Multi-Tenant Geo-Distributed Data Analytics

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Albert Jonathan

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Abhishek Chandra and Jon B Weissman

July, 2019

Acknowledgements

There are many people that have earned my gratitude for their contribution to my time in graduate school. First of all, I would like to thank my doctoral advisers, Prof. Abhishek Chandra and Prof. Jon Weissman, for their guidance and support over the last five years. They introduced me to research as an undergraduate student and patiently taught me how to think critically, approach a problem, and improve my communication skill. Such invaluable lessons will be very helpful on my future career.

Besides my advisers, I would also like to thank Prof. Zhi-Li Zhang, Prof. Marc Riedel, Prof. Tian He, and Prof. David Lilja for serving as my thesis and preliminary examination committees and providing constructive feedback throughout my degree progress. I am also grateful to Prof. Mohamed Mokbel who supported and encouraged me to pursue a doctoral degree.

I would also like to thank my friends and colleagues at the University of Minnesota and members of Distributed Computing Systems Group. In particular, I am grateful to Benjamin Heintz, Kwangsung Oh, Francis Liu, and Rankyung Hong for making the lab a family-like working environment. A special thank you to Ben, Kwangsung, and Francis who welcomed me when I first joined the lab and guided me to research.

Finally and most importantly, I would like to thank my family. I am extremely grateful to my parents, Sandford Jonathan and Sutini Setiadi, for their unconditional love, faith, support, and sacrifice. I will forever be indebted to them. I am also genuinely thankful to my brothers, Christopher Jonathan and Alvin Jonathan, who have always been very supportive. I will always cherish the memories I made with them in pursuing our Ph.D. careers together. Everything that I have achieved in life would have been impossible without my family.

Dedication

To my beloved family.

Abstract

Geo-distributed data analytics has gained much interest in recent years due to the need for extracting insights from geo-distributed data. Traditionally, data analytics has been done within a cluster/data center environment. However, analyzing geo-distributed data using existing cluster-based systems typically cannot satisfy the timeliness requirement of most applications and result in wasteful resource consumption due to the fundamental differences of the environments, especially due to the scarce, highly heterogeneous, and dynamic nature of the wide-area resources: compute power and network bandwidth.

This thesis addresses the challenges faced by geo-distributed data analytics systems in ensuring high-performance and reliable execution of multiple data analytics applications/queries. Specifically, the focus is on sharing resources across multiple users, applications, and computing frameworks. Sharing resources is attractive as it increases resource utilization and reduces operational cost. However, ensuring high-performance execution of multiple applications in a shared environment is challenging as they may compete for the same resources, especially in a wide-area environment with scarce resources. Furthermore, dynamics such as workload variation, resource variation, stragglers, and failures are inevitable in large-scale distributed systems. These can cause large resource perturbation that significantly affect the performance of query executions.

This thesis makes the following contributions. First, we present a resource sharing technique across multiple geo-distributed data analytics frameworks. The main challenge here is how to elastically partition resources while allowing high locality scheduling to each individual framework, which is critical to the execution performance of geo-distributed analytics queries. We then address the problem of how to identify and exploit common executions across multiple queries to mitigate wasteful resource consumption. We demonstrate that traditional multi-query optimization may degrade the overall query execution performance due to its lack of support for network awareness. Finally, we highlight the importance of adaptability in ensuring reliable query execution in the presence of dynamics, both for single and multiple query executions. We propose a systematic approach that can selectively determine which queries to adapt and how to adapt them based on the types of queries, dynamics, and optimization goals.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recent years have seen an increasing amount of data that are naturally *born* geo-distributed. This results from the way people rely on the Internet in their daily activities such as communicating with others through social networks, spending time enjoying online entertainment, and accessing information and news through online media. For example, Facebook reported that more than one billion people around the world actively interact with their friends and upload millions of photos and videos on a daily basis [**?**]. Studies have also reported that Twitter's users are actively generating an average of 500 million tweets per day and they rely on Twitter feeds to learn what events are happening around the world in real time [**?**]. Meanwhile, millions of people spend more than 100 million hours watching videos from Netflix [**?**]. These global activities have resulted in a vast amount of data continuously being generated around the globe.

To provide low-latency service delivery to end-users, many organizations such as Google, Facebook, Microsoft, Amazon, and Netflix deploy their services over tens of data centers and hundreds of *edge* infrastructures that are distributed around the globe [**?**, **?**, **?**, **?**, **?**]. Each site (edge cluster or data center) stores the information produced/consumed by co-located users (e.g., photos, videos, status updates) while in turn generates additional information (e.g., system and user-access logs). Collectively analyzing these geo-distributed data is critical for many business and operational tasks. For example, Twitter data are often analyzed to detect emergency events in real time [**?**, **?**]. Public services are monitoring live video streams from thousands of cameras installed all over a city for traffic control and surveillance purposes [**?**, **?**]. Similarly, video stream

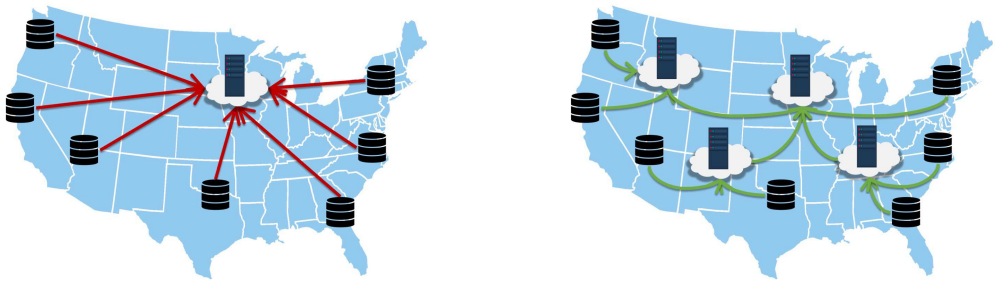
providers continuously analyze their user-access logs from their Content Delivery Network (CDN) servers for better user-experience, marketing, and billing [?, ?]. Since many of these analytics applications rely on timely information, achieving low-latency analysis is crucial.

1.1 Challenges in Geo-Distributed Data Analytics

Traditionally, data analytics has been done in a centralized manner within a cluster or data center, comprising large number of computing machines that are connected by a high-speed network. Several data analytics/computing frameworks and research prototypes have been developed for this environment for *batch*-oriented processing [?, ?, ?], *stream*-oriented processing [?, ?, ?, ?], and graph processing [?, ?, ?]. However, using these cluster-based computing frameworks to collectively analyze geo-distributed data is not trivial. For example, most data analytics frameworks assume that all input data and computing machines are co-located within the same site and any data communication among the machines is done over a high-speed local-area network (LAN). However, such an assumption is invalid for geo-distributed data processing as the data may not be co-located with the compute cluster and transmitting geo-distributed data to a centralized compute cluster is done over a relatively low-bandwidth, high-latency, and highly heterogeneous wide-area network (WAN).

One possible approach to analyze geo-distributed data is to first send all the data into a single rendezvous cluster, and process them all together using an existing cluster-based data analytics framework. We refer to this approach as a *centralized* processing model (Figure 1.1(a)). Unfortunately, transmitting a large amount of data over the WAN typically incurs very high transmission delay due to the scarce WAN bandwidth. Studies have shown that today's wide-area network bandwidth is significantly lower than the local-area network bandwidth by several orders of magnitude [?, ?]. Furthermore, the public Internet that is often used by private Clouds and edge infrastructures has even more constrained bandwidth, with an average of less than 10Mbps [?, ?]. Thus, the centralized processing model typically limits the timeliness of the results and hence, is not practical for many data analytics applications [?, ?, ?, ?, ?].

To address the limitation of the centralized approach, recent work has proposed an alternative processing model called *geo-distributed analytics* (GDA). This model processes geo-distributed data in a geo-distributed fashion by utilizing computing resources/nodes that are located close[1] to the data (Figure 1.1(b)) [?, ?, ?, ?, ?]. Data analytics systems that adopt the GDA processing model treat all geo-distributed nodes to form a single logical data center, but account for the limitation and heterogeneity of the resources. Such a decentralized processing model can significantly improve the timeliness of the analysis and mitigate wasteful resource consumption by pre-processing geo-distributed data in-place and reducing the total amount of data that need to be sent over the WAN for final processing [?, ?, ?, ?, ?].



(a) Centralized approach.

(b) Decentralized (geo-distributed) approach.

Figure 1.1: Centralized vs. decentralized processing model of geo-distributed data.

Most of the work in GDA has focused on incorporating WAN awareness in scheduling different types of data analytics jobs in wide-area settings, for batch analytics [?, ?, ?, ?, ?, ?], streaming analytics [?, ?, ?, ?, ?, ?], or graph analytics [?, ?]. The main challenge here is due to the scarce, highly heterogeneous, and dynamic nature of the wide-area resources. Thus, many data analytics applications are often constrained by the limitations of the wide-area resources in achieving their desired requirements. For example, most streaming analytics queries require low-latency and high-throughput processing but wide-area network incurs high network delay and has low bandwidth. Similarly, long-running queries require stable and high-performance processing, but they are often challenged by the high variation of wide-area network bandwidth. These

[1] The distance is often measured based on the network bandwidth availability between two sites instead of the actual geographic distance.

unique characteristics of wide-area resources are not taken into account by existing cluster-based data analytics systems. Thus, optimizing existing data analytics systems to wide-area settings requires rethinking some of their designs.

This thesis addresses the challenges faced by geo-distributed data analytics systems in ensuring a high-performance and reliable execution of *multiple* data analytics applications/queries[2] . Specifically, the focus is on sharing resources across multiple users, applications, and computing frameworks. Sharing resources across multiple applications is attractive as it reduces the ownership and operational cost of a cluster and it improves resource utilization by allowing applications to elastically scale their resource needs based on their workload. Yet, ensuring high-performance and reliable execution of multiple queries in a shared environment is challenging as they may compete for the same resources, especially in a wide-area environment with limited resources. Furthermore, runtime dynamics such as unpredictable workload pattern, network bandwidth variation, continuous job arrivals/completions, occurrence of stragglers, and failures are inevitable in large-scale distributed systems. In a resource-constrained environment, such dynamics can cause large resource perturbation that significantly affect the overall execution performance of queries.

1.2 Thesis Contributions and Outline

To address the above challenges faced by geo-distributed data analytics systems, this thesis answers the following research questions: (1) How to schedule *batch* and *streaming* analytics jobs in a wide-area environment? (2) How to share wide-area resources across multiple data analytics frameworks? (3) How to optimize multiple query executions to mitigate any wasteful resource utilization? and (4) How to adapt query executions in the presence of dynamics? We summarize the contributions and the organization of the remainder of this thesis as follows:

- Chapter 2 addresses the problem of resource sharing across multiple batch analytics frameworks in a wide-area environment. We propose a lease-based resource sharing technique and a variant of a delay scheduling that allow each framework to schedule its jobs with high locality.

[2] A query refers to a processing model of an application.

- Chapter 3 explores the opportunity of incorporating multi-query optimization in the context of wide-area streaming analytics. Specifically, we look at the opportunity of sharing common executions across multiple queries to eliminate any redundant data processing and transmission over the wide-area network. We further highlight the importance of network awareness in applying multi-query optimization in a wide-area environment. We show that traditional multi-query optimization may degrade the overall query execution performance due to its lack support for network awareness.

- Chapter 4 addresses the importance of adaptability for long-running streaming analytics queries in maintaining their execution performance regardless of dynamics. We study several adaptation techniques, extend them to handle various wide-area dynamics, and propose a systematic approach that can automatically determine which adaptation action to take depending on the types of queries, dynamics, and optimization goals. We show that the proposed adaptation technique can handle various dynamics with low overhead and without compromising the accuracy/quality of the results.

- Chapter 5 further extends the proposed adaptation technique to a multi-query environment where multiple queries may be constrained and competing for common resources. We propose a notion of adaptation cost that considers both the overhead and the resource consumption cost of adapting a query, and use this adaptation cost metric to determine which queries that need to be adapted as well as how to adapt them. The proposed adaptation technique results in a more efficient adaptation that improves the overall performance and stability of multiple query executions.

- Chapter 6 presents the future research directions.

- Lastly, Chapter 7 highlights the key contributions of the thesis.

Chapter 2

Resource Sharing in Geo-Distributed Edge Cloud

2.1 Introduction

Many Cloud providers such Amazon, Google, and Microsoft operate and deploy their services over tens of data centers and hundreds of edge servers around the globe to provide low-latency service delivery to their end-users [**?**, **?**, **?**]. These services in turn continuously generate large amounts of data across multiple sites/locations. Collectively analyzing these geo-distributed data is crucial for many operational tasks. For example, analyzing user-access logs from multiple geo-distributed CDN servers provides insights on the popularity of a specific event across different countries, which can be further used for advertisement purpose. Another example includes analyzing system logs to find common security threats across multiple edge servers.

Most data analytics systems have been primarily designed to run within a cluster or data center. However, such a centralized platform is not well suited for geo-distributed data analytics applications/queries since it requires sending all the data into the cluster before starting the analysis. This geo-distributed data transmission over the wide-area network (WAN) typically results in high overhead due to the bandwidth limitation of the network. Thus, such an approach typically limits the execution performance of geo-distributed data analytics queries. To address the limitations of the centralized processing model, recent work has proposed a decentralized processing model that processes

geo-distributed in a geo-distributed fashion using computing machines that are located at the *edge of the network*. Such a platform is often called an *Edge Cloud* [**?**, **?**, **?**]. Edge Cloud can improve the execution performance of analyzing geo-distributed data by utilizing co-located computing resources.

Data-intensive applications are diverse in terms of their characteristics, requirements, and processing model and hence, they require different execution models to process the data efficiently. This has led to recent developments of a number of distributed data analytics frameworks/systems such as MapReduce [**?**], Dryad [**?**], Pregel [**?**], and others [**?**, **?**, **?**]. Since most of the frameworks have been designed for a cluster/data center environment, recent attempts have looked at the opportunity and challenges of adapting and optimizing them to a geo-distributed environment for geo-distributed data analytics queries [**?**, **?**, **?**, **?**]. We believe that the increasing trend of geo-distributed data will trigger more computing frameworks to be developed or adapted to a wide-area environment. Yet, this imposes new challenges in sharing the resources across multiple data analytics frameworks in a wide-area environment.

In general, resource sharing provides hardware cost benefits and improves resource utilization as it allows each framework to elastically adapt its resource share based on its workload. Although the problem of resource sharing across multiple frameworks has been extensively studied in a large cluster/data center environment [**?**, **?**, **?**, **?**], existing cluster-based resource sharing techniques do not scale well to a wide-area environment due to the fundamental differences between the two environments. In particular, they lack the support for network awareness that is critical to achieving high-performance query execution in geo-distributed settings.

To address the challenges of sharing resources across multiple geo-distributed data analytics frameworks, we introduce `Awan` [1] - a resource management system for geo-distributed Edge Clouds. The main goal of `Awan` is to provide a generic resource sharing mechanism that allows each data analytics framework to schedule and deploy its jobs with high locality, which is crucial to the overall query execution performance of geo-distributed data analytics queries. `Awan` achieves this goal by implementing a *resource lease* abstraction, which defines a set of computing resources that have been allocated to a specific framework for a specific amount of time. A lease provides a guarantee on the

[1] Awan is an Indonesian word of "Cloud".

duration for which resources will be held by a particular framework. `Awan` shares every lease information to all frameworks, enabling them to make better scheduling decisions by considering the future availability of all of the computing resources. We further propose a locality-based priority scheduling algorithm based off a delay scheduling algorithm that allows framework schedulers to prioritize high-locality jobs in scheduling multiple data analytics jobs.

Our experimental evaluation with the Nebula Edge Cloud [**?**] on a real geo-distributed system deployment using PlanetLab [**?**] shows that `Awan` outperforms existing resource sharing techniques for geo-distributed data-intensive applications. Specifically, it increases the number of tasks that can be scheduled locally by each framework by approximately 28%. This locality improvement results in a reduction of the overall job execution time by approximately 18%. The use of delay scheduling algorithm further improves the locality, which results in a decrease in the average job turnaround time by an additional 13%.

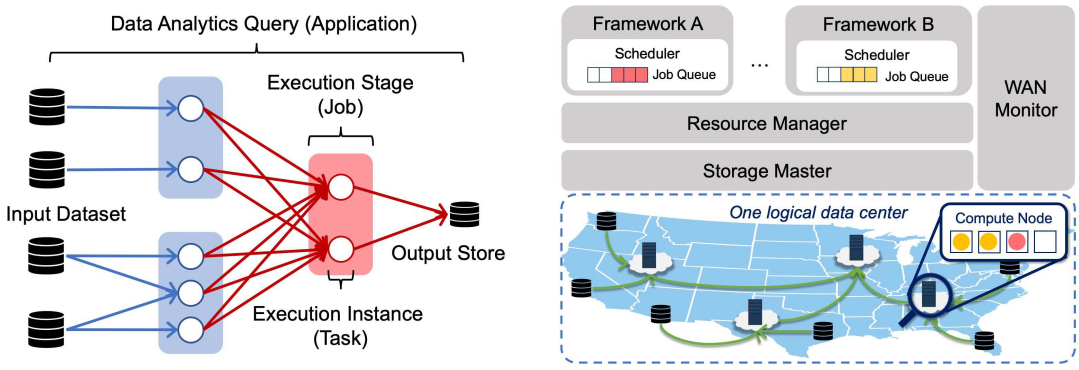
2.2 Problem Context

In this section, we describe the application model and the Edge Cloud system model that we consider throughout the chapter.

2.2.1 Application/Query Model

We consider batch-oriented data analytics applications/queries whose inputs are distributed across multiple locations. Figure 2.1(a) shows an example of the query execution model. Each application that is submitted to the system needs to specify its input dataset, its processing model, and the final output location. Here, the processing model of an application corresponds to which execution framework the application will be scheduled by (will be discussed in Section 2.2.2). For example, an analyst may submit a WordCount application to a MapReduce [**?**] framework to find errors from multiple geo-distributed log files.

In general, an application/query, Q, consists of one or more execution stages (*jobs*), $Q = \{J_1, \ldots, J_n\}$ where $n > 1$, and a job may have dependencies with other jobs. For example, a MapReduce application consists of 2 jobs: Map job and Reduce job, and



(a) Application/query execution model. (b) Edge Cloud system model.

Figure 2.1: Query execution and Edge Cloud system model

the Reduce job depends on the execution of the Map job, as its inputs are generated by the Map job. A job can start its execution only after all of its dependency jobs have completed their executions. A job J_i can be further broken down into multiple execution instances (*tasks*) that can run in parallel: $J_i = \{T_{i,1}, \ldots, T_{i,p}\}$, where p is the parallelism of the job. The parallelism value of a job is typically determined based on its input workload. A query execution is considered complete/finish when all of its jobs have completed, while a job is considered complete when all of its tasks have completed.

