been received.

Figure 5.7 shows a task-level illustration of Incremental processing with Shuffled followed by Partitioned parallelism of window W_2. There are three operators: window operator, GA operator for generating intermediate results, and another GA operator for generating the final aggregation result per window; all of which are executed by multiple parallel tasks. The W tasks randomly send the output tuples to the GA tasks, instead of partitioning on the grouping key. The first set of GA tasks hash on the grouping key to determine the next GA task to send the intermediate results. All tasks handle the synchronization, merging, and grouping of tuples by

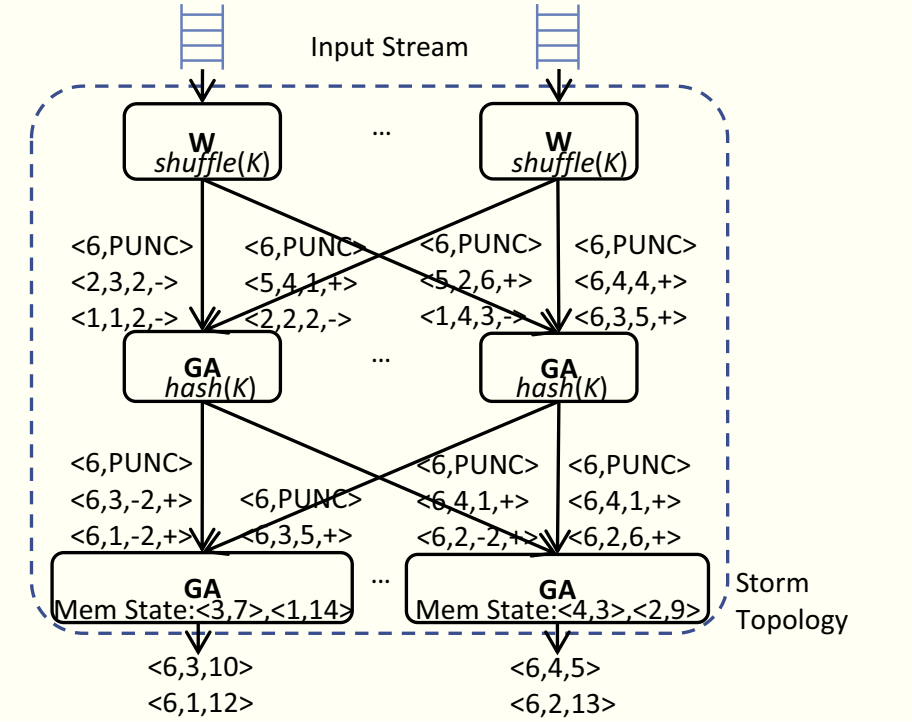


FIGURE 5.7: Task-level illustration of a Storm topology performing Incremental processing with Shuffled followed by Partitioned parallelism for window W_2 of the example stream.

maintaining separate queues of input tuples.

Note that the Incremental version of Shuffled followed by Partitioned parallelism is slightly different from the implementation of Incremental Partitioned parallelism. As before, the W tasks only output the plus and minus tuples. However, the first set of GA tasks does not maintain any synopsis (e.g., aggregation results from previous window). These tasks perform grouping and partial aggregation solely based on the tuples they receive for that particular window. For example, consider the first GA task on the right in Figure 5.7. For window W_2, it receives 4 tuples: $\langle 1,4,3,- \rangle$ and $\langle 5,2,6,+ \rangle$ from one W task, and $\langle 6,3,5,+ \rangle$ and $\langle 6,4,4,+ \rangle$ from the other W task. For each group, aggregation is performed on the input tuples that results in the following tuples: $\langle 6,2,6,+ \rangle$, $\langle 6,3,5,+ \rangle$, and $\langle 6,4,1,+ \rangle$. Each result is then sent to its

corresponding next GA task, which applies the result to its synopsis (e.g., aggregation result of window W_1).

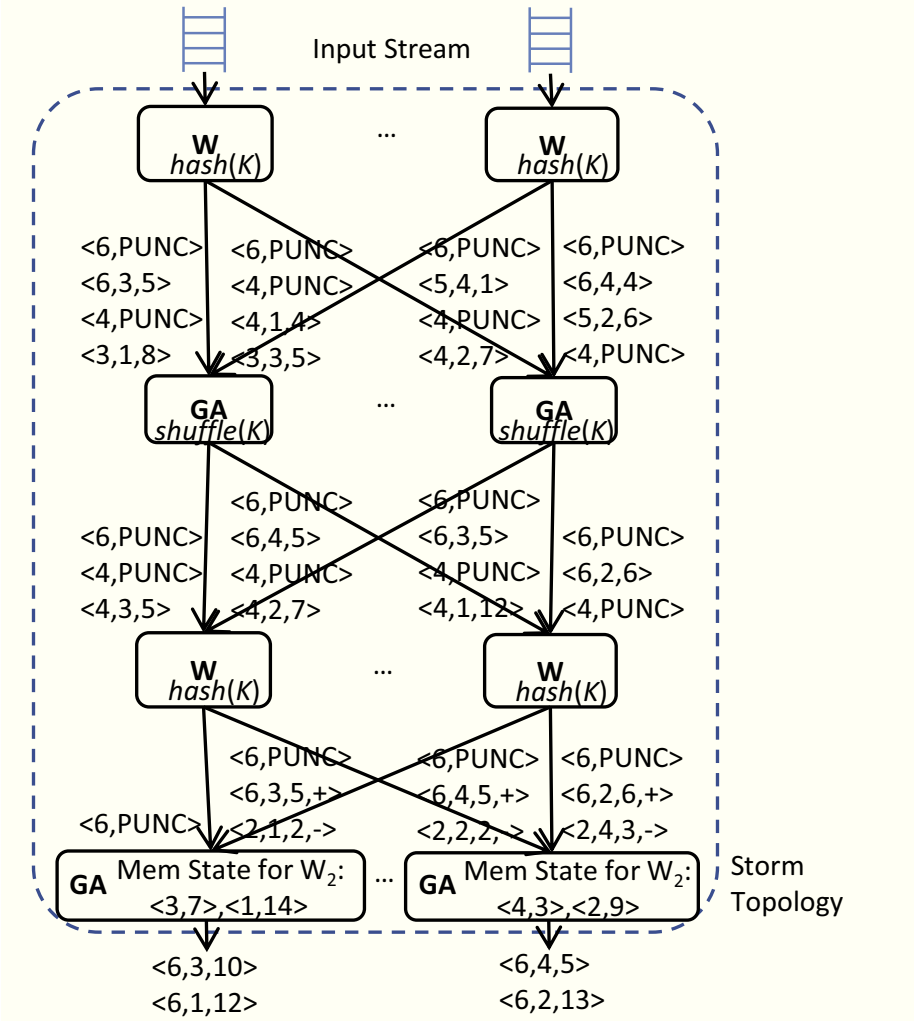


FIGURE 5.8: Task-level illustration of a Storm topology performing Non-incremental processing with Partitioned parallelism of the non-overlapping subwindows and then using the partial aggregation results to perform Incremental processing with Partitioned parallelism for window W_2 of the example stream.

The Storm implementations for Incremental and Non-incremental Hierarchical processing have some differences with those in Esper. Like in Esper, there are two sets of window and grouping-aggregation operators. However, as with the other Storm implementations, these operators use task-level parallelism and replicate the

punctuation tuples across all tasks. Figure 5.8 shows the task-level illustration of Incremental Hierarchical processing with partitioned parallelism for W_2. The first set of W and GA tasks processes the subwindows. In particular, the W tasks pull tuples from the input stream and generate punctuation tuples for subwindows. For each subwindow, the first set of GA tasks applies the aggregation function. The results are pushed to the second set of W tasks which are responsible for keeping track of the tuples that belong to the actual window specification of the windowed aggregation query. For Incremental processing, the second set of W tasks only outputs the plus and minus tuples (distinguished by the +/- label in the figure).

5.4.3 Hadoop: Distributed and Repeated-Batch

Hadoop is a distributed repeated-batch system that runs jobs composed of data-parallel map and reduce tasks. It has a built-in mechanism that partitions, sorts, and groups outputs of map tasks before sending as inputs to reduce tasks. In contrast to the previous two systems, tasks are not running continuously but are scheduled on the cluster by Hadoop's task scheduler. Processing a window in Hadoop involves executing one or more jobs. Moreover, Hadoop requires a control program for submitting jobs along with the information about the input dataset that each job should process.

There is no window operator internally in a Hadoop job. Instead, the sequence of the input stream that belongs to a window is set as part of the input datasets of the Hadoop job by the control program. For Non-incremental processing with Partitioned parallelism, a single Hadoop job is responsible for generating results of each window. The map tasks simply set the grouping key K of each input tuple as the map output key. The Hadoop framework then handles the partitioning and grouping of the output of map tasks. The reduce tasks then apply the aggregation function to each group.

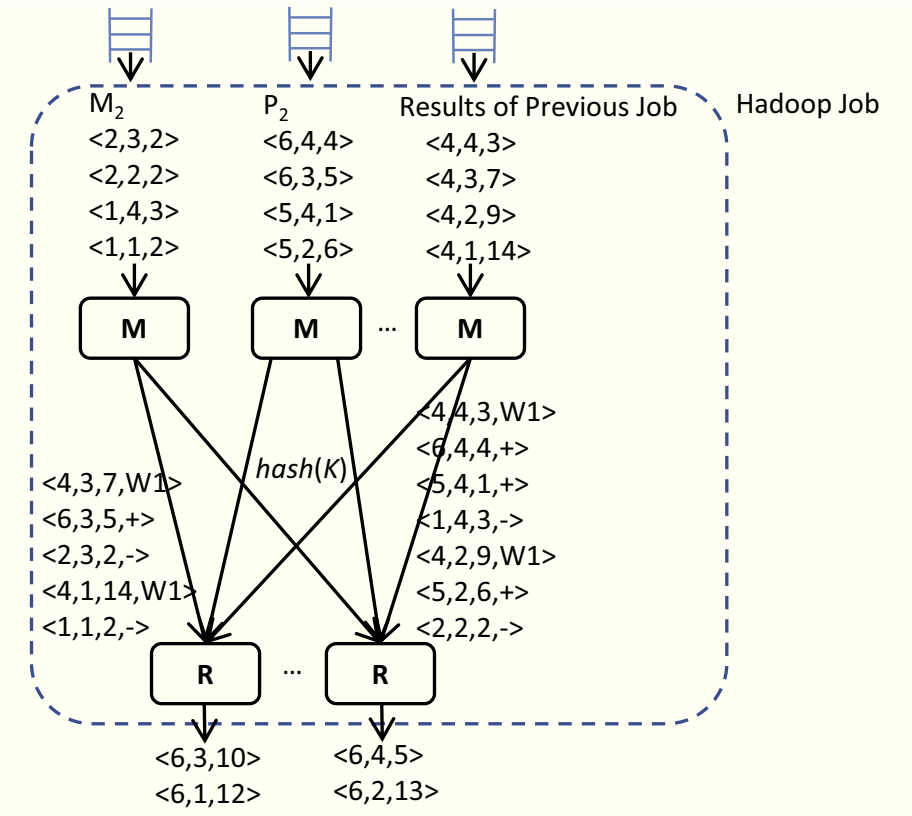


FIGURE 5.9: Task-level illustration of a Hadoop job performing Incremental processing with Partitioned parallelism for window W_2 of the example stream.

In contrast to Esper and Storm, the Hadoop implementation for Incremental processing with Partitioned parallelism is different because the Hadoop framework does not share memory state across jobs (i.e., each new job starts with a clean slate). Figure 5.9 shows the implementation of Incremental processing with Partitioned parallelism of window W_2. Notice that the synopsis is also read as input by the map tasks. Moreover, the control program must also set the input datasets of the job to contain the plus (P_2) and minus (M_2) tuples for the current window. The map tasks then tag each input tuple based on its type (+,-, or W1 in the figure). The reduce tasks are then responsible for applying aggregation based on the plus (tuples with + tag) and minus tuples (tuples with - tag) to the corresponding synopsis (tuples with

W1 tag).

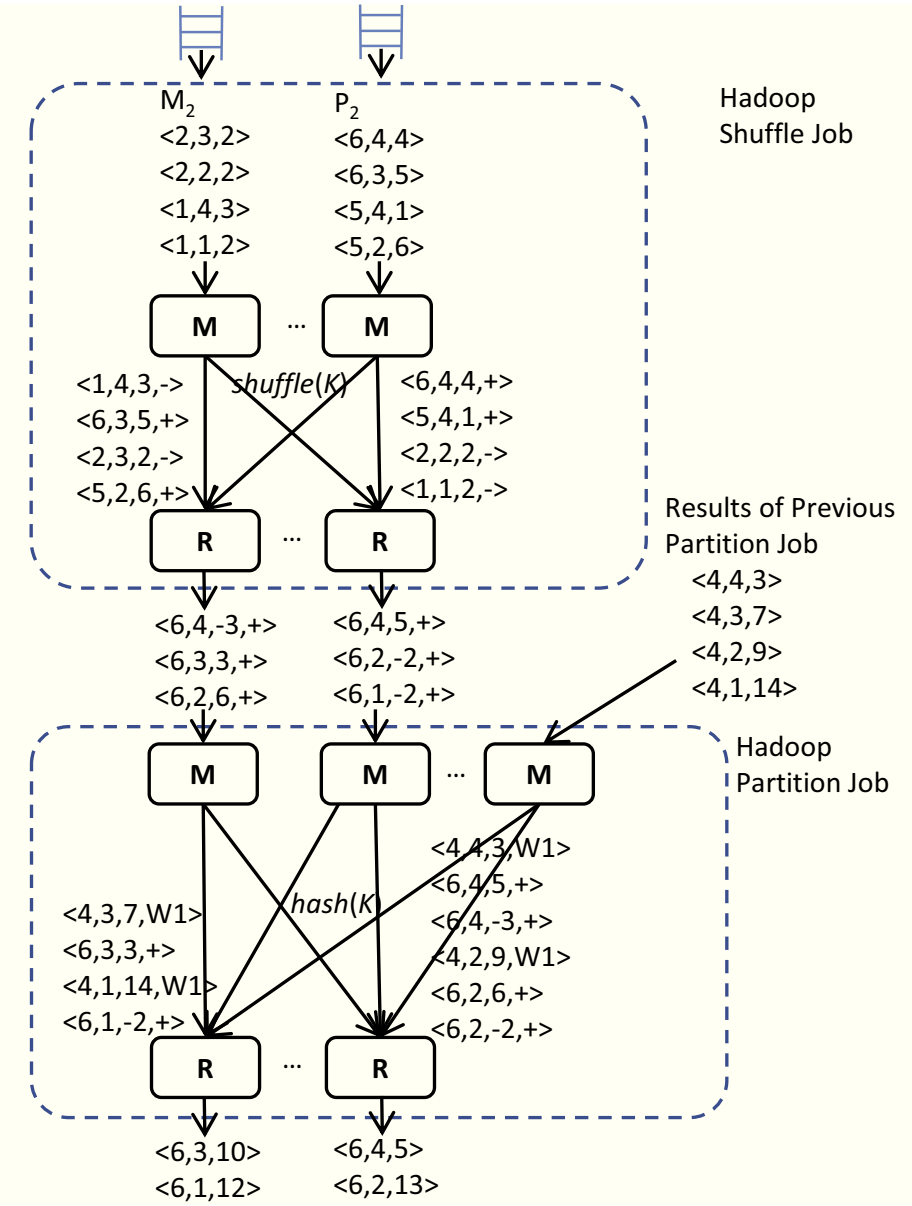


FIGURE 5.10: Task-level illustration of a Hadoop job performing Incremental processing with Shuffled parallelism and then another Hadoop job performing Incremental processing with Partitioned parallelism for window W_2 of the example stream.

For plans with Shuffled followed by Partitioned parallelism, the implementation contains two jobs. For Non-incremental processing, the first job randomly sends the output tuples to the reduce tasks, which can be achieved by implementing a

custom partitioning function in Hadoop. The reduce tasks then apply the aggregation function to each group that they receive. The second job then uses the output of the first job as input. Similar to the implementation of the Hadoop job for Non-incremental processing, the map tasks of the second job only set the group key of each input tuple as map output key. For each group, the reduce tasks generate the final aggregation results of the window.

While the plus tuples, minus tuples, and synopsis are set as input to the job for the implementation of Incremental processing with Partitioned parallelism, only plus and minus tuples are set as input to the first job of Incremental processing with Shuffled followed by Partitioned parallelism (see the inputs labeled P_2 and M_2 of the first job in Figure 5.10). The map tasks tag the input tuples as either + or - and then randomly distribute the tuples to the reduce tasks. For each group that the reduce tasks receive, the aggregation function is applied. The second job then uses these results and the synopsis (results of the previous window) as inputs. The map tasks tag the input tuple appropriately as + or W1. The reduce tasks then generate the final aggregation results by applying tuples to their corresponding synopsis (tuples with W1 tag).

For Hierarchical processing, each subwindow is processed by a separate job. The outputs of these jobs are then used by another job to generate the aggregation results of its corresponding window. As mentioned in Section 5.3, subwindows and windows can be pipelined. In our implementation, pipelining happens for job submissions. Specifically, jobs for processing subwindows are submitted to the Hadoop cluster only based on the availability of the input stream and does not have to wait for the jobs that are processing windows. Similarly, if all subwindow results for a particular window are available, then the corresponding job is submitted.

Figure 5.11 shows the task-level implementations of Hadoop jobs for Incremental Hierarchical processing with Partitioned parallelism. The control program has to

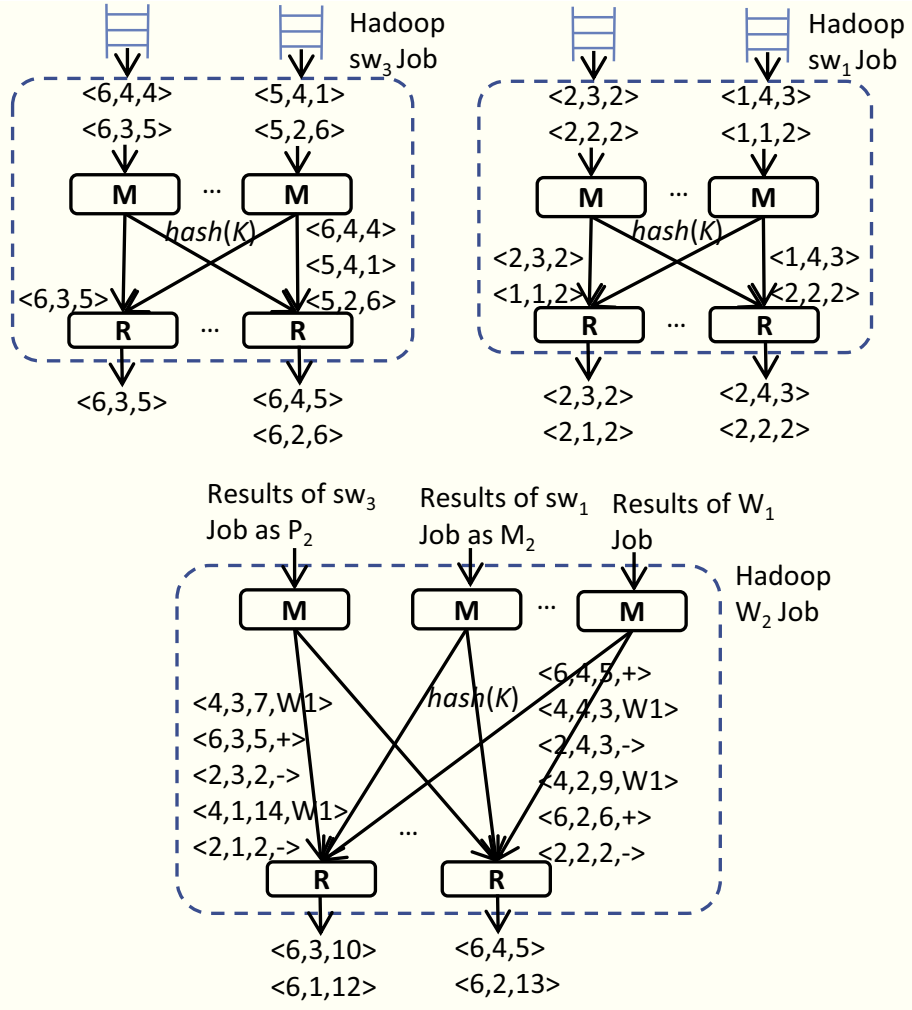


FIGURE 5.11: Task-level illustration of Hadoop jobs for Incremental Hierarchical processing with Partitioned parallelism of window W_2.

properly set the corresponding results of the subwindow jobs and previous window job as plus tuples, minus tuples, and synopsis. For the W_2 job, the results of sw_1 job are set as M_2, the results of sw_2 job are set as P_2, and the results of the W_1 job are set as the synopsis.

5.5 Cost-based Optimization

The previous section described the execution plan space for windowed aggregation queries. In this section, we describe the design and implementation of *Cyclops*, which is a cost-based optimizer that picks the best execution plan—which is the combination of logical plan and execution engine that gives minimum cost—to run the query. Internally, our optimizer has a *Bootstrap* module, *What-if engine*, and *Optimization* module.

Bootstrap: The goal of the Bootstrap module is to generate training data that can be used to learn cost models for the various execution plans. It currently supports a total of 16 different execution plans based on the various combinations of logical plan and system described in Section 5.4. Moreover, the aggregation function can be a black-box (i.e., user-defined aggregate or UDA). It is natural to expect that the cost model for sum aggregation will be different from the cost model for a complex UDA. Thus, a cost model is generated for each unique combination of aggregation function and execution plan. A couple of points are noteworthy. If the aggregation function is holistic or semi-holistic, then not all 16 execution plans are valid for that function. (If nothing is known about a UDA, then the optimizer assumes for safety that the UDA is holistic.) Also, the Bootstrap module needs to be run only once for a given back-end cluster, usually, when the aggregation function is developed by a user.

The metric of cost for an execution plan is the plan's *latency* l, namely, the average time to process a window. For modeling the cost of any plan, we first use our domain knowledge to identify the variables that can affect the plan's cost. These variables can be derived from the properties mentioned in Section 5.2 and include the amount of data processed per window, the number of unique groups per window, and the skew in the distribution of tuples within these groups.

The amount of data processed per window, n, can be estimated from the Slide s, Range r, and arrival rate a. For plans with Non-incremental processing, $n = r \times a$. For plans with Incremental processing, $n = 2 \times s \times a$. For plans with Hierarchical processing, $n = s \times a$ in the first aggregation pass which is always Non-incremental. The number of unique groups is determined by the domain size property u. We currently model the skew in the data distribution across groups as a Zipfian distribution with skew factor z, where $z = 0$ means uniformly distributed data. Thus, the cost model has the form $l = f(n, z, u)$ with three input variables and one output variable.

In order to build the cost model for any given aggregation function, the Bootstrap module first generates training data by running and recording the latency l to process a window for various property values of n, z, and u of each valid plan. After the training data is generated, a cost model function $l = f(n, z, u)$ is learned from the data. We currently use linear regression. We would like to point out that the design is general and extensible, so other modeling or regression techniques can be added in the future.