2.2.2 Edge Cloud System Model

Figure 2.1(b) depicts the Edge Cloud system model that we consider: it consists of several components that are deployed in a reliable/dedicated server and multiple storage and compute nodes that are geo-distributed. Both the storage and compute nodes can be shared by multiple applications. A node in our environment may perform as a compute node, storage node or both. A compute node consists of one or more *computing slots*, which is the smallest granularity of a computing resource that can be assigned to exactly one task at a time. In this work, we consider all slots to be homogeneous (e.g., a slot corresponds to 1 CPU and 1GB memory). However, the number of computing slots across nodes may vary depending on the computing power of each node itself. When a task is deployed to a particular computing slot, it will first download all of its input dataset, process them, and store the final result to one or more storage nodes.

We assume that the Edge Cloud consists of following modules:

- **Storage Master.** Throughout the discussion, we consider a file as the smallest granularity of an input data shard of a particular task. All files that are stored in the system are managed by the *Storage Master*. It is responsible for (1) maintaining all files' metadata, (2) ensuring the availability of all files by determining the replication factor of each file, and (3) determining where each file and its replicas should be stored [**?**].

- **WAN Monitor.** The *WAN Monitor* is responsible for monitoring the end-to-end network bandwidth availability (both uplink and downlink) between nodes. This bandwidth information is used to determine the network bandwidth distance between nodes, which is used by the *Storage Master* to determine *where* each file should be stored, and by each *Data Analytics Framework*'s scheduler to schedule its jobs with network awareness. In this work, we consider a compute node to be *local* to a storage node if they share the same physical machine or the network bandwidth between them is higher than a specific bandwidth threshold.

- **Data Analytics Framework.** A *Data Analytics Framework* (or framework for short) consists of a scheduler that is responsible for (1) scheduling and deploying any submitted job, and (2) monitoring the execution state of all of its jobs. Each submitted job will put into a *job queue* and the scheduler will schedule the jobs (i.e., job whose dependencies have been resolved) based on their priorities on a per-job basis. Specifically, the scheduler will determine *where* each task of the job should be deployed. Different frameworks in the system may have different scheduling policies. For example, a framework may schedule its jobs with the goal of minimizing the overall job execution runtime, while another framework may schedule its jobs with the goal of minimizing WAN bandwidth consumption. In the former case, the scheduler will attempt to schedule its jobs with high locality because data transmission over WAN is usually the dominant factor to the overall query execution time. In this case, locality can be achieved by scheduling a task on a compute slot that is closest to its inputs' locations. We refer to this scheduling technique as *locality scheduling*.

- **Resource Manager.** Since different analysts may want to run a different types of applications, the resources in an Edge Cloud is typically shared by multiple frameworks. The main goal of the *Resource Manager* is to provide a resource sharing mechanism and policies across multiple computing frameworks. It keeps track of the availability of all computing slots (e.g., vacant or has been allocated to a particular framework) and may allocate any of the available slots to any of the frameworks. For clarity reason, we assume that a slot can only be allocated to exactly one framework at a time. Note that a system may not have a *Resource Manager*, in which case each framework will compete for the resources directly [**?**]. However, such a resource sharing mechanism introduces new challenges in coordinating and enforcing any global resource sharing policy across multiple frameworks.

2.3 Awan: Geo-Distributed Resource Manager

There have been several resource management systems developed to share computing resources among multiple frameworks in a cluster/data center environment [**?**, **?**, **?**, **?**]. However, they do not account for the network bandwidth limitation and heterogeneity between the nodes in the system since they are primarily designed for a relatively homogeneous environment. In this section, we first identify and highlight some of the limitations of existing cluster-based resource management techniques when they are deployed in wide-area settings, and then present `Awan`: a resource sharing technique that we have designed specifically for a geo-distributed environment.

2.3.1 Limitations of Existing Cluster Resource Managers

One possible approach to share resources across multiple frameworks is to use a *Monolithic Scheduler*: a global resource scheduler that determines how to allocate resources to each framework. Such a scheduler typically implements a generic scheduling policy (e.g., fair sharing) that can be used by various types of frameworks. Although a Monolithic Scheduler generally simplifies the resource sharing problem, such a scheduler is often difficult to extend with new framework-specific policies and optimization.

In order to optimize the execution deployment of various data analytics applications, researchers have implemented multiple data analytics frameworks with different scheduling policies, each of which is optimized for a specific type of applications. However, sharing limited number of resources across different frameworks introduces new challenges such as (1) How to partition the resources across frameworks? (2) How to handle concurrency issues among frameworks? and (3) How to prioritize certain jobs from different frameworks? One possible approach would be to statically partition the resources in advance and allocate each framework a predetermined share of resources. We refer to this approach as a *static resource partitioning*. The main drawback of statically partitioning the resources is that it leads to an external resource fragmentation problem, resulting in a low cluster resource utilization. Determining the size of each partition in advance may also be difficult since each framework may have a dynamic and unpredictable workload that changes over time. Moreover, a static partitioning approach is not suitable for a geo-distributed environment since it typically limits the locality scheduling of each framework.

Dynamic resource partitioning solves the external fragmentation problem by elastically adapting the resource share of each framework based on their workloads. There are several dynamic resource partitioning techniques that have been proposed to share computing resources in a cluster environment. The first type is a *two-level scheduling*. It consists of a single logical *Resource Manager* that performs as an abstraction layer between the resources and the frameworks. Each framework interacts with the *Resource Manager* in order to acquire resources. Mesos [**?**] is a popular resource management system that uses a two-level scheduling approach. In Mesos, all frameworks acquire resources from the Mesos *Resource Manager* using a *resource offer* mechanism. In this model, each framework would request for the availability of the resources from the *Resource Manager* whenever there is a job that needs to be scheduled. Upon receiving this request, the *Resource Manager* would offer a set of the available resources based on its resource partitioning policy. In return, a framework may either accept or reject the offer if the offered resources do not adequately satisfy the requirements.

The resource offer mechanism uses a pessimistic concurrency control, meaning that the resources that are currently offered to one framework will not be offered to the other frameworks at the same time. This ensures no conflict between frameworks in allocating

resources. The drawback of the pessimistic approach is that only one framework can acquire a particular set of resources at a time. Thus, other frameworks may have to wait for a long time for the *Resource Manager* to offer the desired resources to them. An alternative approach would have the *Resource Manager* perform global resource allocation for all the requests from the frameworks, similar to the approach used in YARN [**?**]. This, however, makes the two-level architecture effectively monolithic since the resource allocation is determined by a single global resource allocator.

A *shared-state model* that was introduced in Google Omega [**?**] removes the role of the *Resource Manager*. In this case, each framework can directly schedule its tasks on any of the available resources. In this resource sharing model, the state of all resources are shared by all of the frameworks and they can schedule their tasks in parallel using an optimistic concurrency control. This mechanism gives all of the frameworks knowledge about the state of each of the resources (i.e., available or unavailable). This knowledge, however, is only used to avoid a framework trying to acquire resources are currently being used by other frameworks. While the shared-state model is useful in a cooperative environment, using it in an Edge Cloud with limited number of resources may lead to significant issues such as fairness and starvation because multiple frameworks may be competitive and try to hoard resources. Furthermore, since a shared-state model does not have a coordinator that controls the resource shares among frameworks, enforcing global policies across frameworks (e.g., fair sharing across frameworks) is not trivial.

2.3.2 Awan Resource Manager

To address the limitations of the existing cluster-based resource managers in an Edge Cloud environment, we propose a new resource manager called `Awan`. The goal of `Awan` is to provide a scalable resource sharing mechanism in a geo-distributed environment that allows each framework to achieve high locality scheduling that is critical to achieving high-performance execution of geo-distributed analytics queries. `Awan` combines the desirable features of the two-level model with those of the shared-state model, while providing explicit support for locality-aware scheduling. Figure 2.2 shows the two-level model of `Awan`. We incorporate the shared-state mechanism by sharing the states of all the resources to all the frameworks. In our system the states of the resources are shared by the *Resource Manager* and not directly by the frameworks.

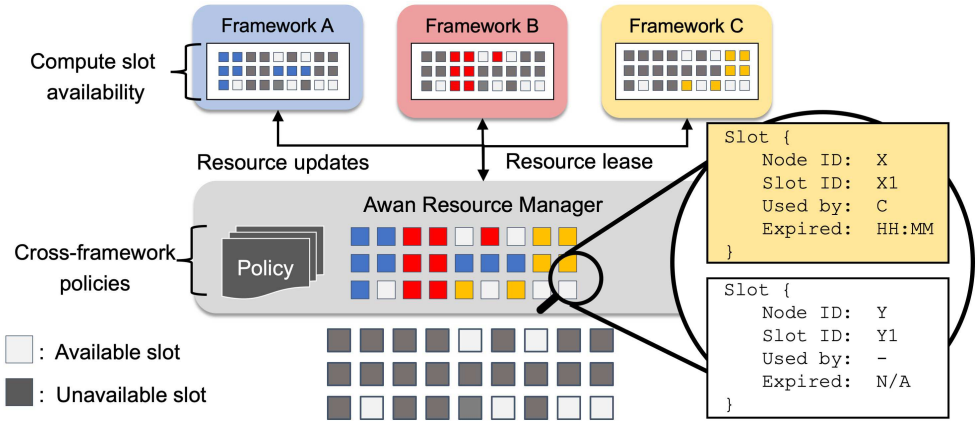


Figure 2.2: Awan two-level resource sharing model.

`Awan` *Resource Manager* provides the states of *all* resources to all frameworks, including those that are not currently available or have been allocated to a particular framework. Each framework acquires and schedules its tasks to the available resources using a *resource lease* mechanism with an optimistic concurrency control. However, the optimistic concurrency control in acquiring resources is coordinated by the *Resource Manager*. In this case, a framework will first lease a set of resources for a specific amount of time to the *Resource Manager* before deploying its tasks (the leasing mechanism will be discussed in Section 2.3.3). To handle conflicting resource acquisition, the resource allocation will be done in an atomic manner. The *Resource Manager* can also enforce any global policy that has to be obeyed by the frameworks. For example, the *Resource Manager* may implement a fair sharing policy by limiting the number of resources that can be acquired by a framework.

2.3.3 Resource Lease

In a geo-distributed environment, the resource offer mechanism that is used by Mesos suffers from the potential lack of locality in task scheduling. The main reason is due to the limited knowledge of the resource availability that is provided by the *Resource Manager*, since the resource offer mechanism only offers currently available resources. At a glance, offering only the available resources seems reasonable because tasks can only be deployed on computing slots that are available. An unavailable slot, however,

may actually provide better data locality for a task than any of the available slots *when* it becomes available in the future. Mesos handles this issue by incorporating a delay scheduling [**?**] for short running tasks which makes a scheduler delay scheduling a job if its task cannot be launched locally. However, delay scheduling a job without any knowledge of the future availability of the resources, especially if most of the tasks are long-running, may introduce unnecessary waiting time. Instead, it would be desirable for a scheduler to wait (or not) on busy resources depending on the expected waiting time. If the scheduler is aware that the local resources will be available soon, it may decide to delay scheduling a certain job. On the other hand, if the waiting time is too long, the scheduler may decide to deploy its tasks on any of the available slots. Thus, sharing the future availability information of unavailable resources can help a scheduler make a better scheduling decision.

A lease in the *resource lease* mechanism has a *lease expiration time* associated with it which provides a guarantee that the acquired resources will be held by a framework for no longer than the lease time (with a possibility of some grace period). After the lease time expiry, the *Resource Manager* will make the resource available to the other frameworks. Sharing the lease expiration time enables framework schedulers to estimate their waiting time for desired busy resources, leading to improved scheduling decisions.

When a framework tries to acquire a set of available resources $R_1, \ldots, R_n$, the scheduler sends a *lease request* $< L_1, \ldots, L_n >$ to the *Resource Manager*. Here, L_i is the lease time on resource R_i. If the *Resource Manager* agrees on the request and the resources are available, the resources will be allocated to the framework and the resources will be marked unavailable for other frameworks. A lease request may also contain an *atomic request* flag specifying that all resources must be acquired atomically. If the flag is set, the request will be granted *iff all* the leases can be satisfied, i.e., all resources in the request are available. Otherwise, the leases on available resources will be granted and the scheduler will be notified if any of the leases failed. Failure in leasing may happen because of the optimistic concurrency control used in our resource management mechanism, where multiple schedulers may try to acquire overlapping sets of available resources at the same time. We provide such an atomic option because some frameworks (e.g., MPI) require all resources to be available to start the execution, while others (e.g., MapReduce) can start a job partially and add more resources later.

The *Resource Manager* keeps track of the leases and share them to all of the frameworks. Since each lease contains information about the future availability of a specific resource, a scheduler may decide whether to schedule its tasks on the available resources or wait for the busy resources. If a scheduler decides to wait on one or more resources, the scheduler should be able to dynamically change its scheduling decision over time. This is useful for a few reasons. First, the network bandwidth availability between nodes is constantly changing over time, especially in a system that is connected via dynamic WAN. Thus a scheduler may change its scheduling decision if the resource availability has drastically changed. Second, stragglers and failures are inevitable in large-scale distributed systems. Thus, a scheduler should be able to re-evaluate its scheduling decision if the resources it has been waiting for have failed. Third, the lease estimation that is provided by other frameworks may not be perfectly accurate (in practice it is unlikely to have a 100% prediction accuracy especially in a resource constrained environment that may have large resource perturbation). If a computing slot becomes available sooner than the estimated time, a framework should be be able to acquire the slot immediately. Lastly, in an optimistic concurrency control, multiple frameworks may wait for the same set of resources. Since only one framework is able to acquire the resources, the other frameworks should be able to reschedule their tasks on different resources if they have failed acquiring the desired resources. Here, a *waiting list* can be added to each resource so that a framework may decide to ignore resources that have long waiting queues.

Our design uses a two-level model to allow global policies to be applied to every framework easily. Some policies that have been incorporated in our implementation are: (1) the capability of rejecting a lease request for an unreasonable long time and (2) terminating a process/task that takes longer than the agreed lease time. If fairness across frameworks is the main priority, fair sharing may also be applied by limiting the number of resources that can be leased using max-min fair sharing.

2.3.4 Lease Estimation and Enforcement

When a framework tries to acquire a resource, it needs to estimate the time needed to complete the task on that particular resource. The lease time in our implementation is estimated by combining the data transmission time and task's running time from its statistical history. However, having a perfect accuracy in estimating the lease time is not

possible for many reasons. Public Internet is highly dynamic in practice, and networking problems such as packet loss may also influence the data transfer time. Estimating the network bandwidth between nodes interconnected via WAN is a challenging problem [**?**, **?**]. In our implementation, the network cost estimation is best-effort and it is estimated based on the time to transmit data over a network link. Although, the data transmission time is typically the dominant factor of the total execution time for geo-distributed data-intensive jobs, we also consider the computing factors into the lease calculation to have a better lease estimation accuracy. These factors are estimated as a per-node compute performance history of running similar tasks, which are maintained by each framework. In general, the lease time of a slot of a particular node, L_a, is estimated as follow:

$$L_a = max\Big(\frac{|input_b|}{B_b^a}\Big) + \bar{C_k}\Big(\sum input_b\Big) + \delta, \forall b$$

L_a is the estimated time needed to process a task on node located at site a. B_b^a is the available network bandwidth between sites b and a, where b is the location of an input data shard $input_b$. $\bar{C_k}$ is the average task processing time of the latest k^{th} tasks. The number of records, k, is used to avoid including obsolete records into the calculation. The similarity of the task can be categorized based on its computing model and its input data size. If there is no similar task profile in the history, $\bar{C_k}$ will be estimated from the performance of running a similar task on a different node. If none of the nodes have ever processed a similar task, the scheduler will lease a slot for a pre-defined amount of time. This might cause a large inaccuracy in running a task for the first time. A small slack factor, δ, is added to the estimation to avoid an overly optimistic prediction.

Both underestimation and overestimation of a lease time can lead to problems. If the lease time is underestimated, i.e., the time needed to complete a task is longer than the lease time, the task would not be completed before the lease expiration time. If the *Resource Manager* terminates the task and revokes the expired slot, this will result in wasted resources and increase the turnaround time of the job. The overall system utilization and performance will deteriorate significantly with a high number of lease underestimations. On the other hand, if the lease time for a slot is overestimated, fewer schedulers may wait for this slot to become available, and may instead schedule their tasks on less desirable non-local slots. With a high number of lease overestimation, most schedulers will ignore the busy slots and schedule their tasks on any of the available

slots, effectively reducing to a resource offer-based mechanism. This may result in a low number of local task executions and decrease the overall system performance.

The *Resource Manager* can also use different policies to handle expired leases. The simplest approach is to terminate the running task upon lease expiry and set the slot to be available to other frameworks. However, terminating a process that has not finished on an expired lease requires the task to be rescheduled on a different slot, and will result in wasted resources if the task is almost finished. A better approach is to give some *grace period* for the node to clean up or finish. If the task is a long-running task and the progress of the process is far from completion, the state of the task could be saved and its temporary results should be stored to a storage node such that the task can be continued by another slot instead of restarting the whole task. The *Resource Manager* should carefully determine what is the appropriate grace period. If the grace period is too low, it is likely to result in large number of terminated tasks. On the other hand, if the grace period is too high, it may lead to much higher waiting times for other frameworks waiting for the slot to become available.

2.4 Locality-based Priority Scheduling

Our discussion so far has focused on how to share resources between multiple frameworks. In this section, we focus on improving locality in scheduling multiple jobs for a batch analytics framework. A notable technique to improve locality in scheduling is by using a delay scheduling algorithm [**?**], which would skip a job that is at the head of the job queue if any of its tasks cannot be scheduled locally. To avoid starvation, the scheduler can limit the number of times a job is skipped. Once the number of skips reaches a predetermined threshold, the job will be scheduled even if its tasks cannot be scheduled locally. This technique is feasible if most of the tasks are short-running where a short delay is often sufficient to have a higher locality.

Introducing a delay in scheduling long running tasks, however, works well only if a scheduler has a complete knowledge of the status of all task slots (including those that are currently unavailable). If a scheduler is only aware of the slots that are available (on which it can launch its tasks without any delay), delaying task placement may incur unnecessary waiting time since it is possible that a task cannot be scheduled locally in

any of the slots. In our resource sharing mechanism, the waiting time for a particular task slot can be obtained from the lease information that is shared by the *Resource Manager*. If the waiting time is too long, the scheduler should be able to schedule the task to a different resource right away.

We generalize the delay scheduling algorithm by introducing a *minimum locality level constraint*. Instead of immediately skipping a job if any of its tasks cannot be scheduled locally, we compare the locality level of the job with the minimum locality level constraint. The locality level of a job, $Locale_J$, is defined as the fraction of tasks that the scheduler can launch locally, which can be computed as follow:

$$Locale_J = \frac{\sum_{T \in J} \begin{cases} 1 & \text{if } B \geq B_{min} \\ 0 & otherwise \end{cases}}{|J|}$$

T is a task of a job J, $|J|$ is the total number of tasks that need to be scheduled, B is the bandwidth demand to deploy the task, and B_{min} is the minimum bandwidth threshold that determines the locality scheduling of the job. A job J will only be scheduled if $Locale_J \geq Locale_{min}$, where $Locale_{min}$ is the minimum locality threshold that is set by the scheduler. A minimum locality level of 0 means that the job will be scheduled regardless of the number of tasks that can be scheduled locally. On the other hand, a minimum locality level of 1 means that a job can be scheduled only if all of the tasks can be scheduled locally (effectively similar to the delay scheduling algorithm).