What-If Engine: The What-if engine maintains the cost models generated by the Bootstrap module. During optimization, the What-if engine estimates the latency of a given plan using the appropriate model. It accepts as inputs the Slide, Range, skew factor, number of unique groups, arrival rate parameters, type of aggregation function, and the query plan. The What-if engine first uses one of the three previously mentioned equations to convert the window properties to the amount of data processed in a window. It then uses the corresponding model of the given plan to generate the estimated latency.

Optimization: Given a windowed aggregation query, the optimization module first enumerates the valid execution plans. It then uses the What-if engine to estimate

the cost of each valid plan, and picks the plan with the minimum cost.

5.6 Experiments

In our experiments, we use a cluster of 11 m1.xlarge Amazon EC2 nodes. The respective master processes of Hadoop and Storm are run on one of the nodes, and the worker processes are run on the remaining 10 nodes. Esper, being centralized, runs on a single m1.xlarge node. Streams are stored in and read from a distributed fast read-write store (Hadoop Distributed File System) on the cluster.

5.6.1 Methodology

We have two primary goals in our evaluation. The first goal is to empirically study the execution plan space to understand the situations under which each plan can dominate the others. With this goal, we study and show different aspects of the plan space (Sections 5.6.2, 5.6.3, 5.6.4, 5.6.5, and 5.6.6). We want to bring out the complexity of selecting one plan over another plan, thus motivating Cyclops's ability to automatically find a good execution plan for a given windowed aggregation query. The second goal is to study the effectiveness of Cyclops' cost-based optimizer to find plans that are either the best or close to the best under any given scenario (Section 5.6.7). We use the non-holistic sum aggregation function throughout because all the 16 plans described in Section 5.4 are valid plans for non-holistic aggregations. Our methodology is as follows:

1. For the experiments related to the first goal, we vary one query property (e.g., Range, Slide, arrival rate, skew factor, domain size) at a time, while keeping the others constant. We compare the performance of relevant subsets of the plan space. We then evaluate the performance of the execution plan space on representative queries with multiple varying properties.

2. For the second goal, we first evaluate the accuracy of the What-if engine by comparing the actual and estimated performance of plans for a number of windowed aggregation queries. We then evaluate the performance improvements achieved by Cyclops.

We use latency and/or throughput as performance metrics. Latency is defined as the average time to process a window. Throughput is defined as the total time to process a fixed amount of the stream (e.g., time to process x seconds worth of an input stream). We summarize the insights learned from the experiments in Section 5.6.8.

5.6.2 Comparison of Systems

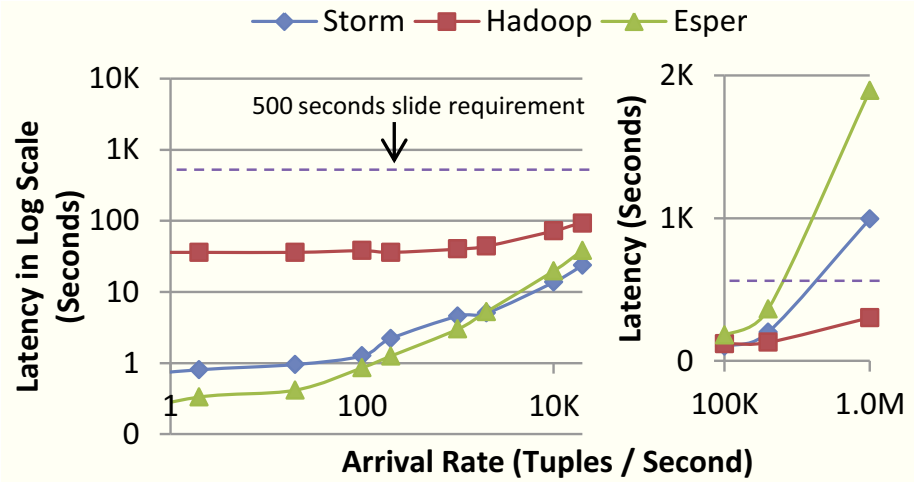


FIGURE 5.12: The latency of Esper, Storm, and Hadoop for different stream arrival rates. The Storm and Hadoop plans include Partitioned parallelism.

We will first show that none of the three systems—Esper, Storm, and Hadoop—dominates the others in all situations. In this experiment, we ran a windowed aggregation query with a Range and Slide of 500 seconds. The input stream has a domain size of 100 thousand with uniform distribution. We first obtained the best plan of each system, which is the corresponding execution plan with the lowest latency on each system. We then compared the latency of the best plans across

systems, which is Non-incremental processing with Pipelined parallelism for Esper and Non-incremental processing with Partitioned parallelism for Storm and Hadoop. We varied the arrival rate up to 1 million tuples/second so that 500-second window sizes go up to half a billion tuples.

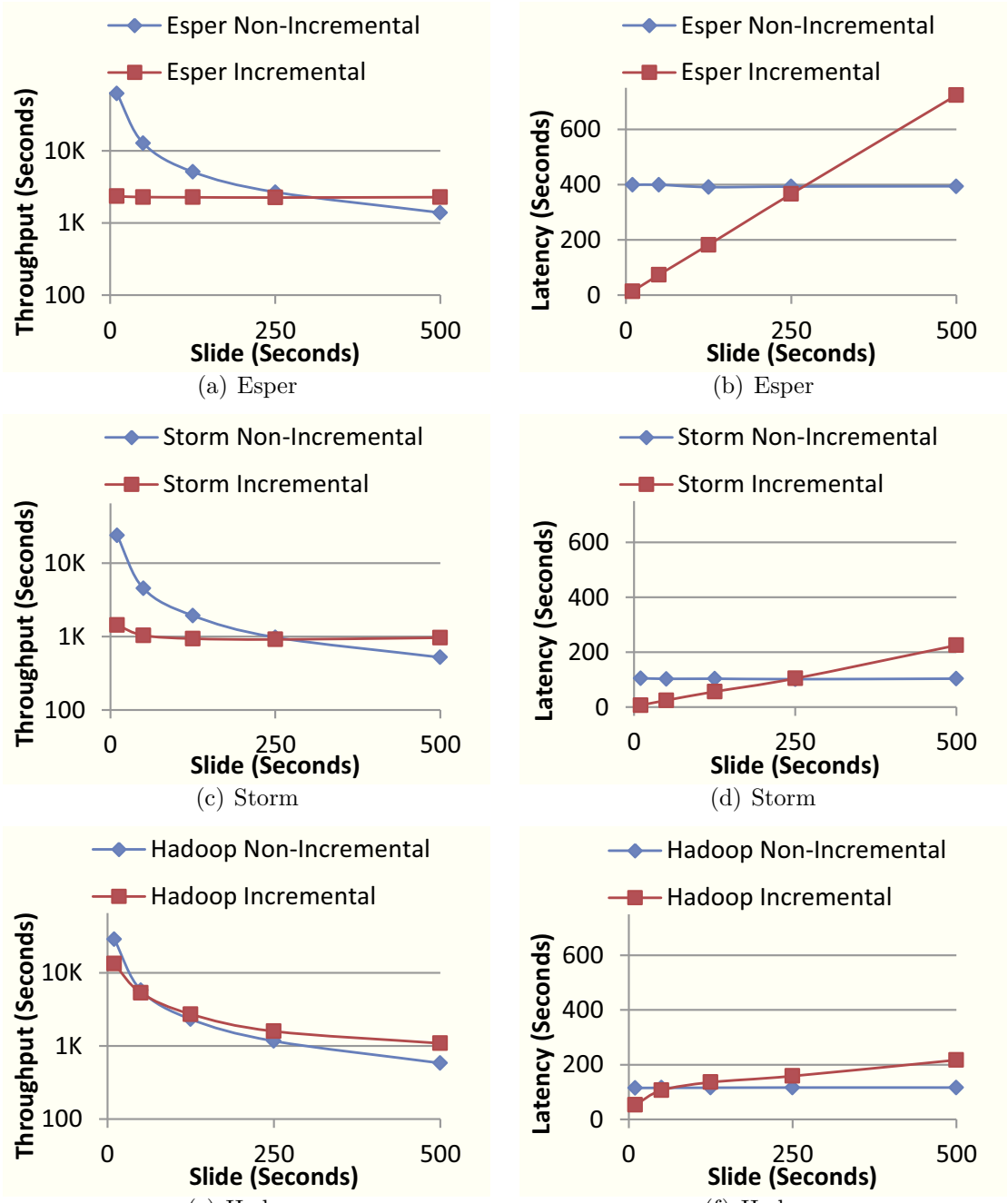
Figure 5.12 shows the latency versus arrival rate plot for this experiment. We split the plot into two pieces to clearly show the difference in latency of the systems. Note that the Y-axis of the arrival rate between 0 and 10 thousand tuples per second is in logarithmic scale. Note that no system always outperforms the others.

On the lower end of the arrival rate spectrum, Esper outperforms Storm and Hadoop. The reason is that Esper can utilize resources efficiently and does not have the overheads needed to support scalability (e.g., task scheduling, network communication). As the arrival rate is increased, the average window size increases; thereby increasing the compute and memory needs. Here, Storm's distributed nature, while still doing in-memory processing, enables it to outperform the others. However, as the arrival rate increases even further as shown in Figure 5.12(b), Hadoop performs the best because of its ability to use both CPU and I/O resources effectively and to avoid the per-tuple overheads found in the other two systems.

If the latency (average time to process a window) is greater than the Slide, then the system will not be able to keep up with the arrival rate. With each slide, the system will fall further behind with the query execution. Note from Figure 5.12 that, for an arrival rate of 1 million tuples per second, only Hadoop can keep the latency below the Slide of 500 seconds.

5.6.3 Comparison of Incremental and Non-Incremental Processing

This section studies the performance tradeoffs between Incremental and Non-incremental processing. In particular, whether there are cases of one type of processing outperforming the other. In this experiment, we have a windowed aggregation query with



(a) Esper (b) Esper

(c) Storm (d) Storm

(e) Hadoop (f) Hadoop

FIGURE 5.13: The throughput (Figures 5.13(a, c, and e)) and the latency (Figures 5.13(b, d, and f)) of Non-incremental and Incremental processing implementations of Esper, Storm, and Hadoop with different Slide. The Storm and Hadoop implementations include Partitioned parallelism.

a Range of 500 seconds. The stream has an arrival rate of 100 thousand tuples per second and a domain size of 100 thousand with uniform distribution. We varied the Slide of the query from 10 seconds to 500 seconds.

Figures 5.13(a, c, and e) show the throughput for processing 2500 seconds worth of input stream data on Esper, Storm, and Hadoop, respectively. As expected, with smaller Slide, Non-incremental processing takes a longer time to process the whole stream because, for each window, it has to process all tuples in the Range (500 seconds worth of tuples). In contrast, incremental processing only has to process the plus and minus tuples, which is determined by the Slide property. However, with large Slide (e.g., Slide of 500 seconds in the figure), Incremental processing performs worse because it has to process each tuple twice per window, once as a plus tuple and the other time as a minus tuple.

The result for Hadoop (Figure 5.13(e)) is also interesting because Non-incremental processing outperforms incremental processing earlier in the plot (at the point of 125 second Slide). Even though with 125 second Slide, the number of tuples to process per window incrementally is smaller than to process non-incrementally (250 thousand tuples vs 500 thousand tuples), it still performs worse. The main reason is that windows are non-continuously executed in Hadoop (i.e., new jobs for each window). Thus, to incrementally process a new window of a query, the job also has to read the synopsis as input (e.g., output of the previous job).

Figures 5.13(b, d, and f) show the latency for Incremental and Non-incremental processing on the three systems. Regardless of Slide, Non-incremental processing is constant because each window has to process all the tuples in the Range. On the other hand, the number of tuples to process a window for Incremental processing is dependent on the the Slide property.

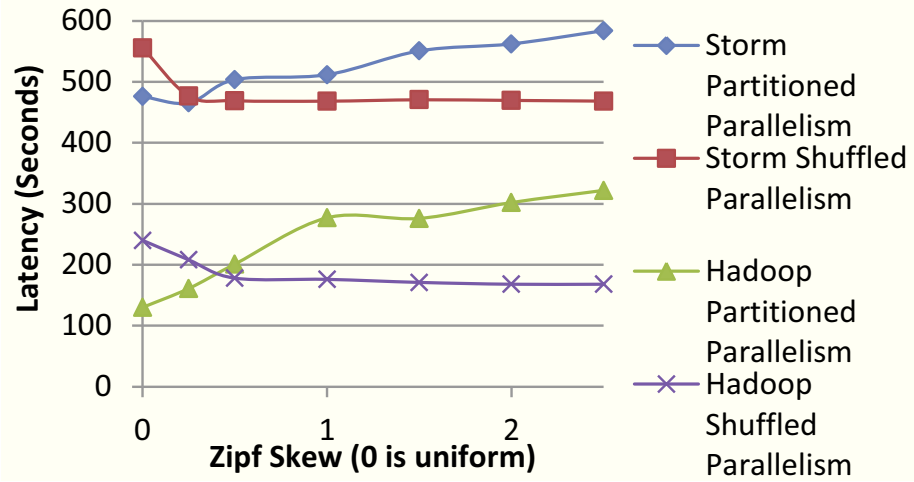


FIGURE 5.14: The latency of Partitioned parallelism and Shuffled followed by Partitioned parallelism of Storm and Hadoop for different skew factors.

5.6.4 Comparison of Partitioned Parallelism and Shuffled Parallelism

In this experiment, we are interested in comparing the performance of Partitioned parallelism and Shuffled parallelism and whether the behavior is comparable across different systems. We ran a windowed aggregation query with Range and Slide of 1000 seconds. The input stream has a domain size of 1 million and arrival rate of 100 thousand tuples per second. We varied the skew factor of the stream from 0 (uniform) to 2.5 (Zipfian distribution skew). For each system, we used the best plan (Non-incremental processing) with both types of parallelism.

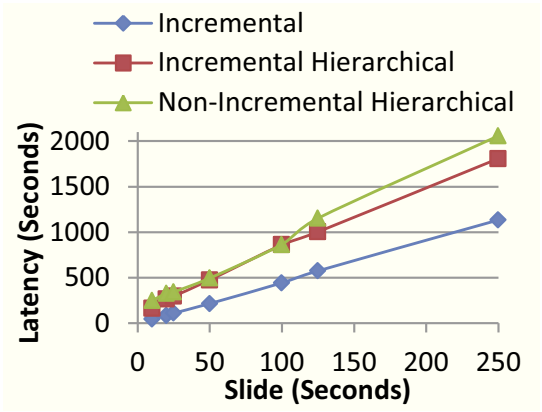
Figure 5.14 shows the latency versus skew factor plot of Partitioned parallelism and Shuffled followed by Partitioned parallelism on Storm and Hadoop. In both systems, as the skew increases, the difference in performance between Partitioned and Shuffled followed by Partitioned increases. In particular, the benefits of first randomly shuffling the tuples is proportional to the skew in the input stream. On a uniformly distributed stream, shuffling tuples is actually a wasted computation, which results in higher latency than Partitioned parallelism. It is also interesting that the cross-over point between plans happens earlier in Storm than Hadoop, which

makes the choice of choosing one plan over another depend on the system used. The pattern of decrease in latency from a uniformly distributed to a skewed distributed stream can be attributed to the size of the synopsis maintained for processing (i.e., a highly skewed stream will have smaller number of groups in a window).

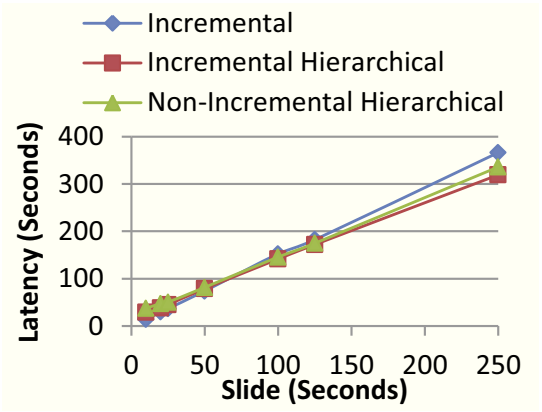
5.6.5 Comparison of Incremental and Hierarchical Processing

We are interested in comparing the performance of Incremental processing and the two Hierarchical processing plans. We use Incremental processing as our baseline for comparing with Hierarchical processing plans because they all share the goal of reusing work (e.g., processing only the differences for Incremental processing and processing subwindows for Hierarchical processing). Thus, we ran a windowed aggregation query with a Range of 500 seconds and the input stream has an arrival rate of 100 thousand tuples per second that is uniformly distributed across a domain size of 1 million. We varied the Slide from 10 to 250 seconds.

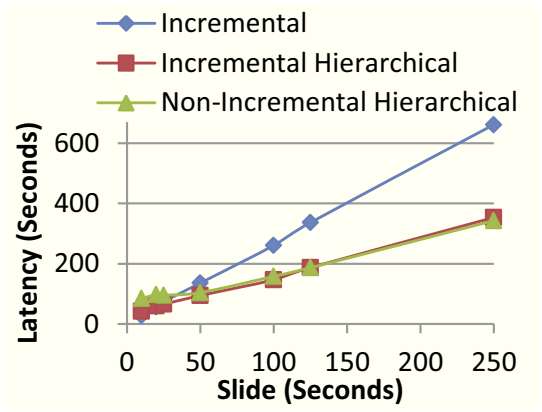
Figures 5.15(a, c, and e) show the latency per window of each plans on the three systems. As expected, for Esper, the plans with Hierarchical processing perform worse than incremental processing. This is due to the fact that with Hierarchical processing, there are more operators at work and more tuples (i.e, intermediate results) that need to be maintained (1 million groups per subwindow). For Storm and Hadoop, the Hierarchical plans actually perform better, especially for large Slide. The reason is that Partitioned parallelism in these systems reduces the amount of data maintained per node. Moreover, there is a batching effect happening in Hierarchical processing where input tuples for non-overlapping subwindows are aggregated and reused to generate the aggregation results of each window. As mentioned previously, Hierarchical processing also enables pipelining within operations. Specifically, the operators processing the subwindows are not blocked (or dependent) on the operators processing the windows. They are continuously processing the incoming



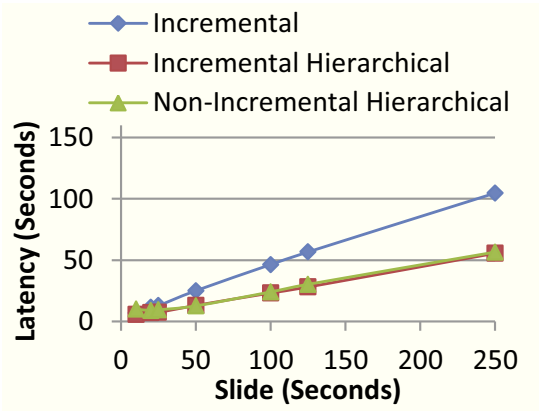
(a) Esper



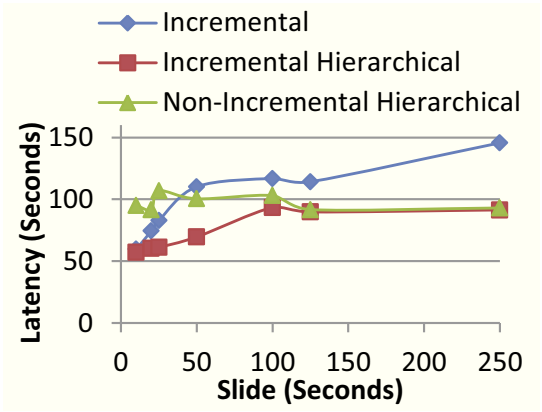
(b) Esper



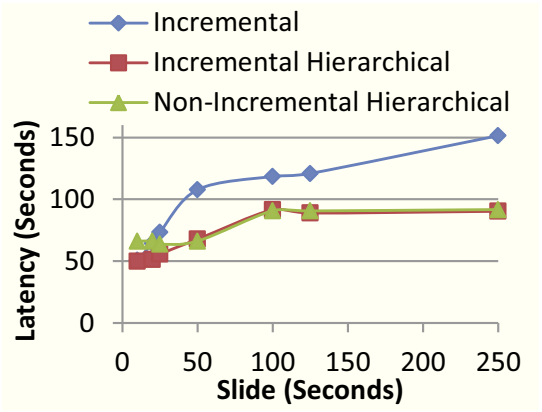
(c) Storm



(d) Storm



(e) Hadoop



(f) Hadoop

FIGURE 5.15: The latency per window of Incremental processing, Incremental Hierarchical processing, and Non-incremental Hierarchical processing implementations of Esper, Storm, and Hadoop with different Slide on an input stream with domain size of 1 million (a, c, and e) and 100 thousand (b, d, and f), respectively. The Storm and Hadoop implementations include Partitioned parallelism.

streams and generating results for the subwindows.

To verify this hypothesis, as shown in Figures 5.15(b, d, and f), we also ran the same experiment on an input stream with a smaller domain size (100 thousand). The plans with Hierarchical processing on Esper now perform slightly better than Incremental processing. Both Storm and Hadoop show similar performance improvements of Hierarchical processing over incremental processing as before, which can be attributed to the fact that the amount of data per node is already small in both experiments. We further ran Esper on a stream with a domain size of only 100. The result indicated even more substantial improvement of latency of the Hierarchical plans (around 126 seconds) over Incremental processing (around 235 seconds). Thus, there are tradeoffs based on the properties of the input stream that make one plan better than the other plans.

5.6.6 Plan Space Characterization

Whereas we empirically compared the plan space by varying a single query property in the previous sections, we are also interested in comparing the logical plan space within and across each system on different queries with multiple varying query properties. We use four representative windowed aggregation queries for the experiments in this section and the properties of the queries are shown in Table 5.2 (labeled a, b, c, and d). We ran all 16 possible plans on each of these queries.

Figure 5.16 shows the latencies of all plans on the four queries (for now focus on the vertical axis that is labeled Actual Latency). It can be clearly seen in the figure that there is a large difference in latencies in the plan space. The ratio of the worst overall plan (highest latency) to the best overall plan (lowest latency) ranges from 5 to 76. Table 5.3 also breaks down the best and worst plan within each system. Even within each system, the ratio of worst to best plan can be large. At worst, Esper has a ratio of 4 for query d, Storm has a ratio of 9 for query d, and Hadoop has a ratio

Table 5.2: The properties of the four windowed aggregation queries.

	Query				
	Slide	Range	Skew Factor	Domain Size	Arrival Rate
a	100s	300s	0	500K	100K
b	240s	480s	0.5	10M	62.5K
c	200s	200s	1	1M	125K
d	40s	160s	1.5	100M	250K

Table 5.3: The latencies of the best plan and worst plan on each system, and the latency of Cyclop's chosen plan, when running the four windowed aggregation queries (refer to Table 5.2).

	Esper		Storm		Hadoop		Cyclops
	Best Plan	Worst Plan	Best Plan	Worst Plan	Best Plan	Worst Plan	Chosen Plan
a	286s	445s	151s	506s	77s	183s	97s
b	104s	149s	421s	1038s	108s	171s	108s
c	84s	160s	341s	2481s	129s	183s	146s
d	31s	126s	271s	2446s	89s	168s	32s

of 2.3 for query a.