A minimum locality level constraint of 0 might result in locality scheduling for a low-intensity workload since each scheduler tries to schedule its tasks locally and can find such resources available. On the other hand, an overly high minimum locality constraint may lead to high waiting time due to the restriction on scheduling. In practice, adjusting the minimum locality threshold between the two extremes $0 < Locale_{min} \leq 1$, may increase the number of tasks that can be scheduled locally since it could prioritize jobs that have higher locality. The maximum number of skips should also be set carefully since a higher number of skips would result in a higher waiting time. In summary, $Locale_{min}$ should be adjusted to the scheduler's workload and the average number of tasks that can be scheduled locally from the statistical history.

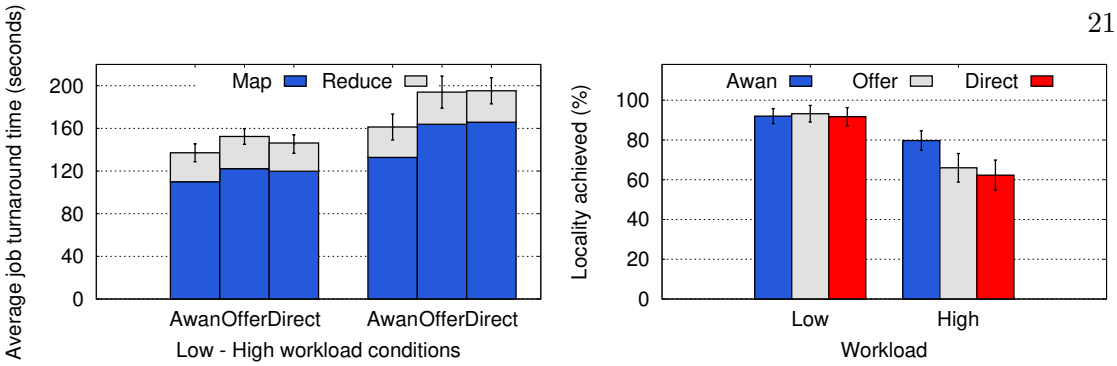
2.5 Experimental Evaluation

System Setup. We have implemented `Awan` on the Nebula Edge Cloud [**?**]. We have modified the default monolithic resource scheduler in Nebula to the two-level scheduler with the leased-based resource management mechanism. Any code execution is carried out inside Google Chrome Web browser-based Native Client (NaCl) sandbox [**?**] that is used in Nebula. Although our evaluation is based on the Nebula system, it is worth noting that the proposed technique itself is not tightly coupled with the platform and could be implemented into any Edge Cloud system that provides the system model discussed in section 2.2.

We deployed 40 compute nodes and 32 storage nodes on PLE PlanetLab [**?**] nodes that are geo-distributed across the Europe. The nodes are heterogeneous in terms of their computation power and network bandwidth (varying from 1Mbps to more than 10Mbps). Most of the storage nodes that we deployed had at least one local compute node, but not every storage node has a nearby compute nodes. Here, a node was considered "local" if the network bandwidth between the nodes was greater than 8Mbps. All centralized components such as the *WAN Monitor*, *Storage Master*, *Resource Manager*, and *Data Analytics Frameworks* were hosted on a dedicated machine with an Intel Xeon CPU E5-2609 and 16GB of memory.

Baseline Comparison. We compare `Awan`'s lease-based resource sharing mechanism (`Awan`) with the two-level resource offer (`Offer`) and the direct share state mechanism (`Direct`). We deployed 3 frameworks with different scheduling policy: (1) Lease-aware scheduling (`Lease`), (2) First-Come-First-Serve scheduling (`FCFS`), and (3) Random (`Random`) scheduling policy. Both `Lease` and `FCFS` implemented a network-aware task placement algorithm proposed by prior work in geo-distributed MapReduce [**?**]. The difference between them is that, `Lease`'s scheduler considered the future availability of the unavailable slots and it might delay scheduling jobs that had low locality. In contrast, `FCFS`' scheduler was not aware of the future availability of any unavailable resource. Thus, it would never wait for resources that were not available during the scheduling time.

Workload. All of the schedulers scheduled the same set of MapReduce WordCount jobs with input data size varied from 256MB to 512MB. Each input data had been



(a) Average job turnaround time.

(b) Fraction of tasks scheduled locally.

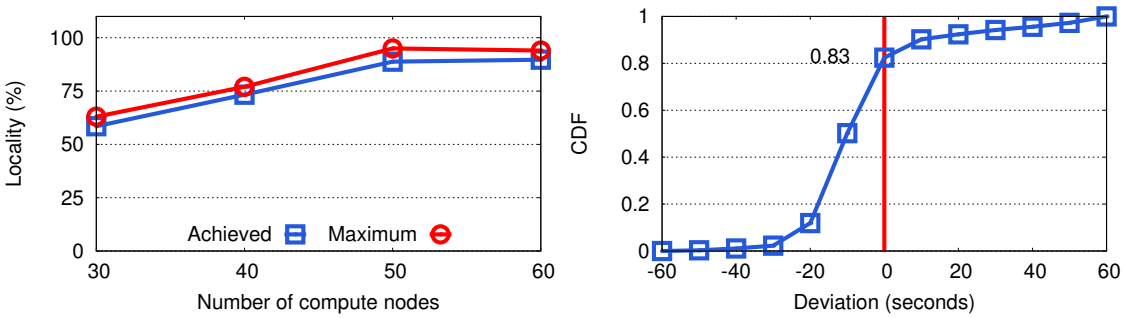
Figure 2.3: Benefit of lease-based resource sharing technique.

partitioned into 16MB chunks, and distributed randomly to different locations. The critical path of each of the MapReduce jobs in our experiments was in the Map job which: downloaded the input data set from geo-distributed storage nodes, processed them, and stored the result to one or more storage nodes. The computation of the Reduce tasks in our experiments were not critical to the overall MapReduce performance since the inputs of the Reduce tasks (the output of the Map tasks) were much smaller compared amount of data processed by the Map tasks.

2.5.1 Leased-based Resource Sharing

In this experiment, we used 5 rounds of Poisson distribution-based simulation with 12 jobs/round. We evaluated the benefit of our lease-based resource sharing technique under two different workload conditions: (1) *Low workload* with a job inter-arrival rate of 100 seconds and (2) *High workload* with a job inter-arrival rate of 50 seconds. The low and high workload conditions resulted in 1 to 2 and 2 to 4 concurrent jobs respectively. On average, a task could be completed in approximately 40 to 60 seconds if it was deployed on a local node and took more than 100 seconds for most of the time if it was deployed on a distant node. We allowed 20 second grace period to every lease upon its expiration. We also included a 95% confidence interval in the results.

Figure 2.3(a) shows the overall job turnaround time of of the frameworks deployed on top of `Awan`, `Offer`, and `Direct`. We can see that during the low workload condition, all of them performed comparably because most jobs could be scheduled locally. However, as the workload increases, the `Lease`'s framework scheduler in `Awan` was able



(a) Distance to maximum locality.

(b) Lease estimation deviation.

Figure 2.4: Lease estimation.

to schedule more tasks locally (28% higher locality as shown in Figure 2.3(b)) since the `Lease`'s scheduler were able to delay scheduling some of the tasks waiting for local slots. This led to the 18% reduction of the overall job turnaround time. These results show that knowing the future availability of the slots allows framework schedulers to better schedule their tasks with higher locality. The framework schedulers in the `Offer` case were not aware of the existence of the busy resources because the *Resource Manager* offered only the available slots. In this case, a scheduler would always try to schedule its tasks locally based on the offered resources, which led to a lower locality scheduling that could be achieved. Similarly, the schedulers in `Direct` would schedule their tasks locally only if the local shots were available since they did not know about the future availability of the busy resources.

2.5.2 Lease Estimation

We further evaluated the locality that could be achieved by the `Lease`'s scheduler on `Awan` compared to the maximum locality that could have been achieved if the future availability of every slot is known in advance, which is impractical in real deployment. In this experiment, we used 60 storage nodes and varied the number of compute nodes from 30 to 60 nodes. Figure 2.4(a) shows locality difference between the locality that was achieved by `Lease` and the maximum locality. The maximum locality was lower than 100% when there were fewer compute nodes because the data were randomly distributed throughout a much higher number of storage nodes and some of them did not have any local compute node. The figure also shows the distance between the locality

that was achieved to the maximum locality is within 5%. This indicates that the `Lease`'s scheduler could schedule its tasks close to the best possible locality regardless of the number of compute nodes. We also evaluated our lease estimation accuracy that is critical to the decision made by the `Lease`'s scheduler. We define the accuracy as:

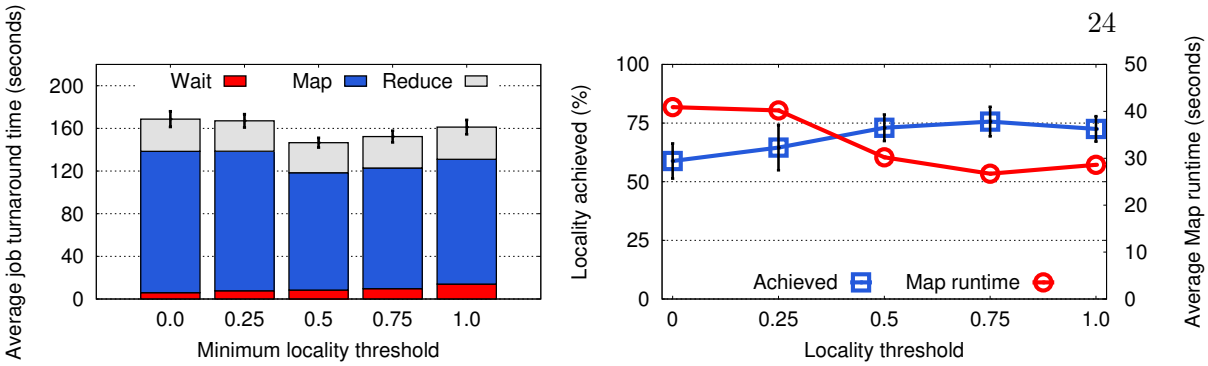
$$accuracy = 1 - \frac{|t_{actual} - t_{predicted}|}{t_{actual}}$$

Figure 2.4(b) shows the accuracy CDF of the task execution prediction made by the scheduler to the actual running time. On average, the lease estimation we used results in 83% accuracy. We observed that the accuracy had some variations due to the initial scheduling of the jobs that had not been previously seen by the scheduler and the dynamics that occurred during the experiments. In the former case, the scheduler estimated the task's running time using a predefined value (100 seconds was used in the experiment). When similar jobs were posted later, the scheduler could predict the task running time based on its statistical records and resulted in a higher accuracy prediction. However, sometimes the accuracy still fluctuated over time even if similar jobs were posted due to the dynamic nature of the wide-area environment.

2.5.3 Locality-based Priority Scheduling

Lastly, we evaluated the benefit of prioritizing higher locality jobs based on the *minimum locality* threshold. This goal of this approach is to prevent scheduling jobs with low locality unless they have been waiting for a long time. The main problem of scheduling low locality jobs in a resource-constrained environment is that it may reduce the locality level that could be achieved for the other jobs. A job that could not achieve the minimum locality threshold will be skipped for no more than 10 seconds. In this experiment, we varied the minimum locality level from 0 to 1. A locality level 0 means that a job would be scheduled regardless of the number of local tasks. On the other hand, a locality level 1 means that a job could only be scheduled if all of its tasks were able to run locally.

In this experiment, the resources were shared by 3 frameworks, whose schedulers implemented the `Lease` aware task scheduling. We only show the results from the high workload since during the low workload condition, the 3 schedulers were be able to schedule their tasks with high locality for most of the time regardless of the threshold.



(a) Average job turnaround time.

(b) Locality level achieved.

Figure 2.5: Effects of varying the minimum locality threshold.

The jobs were posted using a Poisson Process with a rate of 50 seconds and we posted 2 jobs at a time instead of 1 job to allow multiple jobs to reside in the job queue.

Figure 2.5(a) shows the average job turnaround time over different minimum locality threshold. We can see that the average job turnaround time can be further improved by 13% when the minimum locality level was increased from 0 to 0.5. However, as the threshold increased from 0.5 to 1.0, the percentage of tasks that could be scheduled locally were relatively stable as shown in Figure 2.5(b). Setting the threshold too high even increased the average job turnaround time. The reason is that, more jobs would have higher waiting time in the queue due to the limited number of local tasks. However, most of the time spent in the queue was unnecessary since most of the jobs could not be scheduled with a perfect locality even when all of the nodes were available.

2.6 Related Work

Geo-distributed Data Analytics Systems. Recent work has considered utilizing multiple data centers/edge clusters for geo-distributed batch analytics with the goal of minimizing query execution time or efficiently utilizing WAN bandwidth [**?**, **?**, **?**, **?**]. Although our work also considers a similar environment, we focus on the problem of sharing resources across multiple data analytics frameworks. Thus, our work in this chapter is orthogonal to theirs.

Resource Management. The problem of sharing resources across multiple data

analytics frameworks has been studied extensively [?, ?, ?, ?]. However, they are intended for a cluster/data center environment. In contrast, this chapter address the problem of sharing resources in a wide-area environment. As we have shown in this work, existing cluster-based resource management needs modification to a wide-area environment due to the critical needs for locality job/task scheduling. Although our resource sharing model is similar to Mesos, our proposed technique uses a lease-based mechanism in contrast to the resource-offer mechanism in Mesos. Furthermore, we incorporate the state sharing mechanism used in Shared State resource sharing model that shares the future availability of all resources to all frameworks to help framework schedulers apply delay scheduling effectively.

Job Scheduling in a Shared Environment. Researchers have also considered optimizing individual framework scheduler in a cluster-based environment [?, ?, ?, ?, ?, ?]. Torque [?] is a batch scheduler for HPC cluster where data locality is not the main issue for such jobs. Delay scheduling [?] and Quincy [?] incorporate techniques to handle locality and fair sharing in a shared cluster environment. In contrast to their work, we consider a wide-area environment, which requires a different job scheduling decision due to the differences in the underlying environments.

2.7 Conclusion

In this chapter, we present `Awan`: a resource manager for data-intensive applications in a geo-distributed Edge Cloud environment. `Awan` uses a lease-based resource sharing mechanism to dynamically partition resources across multiple data analytics frameworks. A lease provides a guarantee that a particular resource will be held by a framework for no longer than the lease time. The lease information is shared to all frameworks so that each framework can determine whether to wait (or not) for the desired unavailable resources for better scheduling decision. This results in a higher locality scheduling that can be achieved by each framework and improves the overall query execution performance of geo-distributed data-intensive queries. We further propose a locality-based priority scheduling that can further improve the locality job scheduling of geo-distributed data analytics frameworks.

Chapter 3

Multi-Query Optimization in Wide-Area Streaming Analytics

3.1 Introduction

Recent years have seen a growing interest in wide-area streaming analytics, where analysts need to extract timely information from large amounts of data that are continuously being generated across multiple locations. Examples of these data include not only traditional log updates from content distribution networks (CDN) but also user-generated microblogs, sensor data from distributed IoT devices, and video streams from distributed surveillance and traffic control cameras. Such data are naturally produced in a geo-distributed manner *near the edge*. The main challenge in analyzing these data is in extracting information in a timely manner [**?**, **?**].

The interest in real-time analysis over continuous data streams has resulted in the recent development of various distributed stream processing systems [**?**, **?**, **?**, **?**, **?**, **?**]. However, these systems have been designed primarily for a *centralized*, *tightly-connected* cluster environment where compute nodes are inter-connected with high-speed network. Using such systems to analyze geo-distributed data streams is impractical since it requires transmitting large amounts of data continuously over the wide-area network (WAN) that has limited bandwidth, slow, and expensive. This *centralized* approach typically leads to wasteful WAN bandwidth consumption and is often unable to satisfy the timeliness requirements of most applications [**?**, **?**, **?**, **?**].

Most of the work in geo-distributed data analytics has instead focused on batch-oriented processing, where finite input data sets are available prior to a query execution [?, ?, ?, ?, ?, ?]. In this case, the main challenge is to schedule each query that minimizes either the overall execution time or WAN bandwidth consumption. Others have also looked at the problem of geo-distributed data analytics in the context of stream-oriented processing where long-running queries are deployed to extract information from continuous data streams [?, ?, ?, ?]. However, most of them focused on optimizing an individual query execution. In contrast, we consider optimizing multiple queries by applying multi-query optimization in a WAN-aware manner.

In practice, the multi-tenancy nature of a Cloud environment leads to multiple queries running concurrently and competing for limited, shared resources. Recent work has indicated that it is common in a production environment for multiple queries to exhibit common executions, whether in reading the same set of inputs or performing the same data processing, especially for queries from the same application domain or those that rely on popular data [?, ?, ?, ?, ?, ?, ?, ?]. Furthermore, as more and more data are increasingly geo-dependent and made available to the public, it is increasingly likely that more geo-distributed data analytics queries will share common executions. As a concrete example, Twitter data streams are commonly analyzed for different purposes including sentiment analysis [?], finding relevant audiences for an advertisement [?], and detecting trending topics in a certain area or globally [?, ?]. Another example includes CDN logs that are continuously monitored for high quality service assurance, network monitoring, and user behavior analysis.

Based on this insight, we examine the opportunity of applying multi-query optimization in the context of wide-area streaming analytics. Our goal is to efficiently and effectively utilize the limited WAN bandwidth while providing low-latency and high-throughput execution of multiple concurrent queries. We first study different types of cross-query sharing opportunities: (1) *input-sharing*: where multiple queries share a common subset of input data, (2) *operator-sharing*: where multiple queries perform the same data processing on the same inputs, and (3) *output-sharing*: where multiple queries additionally share partial output (or intermediate) results. Furthermore, we demonstrate the importance of *WAN awareness* in applying multi-query optimization in a wide-area environment: both for query planning and for operator scheduling.

There are several challenges in applying multi-query optimization (MQO) in the context of wide-area streaming analytics. First, multiple queries may be submitted to the system independently at different times by different users and hence, it may not be possible to optimize these queries together prior to their deployment using the MQO techniques proposed for batch-oriented workloads [**?**, **?**, **?**]. Second, most streaming analytics queries are long-running and latency sensitive [**?**, **?**, **?**]. Thus, it is very inefficient and impractical to interrupt existing query executions whenever a new query arrives to optimize them together. Instead, our technique optimizes multiple query executions in an *online* manner by allowing queries to share their common executions *incrementally* without disrupting any of the existing executions. The wide-area environment further imposes unique challenges in applying multi-query optimization due to the highly heterogeneous and limited bandwidth availability of the wide-area network. We show that applying MQO designed for a local environment in a wide-area environment without network awareness is sub-optimal and may lead to performance degradation due to the assumptions of homogeneous and high-bandwidth network that are invalid in a real wide-area system deployment.