The experiment results show that using the wrong plan for running a windowed aggregation query can result in vastly suboptimal performance. Moreover, our analysis indicates that the best and worst plans for each system can be different, which further reinforces the complexity of selecting one plan over the another plan. For example in query b, Esper's best and worst plans are Incremental Hierarchical processing with Pipelined parallelism and Non-incremental processing with Pipelined parallelism, respectively. Storm's best and worst plans are Non-incremental processing with Shuffled parallelism and Non-incremental processing with Partitioned parallelism, respectively. Hadoop's best and worst plans are Non-incremental Hierarchical processing with Partitioned parallelism and Incremental processing with Shuffled parallelism, respectively. Finally, our results also show that the performance of the logical plans can overlap across different systems.

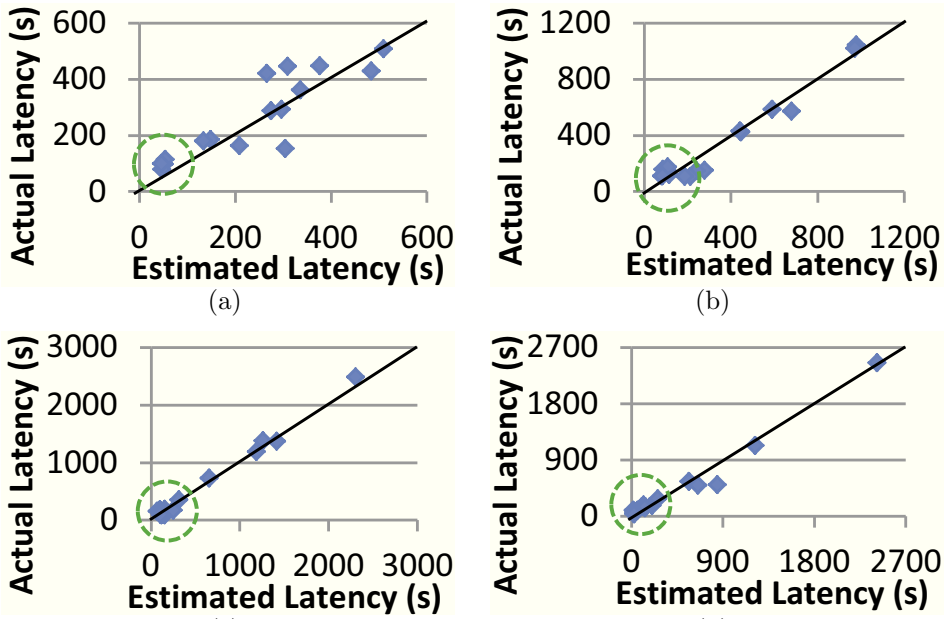


FIGURE 5.16: The actual vs estimated (by the What-if engine of Cyclops) latency of different execution plans on four windowed aggregation queries.

5.6.7 Analysis of Cyclops

The complexity of the plan space motivates the need for automatically selecting a good plan. In this section, we evaluate the performance of Cyclops.

We ran the bootstrap module with the following input values for the amount of data in a window n ={100 thousand, 1 million, 10 million, 50 million} tuples, skew factor z ={0, .5, 1.5}, and domain size u ={100 thousand, 10 million, 100 million} to generate a cost model for each plan.

Figure 5.16 shows the scatter plot of the actual and estimated (by the What-if engine) latency for the 16 plans on the queries shown in Table 5.2. Ideally the points should fall on the solid line. The inaccuracies can be attributed to regression error. However, note that there is still a clear identification of plans with the best and worst performance (the cluster of top best plan performances is marked by the dotted circle). For the task of choosing a good plan, as opposed to exact latency

prediction for all plans, this level of accuracy works reasonably well.

Table 5.3 also shows the latency of the best plan, worst plan, and Cyclops' chosen plan for each query. In particular, the latency of Cyclops' chosen plan for the four queries is relatively close to the latency of the best plan. The error is usually small and can be reduced further by generating more observations to train the cost models. The table also shows that Cyclops is able to achieve between 5-76X speedup over the worst plan. Such speedups clearly motivate the need for Cyclops because the majority of users do not have a deep understanding of the plans and engines for executing queries; and may end up choosing plans that are vastly suboptimal.

5.6.8 Summary

We summarize the observations from our experiments in the following list.

- No one type of system completely dominates the other types of systems.
- Non-incremental processing provides performance stability, regardless of the size of the Slide. In contrast, the performance of Incremental processing is dependent on the size of the Slide.
- Shuffled parallelism provides more performance stability as the skew of the stream increases (i.e., the latency remains roughly constant). In contrast, the performance without shuffled parallelism degrades sharply as the skew of the stream increases.
- For Hierarchical processing, there is a tradeoff between the size of the partial results and the extra work from applying multiple passes of aggregation. Performance improvement over Non-hierarchical processing is substantial with small partial results. On the other hand, on large partial results, the extra work from applying multiple passes of aggregation can result in worse performance than Non-hierarchical processing.

- The performance crossover point between plans can vary depending on the type of system.

- Training data to build cost models of the plan space can produce reasonable estimations of plan performance.

5.7 Related Work

Query Plan Space: To our knowledge, we are the first to systematically characterize the full execution plan space for windowed aggregation queries. One of the reasons that allowed us to explore the full space is that we consider all types of systems, which does not restrict the choice of plans. Nevertheless as we have described in Sections 5.3 and 5.4, there is related work that has discussed or explored a subset of the full space.

SECRET [20] proposes a model for comparing the underlying query semantics of different stream processing systems, which is orthogonal to the goals of this chapter of characterizing the execution plan space for the query semantics described in Section 5.2.

Windowed aggregation (any continuous query for that matter) can be considered as a special case of materialized view maintenance. Incremental processing was described by Gupta et al. [46], in the context of incremental view maintenance in database systems. Palpanas et al. [95] describes incremental view maintenance on different types of aggregation functions. More recently, REX [85] used Incremental processing, which they called delta-based computation, for processing iterative queries.

The pane-based approach [72] to processing windowed aggregation queries falls under the Hierarchical processing dimension of the logical plan space. Recent work on distributed repeated-batch systems have also used the hierarchical approach to

processing continuous queries [49, 79]. For example, Comet [49] automatically splits a query into subqueries and reuses results of previous subqueries to generate query results.

A subset of the choice of parallelism has also been discussed by a number of papers. For example, Backman et al. [16] describes the usage and replication of punctuation tuples for partitioned parallelism in distributed streaming systems. Pipelined and partitioned parallelism are natively supported and can easily be used in most of the systems we have previously mentioned (e.g., [1, 10, 25]). Moreover, Hive [54] has native support for Shuffled parallelism. However, as with the other choices in the full plan space, none of this work has empirically or analytically compared the tradeoffs of one choice over the others for executing windowed aggregation queries.

Systems: As Table 5.1 shows, there is a wide range of systems that can execute continuous queries. Designing and implementing a new continuous query system is orthogonal to the goals of this work. Rather, the dimensions that categorize the systems are part of the full execution plan space for continuous queries. As we showed in Section 5.6.2, no one type of system always outperforms all the other systems. Our recent position paper [73] also made this case and called for the need to manage these systems in a unifying management system, called DBMS$^+$.

Cost-based Optimization: There is research work related to the overall goals of Cyclops. MaxStream [19] is a middleware platform that integrates stream processing systems, but lacks a cost-based optimizer for selecting the most suitable plan and system. Although focusing on queries on sensor networks, the ASPEN project [78] has a federated optimizer for optimizing queries across different systems, but lacks support for orchestration and management of queries. In contrast, Cyclops generates training data and uses regression techniques to build models for the plan space; which is extensible and avoids challenges of reconciling the cost metrics obtained from each system's optimizer. Cyclops directly uses the systems without requiring an engine-

specific What-if Engine or cost-based optimizer. Most of the newer systems like Storm and Hadoop lack these modules anyway.

5.8 Conclusions

Windowed aggregation queries represent an important class of continuous queries that has a wide range of applications. However, the execution plan space for these queries is complex. We systematically characterize the plan space by first describing three dimensions that comprise the logical plan space: i) Incremental Vs. Non-incremental processing, ii) Hierarchical processing, and iii) the type of parallelism. The execution plan space includes the choice of system, which can be described as i) centralized Vs. distributed processing, and ii) streaming Vs. repeated-batch execution. We provide implementations of the execution plans on Esper, Storm, and Hadoop, which capture the broad spectrum of systems found in the backend tier of data-intensive services, and also describe query and stream properties that affect performance. A comprehensive experimental evaluation shows the interesting tradeoffs in the plan space. Finally, we describe the design and implementation *Cyclops*, a cost-based optimizer that picks a good plan for executing a windowed aggregation query.

6

Conclusions and Future Work

As we have described in Section 1, data-intensive services are comprise of multiple complex systems, which currently requires careful management from application developers and system administrators. This dissertation outlined our contributions toward automatically managing the workloads of data-intensive services. We have categorized the systems found and used in a typical data-intensive service into three tiers: display tier, storage tier, and analytics tier (refer to Figure 1.1). We presented automated workload management solutions in each of these tiers.

Display Tier: For stateless systems deployed in the cloud, such as clustered Web servers, workload can be managed through dynamic provisioning of resources. We introduced a new control policy, called proportional thresholding, which addresses the challenges posed by cloud providers, such as coarse-grained actuators (i.e., resources are defined in terms of virtual machines with fixed configurations).

Storage Tier: *Elastore* automatically manages the workload of storage systems typically found in the storage tier by dynamically provisioning storage nodes. It coordinates two controllers: Horizontal Scale Controller for managing the size of the

cluster, and Data Rebalance Controller for ensuring data is always balanced across the cluster, to ensure that the managed storage system achieves its service level objective, while still being resource efficient.

Analytics Tier: *Stubby* automatically optimizes batch data-parallel workflows in a cost-based manner. We characterized the large plan space for transforming the workflows. Stubby works with any number of interfaces for generating workflows and can efficiently search through the transformation plan space to apply the set of transformations that results in minimum workflow execution time.

Similarly, we have also characterized the execution plan space for another important type of workload running in this tier: continuous data-parallel workflows. By studying this type of workload, we have developed *Cyclops*, which is a cost-based optimizer that can select the most suitable plan. Since it is also common to have multiple systems running in the analytics tier [84], Cyclops also selects the most suitable system for executing a given continuous workflow.

While this dissertation has shown that with the right sets of policies and mechanisms, data-intensive services can be managed automatically to ensure that workloads are running at an acceptable level of performance, there are still a number of challenges that need to be addressed in order to have a fully integrated end-to-end workload management solution. We now list possible future research directions.

- In this dissertation, we assumed that fine-grained resources are not shared across different systems of data-intensive services. Specifically, a virtual machines instance from the cloud run a specific system, which results in coarse-grained resource partitioning of cluster resources. However, there is now an emerging trend to run multiple systems on the same cluster resources using platforms that can balance resource sharing and isolation [53, 131], which pose new challenges, such as minimizing performance interference across different

systems, and finding the optimal fine-grained resource allocations for all systems.

- How the data is stored and accessed by systems can affect the performance of workload, which presents an interesting future research direction. There are recent work that has made accessing data on systems more efficient, e.g., column stores on HDFS and serialization formats for storing structured data [100]. Moreover, a number of systems can now utilize different types of resources to store data [136]. Depending on the workload requirements, one choice of layout of data may be better than another. There is thus an opportunity to automatically find the best layout of data that are globally optimal for the systems that comprise data-intensive services.

- As mentioned in a recent paper [73], there are a number of specialized systems built over the years. Often times, these systems can run the same type of workload, albeit with different performance implications. It is a challenge to determine which systems to include for a particular data-intensive service because as we have shown in Chapter 5, the performance cross-over points are not clear cut. It would be interesting to develop a methodology or benchmark to characterize and categorize systems across different types of workload and workload requirements. This can then help administrators to systematically decide which systems to include for their services.

- In this dissertation, we developed controllers and optimizers for managing workloads in each specific tier. However, workloads in each tier interacts with each other. For example, the results of workloads processed in the analytics tier are used by the display tier. A fully integrated end-to-end solution require coordination of each management systems to ensure that choices made do not

interfere with each other. It is a research challenge to determine the acceptable level of coordination required to achieve the workload requirements.

- We focused on performance as the metric for workload management. However, data-intensive services usually have multiple (possibly complex) requirements. Other requirements include availability, consistency, and cost. In this case, managing workloads now involves comparing choices across multiple dimensions.

Appendix A

Additional Details on Stubby

A.1 Proof of Intra-job Vertical Packing Transformation

We present a formal proof by contradiction to show that the preconditions for a plan P along with the postconditions for a plan P' are sufficient to ensure P is equivalent to P' during an intra-job vertical packing transformation.

Proof.

1. Given: Let j_p be the producer job, j_c the consumer job, R_p the reduce function of the producer job, M_c the map function of the consumer job, R_c the reduce function of the consumer job, p the initial plan satisfying preconditions, and P' the generated plan satisfying postconditions.

2. Assume: P is not equivalent to P'. Since the intra-job vertical packing transformation will remove the shuffle phase—and hence, the sorting and partitioning—from the consumer job j_c, the non-equivalence can occur in one of two ways: (a) two (or more) key-value pairs with the same key $j_c.K_2 = k$ appear in the input of R_c of two (or more) tasks in job j_c in plan P', or (b) two (or more)

key-value pairs with the same key $j_c.K_2 = k$ appear in two (or more) different groups in the input of R_c of a task in job j_c in plan P'.

3. First, consider two key-value pairs with the same key $j_c.K_2 = k$ that appear as input to R_c for two different tasks in job j_c in plan P'.

4. The precondition states that M_c can output a key-value pair with $j_c.K_2 = k$ only from one or more key-value pairs with $j_c.K_2 = k$ given as input to the reduce function R_p of j_p. Hence, all key-value pairs with the same key $j_c.K_2 = k$ that appear in the input of R_c of more than one tasks of j_c of plan P' must be from the output of M_p of any map task of j_p.

5. The first postcondition (i.e., setting the partition function of j_p) for P' ensures that all key-value pairs with key $j_c.K_2 = k$ are shuffled to a single task of R_p.

6. The second postcondition ensures that the entire output of a single R_p is read by a single task of M_c and, consequently, consumed by the chained R_c.

7. It follows from Step 6 that all key-value pairs with key $j_c.K_2 = k$ will reach a single R_c, which contradicts the statement in Step 3.

8. Second, consider two key-value pairs with the same key $j_c.K_2 = k$ that appear in two different groups in the input of R_c of a single task in job j_c in plan P'.

9. Using the precondition mentioned in Step 4, key-value pairs with the same key $j_c.K_2 = k$ that appear in the input of R_c of a single task in plan P' must be from the output of M_p of any map task of j_p.

10. The first postcondition for P' ensures that key-value pairs with the same key $j_c.K_2 = k$ are sorted and grouped together. Hence, a single group containing all key-value pairs with the same key $j_c.K_2 = k$ are given as a single input to R_p.

11. The second postcondition ensures that the entire output of a single R_p is read by a single task of M_c and, consequently, consumed by the chained R_c.

12. It follows from Step 11 that all key-value pairs with key $j_c.K_2 = k$ will reach a single R_c in the same group, which contradicts the statement in Step 8.

13. Both Steps 3 and 8 have lead to the corresponding contradictions in Steps 7 and 12. Therefore, the original assumption that P is not equivalent to P' leads to contradictions.

This completes the proof. □

A.2 Pig Latin Queries Used in the Experiments

Information Retrieval:

```
REGISTER stubby_udfs.jar
words = LOAD '/randomtext' USING PigStorage('\t') AS (docid,
    wordid);
-- Calculate the frequency of word per doc
docid_wordid = GROUP words BY (docid, wordid);
wordfreqindoc = FOREACH docid_wordid GENERATE FLATTEN(group),
   COUNT(words) as word_count_per_doc;
-- Calculate the total words per doc
groupedbydocid = GROUP wordfreqindoc BY docid;
wordcountsfordoc = FOREACH groupedbydocid GENERATE group as docid
    , FLATTEN(wordfreqindoc.(wordid, word_count_per_doc)) AS (
    wordid, word_count_per_doc), SUM(wordfreqindoc.
    word_count_per_doc) AS word_counts_for_doc;
-- Calculate the number of docs that has the word
groupedbywordid = GROUP wordcountsfordoc by wordid;
docfreqbywordid = FOREACH groupedbywordid GENERATE group as
    wordid, FLATTEN(wordcountsfordoc.(docid,word_count_per_doc,
    word_counts_for_doc)) AS (docid, word_count_per_doc,
    word_counts_for_doc), COUNT(wordcountsfordoc) AS
    doc_freq_by_word;
-- Calculate TF-IDF
tf_idf = FOREACH docfreqbywordid GENERATE wordid, docid, edu.duke
    .stubby.pig.udfs.TFIDF(word_count_per_doc, word_counts_for_doc
    , 1000L, doc_freq_by_word) AS tfidf;
STORE tf_idf INTO '/tfidf' USING PigStorage();
```

Social Network Analysis:

```
REGISTER stubby_udfs.jar
paper_author = LOAD '/paperauthor' USING PigStorage('\t') AS (
    paperid:long, authorid:int);
-- Create paper-author grouping
groupedbypaper = GROUP paper_author BY paperid;
-- Count the frequency of authorpairs
authorpair = FOREACH groupedbypaper GENERATE $0 AS paperid,
    FLATTEN(edu.duke.stubby.pig.udfs.AUTHORPAIR($1)) AS
    author_pair;
groupedbyauthorpair = GROUP authorpair by author_pair;
pair_count = FOREACH groupedbyauthorpair GENERATE group AS
    author_pair, COUNT(authorpair) AS count;
-- Get the top 20 pairs
ordered_authorpair = ORDER pair_count BY count DESC;
top20 = LIMIT ordered_authorpair 20;
STORE top20 INTO '/coauthor' USING PigStorage();
```

Log Analysis:

```
rankings = LOAD '/complex/rankings' USING PigStorage('|') AS (url
    :chararray, page_rank:float, duration);
uservisits = LOAD '/complex/uservisits' USING PigStorage('|') AS
    (url:chararray, ip, visit_date:chararray, ad_revenue:float,
    user_agent, country, lang_code, search, duration);
--join by url and also filter by date
filtered_uservisits = FILTER uservisits BY visit_date >= '1974
    -1-1' AND visit_date <= '1974-7-1';
rankings_uservisits = join rankings BY url, filtered_uservisits
    BY url;
-- get the sum of ad_revenue and average of page_rank, grouped by
     IP
groupedbyip = GROUP rankings_uservisits BY ip;
pagerank_revenue = FOREACH groupedbyip GENERATE group AS ip, AVG(
    $1.page_rank) As avg_pagerank, SUM($1.ad_revenue) AS
    sum_revenue;
--get the top revenue
ordered_pagerank_revenue = ORDER pagerank_revenue BY sum_revenue
    desc;
top = LIMIT ordered_pagerank_revenue 1;
STORE top INTO '/complex/results' USING PigStorage();
```

Web Graph Analysis:

```
REGISTER stubby_udfs.jar
```

```
rankings = LOAD '/pagerank/rankings' USING PigStorage('\t') AS (
    pageid:int, rank:double);
pages = LOAD '/pagerank/pages' USING PigStorage('\t') AS (pageid:
    int, out_links:chararray);
-- Join Rankings and pages by pageid
joined_table = COGROUP rankings BY pageid, pages BY pageid;
-- emit each outgoing links
outlink_table = FOREACH joined_table GENERATE $0 AS pageid,
    FLATTEN(edu.duke.stubby.pig.udfs.OUTGOINGLINKS($2)) AS
    out_link_id:chararray, edu.duke.stubby.pig.udfs.INTERMRANK($1,
    $2) AS interm_rank;
-- group by out_link_id
outlink_table_by_out_id = GROUP outlink_table BY out_link_id;
new_ranks = FOREACH outlink_table_by_out_id GENERATE $0, edu.duke
    .stubby.pig.udfs.PAGERANK($1) AS page_rank;
STORE new_ranks INTO '/pagerank/newranks' USING PigStorage();
```

Business Analytics Query:

```
lineitem = LOAD '/tpch17/lineitem' USING PigStorage('|') AS (
    partkey:long, orderkey:long, suppkey:long, linenumber:long,
    quantity:float, extendedprice:float, discount:float, tax:float
    , returnflag, linestatus, shipdate, commitdate, receiptdate,
    shipinstruct, shipmode, comment);
part = LOAD '/tpch17/part' USING PigStorage('|') AS (partkey:long
    , name, mfgr, brand:chararray, type, size:int, container:
    chararray, retailprice:float, comment);
-- join by partkey, where p_brand = 'Brand#54', p_container = '
    MED CAN'
filtered_part = FILTER part BY brand == 'Brand#54' AND container
    == 'MED CAN';
cogroup_line_part = COGROUP lineitem BY partkey, part by partkey;
-- project the average quantity of for each part key
project_lp = FOREACH cogroup_line_part GENERATE group AS partkey,
     .2 * AVG($1.quantity) AS avg_quantity;
-- perform a filtered join of lineitem and project_lp where
    quantity < avg_quantity
join_line_project_lp = join lineitem BY partkey, project_lp BY
    partkey;
filtered_join_line_project_lp = FILTER join_line_project_lp BY
    quantity < avg_quantity;
-- aggregate the extended_price
grouped_filtered = GROUP filtered_join_line_project_lp ALL;
result = FOREACH grouped_filtered GENERATE SUM($1.extendedprice)
    / 7 as avg_yearly;
STORE result INTO '/tpch17/results' USING PigStorage();
```

Business Report Generation:

```
lineitem = LOAD '/tpch/lineitem' USING PigStorage('|') AS (
    orderkey:long, part:long, supp:long, linenumber:long, quantity
    :float, extendedprice:float, discount:float, tax:float,
    returnflag, linestatus, shipdate, commitdate, receiptdate,
    shipinstruct, shipmode, comment);
-- perform filter on orderkey
lineitem1 = FILTER lineitem BY (part <= 2840000);
-- get the sum of extendedprice, grouped by orderkey,part
lineitembyorderpart1 = GROUP lineitem1 BY (orderkey, part);
aggbycustprio1 = FOREACH lineitembyorderpart1 GENERATE FLATTEN(
    group) AS (orderkey,part), SUM($1.extendedprice) AS sum_price;
-- get the total extended price per order
groupbycustkey1 = GROUP aggbycustprio1 BY orderkey;
aggbycust1 = FOREACH groupbycustkey1 GENERATE group AS orderkey,
    SUM($1.sum_price) AS total_price_per_order;
-- count the number of orders
groupbyprice1 = GROUP aggbycust1 BY total_price_per_order;
aggbyprice1 = FOREACH groupbyprice1 GENERATE group AS
    total_price_per_cust, COUNT($1) AS countcust_per_price;
STORE aggbyprice1 INTO '/results/consumer1' USING PigStorage();
-- perform filter on orderkey
lineitem2 = FILTER lineitem BY (part > 139160000);
-- get the max of extendedprice, grouped by orderkey,supp
lineitembyorderpart2 = GROUP lineitem2 BY (orderkey, supp);
aggbycustprice2 = FOREACH lineitembyorderpart2 GENERATE FLATTEN(
    group) AS (orderkey,supp), MAX($1.extendedprice) AS max_price;
-- get the global max price per order
groupbycustkey2 = GROUP aggbycustprice2 BY orderkey;
aggbycust2 = FOREACH groupbycustkey2 GENERATE group AS orderkey,
    MAX($1.max_price) AS max_price_per_cust;
-- count the number of prices per order
groupbyprice2 = GROUP aggbycust2 BY max_price_per_cust;
aggbyprice2 = FOREACH groupbyprice2 GENERATE group AS
    max_price_per_cust, COUNT($1) AS countcust_per_max_price;
STORE aggbyprice2 INTO '/results/consumer2' USING PigStorage();
```