We have implemented our WAN-aware multi-query optimization into a system prototype called `Sana`: an Apache Flink [**?**]-based stream processing system that we have adapted for wide-area deployments. We quantitatively evaluated `Sana` using 14 geo-distributed EC2 [1] data centers. Experimental evaluation using multiple streaming analytics queries [**?**, **?**] on a real Twitter trace shows that `Sana` is able to achieve 21% higher throughput while saving WAN bandwidth consumption by 33% compared to the state-of-the-art WAN-aware, sharing-agnostic system.

We summarize our contributions in this chapter as follows:

- We propose a multi-query optimization in the context of wide-area streaming analytics that allows multiple queries to *incrementally* share their common executions in an *online* manner (Section 3.4).

- We highlight the importance of *network awareness* in applying multi-query optimization in a wide-area environment, both in planning and scheduling multiple query executions (Section 3.5).

[1] https://aws.amazon.com/ec2/

- We have implemented our WAN-aware multi-query optimization techniques in a system prototype based on Apache Flink (Section 3.6).
- We experimentally demonstrate the effectiveness of our WAN-aware multi-query optimization through a real system deployment across geo-distributed EC2 data centers using Twitter trace-driven queries (Section 3.7).

3.2 Background and Motivation

In this section, we discuss the background of wide-area streaming analytics and illustrate through an example the benefits of applying multi-query optimization to this context.

3.2.1 Wide-Area Streaming Analytics

Stream Execution Model

Stream processing systems can be generally classified into two different classes based on their computational model: (1) the *dataflow* model [**?**, **?**, **?**, **?**], and (2) the *bulk-synchronous parallel* (BSP) model [**?**, **?**, **?**]. Here, we focus on the dataflow model where data streams flow continuously from one or more data sources into the system and are transformed by a set of *stream operators*. We consider this model over the BSP model for two reasons. First, it allows data streams to be processed with lower latency and higher throughput [**?**, **?**]. Second, the BSP model incurs higher communication overhead due to the frequent synchronization at every micro-batch boundary [**?**], which will be inefficient in a wide-area environment. However, our proposed techniques are not limited to the dataflow processing model, and can be adapted to the BSP model.

A streaming analytics query is typically written using a high-level, SQL-like language [**?**, **?**]. The query is (1) translated and optimized by a *Query Optimizer* into its corresponding execution plan, represented using a directed acyclic graph (DAG), and (2) deployed by a *Job Scheduler*. A query execution graph, denoted as $G = (V, E)$, consists of vertices V and edges E. Each vertex $v \in V$ corresponds to a stream operator f_v that consumes input streams I from its predecessor (upstream) vertices and produces output streams O to its successor (downstream) vertices ($O = f_v(I)$). Each edge $e \in E$ represents a data flow between two vertices. Example of stream operators

include *source*, *map*, *reduce*, *join*, *filter*, and *sink*. The *source* and the *sink* operators are specialized operators that receive input streams from external sources and output the results to final destinations respectively.

Geo-Distributed Stream Processing

We consider a stream processing system comprising multiple *compute nodes* that are geo-distributed across multiple sites, and a *master node* located in one of the sites. A streaming analytics query is submitted to the master node comprising a *Query Optimizer* and a *Job Scheduler*. The *Query Optimizer* will optimize the execution plan of the query (e.g., parallelize and chain multiple operators) and the *Job Scheduler* will deploy each parallel execution instance (task) on a compute node.

The inputs of a wide-area streaming analytics query are produced by multiple sources that are geo-distributed, and they are continuously ingested into nearby edge clusters or data centers. Examples of such data streams include sensor readings, microblogs from social network applications, and log updates from distributed CDN servers. Each query continuously reads these geo-distributed input streams, processes them, and outputs its results to one or more final locations, e.g., stored in databases, displayed on a monitoring dashboard, or streamed back as new inputs for iterative analysis.

To minimize data transfer overhead between operators, the *Job Scheduler* will deploy connected operators on the same site. However, common operators such as *union*, *shuffle*, and *join* may require cross-site data transmission since their inputs may be generated at different locations. Thus, the *Query Optimizer* and the *Job Scheduler* should be aware of the underlying WAN to generate an optimized execution plan and a scheduling decision respectively that can effectively utilize WAN bandwidth [**?**, **?**, **?**].

3.2.2 Benefits of Multi-Query Optimization in Wide-Area Settings

Multi-Query Optimization in Data Analytics World

Multi-query optimization (MQO) is a well-studied topic in the database community to improve the performance of multiple query executions, especially in relational databases [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. Since many data analytics queries often rely on common popular data sets and may perform common executions, recent work has argued that it is imperative

to apply MQO in the context of data analytics to improve the performance of multiple data analytics queries [**?**, **?**, **?**, **?**, **?**]. Here, the *Query Optimizer* needs to identify the commonality between queries and potentially combine their executions to mitigate redundant executions. The combined execution must produce the same outputs as those produced by executing the queries independently.

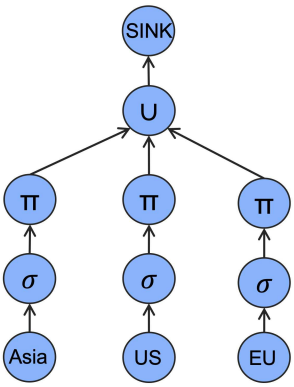
Applying multi-query optimization in a wide-area environment can reduce WAN bandwidth consumption by eliminating the redundancy in processing and transmitting duplicate data over the WAN. In the face of bandwidth constraints, this can improve the overall performance of concurrent query executions. Although there have been attempts that look at the opportunity of optimizing multiple queries in the context data analytics, their focus have been largely on batch-oriented workloads [**?**, **?**, **?**]. These approaches are not applicable for stream-oriented workloads because most streaming analytics queries are long running: *deployed once and run indefinitely* [**?**, **?**]. Thus, applying MQO in streaming analytics should be done in an *online* manner as new queries arrive by sharing any common execution *incrementally*. Previous attempts have also looked at the opportunity of applying multi-query optimization for stream-oriented queries over continuous data streams, but focused on memory limitations because they were designed for a single-server deployment [**?**, **?**, **?**]. On the other hand, we consider a wide-area environment where the limited WAN bandwidth is typically the main constraint.

Illustrative Example

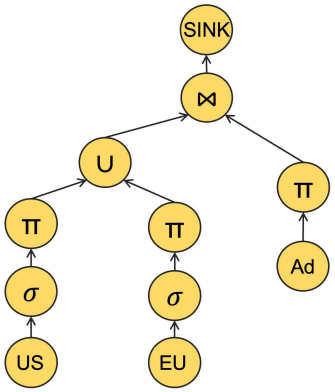
To see the opportunity of applying multi-query optimization in wide-area streaming analytics, consider the following illustration. Suppose there are 2 different analytics queries that are submitted to the system:

`Query 1`: A marketing group is periodically monitoring the trending topics in Twitter across the US, Europe, and Asia to support their operational decisions:

```
SELECT Time, Topic, COUNT(*)
FROM   Host.US, Host.EU, Host.Asia
GROUP BY WINDOW(Time.Minutes(1)), Topic
HAVING COUNT(*) > 100
```



(a) Logical execution of Query 1.



(b) Logical execution of Query 2.

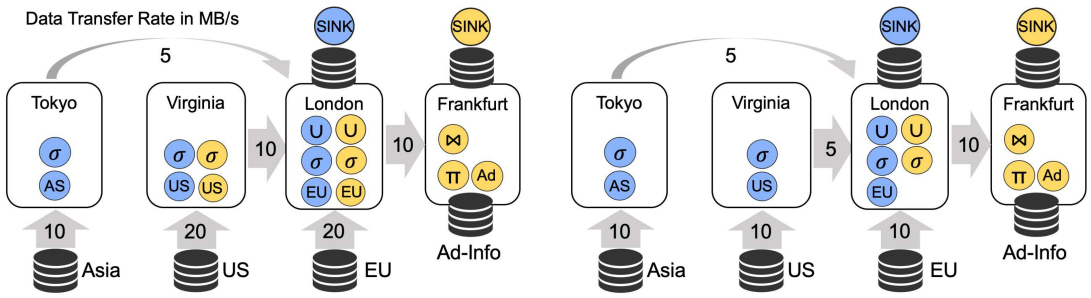
Figure 3.1: Logical execution plans of Query 1 and Query 2.

`Query 2`: Another group of analysts is monitoring the impressions from Twitter users in the US and Europe that are related to a specific type of campaign:

```
SELECT Time, AdInfo.Campaign
FROM (SELECT Time, Topic
      FROM Host.US, Host.EU
      GROUP BY WINDOW(Time.Seconds(30)), Topic
      HAVING COUNT(*) > 100) AS Tweet, AdInfo
WHERE AdInfo.Topic = Tweet.Topic
```

Figure 3.1 shows the logical execution plans of both queries. In this example, both queries subscribe to common input sources (`US` and `EU`), deserialize, filter, reduce the data (σ and π) to remove irrelevant information (e.g., discard user profile), aggregate the results ($\cup$), and send only the relevant information to their corresponding final locations. In the case of `Query 2`, the intermediate results are further joined ($\bowtie$) with static data that are stored in `AdInfo`.

Figure 3.2(a) shows the independent deployment of the two queries. For clarity reasons, suppose the input stream rate from each source is $10MB/s$ and the *selectivity* of each selection and projection operator is 0.5. We also consider the data transfer overhead within a site to be negligible since intra-data center bandwidth is typically 1-2 orders of magnitude higher than inter-data center bandwidth [**?**]. In this case, deploying



(a) Independent deployment.

(b) Shared deployment.

Figure 3.2: Execution sharing between Query 1 and Query 2.

the two queries independently will consume WAN bandwidth with a rate of $75MB/s$ ($40MB/s$ for `Query 1` and $35MB/s$ for `Query 2`).

However, we can see that both queries partially share common input streams (`US` and `EU`) and perform similar data processing (e.g., filtering user info). If the *Query Optimizer* is able to identify these commonalities, it may combine their common executions, which will significantly reduce the WAN bandwidth consumption rate to $50MB/s = 40MB/s + 10MB/s$ (Figure 3.2(b)), which saves ~33% of the original bandwidth consumption. This illustration shows that optimizing multiple query executions in wide-area streaming analytics can significantly save WAN bandwidth consumption.

In addition to saving WAN bandwidth consumption, sharing common executions between multiple queries can also improve the overall performance in the face of bandwidth constraints. In the previous example, if the available bandwidth from the Virginia data center to the London data center is less than $10MB/s$, deploying the two queries independently will result in a bandwidth contention between the two queries. One possible solution is to reduce the data transmission rate over the bottleneck link through an approximation, aggregation, or data reduction, which trades the output's quality for higher overall performance [**?**, **?**, **?**, **?**]. Alternatively, the *Query Optimizer* may choose a less optimal query execution plan that avoids the congested network link [**?**]. However, we argue that making this trade-off is unnecessary if the system is able to detect that the problem arises due to redundant data transmission. Furthermore, these techniques still result in a wasteful bandwidth consumption that could be reduced.

3.3 Sana: System Architecture

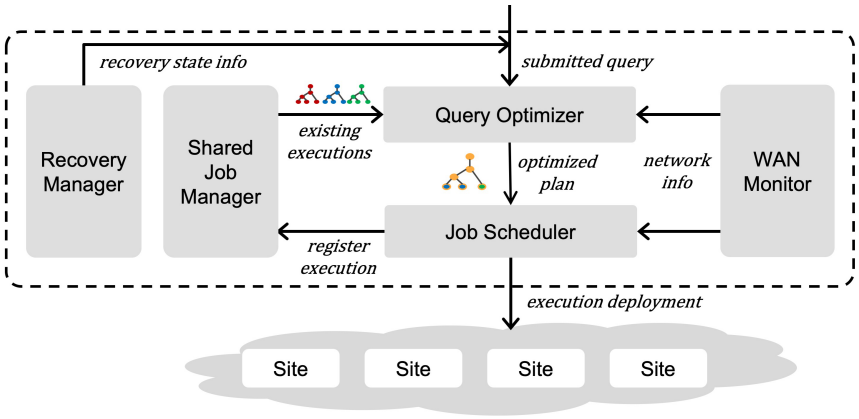


Figure 3.3: Sana system architecture.

We propose a geo-distributed stream processing system called `Sana` which implements multi-query optimization in a WAN-aware manner. Figure 3.3 shows the system architecture of `Sana`. When a new (possibly a recovery) query is submitted to the system, the *Query Optimizer* will optimize its execution plan while considering the inter-site bandwidth information that is periodically monitored by the *WAN Monitor*. This inter-site network information is particularly important to optimize the execution plan and the task placement of a wide-area data analytics query [**?**].

When applying multi-query optimization, the *Query Optimizer* will also consider the deployment of the existing queries that is provided by the *Shared Job Manager* to identify any commonality between the newly submitted query and the existing ones (Section 3.4). After the optimized query execution plan has been generated, the *Job Scheduler* will schedule and deploy each operator instance on a compute node in a WAN-aware manner to minimize the overall query execution latency and/or WAN bandwidth consumption (Section 3.5). Once a query has been deployed, it may periodically checkpoint its execution state and report the state metadata to the *Recovery Manager*. This mechanism allows the system to replay a query execution from its latest checkpoint state in the case of failures. The implementation details will be discussed in Section 3.6.

3.4 Multi-Query Optimization

In this section, we look at how the *Query Optimizer* optimizes multiple query executions by sharing any commonality between them. We first study different types of sharing opportunities that can be exploited between two queries (Section 3.4.1), and show how to apply them across multiple queries (Section 3.4.2). We will discuss the WAN awareness in optimizing multiple query executions in Section 3.5.

3.4.1 Sharing Opportunities

Input-Operator Sharing

A natural way to determine whether two queries share common executions is to compare their vertices. Two vertices v_1 and v_2 are considered equivalent *iff* they share the same input streams $I_{v_1} = I_{v_2}$, perform the same transformation function $f_{v_1} = f_{v_2}$, and thus produce the same output streams $O_{v_1} = O_{v_2}$. We refer to this type of sharing as `IN-OP`. In this case, deploying the two vertices independently will result in a full redundancy in both transmitting and processing duplicate data. This redundancy can be eliminated by deploying only one of the vertices. In this case, the *Query Optimizer* can merge the two vertices together, i.e., let the *Job Scheduler* know that v_2 does not need to be scheduled if v_1 has already been deployed.

In practice, two vertices may share common inputs and operators, but output the results to a different set of downstream vertices (possibly with some overlap). We denote the set of v's downstream vertices as $D_v = \{d_v^1, \ldots, d_v^n\}$. These conditions are especially common in the early stages of executions where multiple queries may read the same input streams from the same data sources although their downstream vertices tend to be more specific to each individual query. In this case, the output streams to any of the downstream vertices that are not shared by the two vertices need to be replicated, while the common outputs can be transmitted only once (Figure 3.4).

Input-Only Sharing

Since multiple vertices with different operators/transformation functions may rely on a common set of input streams, we relax the sharing requirement of the `IN-OP` type of

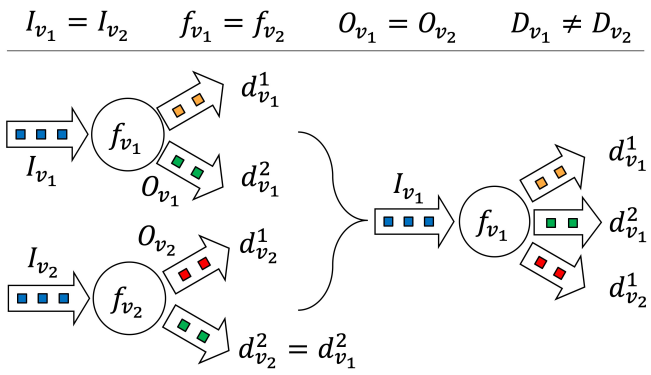


Figure 3.4: IN-OP: Input-Operator Sharing. Here, v_1 and v_2 share common input streams and operators, but only partially share the output streams.

sharing by removing the operator-equality condition, i.e., $f_{v_1} \neq f_{v_2}$, therefore $O_{v_1} \neq O_{v_2}$. This allows two vertices to share their common input streams even though they have different operators. We refer to this input-only sharing as `IN`. In this case, independently deploying the two vertices will result in redundancy in transmitting duplicate input data. Unlike `IN-OP`, this type of sharing requires both vertices to be deployed since they rely on different transformation functions. However, applying this type of sharing will eliminate the redundancy in transmitting duplicate input streams from their common upstream vertices, which can be highly beneficial in the case where the inputs are transmitted over slow and limited bandwidth links, as in a wide-area environment.

In wide area settings, the `IN` type of sharing can be exploited by deploying the two vertices on the same site (or the same node). However, the physical deployment of a stream operator is typically determined by the *Job Scheduler* after the query execution plan has been generated by the *Query Optimizer*. Thus, the *Query Optimizer* needs to provide a *hint* to the *Job Scheduler* in exploiting this type of sharing. The co-location deployment of two vertices does not necessarily eliminate the redundancy in transmitting duplicate data because they are still considered as two independent stream edges to their respective downstream vertices. To exploit this type of sharing, we introduce a lightweight *router* operator R which (1) keeps track of the input edges of each input stream originated from remote vertices, and (2) forwards each record to every downstream vertex without performing any data transformation. Note that the *router* operator does not buffer nor batch the records, instead it only routes the records to

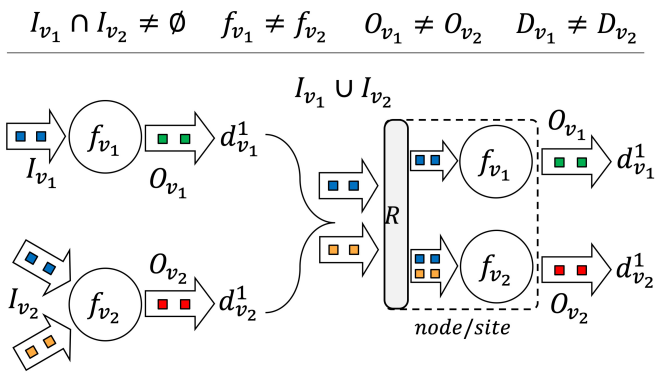


Figure 3.5: IN: Input Sharing. v_1 and v_2 only partially share common input streams.

multiple operators, similar to the task of router in networks. Thus, the overhead of the router operator is negligible as shown in Section 3.7.

Partial Input Sharing. In the case of `IN-OP`, two vertices that share common operators must rely on the same *exact* input streams since in general applying the same transformation to different input sets does not guarantee the same resulting outputs. This strict input-stream-equality can further be relaxed in the case of `IN` since the two vertices do not rely on the same results. Thus, the `IN` type of sharing allows two vertices with different operators to *partially* share their input streams (Figure 3.5).

3.4.2 Sharing Across Multiple Queries

Having discussed different sharing opportunities that can exist between two queries, we will now look at how the *Query Optimizer* exploits these opportunities across *multiple* queries. Since most streaming analytics queries are long-running, it is possible that a newly submitted query exhibits common executions with multiple existing queries that may have already been deployed. Thus, the *Query Optimizer* needs to determine with *which* of the queries it should share the new query.