Post-Processing Jobs:

```
part = LOAD '/tpch/part' USING PigStorage('|') AS (partkey:int,
    name:chararray, mfgr:chararray, brand:chararray, type:
    chararray, size:int, container:chararray, retailprice:float,
    comment:chararray);
-- Calculate the correlation grouped, by container
consumer1_inter = GROUP part BY container;
```

```
consumer1 = FOREACH consumer1_inter GENERATE $0 AS container, COR
    ($1.partkey, $1.size, $1.retailprice) AS cor_part_size_retail;
STORE consumer1 INTO '/results/consumer1' USING PigStorage();
-- Calculate the covariance grouped, by container
consumer2_inter = GROUP part BY container;
consumer2 = FOREACH consumer2_inter GENERATE $0 AS container, COV
    ($1.partkey, $1.size, $1.retailprice) AS cor_part_size_retail;
STORE consumer2 INTO '/results/consumer2' USING PigStorage();
```

User-defined Logical Splits:

```
lineitem = LOAD '/tpch/lineitem' USING PigStorage('|') AS (
    orderkey:long, partkey:long, suppkey:long, linenumber:long,
    quantity:float, extendedprice:float, discount:float, tax:float
    , returnflag, linestatus, shipdate, commitdate, receiptdate,
    shipinstruct, shipmode, comment);
-- perform grouping of lineitem by partkey
grouped = GROUP lineitem BY partkey;
grouped_part = FOREACH grouped generate $0 AS partkey, FLATTEN($1
    .(orderkey,suppkey,linenumber,quantity,extendedprice,discount,
    tax,returnflag,linestatus,shipdate,commitdate,receiptdate,
    shipinstruct,shipmode,comment)) AS (orderkey:long, suppkey:
    long, linenumber:long, quantity:float, extendedprice:float,
    discount:float, tax:float, returnflag, linestatus, shipdate,
    commitdate, receiptdate, shipinstruct, shipmode, comment);
-- consumer 1 filters by partkey, group by orderkey and aggregate
     the quantity
filter1 = FILTER grouped_part BY partkey < 400000;
consumer1 = GROUP filter1 BY orderkey;
consumer1_agg = FOREACH consumer1 GENERATE $0 AS orderkey, SUM($1
    .quantity) AS sum_quantity;
-- consumer 1 filters by partkey, group by suppkey and aggregate
    the extendedprice
filter2 = FILTER grouped_part BY partkey > 141600000;
consumer2 = GROUP filter2 BY suppkey;
consumer2_agg = FOREACH consumer2 GENERATE $0 AS suppkey, AVG($1.
    extendedprice) AS avg_price;
STORE consumer1_agg INTO '/results/consumer1' USING PigStorage();
STORE consumer2_agg INTO '/results/consumer2' USING PigStorage();
```

A.3 Extracting Annotations

As mentioned in Section 4.1, there is a growing number of interfaces for generating MapReduce Workflows (refer to Figure 4.2). These interfaces can be grouped into

two categories: i) high-level declarative languages, such as Pig, Hive, Cascading, and SCOPE, and ii) programming languages, such as Java. In this Section, we describe how annotations are extracted from the MapReduce workflows generated from these two categories of interfaces.

A.3.1 High-Level Declarative Languages

In Section 4.6, we described how we implementated automatic annotation extractions in Pig. Specifically, queries written in Pig Latin already contains rich information regarding the MapReduce workflow, such as schema information, and as part of Pig's compilation process of translating queries into MapReduce programs, we have implemented a module to automatically extract the annotations from these information. Similarly, this technique can be used in other high-level declarative languages, such as Hive, Cascading, and SCOPE.

Specifically in Pig, we automatically extract the schema and filter annotations by modifying the LogToPhysTranslationVisitor class, which handles the translation from a logical plan of a Pig Latin query into a physical plan composed of physical operators that will be packed into a set of MapReduce jobs. It should be noted that in Pig Latin, the schema of a dataset is already specified by the user (i.e., the syntax for loading a dataset is LOAD 'data' [USING function] [AS schema];) and is already stored as schema objects in the logical operators of the logical plan. However, the schema information was not propagated down to the physical plan. Thus, we use the schema information between logical operators to extract the information regarding the flow of data and whether they flow unchanged (i.e., information regarding the input and output keys and values of operators). This information is then stored and included in the physical plan. Similarly, we automatically extract the filter annotations from the filter operators of logical plan, which is specified by the user through the filter syntax in Pig Latin.

The dataset annotations for the intermediate datasets, such as partitioning, ordering, and compression, is already present in the Pig-generated plans. However, partitioning and ordering information regarding base (input) datasets is not available. Similar to how users specify the input dataset schemas, we provide users with the ability to specify the partitioning and ordering of base datasets, which is then included in the plans submitted to Stubby.

A.3.2 Programming Languages

For MapReduce programs written directly in some programming language, such as Java, annotations can be extracted and given to Stubby in two ways:

1. Stubby provides and accepts as input an internal representation language for annotations. Users or applications can also manually specify the annotations of the MapReduce programs in this language and then directly given to Stubby.

2. Annotations can also be automatically extracted and provided to Stubby through program-analysis-based techniques. Related work, such as HadoopToSQL [58] and Manimal [59] can be used to generate the annotations.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB J.*, 12(2):120–139, 2003.

[3] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proceedings of the VLDB Endowment*, 5(10), 2012.

[4] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya. Storage Infrastructure Behind Facebook Messages: Using HBase at Scale. *IEEE Data Eng. Bull.*, 35(2):4–13, 2012.

[5] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Kirilov, A. Ananthanarayan, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. D. Nicola, X. Wang, D. Maier, I. Santos, O. Nano, and S. Grell. Microsoft CEP Server and Online Behavioral Targeting. *Proceedings of the VLDB Endowment*, 2(2), 2009.

[6] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 1–14. ACM, 2009.

[8] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A. R. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An Experimental Study of Data Migration Algorithms. In *Algorithm Engineering*, pages 145–158. Springer, 2001.

[9] E. Anderson and J. Tucek. Efficiency Matters! *ACM SIGOPS Operating Systems Review*, 44(1):40–45, 2010.

[10] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Scale-Up Strategies for Processing High-Rate Data Streams in System S. In *Proceedings of the IEEE 25th International Conference on Data Engineering*, pages 1375–1378. IEEE, 2009.

[11] Animoto's Facebook scale-up. http://blog.rightscale.com/2008/04/23/animoto-facebook-scale-up.

[12] Aptana Cloud. http://aptana.com/cloud/.

[13] A. Arasu and J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, volume 30, pages 336–347. VLDB Endowment, 2004.

[14] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-Independent Storage for Social Computing Applications. In *Proceedings of the 4th Biennial Conference on Innovative Data Systems Research*, 2009.

[15] S. Babu and J. Widom. Continuous Queries over Data Streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

[16] N. Backman, R. Fonseca, and U. Çetintemel. Managing Parallelism for Stream Processing in the Cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, page 1. ACM, 2012.

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[18] D. Borthakur, J. Gray, J. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache Hadoop goes Realtime at Facebook. In *Proceedings of the 2011 International Conference on Management of Data*, pages 1071–1080. ACM, 2011.

[19] I. Botan, Y. Cho, R. Derakhshan, N. Dindar, L. Haas, K. Kim, C. Lee, G. Mundada, M.-C. Shan, N. Tatbul, Y. Yan, B. Yun, , and J. Zhang. Design and Implementation of the MaxStream Federated Stream Processing Architecture. Technical report, ETH Zurich, 2009. `ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/632.pdf`.

[20] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: a Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proceedings of the VLDB Endowment*, 3(1-2), 2010.

[21] R. Cáceres, L. Cox, H. Lim, A. Shakimov, and A. Varshavsky. Virtual Individual Servers as Privacy-Preserving Proxies for Mobile Devices. In *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*, pages 37–42. ACM, 2009.

[22] Catalina Robot. http://www.isi.edu/robots/catalina/.

[23] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[24] B. Chandramouli, J. Goldstein, and S. Duan. Temporal Analytics on Big Data for Web Advertising. In *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE)*, pages 90–101. IEEE, 2012.

[25] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[26] J. Chase, L. Grit, D. Irwin, V. Marupadi, P. Shivam, and A. Yumerefendi. Beyond Virtual Data Centers: Toward an Open Resource Control Architecture. In *Selected Papers from the International Conference on the Virtual Computing Initiative (ACM Digital Library)*, 2007.

[27] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 103–116. ACM, 2001.

[28] D. Chatziantoniou and K. A. Ross. Querying Multiple Features of Groups in Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 295–306, 1996.

[29] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *ACM SIGMOD Record*, 29(2):379–390, 2000.

[30] Cloudera: 7 Tips for Improving MapReduce Performance. http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance.

[31] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, pages 21–21, 2010.

[32] B. F. Cooper, E. Baldeschwieler, R. Fonseca, J. J. Kistler, P. P. S. Narayan, C. Neerdaels, T. Negrin, R. Ramakrishnan, A. Silberstein, U. Srivastava, and R. Stata. Building a Cloud for Yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

[33] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of data*, pages 647–651. ACM, 2003.

[34] S. Das, D. Agrawal, and A. El Abbadi. ElasTras: An Elastic Transactional Data Store in the Cloud. In *Proceedings of the First USENIX Workshop on Hot Topics in Cloud Computing*, 2009.

[35] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[36] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the Twenty First ACM Symposium on Operating Systems Principles*, volume 14, pages 205–220, 2007.

[37] Esper. http://esper.codehaus.org/.

[38] Eucalyptus. http://eucalyptus.cs.ucsb.edu/.

[39] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[40] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[41] K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye. Building LinkedIn's Real-time Activity Data Pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.

[42] G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. In *Proceedings of the 14th SIGMOD International Conference on Management of Data*, pages 160–172, 1987.

[43] Groovy - an Agile Dynamic Language for the Java Platform. http://groovy.codehaus.org/.

[44] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 85–98, 2009.

[45] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, and P. Valduriez. StreamCloud: A Large Scale Data Streaming System. In *Proceedings of the IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 126–137. IEEE, 2010.

[46] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *ACM SIGMOD Record*, volume 22, pages 157–166. ACM, 1993.

[47] Apache Hadoop. http://hadoop.apache.org/.

[48] Hadoop distributed File System. http://hadoop.apache.org/hdfs/.

[49] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 63–74. ACM, 2010.

[50] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[51] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[52] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[53] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 22–22. USENIX Association, 2011.

[54] Apache Hive. http://hive.apache.org/.

[55] Hyperic HQ Open Source Web Infrastructure Management Software. http://www.hyperic.com/.

[56] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proceedings of the USENIX Annual Technical Conference*, pages 199–212, 2006.