One possible approach to determine *which* query to share is by finding a query that exhibits the highest *similarity score* using a subgraph-matching algorithm [**?**, **?**]. However, we argue this approach is sub-optimal since it limits the sharing opportunities to only 1 query. Instead of finding the similarity in a *query-centric* manner, we adopt a *vertex-centric* philosophy where a query may share its vertices with *multiple* queries.

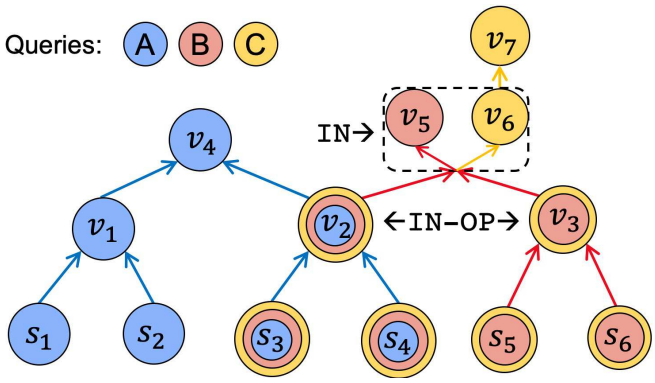


Figure 3.6: Cross-query execution sharing: C shares its execution with A and B.

This will result in a higher overall degree of sharing. We compare a new query with each of the existing queries topologically from the *source* vertices. Traversing the vertices in topological order gives the benefit of early termination in traversing a graph. If two vertices are not equivalent ($v_1 \neq v_2$), by definition, none of their downstream vertices are equivalent, and hence they do not need to be compared.

Although finding common vertices among multiple queries can be computationally expensive, this step is only performed during the query planning stage. Since most streaming analytics queries are long-running, this overhead is justified for higher overall execution performance and better resource utilization. To reduce the analysis cost, the *Query Optimizer* may limit the number of queries to be analyzed or adopt a group-based analysis, as proposed by existing work in Internet Databases [**?**], which reduces the number of vertices that needs to be analyzed.

Figure 3.6 shows an example where query (`C`) shares its execution with existing queries (`A` and `B`). When `C` arrives, the *Query Optimizer* finds that `C` shares (1) common input-operators with `B` at s_5, s_6, and v_3, with both `A` and `B` at s_3, s_4, and v_2, and (2) input streams with `B` ($I_{v_5} \cap I_{v_6} \neq \emptyset$). In this case, the *Query Optimizer* may exploit these sharing opportunities by merging the common executions of these queries. Thus, the *Job Scheduler* only needs to deploy two additional vertices for `C`: v_6 that exploits `IN` sharing with v_5, and v_7 that does not exhibit any sharing opportunity with the rest of the vertices, while s_3, s_4, s_5, s_6, v_2, and v_3 are shared with `IN-OP` sharing.

3.5 WAN-Aware Optimization

Our discussion so far has focused on the sharing opportunities between multiple queries without considering the wide-area constraints. In this section, we focus on addressing the challenges of applying these sharing opportunities in a wide-area environment. Specifically, we propose a WAN-aware optimization to the *Query Optimizer* in generating and optimizing query execution plans while considering the sharing opportunities with existing query executions (Section 3.5.1) and WAN-aware operator placement to the *Job Scheduler* in deploying stream operators (Section 3.5.2).

3.5.1 WAN-Aware Query Planning

In the context of wide-area data analytics, the *Query Optimizer* needs to consider the inter-site bandwidth availability to generate an optimized query execution plan for each query [**?**]. Similarly, the *Query Optimizer* must also optimize multiple query executions in a WAN-aware manner. The WAN awareness in this context is used to determine whether a query should share its execution with other queries (when possible) based on the current WAN bandwidth availability between sites. Without WAN awareness, sharing executions across multiple queries may result in WAN bandwidth contention that will degrade the performance of either or both the new and the existing queries.

Since our *Query Optimizer* analyzes the commonality between queries in a vertex-centric manner, a vertex may exhibit more than one sharing opportunities with multiple vertices from different queries. Figure 3.7 shows a situation where vertex v can share both its inputs and operator with v_2, or partially share its inputs with either v_1 or v_3. In this case, the *Query Optimizer* needs to determine *which* of these sharing opportunities should be exploited, or decide not to share the execution at all.

One possible approach is to choose a vertex that maximizes the degree of sharing since intuitively it will maximize the duplicate elimination. However, this naive approach may result in a performance degradation. Consider the scenario shown in Figure 3.7. If the *Query Optimizer* always tries to maximize the sharing regardless of the network conditions, it will exploit the `IN-OP` type of sharing with v_2 since the input streams of vertex v are fully covered by v_2. However, we can see that Site B does not have sufficient bandwidth capacity for transmitting its output streams. Thus,

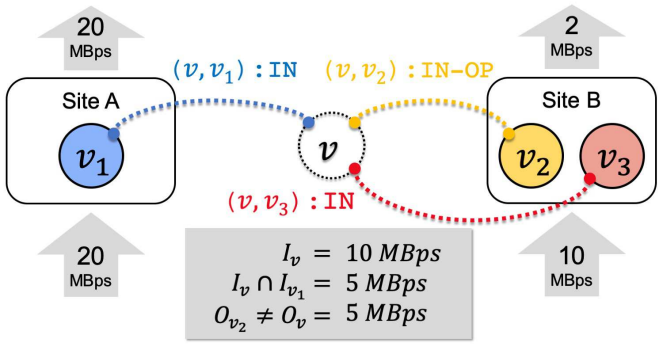


Figure 3.7: Sharing opportunities: v exhibits IN-OP with v_2, and IN with v_1 and v_3.

exploiting `IN-OP` with v_2 may result in bandwidth contention between v, v_2, and v_3. On the other hand, if the *Query Optimizer* is aware of the bandwidth constraints, it may exploit the `IN` type of sharing with v_1 by partially sharing their input streams at Site A. This decision is preferable because it does not cause any bandwidth contention that will degrade the overall query execution performance. Thus, there is a trade-off between minimizing bandwidth consumption (maximizing sharing) and maximizing the performance of concurrent executions.

Algorithm 1 shows how the *Query Optimizer* considers WAN bandwidth availability to determine *which* sharing opportunities (if any) to be exploited. In the case of `IN-OP`, the *Query Optimizer* needs to ensure that the site where the shared vertex v_i has been deployed, has sufficient egress bandwidth capacity to transmit additional output streams (Line 7). This can be estimated proportionally to the increase in the number of output stream consumers since both vertices rely on the exact same output data streams ($O_v = O_{v_i}$), and only their downstream vertices are different. In the case of `IN` where vertices only share partial input streams, the *Query Optimizer* needs to further ensure there is sufficient bandwidth in both the ingress and egress links to transmit additional input and output streams respectively. If the *Query Optimizer* predicts that exploiting the opportunity can potentially result in bandwidth contention, it will not exploit the opportunity, which trades bandwidth utilization for higher overall performance.

Note from Lines 8 and 13 that the *Query Optimizer* outputs a set of vertices that can be shared by each vertex (if any) instead of only a single vertex as long as they

Algorithm 1 WAN-aware execution sharing

```
1: procedure FIND-COMMON-VERTICES(v, V)
2:     for v_i ∈ V topologically do
3:         share ← getShareType(v, v_i)
4:         (B_in, B_out) ← getBandwidth(v_i)
5:         if share == IN-OP then
6:             ΔO ← (|D_v ∪ D_{v_i}| / |D_{v_i}|) × O_v
7:             if B_out > ΔO then
8:                 add v_i to the set of IN-OP vertices
9:             end if
10:        else if share == IN then
11:            ΔI ← I_v − I_{v_i}
12:            if B_out > O_v and B_in > ΔI then
13:                add v_i to the set of IN vertices
14:            end if
15:        end if
16:    end for
17: end procedure
```

ensure sufficient bandwidth for deployment. In this case, the *Job Scheduler* needs to choose *which* vertex to be shared. We adopt this design to give the *Job Scheduler* a flexibility to apply different optimization in scheduling different queries. For example, some queries may tolerate higher delay for lower bandwidth consumption while others may require real-time results even though they consume more bandwidth.

3.5.2 WAN-Aware Operator Scheduling

While the previous section focuses on bringing WAN awareness to the *Query Optimizer* in planning a query execution, this section focuses on incorporating WAN awareness to the *Job Scheduler* in deploying the execution. Once the *Query Optimizer* has identified a set of vertices that can be shared for each vertex in the query execution plan, the *Job Scheduler* is responsible for the actual deployment of the vertices themselves. Algorithm 2 shows how the *Job Scheduler* schedules each operator while considering the sharing opportunities that have been identified by the *Query Optimizer*. The *Job Scheduler* will place and deploy each operator in the physical execution graph topologically based on the deployment of its upstream vertices. Although this approach may not result in the most optimal end-to-end deployment of the entire graph, this has been shown to work reasonably well in practice with significantly lower complexity [**?**].

Algorithm 2 WAN-aware operator placement

```
1: procedure SCHEDULE(v)
2:     if find v_i ∈ set IN-OP then
3:         add edges from v_i to ΔD ← D_v \ D_{v_i}
4:     else if find v_i ∈ set IN then
5:         deploy v at the same site as v_i
6:     else if I_v are local input streams then
7:         site-locality deployment
8:     else                                  ▷ neither share-able nor a local operator
9:         WAN-aware deployment
10:    end if
11: end procedure
```

In exploiting the sharing opportunities, the *Job Scheduler* prioritizes exploiting `IN-OP` over `IN` because the gain of `IN-OP` $\geq$ `IN` in terms of minimizing WAN bandwidth consumption since the former type of sharing covers the benefits of the latter. Note that exploiting any of these opportunities guarantees sufficient bandwidth deployment since the *Query Optimizer* has already omitted those that may result in a bandwidth contention. If a vertex exploits the `IN-OP` type of sharing with any of the existing vertices, the *Job Scheduler* does not need to deploy the vertex. However, the *Job Scheduler* may need to update the existing execution by creating additional edges from the shared vertex to any of the additional downstream vertices that is not shared by the two executions (Line 3). On the other hand, vertices that exhibit `IN` type of sharing will be deployed on the same site as their corresponding shared vertices to mitigate redundant data transmission over the WAN (Line 5).

If a vertex can be shared with multiple vertices of the same sharing type (e.g., v exhibits `IN` with both v_1 and v_3 in Figure 3.7), the *Job Scheduler* needs to determine *which* of the vertices should be shared (Lines 2 and 4). Since our goal is to minimize WAN bandwidth consumption, our *Job Scheduler* will choose a vertex that maximize the sharing. Although maximizing sharing may not necessarily minimize latency, in practice this will result in an improved execution performance [**?**]. If the goal is to minimize delay, the *Job Scheduler* may choose the vertex that minimizes latency.

Vertices that do not exhibit any sharing opportunity will be deployed based on the locations of their upstream vertices. Those that rely only on local input streams will be deployed on the same site as their upstream vertices to minimize the communication

overhead, especially the high latency of the wide area network. On the other hand, vertices that rely on one or more input streams originated from remote sites will be deployed using a WAN-aware operator deployment. We adapt the cost model from Hourglass [**?**] which optimizes stream operator placement that balances WAN bandwidth consumption and latency, by minimizing $\sum_{l \in L} \frac{DR_l(\ell_l)^2}{B_l}$ where l is a link between two sites, DR_l is the data rate transmitted over the link, ℓ_l is the latency overhead, and B_l is the available bandwidth of the link. Any update of a link will be reflected in the bandwidth availability that is continuously being monitored by the *WAN Monitor*.

3.6 Implementation

We have implemented `Sana` in a system prototype based on Apache Flink [**?**] - a stream processing system with the dataflow computational model. We have modified and adapted the original Flink system to a wide-area environment by implementing network monitoring and multi-query optimization modules, as well as incorporating WAN awareness to both the *Query Optimizer* and *Job Scheduler*.

- **WAN bandwidth monitoring.** The bandwidth availability between sites is continuously monitored by the *WAN Monitor*. Congested links are detected by the ratio of the current bandwidth utilization over the maximum available bandwidth [**?**]. A ratio of <1 indicates that the network link has spare bandwidth capacity while a ratio >1 indicates that the bandwidth is contended. This bandwidth information is shared with both the *Query Optimizer* and the *Job Scheduler* to implement the WAN-aware query planning (Section 3.5.1) and operator scheduling (Section 3.5.2) respectively.

- **Multi-query optimization.** We have implemented our WAN-aware multi-query optimization module in Flink to find common executions between a newly submitted and existing queries in a WAN-aware manner. To exploit the `IN` type of sharing, the *Query Optimizer* will modify the original query's execution plan by adding a *router* operator for every vertex that rely on remote input streams. The *router* operators are added proactively to prevent suspending the execution of an existing vertex. Although the use of *router* operators would still incur duplicate

data streams from the *router* to the downstream operators, this data forwarding happens within a local environment (within a site or even a node) and hence, its overhead is negligible compared to the overhead from transmitting duplicate data across sites. We show in Section 3.7 that the overhead of the *router* operator is negligible even when it is not shared.

- **WAN-aware scheduling.** The default Flink scheduler has already implemented node-locality scheduling, which tries to schedule a vertex on the same node with any of its upstream vertices. However, if an operator relies on input streams from different nodes, the original scheduler will choose one of the nodes without considering the network condition (bandwidth availability and latency) between them. This simple policy works well in a centralized cluster environment, for which Flink has been designed. However, this scheduling policy may result in a non-optimal operator placement in wide-area settings. We have modified the default Flink's scheduler by incorporating the WAN awareness discussed in Section 3.5.2.

- **Fault tolerance.** A query whose vertices are shared with other queries may be terminated either intentionally (e.g., the analysis is complete) or unintentionally (e.g., failure in one of the vertices in the query plan). To handle these issues, the *Shared Job Manager* keeps track of every vertex that is shared with other queries. Whenever a query that shares a vertex is terminated, it removes the reference to the shared vertex. A vertex execution will only be terminated if *all* queries that share the execution have been terminated. This simple approach prevents cascading failures unless they happen directly on the stream operator logic. Recovering from failures that involves shared vertices is challenging since a stream processing system needs to ensure the exactly-once semantic processing guarantee. `Sana` uses a *checkpoint-and-replay* fault recovery mechanism, where each query periodically checkpoints its processing state and thus the system can restore its execution from the last checkpointed state upon recovering from failures [**?**]. We maintain an independent state for each vertex that is shared by multiple queries. Thus, if a sharing query fails, other queries can continue their executions and update their states independently. When a failed query is restarted, it may not be able to immediately share the vertex it was sharing earlier since the shared vertex may

have a different state. In this case, the query needs to catch up its processing in order to re-share the vertex.

- **Query adaptation.** Since many streaming analytics queries are long running, a query needs to gracefully adapt to runtime dynamics, such as changes in workload or network topology [**?**, **?**]. In this case, the *Query Optimizer* and the *Job Scheduler* may change the execution plan and/or the deployment of the query respectively whenever the environment changes significantly. We will address the problem of adapting a shared query execution later in Chapter 5.

3.7 Experimental Evaluation

Experimental Setup. We experimentally evaluate the effectiveness of `Sana` using a wide-area system deployment across 14 geo-distributed EC2 data centers. The compute nodes were deployed on 8 of the sites (Virginia, California, Canada, London, Frankfurt, Sydney, Tokyo, and Singapore) and the input streams are generated by external sources that were located on the other 6 sites (Ohio, Oregon, Ireland, Seoul, Mumbai, and Sao Paulo). To prevent an inaccurate evaluation caused by the data exchange overhead between the external sources and the system, we follow the design proposed by recent work which uses distributed in-memory data generators instead of message brokers as the external sources [**?**].

We also measured the bandwidth availability and the latency between the sites prior to running the experiments as initial network information to the *WAN Monitor*. Our measurements show that WAN bandwidth between EC2 data centers ranged from 20Mbps to 280Mbps, confirming a similar trend from prior work [**?**, **?**].

Dataset and Queries. All experiments are based on real geo-tagged Twitter trace that was collected from Twitter Streaming APIs [2] in December 2015. It consists of approximately 4 million tweets per day. Since the trace only represents a sample of real Twitter workload, we scaled the playback rate to 6000∼8000 tweets per second to reflect the actual tweet rate [**?**]. The tweets were distributed across the 6 input sources based on their geographic information.

[2] https://developer.twitter.com/en/docs

Table 3.1: Sana query details

Category	Query Examples	Num. Operators
Tweet Statistics	[rate] of [tweet, hashtag] on [country, language, topic]	10-18
Users Analysis	Num. of tweet per [gender, age-group] per [country, language]	12-18
Top-k Analysis	Top-k [hashtag, topic] per [language, country]	10-15
Sentiment Analysis	Sentiment of each [hashtag, country, topic]	12-18
System Load	[rate, count] of [data, request] per [node, region]	6-10

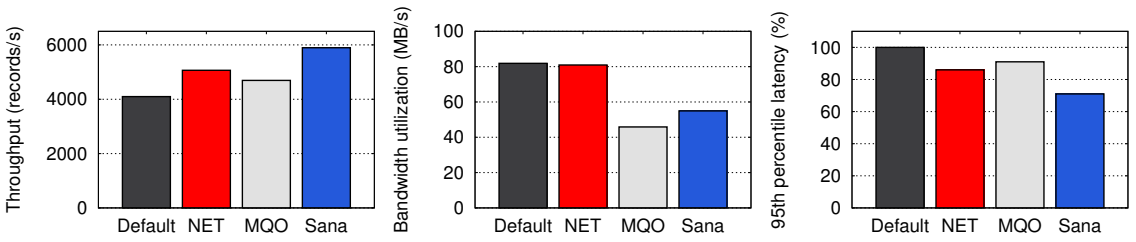
We implemented 12 streaming analytics queries based on actual streaming analytic queries on Twitter streams [**?**, **?**]. Table 3.1 shows the summary of the queries. Each query consisted of various combination of operators including *map*, *reduce*, *filter*, *join*, *union*, and *window*. Each query subscribed to 4-6 input sources and outputs its final result locally at the *sink* operator. Some of the queries also rely on static data sources. For example, in the case of trend analysis, the query discards all the irrelevant words by consulting to an external database. Another example includes a sentiment dictionary used in sentiment analysis. In all of the experiments, each query is submitted independently with a time gap of 10 seconds to mimic the independent deployment of most streaming analytics queries in a practical scenario. Hence, batching multiple queries together prior to their deployment is impractical.

Evaluation Metrics. We use the following metrics to evaluate and compare the performance of the systems:

- *Throughput:* The average rate of *distinct* records/second processed by the system for each query. In the face of bandwidth constraints, the system may trigger a backpressure to reduce the rate of an input stream.

- *WAN Bandwidth Utilization:* The average rate of records (including duplicates) transmitted over the WAN. This is particularly critical in a wide-area environment that typically has limited bandwidth.

- *Latency:* The latency is measured as an *event time* latency, which is the difference between the time when a record is generated at the external data source and when its processed output is written to the final location by the *sink* operator.



(a) Throughput. (b) Bandwidth utilization. (c) Latency w.r.t. Default.

Figure 3.8: Overall system performance comparison.