[57] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[58] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: a MapReduce Query Optimizer. In *Proceedings of the 5th European Conference on Computer Systems*, pages 251–264. ACM, 2010.

[59] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, 2011.

[60] W. Jin, J. S. Chase, and J. Kaur. Interposed Proportional Sharing for a Storage Service Utility. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 37–48. ACM, 2004.

[61] Joyent. http://www.joyent.com/.

[62] A. Kamra, V. Misra, and E. M. Nahum. Yaksha: A Self-Tuning Controller for Managing the Performance of 3-Tiered Web Sites. In *Proceedings of the Twelfth IEEE International Workshop on Quality of Service*, pages 47–56. IEEE, 2004.

[63] C. Karamanolis, M. Karlsson, and X. Zhu. Designing Controllable Computer Systems. In *Proceedings of the 10th conference on Hot Topics in Operating Systems*, volume 10, 2005.

[64] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Differentiation for Storage Systems Using Adaptive Control. *ACM Transactions on Storage (TOS)*, 1(4):457–480, 2005.

[65] M. Karlsson, X. Zhu, and C. Karamanolis. An Adaptive Optimal Controller for Non-Intrusive Performance Differentiation in Computing Services. In *Proceedings of the International Conference on Control and Automation*, volume 2, pages 709–714. IEEE, 2005.

[66] A. Kimball, S. Michels-Slettvet, and C. Bisciglia. Cluster Computing for Web-Scale Data Processing. In *ACM SIGCSE Bulletin*, volume 40, pages 116–120. ACM, 2008.

[67] J. Kreps. Building a Terabyte-Scale Data-cycle at LinkedIn with Hadoop and Project Voldemort, 2009. http://data.linkedin.com/blog/2009/06/building-a-terabyte-scale-data-cycle-at-linkedin-with-hadoop-and-project-voldemort.

[68] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-style Processing of Fast Data. *Proceedings of the VLDB Endowment*, 5(12), 2012.

[69] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, 1996.

[70] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. YSmart: Yet Another SQL-to-MapReduce Translator. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 25–36. IEEE, 2011.

[71] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, and N. Polyzotis. Exploiting Opportunistic Physical Design in Large-scale Data Analytics. *arXiv preprint arXiv:1303.6609*, 2013.

[72] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *ACM SIGMOD Record*, 34(1):39–44, 2005.

[73] H. Lim, Y. Han, and S. Babu. How to Fit when No One Size Fits. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[74] H. Lim, H. Herodotou, and S. Babu. Stubby: A Transformation-based Optimizer for MapReduce Workflows. *Proceedings of the VLDB Endowment*, 5(11):1196–1207, 2012.

[75] H. Lim, A. Kansal, and J. Liu. Power Budgeting for Virtualized Data Centers. In *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC11)*, page 59, 2011.

[76] H. C. Lim, S. Babu, and J. S. Chase. Automated Control for Elastic Storage. In *Proceedings of the 7th International Conference on Autonomic Computing*, pages 1–10. ACM, 2010.

[77] H. C. Lim, S. Babu, J. S. Chase, and S. 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.

[78] M. Liu, S. R. Mihaylov, Z. Bao, M. Jacob, Z. G. Ives, B. T. Loo, and S. Guha. SmartCIS: Integrating Digital and Physical Environments. *ACM SIGMOD Record*, 39(1):48–53, 2010.

[79] D. Logothetis, C. Trezzo, K. Webb, and K. Yocum. In-Situ MapReduce for Log Processing. In *Proceedings ot the 2011 USENIX Annual Technical Conference*, page 115, 2011.

[80] B. Lohrmann, D. Warneke, and O. Kao. Massively-Parallel Stream Processing under QoS Constraints with Nephele. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, pages 271–282. ACM, 2012.

[81] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *Proceedings of the Conference on File and Storage Technologies*, pages 219–230, 2002.

[82] Y. Lu, T. Abdelzaher, and G. Tao. Direct Adaptive Control of a Web Cache System. In *Proceedings of the American Control Conference*, volume 2, pages 1625–1630, 2003.

[83] B. Maggs. Mapping the Whole Internet with Passive Measurements. Technical report, Duke University, 2012.

[84] N. Marz. Realtime Analytics with Storm and Hadoop, 2012. http://www.slideshare.net/Hadoop_Summit/realtime-analytics-with-storm.

[85] S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: Recursive, Delta-Based Data-Centric Computation. *Proceedings of the VLDB Endowment*, 5, 2012.

[86] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *Proceedings of the IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 170–177. IEEE, 2010.

[87] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *Proceedings of the VLDB Endowment*, 3(1):494–505, 2010.

[88] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *Proceedings of the USENIX Annual Technical Conference*, pages 267–273, 2008.

[89] Oozie: Workflow Engine for Hadoop. http://yahoo.github.com/oozie/.

[90] Oracle CEP. http://www.oracle.com/technetwork/middleware/complex-event-processing/.

[91] Open Resource Control Architecture (ORCA). http://www.nicl.cod.cs.duke.edu/orca/.

[92] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 13–26. ACM, 2009.

[93] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. *ACM SIGOPS Operating Systems Review*, 41(3):289–302, 2007.

[94] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford Info Lab, November 1999.

[95] T. Palpanas, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 802–813. VLDB Endowment, 2002.

[96] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using Control Theory to Achieve Service Level Objectives in Performance Management. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management*, pages 841–854. IEEE, 2001.

[97] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 165–178. ACM, 2009.

[98] Apache Pig. http://pig.apache.org/.

[99] Postgresql. http://www.postgresql.org/.

[100] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[101] G. Rosen. State of the Cloud - August 2009. http://www.jackofallclouds.com/2009/08/state-of-the-cloud-august-2009/.

[102] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building Distributed Enterprise Disk Arrays from Commodity Components. In *Proceedings of the ACM SIGARCH Computer Architecture News*, volume 32, pages 48–58. ACM, 2004.

[103] SAP Sybase Event Stream Processor. http://www.sybase.com/products/financialservicessolutions/complex-event-processing/.

[104] M. J. Sax, M. Castellanos, Q. Chen, and M. Hsu. Performance Optimization for Distributed Intra-Node-Parallel Streaming Systems. In *Proceedings of the 8th International Workshop on Self-Managing Database Systems*, 2013.

[105] B. Seo and R. Zimmermann. Efficient Disk Replacement and Data Migration Algorithms for Large Disk Subsystems. *ACM Transactions on Storage (TOS)*, 1(3):316–345, 2005.

[106] A. Shakimov, H. Lim, R. Cáceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky. Vis-à-Vis: Privacy-Preserving Online Social Networking via Virtual Individual Servers. In *Proceedings of the Third International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–10. IEEE, 2011.

[107] A. Silberstein, R. Sears, W. Zhou, and B. F. Cooper. A Batch of PNUTS: Experiences Connecting Cloud Batch and Serving Systems. In *Proceedings of the 2011 International Conference on Management of Data*, pages 1101–1112. ACM, 2011.

[108] A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL Processes in Data Warehouses. In *Proceedings of the 21st International Conference on Data Engineering*, pages 564–575, 2005.

[109] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing Analytic Data Flows for Multiple Execution Engines. In *Proceedings of the 2012 International Conference on Management of Data*, pages 829–840. ACM, 2012.

[110] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for Web 2.0. In *Proceedings of the First Workshop on Cloud Computing and its Applications*, 2008.

[111] G. Soundararajan, C. Amza, and A. Goel. Database Replication Policies for Dynamic Content Applications. *ACM SIGOPS Operating Systems Review*, 40(4):89–102, 2006.

[112] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering*, pages 2–11, 2005.

[113] Storm. http://storm-project.net/.

[114] StreamBase. http://www.streambase.com/.

[115] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 224–237. ACM, 1997.

[116] Apache Tomcat. http://tomcat.apache.org/.

[117] TPC-H Benchmark Specification. http://www.tpc.org/tpch/.

[118] TPC-W. http://www.tpc.org/tpcw/default.asp.

[119] TPC-W Java Implementation. http://tpcw.deadpixel.de/.

[120] Truviso. http://www.truviso.com.

[121] Twitter. http://www.twitter.com/.

[122] P. Upadhyaya, Y. Kwon, and M. Balazinska. A Latency and Fault-Tolerance Optimizer for Online Parallel Query Plans. In *Proceedings of the 2011 International Conference on Management of Data*, pages 241–252. ACM, 2011.

[123] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-Tier Internet Services and its Applications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 291–302. ACM, 2005.

[124] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic Provisioning of Multi-Tier Internet Applications. In *Proceedings of the Second International Conference on Autonomic Computing*, pages 217–228. IEEE, 2005.

[125] S. Uttamchandani, L. Yin, G. A. Alvarez, J. Palmer, and G. Agha. CHAMELEON: A Self-Evolving, Fully-Adaptive Resource Arbitrator for Storage Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 75–88, 2005.

[126] VMware: Virtualization via Hypervisor, Virtual Machine & Server Consolidation. http://www.vmware.com/.

[127] X. Wang, A. D. Sarma, C. Olston, and R. Burns. CoScan: Cooperative Scan Sharing in the Cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.

[128] Y. Wang and A. Merchant. Proportional-Share Scheduling for Distributed Storage Systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pages 4–4. USENIX Association, 2007.

[129] Z. Wang, X. Zhu, and S. Singhal. Utilization and SLO-Based Control for Dynamic Sizing of Resource Partitions. In *Proceedings of the 16th IFIP/IEEE Ambient Networks International Conference on Distributed Systems: Operations and Management*, pages 133–144. Springer-Verlag, 2005.

[130] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query Optimization for Massively Parallel Data Processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 12. ACM, 2011.

[131] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[132] T. Ye and S. Kalyanaraman. A Recursive Random Search Algorithm for Large-Scale Network Parameter Configuration. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):196–205, 2003.

[133] Y. Yu, P. K. Gunda, and M. Isard. Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 247–260. ACM, 2009.

[134] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 1–14, 2008.

[135] A. Yumerefendi, P. Shivam, D. Irwin, P. Gunda, L. Grit, A. Demberel, J. Chase, and S. Babu. Towards an Autonomic Computing Testbed. In *Proceedings of the Workshop on Hot Topics in Autonomic Computing*, 2007.

[136] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[137] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10. USENIX Association, 2012.

[138] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *Proceedings of the IEEE 26th International Conference on Data Engineering (ICDE)*, pages 1060–1071. IEEE, 2010.

Biography

Harold Vinson Chao Lim was born in the Philippines on July 29, 1982. He received his B.S. in Computer Engineering and Computer Science *magna cum laude* from the University of Southern California in 2004. While he was an undergraduate student, he spent a summer working for Sun Microsystems, Inc. as a summer intern. He also received his M.S. in Computer Science from the University of Southern California in 2006. While working towards his masters degree, he worked as a research assistant in the Polymorphic Robotics Laboratory at the Information Sciences Institute. He worked on the CATALINA Project designing and building underwater robots [22]. Afterward, he worked as a full-time software engineer at Northrop Grumman Corporation designing and developing software modules for the Force XXI Battle Command, Brigade-and-Below (FBCB2) Joint Capabilities Release Vehicle product.

In August 2007, he started his doctoral studies in the Department of Computer Science at Duke University in Durham, North Carolina. He has published in systems and database conferences [73, 77, 76, 74, 52]. In addition to his dissertation work, he spent two summers as a summer intern at Microsoft Research. His work on power budgeting for virtualized data centers [75] was selected as pick of the month for May 2012 in the IEEE Computer Society Special Technical Community on Sustainable Computing. He has also worked on privacy-preserving online social networking [106, 21]. He defended his PhD Thesis at Duke University in May 2013. After graduation, he will pursue an industrial career at VMware, Inc.