3.7.1 Baseline System Comparison

We evaluated the benefits of WAN-aware multi-query optimization by comparing the following systems:

- `Default`: The default Flink system that does not allow sharing executions nor does it implement WAN-aware scheduling, but implements node-locality scheduling.
- `NET`: A modified Flink system that adopts the WAN-aware task scheduling algorithm that minimizes query execution time by distributing tasks across sites with sufficient bandwidth. However, it does not allow queries to share common executions. The batch-schedule optimization in Clarinet is not applicable to this context due to the independent deployment of the queries.
- `MQO`: A modified Flink system that allows queries to share common executions. But, it does not implement WAN-aware scheduling (default Flink scheduler).
- `Sana`: Our modified Flink system that incorporates the WAN awareness in both optimizing multiple query executions and scheduling stream operators.

Figure 3.8 compares the overall performance of different systems with the 12 queries running concurrently. We can see from Figure 3.8(a) that `Sana` resulted in 44%, 16%, and 26% higher throughput compared to `Default`, `NET`, and `MQO` respectively. Figure 3.8(b) also shows that `Sana` was able to achieve these performance gains while consuming significantly lower bandwidth compared to both `Default` and `NET` (∼33% less bandwidth utilization). These results indicate that `Sana` could efficiently utilize the WAN bandwidth by preventing transmitting duplicate records over the constrained

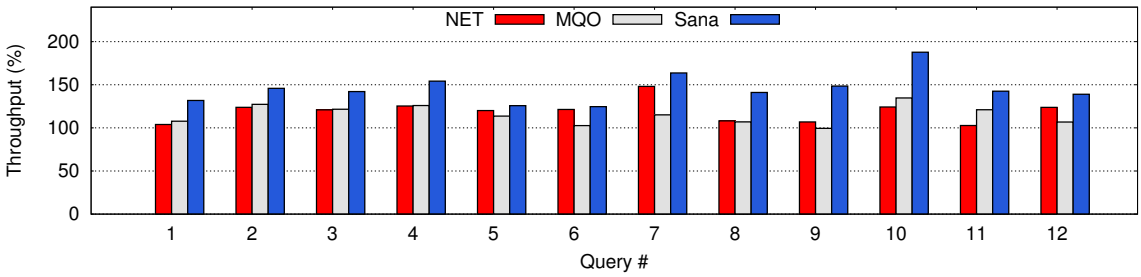


Figure 3.9: Per-query execution throughput.

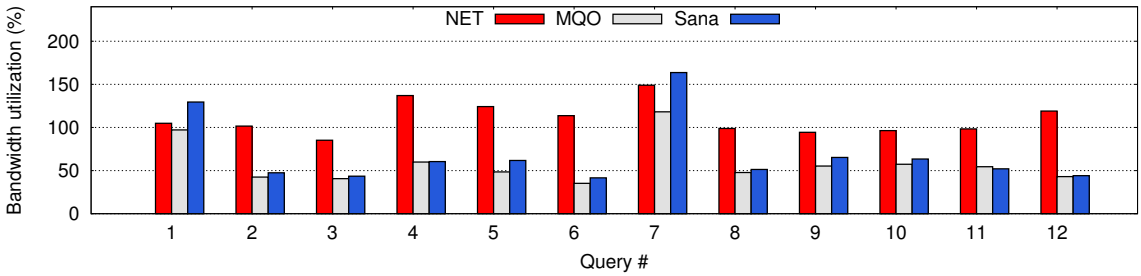


Figure 3.10: Per-query WAN bandwidth consumption.

network links. Although `MQO` consumed less bandwidth with respect to `Sana`, the bandwidth utilized by `Sana` is effectively used to transmit more records. Furthermore, `MQO` resulted in a higher overall query execution latency compared to `Sana` and `NET`, as shown in Figure 3.8(c). This highlights the importance of WAN-aware operator scheduling to *effectively* utilize limited network bandwidth in a wide-area environment. The latency and throughput gains achieved by `MQO` with respect to `Default` is because `MQO` utilized the available bandwidth more efficiently by preventing transmitting redundant data over the WAN while `Default` was not aware of such redundancy.

We further break down the overall performance and WAN bandwidth consumption rate of the queries to observe the gain for each individual query relative to `Default`. We make a few observations. First, we can see from Figure 3.9 that `NET` was able to improve the overall throughput of each query by up to 48% and resulted in 40% lower latency compared to `Default` (see Figure 3.11). However, we can also see from Figure 3.10 that the WAN-aware scheduling in `NET` that tried to minimize query execution latency did not reduce the overall WAN bandwidth consumption even though it resulted in higher throughput. This indicates that `NET` was able to process a higher rate of data streams by avoiding overloaded network links.

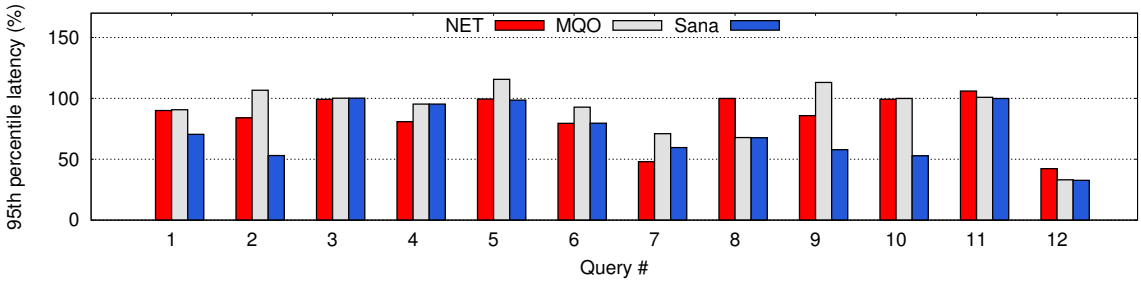


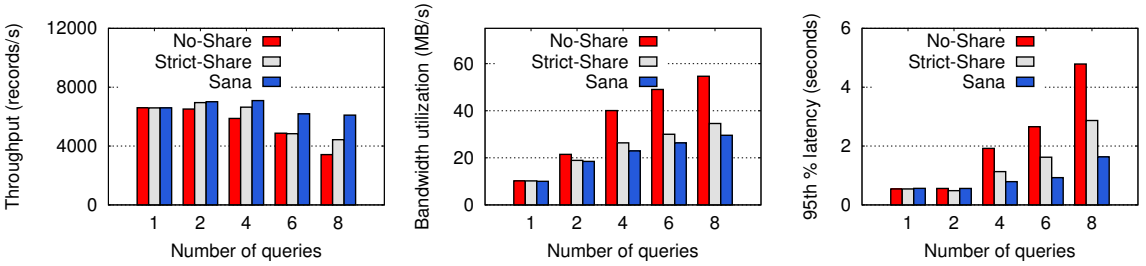
Figure 3.11: Per-query execution latency.

Secondly, we can see from Figure 3.10 that `MQO` was able to significantly reduce the bandwidth utilization by up to 60% by sharing common executions between queries. The only cases where the `MQO` could not reduce the bandwidth utilization were for query 1 and 7 which did not exhibit any commonality with the other queries. However, we can see from Figure 3.9 that query 7 was able to process more data streams with a similar increase. This indicates that the bandwidth was *efficiently* used for transmitting a higher rate of data streams. We can also see from Figure 3.10 and Figure 3.11 that although `NET` consumed higher network bandwidth compared to the `MQO` it was able to outperform `MQO` for most queries in terms of minimizing execution latency. This shows that minimizing WAN bandwidth consumption in a wide-area environment does not necessarily minimize the query execution latency.

Thirdly, we can see that `Sana` improves the overall performance of each query execution while consuming less network bandwidth. It resulted in up to of 87% higher throughput and 68% lower latency compared to `Default`. Similar to the `MQO` case, both query 1 and 7 consumed higher bandwidth, but the extra bandwidth was used for transmitting more data. Furthermore, `Sana` also achieved 21% higher throughput compared to `NET` by eliminating redundant data transmission, as reflected by the reduction in bandwidth utilization for most queries. Lastly, even though `Sana` consumed more bandwidth compared to `MQO`, it resulted in a higher throughput. These results show `Sana` can utilize WAN bandwidth effectively and efficiently.

3.7.2 Impact of Degree of Sharing

In the next set of experiments, we explore the impact of degree of sharing in applying multi-query optimization. Specifically, the benefit of allowing queries to partially share



(a) Throughput. (b) Bandwidth utilization. (c) 95th % latency.

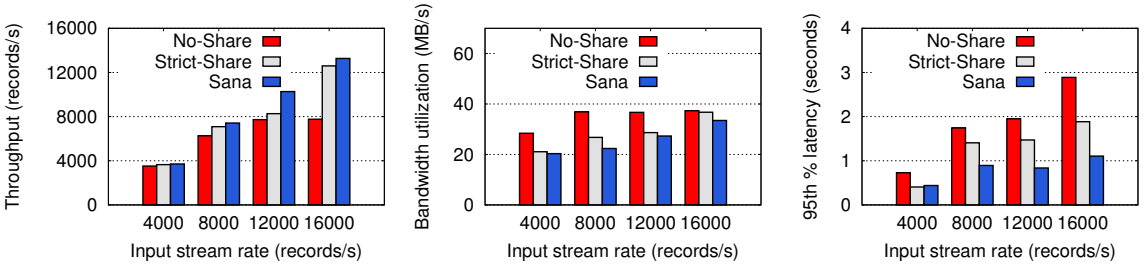
Figure 3.12: Degree of sharing impact over different number of concurrent queries.

common input streams even though their operators are different. All systems in the following experiments applied WAN-aware operator scheduling. Thus, the differences in the results were strictly based on the different execution plans generated by the *Query Optimizer*. We compared `Sana` (which allows `IN` and `IN-OP`) against (1) `No-Share`, which did not consider execution sharing, and (2) `Strict-Share` whose *Query Optimizer* only allowed queries to share vertices if they shared the same inputs and operators (`IN-OP` only). In contrast to `Strict-Share`, `Sana` allowed queries to share partial input streams.

Varying Number of Concurrent Queries

Figure 3.12 compares the three *Query Optimizers* over varying number of concurrent queries. In the case of a single query execution, all the *Query Optimizers* generated the same execution plan. However, as the number of queries increased `Sana` was able to exploit a higher degree of sharing by allowing queries to partially share their executions. This resulted in a lower bandwidth consumption and approximately 78% and 37% higher throughput execution compared to `No-Share` and `Strict-Share` respectively (see Figures 3.12(b) and 3.12(a)). We can see from Figure 3.12(c) that allowing partial sharing can also reduce the execution latency due to the higher bandwidth availability, which provides a higher flexibility to the *Job Scheduler* to deploy the queries.

Figure 3.12(b) shows that although `No-Share` consumed 41% and 65% higher bandwidth compared to `Strict-Share` and `Sana` respectively, it resulted in an overall lower throughput. This indicated there was a large amount of redundant data being transmitted over the WAN. The `Strict-Share` also consumed slightly more bandwidth compared



(a) Throughput. (b) Bandwidth utilization. (c) 95th % latency.

Figure 3.13: Degree of sharing impact over different input stream rate.

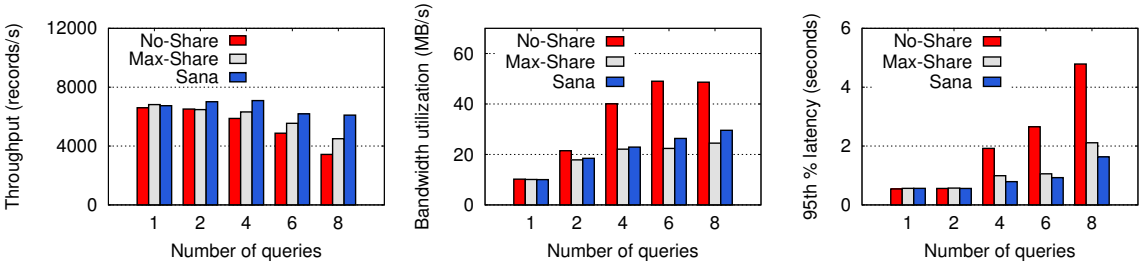
to `Sana` but resulted in a lower throughput, which highlights the importance of (partially) sharing common input streams even for different operators. From Figure 3.12(c), we can also see that the overhead of the *router* operators that were added to route input streams from remote sites was negligible (∼5%) even when they were not utilized, as shown in the case with 1 query execution. Thus, in general the *router* operator can reduce the redundancy in transmitting duplicate data over the WAN.

Varying Input Stream Rates

In this experiment, we evaluate the impact of the degree of sharing over different input stream rates with 4 concurrent queries. Figure 3.13(a) shows that, as the input rate increased, `Sana` resulted in a higher throughput while consuming lower bandwidth compared to both `No-Share` and `Strict-Share` (Figure 3.13(b)). Furthermore, `Sana` was able to reduce the overall execution latency compared to `No-Share` and `Strict-Share`, similar to the effect of increasing the number of queries (Figure 3.13(c)). This shows that (1) the proposed WAN-aware multi-query optimization scales as workload increases, and (2) allowing queries to share common inputs even if they have different operators can improve the overall performance and reduce wasteful bandwidth consumption.

3.7.3 WAN-Aware Sharing: Bandwidth Utilization vs. Performance

In the following experiments, we show the importance of WAN awareness in applying multi-query optimization in a wide-area environment to maintain high-performance executions while reducing WAN bandwidth consumption (Section 3.5.1). We compared `Sana` against (1) `No-Share` which did not exploit any execution sharing and



(a) Throughput. (b) Bandwidth utilization. (c) 95th % latency.

Figure 3.14: WAN-aware planning: bandwidth utilization vs. performance trade-off.

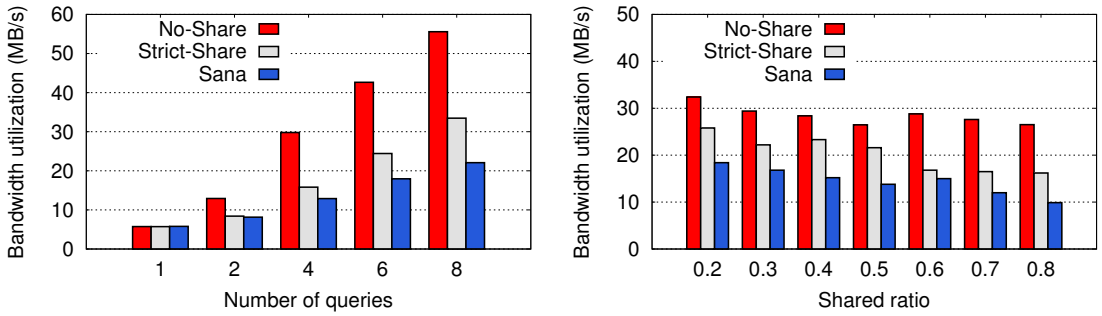
(2) `Max-Share` which allowed queries to share common executions but did not consider the WAN bandwidth availability in sharing executions. In contrast to `Sana`, the latter always tried to exploit any sharing opportunity, which is essentially the traditional multi-query optimization for a local environment. The main problem with maximizing sharing without WAN awareness is that it may result in WAN bandwidth contention among queries, which can degrade the performance of either or both the sharing and the shared executions.

Figure 3.14(a) and Figure 3.14(c) show that `Sana` resulted in 35% higher throughput and 23% lower latency compared to `Max-Share`, but consuming slightly higher bandwidth. The performance gain achieved by `Sana` compared to `Max-Share` is because `Sana`'s *Query Optimizer* prevented exploiting sharing opportunities that led to a bandwidth contention which would degrade the overall performance. We can also see that as the number of queries increases, the performance gap between `Sana` and `Max-Share` also increases. This indicates that the WAN awareness in `Sana` resulted in less number of contended links. Thus, there is a trade-off between minimizing WAN bandwidth utilization and maximizing the overall performance of multiple query executions.

3.7.4 Potential Bandwidth Saving

In the following experiments, we observe the potential bandwidth saving from applying multi-query optimization in the case where network bandwidth is not constrained. We deployed `Sana` on a localized CloudLab[3] environment where the available bandwidth between nodes are higher than the rate of the data streams. In such a condition where

[3] https://www.cloudlab.us/



(a) Bandwidth util. over number of queries.

(b) Bandwidth util. over shared-ratio.

Figure 3.15: Saving WAN bandwidth consumption.

bandwidth is sufficient, reducing data transfer over network is still desirable since WAN bandwidth is expensive in terms of monetary cost [**?**].

Figure 3.15(a) and Figure 3.15(b) show the average bandwidth consumption rate over different number of concurrent queries and sharing ratio respectively. The sharing ratio is defined as the percentage of vertices that are shared between queries. The average sharing ratio between the queries in Figure 3.15(a) was approximately 0.2 whereas the number of concurrent queries in Figure 3.15(b) was set to 4. We can see from both figures that `Sana` greatly reduced the bandwidth consumption as the number of queries and the sharing ratio increased. Specifically, it resulted in up to 60% reduction in bandwidth consumption rate compared to the sharing-agnostic approach. Thus applying multi-query optimization even in an unconstrained wide-area environment can still reduce the bandwidth utilization and save monetary cost.

3.8 Related Work

Geo-Distributed Data Analytics Systems. Table 3.2 shows where `Sana` stands in the world of geo-distributed data analytic systems. Iridium [**?**] proposes a WAN-aware optimization that minimizes query execution latency for batch-oriented workloads by proactively migrating input data prior to the arrivals of queries based on history. Geode [**?**] also relies on recurring queries but focuses on minimizing WAN bandwidth consumption by sending only the *diff* of input data over the wide-area network for subsequent queries. In contrast to both approaches, `Sana` focuses on stream-oriented

Table 3.2: Geo-distributed data analytics systems

Systems	Workload	WAN-aware optimization	Multi-query optimization
Iridium [**?**]	Recurring	Data and task placements prior to query arrivals	N/A
Geode [**?**]	Recurring	*diff* or incremental data transfer over the WAN	N/A
Clarinet [**?**]	Batch	WAN-aware query planning	Multi-job scheduling
Tetrium [**?**]	Batch	Task scheduling over heterogeneous resources	Multi-job scheduling
JetStream [**?**]	Stream	Data aggregation/degradation using data cube	N/A
AWStream [**?**]	Stream	Profiling-based data degradation	N/A
Sana	Stream	WAN-aware operator sharing and scheduling	Execution sharing

workloads where most queries are long-running and consume data streams that are continuously being generated in real time. Furthermore, `Sana` does not make any assumption on the query arrivals. None of these techniques support multi-query optimization.

Both Clarinet [**?**] and Tetrium [**?**] look at optimizing batch-oriented queries in a wide-area environment. Specifically, Clarinet incorporates WAN awareness into the query optimizer to choose a query execution plan based on inter-site bandwidth availability, whereas Tetrium additionally considers the heterogeneity of computational resources across sites in scheduling jobs. In addition to incorporating WAN-aware optimization for single-query deployment, both of them consider optimizing multiple query executions by batch-scheduling multiple queries rather than scheduling each query independently. Their approaches, however, are not feasible for stream-oriented workloads. Furthermore, they do not allow queries to share common executions and hence their techniques would still result in a redundant data transmission and processing. In contrast, `Sana` can eliminate redundant executions and optimize multiple query executions in an incremental manner, which is critical for long-running, continuous queries.

Recent attempts have also considered optimizing stream-oriented queries in a wide-area environment. Photon [**?**] and Ubiq [**?**] address the fault tolerant aspect of geo-distributed data analytics over continuous data streams. JetStream [**?**] handles WAN bandwidth limitation by making a trade-off between output quality and performance, which may not be applicable for queries that rely on exact results. AWStream [**?**] further automates the degradation policies in JetStream based on the resource-accuracy profiles. Heintz et al. [**?**] propose an online algorithm that trades timeliness and accuracy in the context of windowed grouped aggregation. Pietzuch et al. [**?**] examine the problem of operator placement on the open Internet environment. Although they related to `Sana`, they mainly focus on optimizing individual query independently.

Others have also looked at optimizing different types of workloads in a wide-area environment. Gaia [?] proposes a system that optimizes machine learning workloads in a wide-area environment by identifying and eliminating any insignificant update over the WAN. Monarch [?] focuses on geo-distributed graph analytics workloads by optimizing existing graph-processing model to a wide-area environment.

Multi-Query Optimization. The problem of multi-query optimization has been extensively studied in databases [?, ?] and have been adopted for OLAP workloads [?, ?, ?, ?, ?] and later, data analytics [?, ?, ?]. `Sana` adopts the *data-centric* philosophy with pipelining technique [?] to share common executions between streaming queries. Although most of them are related to our work, they focus on a local environment whereas `Sana` focuses on a wide-area environment with different bottleneck. We show that applying traditional multi-query optimization in wide-area settings without WAN awareness may lead to performance degradation.

Others have also examined the problem of multi-query optimization over continuous data streams in streaming databases [?, ?, ?]. Seshadri et al. [?] propose an algorithm to find an optimal execution plan with reduced search space. Rule-based [?] and sketch-based [?] optimization have also been proposed for multiple queries over data streams, and NiagaraCQ [?] addresses the scalability issue in applying multi-query optimization for Internet Databases. Although our work is related, they are mainly concerned with memory constraints since they focus on a single-server deployment.

Incremental Processing and Caching Systems. It is worth mentioning that our work shares similarity with other work in incremental processing [?, ?] and caching systems [?, ?, ?, ?] since they also address the problem of redundant computation. However, they are orthogonal to our work. The incremental processing technique can be applied by each individual query by updating its state incrementally instead of computing from the beginning [?]. However, this is application-specific. Caching intermediate *hot* data also prevents performing redundant data processing, but it may not be applicable for queries that rely on real-time data streams. Thus, these techniques can be used in conjunction with our techniques.

3.9 Conclusion

In this chapter, we present `Sana`, a streaming analytics system that optimizes multiple query executions in a wide-area environment. We demonstrate the opportunity of applying multi-query optimization in the context of wide-area streaming analytics to mitigate wasteful resource utilization in processing redundant operations and transmitting duplicate data over scarce wide-area network bandwidth. We study different types of sharing opportunities and propose a multi-query optimization that allows multiple queries to *incrementally* share common executions in an *online* manner. We also address the importance of *network awareness* in applying multi-query optimization in a wide-area environment. We show that traditional WAN-agnostic multi-query optimization may lead to performance degradation. The evaluation using a wide-area system deployment across multiple geo-distributed EC2 data centers shows that `Sana` resulted in 21% higher throughput while saving WAN bandwidth utilization by 33% compared to a WAN-aware, sharing-agnostic system.

Chapter 4

WASP: Wide-area Adaptive Stream Processing

4.1 Introduction

Wide-area stream processing systems typically comprise multiple *edge clusters* and data centers connected by a wide-area network (WAN) [?, ?, ?, ?]. Such systems require low-latency and high-throughput processing to extract timely insights from geo-distributed data streams. However, ensuring a stable and high-performance execution of long-running queries in a wide-area environment is challenging due to the highly dynamic WAN bandwidth and unpredictable workload patterns. Studies have shown that WAN bandwidth may change at an interval of minutes [?, ?], while Internet workload exhibits strong variability, both temporally and spatially [?, ?]. Furthermore, stragglers and failures are inevitable in large-scale distributed systems [?, ?, ?].

Existing work has addressed the importance of adaptability in distributed stream processing systems, but has focused on addressing only computational bottlenecks in a centralized cluster environment and hence, is WAN-agnostic [?, ?, ?, ?, ?]. As argued by recent work, wide-area data analytics requires WAN-aware optimization due to the fundamental differences in the environments [?, ?, ?]. This can significantly improve the query execution time and/or reduce wasteful resource consumption by several orders of magnitude. Early work in wide-area data analytics has focused on short-lived *batch* processing and assumed that network bandwidth is relatively stable throughout the

runtime of queries [?, ?, ?], which is an invalid assumption for long-running streaming queries. Others have addressed the importance of adaptability in wide-area streaming analytics but force users to trade quality/accuracy for performance through aggregation, degradation, and statistical estimation [?, ?, ?]. We argue that these approaches are application-dependent, and may not apply generally. For example, reducing a frame rate may be applicable for some video analytics applications, but dropping data is not tolerable for queries that require high accuracy such as fraud detection and billing queries [?, ?]. Furthermore, they rely heavily on analysts' expertise to explicitly specify various degradation policies and involve extensive parameter tuning, making it cumbersome in practice.

In this work, we rethink the adaptability property of wide-area stream processing systems. Our goal is to allow queries to maintain low-latency execution while preserving the quality of the results. Instead of immediately degrading data in the face of bottlenecks, we propose a re-optimization adaptation approach that adapts the execution of a query, and only consider data degradation as a last resort. There are several challenges in re-optimizing a query execution in a wide-area environment. First, the system needs to account for the limited and heterogeneous WAN bandwidth. Secondly, wide-area dynamics may happen frequently and hence, any adaptation should be done with low overhead. This is challenging in the case of *stateful* computation where the internal query processing state may be geo-distributed. Thirdly, it is critical to ensure a stable execution without over-allocating resources to avoid wasteful monetary cost.

To address the above challenges, we propose a resource-aware adaptation framework called `WASP` (**W**ide-area **A**daptive **S**tream **P**rocessing). `WASP` adapts queries at runtime through a combination of multiple techniques: (1) task re-assignment, (2) operator scaling, and (3) query re-planning. Both task re-assignment and operator scaling adapt the physical execution of a query, while query re-planning adapts the logical plan. We show that a combination of these techniques are generally applicable for different type of queries. `WASP` can automatically determine *how* to adapt a query depending on the type of dynamics. For example, `WASP` *scales up* a compute-constrained operator by allocating additional resources *within* a site. On the other hand, it handles network bottlenecks differently by *scaling out* the bottleneck operator *across* sites and distributing the workload across multiple network links. Furthermore, `WASP` can quickly recover from failures

and adjust any misconfiguration.

We further study the applicability, overhead, and benefits of different adaptation techniques, and propose a practical approach to determine *which* adaptation action to take based on the types of queries (stateless vs. stateful), bottlenecks (compute vs. network), and optimization objectives. To ensure low-overhead adaptation, we propose a network-aware state migration and a state partitioning technique when adapting queries with stateful operators. We have implemented a `WASP` prototype on Apache Flink [**?**]. Experimental evaluation using the YSB benchmark [**?**] and a real Twitter trace demonstrates that `WASP` can maintain low-latency execution without compromising quality and with low overhead in the face of dynamics.

We summarize our contributions as follows:

- We propose several optimization-based adaptation techniques for wide-area streaming analytics to handle various dynamics without sacrificing the accuracy/quality of the results (Section 4.5).
- We qualitatively compare the applicability, overhead, and benefits between different adaptation techniques, and propose a policy to determine *which* adaptation to use depending on the types of queries, dynamics, and goals (Section 4.6).
- We further highlight the importance of network awareness to ensure an effective and low-overhead adaptation (Section 4.5 and Section 4.6).
- We have implemented our adaptability techniques into a system prototype called `WASP` over Apache Flink, and demonstrated that `WASP` can achieve low-latency execution without sacrificing any data (Section 4.9).

4.2 Background & Motivation

4.2.1 Wide-area Streaming Systems

We consider a wide-area streaming system that spans multiple sites (edge clusters/data centers) that are connected by WAN with diverse inbound and outbound bandwidth and latency [**?**, **?**, **?**, **?**]. A global *Job Manager* running in one of the sites provides

an interface for query submission, and it optimizes and deploys queries across multiple sites. The inputs of a query can be generated or collected at any site.

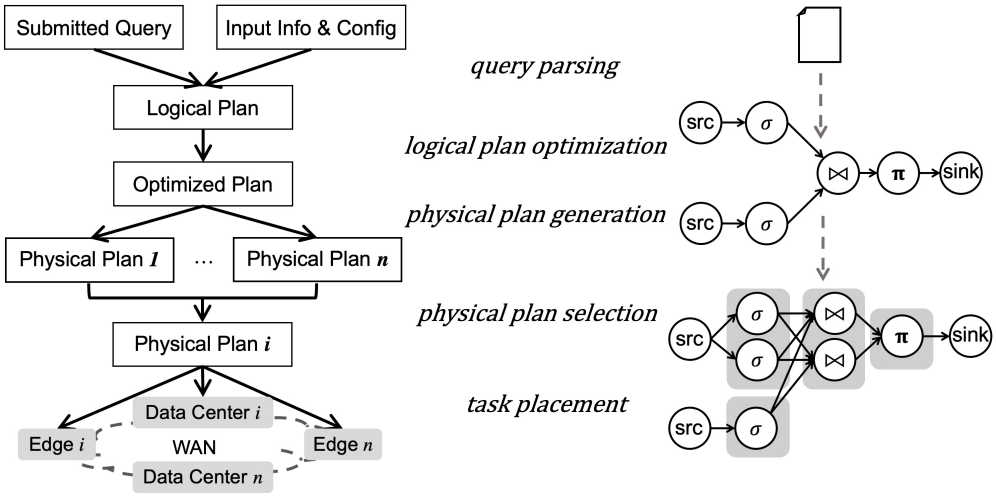


Figure 4.1: Wide-area query execution pipeline.

We consider a streaming dataflow model [?, ?, ?]. Figure 4.1 shows the execution pipeline of a typical streaming analytics query. A query is represented as a logical directed acyclic graph of operators $G = (V, E)$, where the vertices V correspond to stream operators and the edges E refer to data flows between operators. The *source* and *sink* vertices refer to operators that consume input streams and produce final results from/to external systems respectively (e.g., display on a monitoring dashboard or store the results to a file system and database).

Each query's logical plan is optimized (e.g., pushing filter operators upstream to reduce data rates) and translated into multiple physical plan candidates. A query's physical plan consists of one or more *execution stages* (jobs), each of which can run in parallel as *execution instances* (tasks). The number of instances of each stage is typically predetermined by the *parallelism* value in the configuration. The system will deploy the tasks with WAN awareness to minimize query execution latency or WAN bandwidth consumption [?, ?, ?, ?, ?]. A task continuously waits for input streams from one or more upstream operators (except for *source* operators), processes them, and outputs the results to one or more downstream operators (except for *sink* operators).

4.2.2 Wide-area Resource Constraints

Scarce and heterogeneous resources. Extracting real-time insights from large continuous data streams in wide-area settings is challenging due to the highly heterogeneous and scarce WAN bandwidth [**?**, **?**, **?**, **?**]. The emergence of *Edge Computing* comprising small edge clusters further introduces additional heterogeneity [**?**, **?**]. They typically have limited computational resources and they are connected using the public Internet, whose bandwidth is even more constrained, with an average of 7∼10Mbps [**?**].

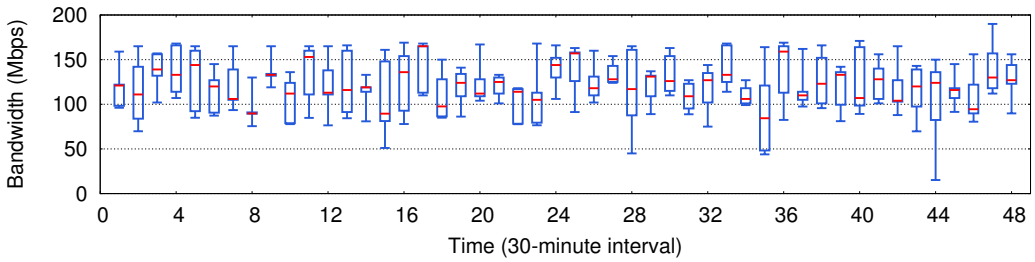


Figure 4.2: WAN bandwidth variability from Oregon to Ohio EC2 data centers.

Resource and workload dynamics. Most of the work in adaptive stream processing systems has focused on a cluster environment where the main source of bottlenecks is due to the limited computational resources. They typically handle this problem by scaling out bottleneck operators *within a cluster* [**?**, **?**, **?**, **?**, **?**]. In wide-area settings, network bandwidth imposes additional dynamic [**?**, **?**, **?**]. We conducted a one-day measurement of WAN bandwidth variation between 8 Amazon EC2 data centers (Oregon, Ohio, Ireland, Frankfurt, Seoul, Singapore, Mumbai, and Sao Paulo). Figure 4.2 shows the bandwidth variation between the Oregon and Ohio data centers. We can see that the bandwidth has a high variation (25% to 93% deviation from the mean). Others have also reported that the inter-data center network topology may change every 5-10 minutes [**?**, **?**], supporting the dynamic nature of WAN bandwidth.

Studies have also reported that many Internet applications have variable workload patterns, both temporally and spatially [**?**, **?**]. For example, Twitter workload exhibits strong spatial and temporal variations, with day hours having 2× higher workload compared to night hours [**?**]. Thus, relying on a static deployment is a poor fit in such a dynamic environment. This leads to performance degradation and wasteful resource utilization during high and low workload periods respectively.

4.3 WASP Overview

In this section, we present the system overview of WASP. Figure 4.3 shows the WASP system architecture. It consists of a *Job Manager* and multiple geo-distributed *Task Managers*. The *Job Manager* consists of a *Query Planner* and a *Scheduler* that are responsible for planning queries and deploying tasks respectively. We define the computational resources provided by each *Task Manager* using a *computing slot* abstraction, each of which can handle exactly one task. Each *Task Manager* continuously monitors and gathers its task's performance metrics (e.g., processing latency and input/output stream rates) through a *Local Metric Monitor* and reports them to the *Global Metric Monitor* ①. The *Global Metric Monitor* uses this information to diagnose unhealthy execution or identify wasteful resource consumption ② and asks the *Reconfiguration Manager* to resolve it ③.

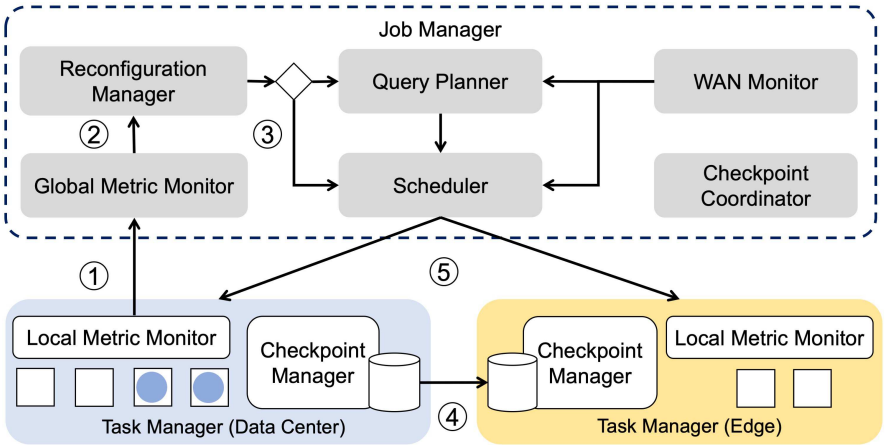


Figure 4.3: System overview of WASP.

Modern distributed stream processing systems support *stateful* computation, where each task tracks its processing progress as a *state* and periodically checkpoints it to a persistent storage system (e.g., HDFS [**?**]) [**?**, **?**, **?**]. This allows a task to start/resume its execution from the last checkpointed state when recovering from failure/adaptation. Examples of state include intermediate aggregation results and topic-partition offsets in Kafka [**?**]. Since tasks in wide-area streaming analytics are geo-distributed, their states are naturally generated in a geo-distributed fashion. To reduce the overhead

Table 4.1: Descriptions of the used notations

Notation	Description
G	operator graph
V	a set of stream operator
E	a set of logical data flow between operators
m	the total number of sites in the system (edge cluster/data center)
$N[s]$	the total number of computing slots at site s
$A[s]$	the total number of available computing slots at site s
B_{s1}^{s2}	available network bandwidth from site $s1$ to $s2$
ℓ_{s1}^{s2}	network latency from site $s1$ to $s2$
β	maximum bandwidth utilization threshold
p	the parallelism/number of instances of an operator
$p[s]$	the number of operator instances deployed at site s
U	a set of upstream/predecessor an operators
D	a set of downstream/successor an operators
λ_I	the observed input rate of an operator
λ_P	the observed processing rate of an operator
λ_O	the observed output rate of an operator
$\hat{\lambda}_I$	the actual/expected input rate of an operator
$\hat{\lambda}_I$	the actual/expected processing rate of an operator
$\hat{\lambda}_O$	the actual/expected output rate of an operator
σ	the selectivity value of an operator
t_{adapt}	the time required to adapt an operator

of checkpointing large state over the WAN, `WASP` stores each state locally instead of in a centralized location. When a task is migrated to a different site, the *Checkpoint Coordinator* will first initiate a state migration ④ and only after the state transfer completes, the task can resume its execution ⑤.

4.4 Query Execution Model & Monitoring

We now look at how `WASP` models and monitors the runtime execution of a query to identify bottlenecks. Table 4.1 summarizes the notations that we use in this chapter.

Runtime Monitoring

Each operator keeps track of its runtime execution metrics such as processing rate (λ_P), output rate (λ_O), and the *selectivity* of the operator (σ) which is defined as the ratio between the output rate and the processing rate. These metrics are periodically

reported to the *Global Metric Monitor* for diagnosis. Here, the execution metric of an operator is computed based on the aggregate runtime information of all of its execution instances/tasks over the past time interval.

$$\lambda_P = \sum_{i=1}^{p} \lambda_P[i] \qquad \lambda_O = \sum_{i=1}^{p} \lambda_O[i] \qquad \sigma = \frac{\lambda_O}{\lambda_P}$$

We consider an operator execution to be *healthy*, i.e., unconstrained by the allocated resources, if the two following conditions are satisfied:

1. The processing rate of an operator is equal to its input rate: $\lambda_P = \lambda_I$.
2. The input rate of an operator is approximately equal to the aggregated output rates of its upstream operators U: $\lambda_I \approx \sum_{u \in U} \lambda_O[u]$.

The first condition ensures that all tasks have sufficient computing power to process their input streams, while the second condition ensures no network congestion in transmitting data streams from its upstream operators. However, these conditions may not always hold due to the dynamic nature of the actual workload and WAN bandwidth.

If $\lambda_P < \lambda_I$, this indicates that there is insufficient computing resources allocated to the operator. This may happen due to the occurrence of stragglers or unpredictable workload variation that is common in practice [**?**]. On the other hand, the second condition may fail ($\lambda_I < \sum_{u \in U} \lambda_O[u]$) if the available network bandwidth between an operator and its upstream operators is constrained or congested. This may happen due to an increasing workload of the operator itself and/or the reduction of network bandwidth availability caused by a change in the underlying network topology or bandwidth contention with other executions. If either or both of these conditions fail, the system needs to adapt the operator execution to maintain the low-latency execution performance of the query.

Estimating the Actual Workload

To identify bottlenecks, modern distributed stream processing systems often use a *back-pressure* mechanism where a bottleneck operator triggers a control-rate message to its upstream operators to reduce the workload [**?**, **?**]. In this case, the observed input and

output rates of an operator do not reflect the actual workload, i.e., the actual stream rates from the *source* operators. Yet, to accurately determine the effective adaptation action that can resolve the bottleneck, the system should rely on the actual workload instead of the observed information. Thus, we estimate the expected input and output rates of each operator based on the actual workload generated by the *source* operators ($\lambda_O[src]$), which is computed recursively as follows:

$$\hat{\lambda}_P = \hat{\lambda}_I = \begin{cases} \sum_{u \in U} \hat{\lambda}_O[u], & \text{if } U \neq \emptyset \\ \lambda_O[src], & \text{otherwise} \end{cases} \qquad \hat{\lambda}_O = \sigma \cdot \hat{\lambda}_I$$

4.5 Optimization-Based Adaptation

Having discussed how we model the runtime execution of an operator and identify performance bottlenecks, we now propose 3 different adaptation techniques to resolve wide-area bottlenecks: task re-assignment (Section 4.5.1), operator scaling (Section 4.5.2), and query re-planning (Section 4.5.3). We compare the 3 techniques and discuss `WASP`'s adaptation policies later in Section 4.6.

4.5.1 Task Re-Assignment

Existing work has addressed the importance of WAN awareness in scheduling tasks in wide area settings with the goal of minimizing query execution latency or WAN bandwidth consumption [**?**, **?**, **?**]. However, they have mainly focused on optimizing the initial task placement and do not consider re-evaluating it after the deployment. In this case, the initial placement may become sub-optimal when the environment or the workload has changed significantly. Note that, re-assigning tasks of a particular stage does not affect the other stages nor change the query's logical plan.

Resource-aware Task Placement

Most wide-area streaming analytics system incorporate WAN awareness in their task placement algorithms based on the deployment of the predecessor (upstream) stages since they typically schedule *one-stage-at-a-time* in a topological order [**?**, **?**, **?**, **?**]. However, re-assigning the tasks of an already running stage without considering the

deployment of its successor (downstream) operators may result in a sub-optimal deployment because the task deployment of its successor stages rely heavily on the original task placement of the operator. This may result in a cascading problem. To prevent this issue, our task re-assignment algorithm considers the deployments of *both* the upstream and downstream stages. Specifically, we compute the number of tasks to deploy in each site ($p[s]$) by solving the following Integer Linear Program (ILP):

$$min \quad \sum_{s=1}^{m} p[s] \cdot (\ell_u^s + \ell_s^d), \forall u, \forall d \tag{4.1}$$

$$s.t. \quad \frac{p[s]}{p} \cdot \hat{\lambda}_{s \neq u}^{in} < \beta B_u^s, \forall s, \forall u \tag{4.2}$$

$$\frac{p[s]}{p} \cdot \hat{\lambda}_{s \neq d}^{out} < \beta B_s^d, \forall s, \forall d \tag{4.3}$$

$$0 \leq p[s] \leq A[s], \qquad \forall s \tag{4.4}$$

$$\sum_{s=1}^{m} p[s] = p, \qquad p \geq 1 \tag{4.5}$$

Our goal is to minimize the network delay of transmitting data streams both from the upstream (u) and to the downstream (d) operators, which equivalently minimizes the average delay incurred by a particular stage. Constraints 4.2 and 4.3 ensure there is sufficient network bandwidth to receive/send data streams from/to the upstream/downstream operators. Constraint 4.4 ensures there are sufficient number of available slots in each site, and Constraint 4.5 ensures that the system deploys all the tasks.

We include a maximum bandwidth utilization threshold, $\{\beta \mid 0 < \beta < 1\}$, in Constraints 4.2 and 4.3 for a few reasons. First, it provides a certain degree of stability in the solution by providing bandwidth *headroom* to handle slight workload and bandwidth variations. Secondly, the headroom makes the system more robust to mis-estimation in measuring the actual inter-site bandwidth availability and data stream rates. Lastly, the reserved bandwidth can be used to process events that are queued during the transitioning process when an execution is adapted. We set $\beta = 0.8$ in our deployment.

Network-aware Task Migration

If the system is able to find an alternative task placement, it may re-assign some of the existing tasks. Those that can be run at the original sites do not need to be

migrated. For example, if the original placement is $S = \{s_1, s_2, s_3, s_4\}$ and the new placement is $S' = \{s_3, s_4, s_5, s_6\}$, only $(S - S') = \{s_1, s_2\}$ need to be migrated to $(S' - S) = \{s_5, s_6\}$. When re-assigning tasks, the system will (1) temporarily halt the execution, (2) instantiate new tasks at the new sites and terminate the old ones, and (3) resume the execution. In the case of stateful operation (e.g., windowed grouped aggregation and join), the system needs to re-distribute/migrate any computation state before resuming the execution. Since the size of a state can be large in practice [**?**, **?**], migrating a state over a low-bandwidth network link may incur high state migration time, making it impractical for frequent dynamics. As the major overhead of migrating a task is determined by the slowest state migration time, we determine the mapping from $(S - S')$ to $(S' - S)$ by solving a *MinMax* problem with the goal of minimizing the slowest state migration: $min\ max(\frac{|state_x|}{B_x^y}), \forall x \in (S - S'), \forall y \in (S' - S)$. We show in Section 4.9.5 that a network-aware state migration can significantly mitigate the overhead of adapting a stateful operator execution.

4.5.2 Operator Scaling

Although task re-assignment is generally applicable for any type of operators, the algorithm may not always be able to find a solution due to its constraint on the initial operator parallelism (Constraint 4.5). Yet, determining the *right* optimal parallelism in advance may not be feasible given the highly dynamic environment. Thus, we consider (1) increasing the parallelism if an execution is constrained by the available resources, and (2) decreasing the parallelism to reduce any wasteful resource consumption.

Scale Up/Out

We define *scale up* and *scale out* in wide-area settings as instantiating new operator instances *within* a site and *across* sites respectively. In general, increasing parallelism can handle computational bottlenecks since it reduces the work performed by each individual task. However, scale up cannot resolve network bottlenecks while scale out can solve this by distributing the workload of any overloaded network link across multiple links.

Figure 4.4 shows how scale up and scale out can handle computational and network bottleneck respectively. When the system observes that a task's processing rate is less

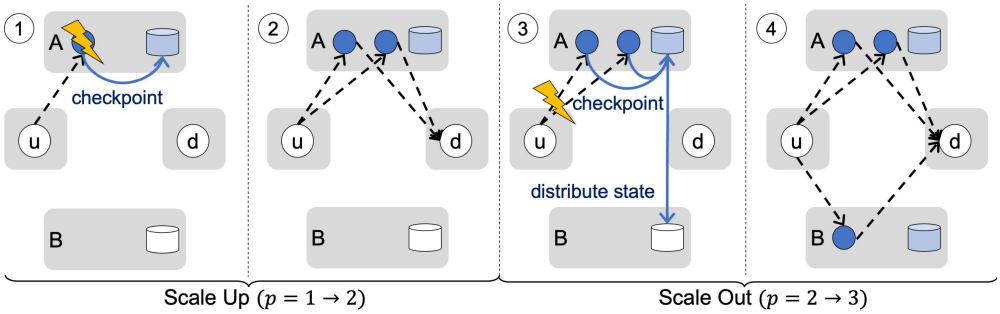


Figure 4.4: Scaling up/out operator within and across sites.

than its expected input rate ①, it may allocate additional slots and launch more instances. To prevent distributing large state over the WAN, the system will prioritize launching the new tasks within the same site ②. ③ shows a scenario where the bandwidth from an upstream task u to Site A is constrained. To reduce the load of the constrained link, the system can instantiate a new task at Site B and distribute the load across 2 links: $u \rightarrow A$ and $u \rightarrow B$ ④. If the operator is stateful, this requires distributing the state across the two sites. Although the example only shows inbound bandwidth contention, scale out can also handle outbound bandwidth contention.

When scaling up/out an operator, the system needs to determine the *scale factor*, i.e., the increase in parallelism. We compute the scale factor based on the operator's execution model proposed in Section 4.4. Specifically, we compute the new parallelism of a bottleneck operator p' based on the ratio between the actual/expected input rate and the operator's processing rate. This is similar to the technique proposed by DS2 in handling computational bottleneck in a cluster-based stream processing system [**?**]:

$$p' = \left\lceil \frac{\hat{\lambda}_I}{\lambda_P} \cdot p \right\rceil$$

This equation gives the *minimum* parallelism value that can effectively resolve the bottleneck. Once the system has computed the new parallelism, it will determine the placement of the tasks by solving Equation 4.1. In the case of scale out, it is computed as the ratio between the stream rate that cannot be handled over the bandwidth availability. In general, increasing the parallelism can handle network bottleneck by distributing the data stream over multiple network links, i.e., $\frac{\lambda}{p'} < \frac{\lambda}{p}$ as $p' > p$, and hence, this can reduce the workload that needs to be transmitted to each network link.

Scale Down

A system may over-allocate resources to a particular job due to several reasons: misconfiguration, pessimistically reserving extra resources to handle peak workload, or as a result of scaling out/up an operator. This results in wasteful resource utilization. If the system identifies such a problem, it should scale down some of the underutilized tasks. To determine *which* tasks to scale down, we prioritize scaling down tasks that are not co-located with their upstream/downstream tasks to reduce the inter-site bandwidth consumption. However, the system needs to ensure the bandwidth to/from any of the sites is higher than the input/output rate after the scaling.

Determining the scale down factor is challenging since it will increase the workload to the remaining tasks. Furthermore it is hard to predict the future availability of the workload. Aggressively scaling down an operator may result in a workload spike if the workload increases after the scale down. To ensure a stable execution, we gradually reduce the parallelism by 1 per iteration. In every iteration, the system needs to ensure that any of the remaining tasks is not constrained, i.e., every task should have sufficient bandwidth and processing capacity to consume the additional workload (relayed data streams) from the terminated tasks. The system will observe its stability and may further scale down the operator in the subsequent iteration as needed.

4.5.3 Query Re-Planning

While task re-assignment and operator scaling focus on adapting the physical deployment of a query, we further consider adapting the logical plan itself. Consider an example in Figure 4.5 which shows 2 different plans for the same query. It consumes input streams from 4 sources that are located at: A, B, C, and D, and joins them using a *full hash join*, which is commutative. A WAN-aware *Query Planner* may choose the first plan if the bandwidth is sufficient since it consumes less bandwidth (70MB/s for the first plan, and 90MB/s for the second plan). However, if the bandwidth between Site C and Site A is constrained, the *Query Planner* may opt for the second plan. This illustration shows that the optimal plan depends heavily on the runtime information *when* the query is deployed [**?**]. Thus, the *Query Planner* may consider adapting the query plan when the environment has changed significantly [**?**].

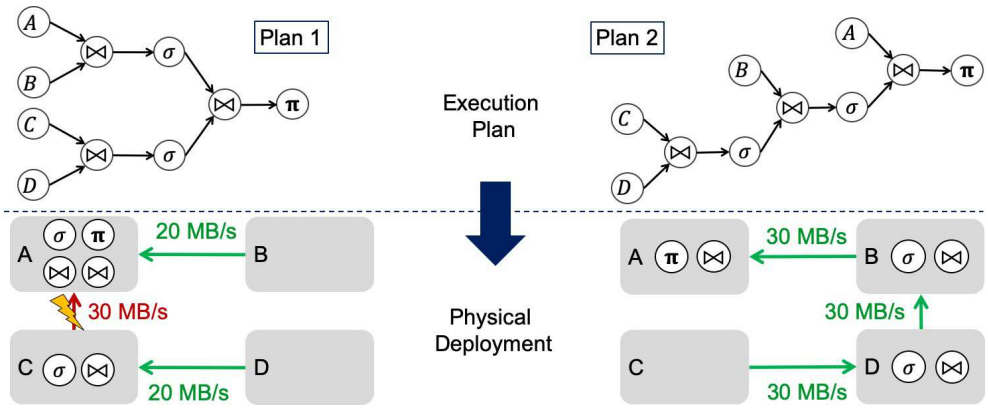


Figure 4.5: Different query execution plans result in different deployments.

To determine the optimal deployment of a query, the *Query Planner* and the *Scheduler* need to jointly optimize the query by evaluating different combinations of logical and physical plans. To avoid computing all possible combinations (that is NP-Hard), we rely on a heuristic cost-based estimation. It first applies any optimization that is independent of the environment (e.g., pushing filter operation upstream) and then evaluates multiple plans with different aggregation ordering. We only consider the ordering of aggregation operators since they are typically the ones that involve cross-site data transmission. The *Scheduler* will compute the optimal task placement for each plan and select the combined plan-placement pair with the lowest estimated delay.

Re-planning Stateful Execution

The main challenge in re-planning a query is in preserving the processing state of a stateful operator. Changing the query plan of a stateless execution can be done by simply replacing the old execution with a new execution. However, in the case of stateful execution, the new execution must restore the state maintained by the previous execution. Although the *Query Planner* guarantees that different plans will output the same results, switching plans in the middle of an execution may not provide this guarantee because different plans may have different stateful operators with different state semantics. For example, the state of $\sigma(A \bowtie B)$ may not be compatible with the state of $\sigma(B \bowtie C)$ since they may have different schema. Thus, the state of $\sigma(A \bowtie B)$ cannot be recovered by the operator instances of $\sigma(B \bowtie C)$.

To continue the progress from the old execution without losing any state, our *Query Planner* will only consider plans that comprise common sub-plans covering the stateful operators. For example, both Plan 1 and Plan 2 in Figure 4.5 exhibit a common sub-plan on $\sigma(C \bowtie D)$. Thus, the new instances of $\sigma(C \bowtie D)$ in the second plan can fully recover the states maintained by the previous plan. However, if $\sigma(A \bowtie B)$ is also stateful, changing from Plan 1 to Plan 2 may not be feasible unless the query can tolerate a certain degree of accuracy/quality loss. Another way to switch query plans for stateful execution is if the operator maintains a short and finite state where reconfiguration can be done at the end of the state interval. For example, in the case of a windowed grouped aggregation with a tumbling window, this can be performed at the end of the window when the state is re-initialized. This is similar to the adaptation during the coordination interval in the *batch synchronous processing* (BSP) model [**?**, **?**].

Re-evaluating both the logical and physical plans of a query typically results in a better adaptation than re-optimizing only its physical plan. However, the former is computationally expensive. Furthermore, query re-planning also has limited applicability for queries that comprise stateful operators. Thus, to preserve the accuracy of the results when re-planning a stateful execution, we only consider alternative plans that exhibit common sub-plans involving the stateful operators.

4.6 WASP's Adaptation Policy

In this section, we qualitatively compare 4 different adaptation techniques: (1) task re-assignment, (2) operator scaling, (3) query re-planning, and (4) data degradation. We then present the policy on how `WASP` determines *which* adaptation technique to use based on several factors.

4.6.1 Adaptability Technique Comparison

Table 4.2 shows a qualitative comparison between different adaptation techniques. Both task re-assignment and operator scaling are generally applicable for any type of operators that can be parallelized whereas query re-planning has limited applicability for queries that comprise stateful operators since their states may not be compatible with the

Table 4.2: Qualitative comparison between different adaptation techniques.

Technique	Adaptation	Applicability	Granularity	Overhead[1]	Quality loss
Task re-assignment	Task placement	General	Stage	Low	No
Operator scaling	Operator parallelism	General	Stage	Low	No
Query re-planning	Query plan	Specific	Query	High	No[2]
Data degradation	Degradation policy	Specific	Query	Low	Yes

operators of a different plan. In contrast, data degradation is application- and algorithm-dependent and may not be applicable for queries that require high accuracy/quality.

The granularity of task re-assignment and operator scaling is on a stage level, but the latter is more flexible since it is not constrained by the initial operator parallelism. On the other hand, query re-planning typically results in a better adaptation since it re-optimizes the whole execution pipeline. However, it comes at the expense of high overhead since it needs to replace the entire execution. The granularity of data degradation depends heavily on the policies [**?**, **?**]. For example, in video analytics, users can specify different frame rates (e.g., 30 and 60 FPS) and fidelity (e.g., 50% and 75%). Lastly, task re-assignment and operator scaling do not affect the output while degrading data may reduce the output's quality.

4.6.2 Determining Factors

Determining the *right* optimal adaptation is complex and may not be feasible since it depends on a lot of factors. Thus, we propose a heuristic approach that considers (1) the type of bottlenecks, (2) the type of operators, (3) overhead, and (4) the type of dynamics. Figure 4.6 shows `WASP`'s adaptability decision.

1. **Type of bottlenecks.** To handle computational bottlenecks, `WASP` allocates additional resources and scales up the bottleneck operator. It will first try to scale the operator within the same site and only consider scaling it to remote sites if the resources are not available since the latter will incur additional network delay and WAN bandwidth consumption. On the other hand, if the execution is constrained by the network bandwidth, it further considers the type of the operator, i.e., stateless or stateful.

[1] Excluding inter-site state migration overhead.
[2] Yes, if the state is not compatible or can be ignored by the new plan.

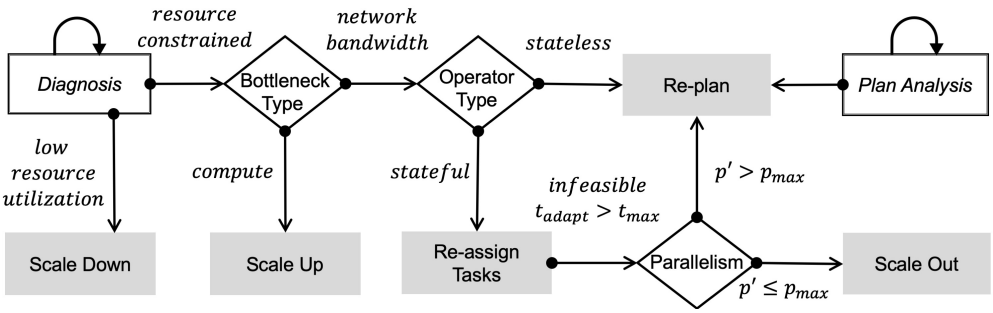


Figure 4.6: Determining *which* adaptability action.

2. **Type of operators.** In the case of stateless execution, WASP will re-optimize the whole execution pipeline: both the logical and physical plans, since the only overhead is in finding such a better combination. In the case of stateful execution, query re-planning may not always be feasible (as discussed in Section 4.5.3). Thus, WASP will first try to re-assign the existing tasks and only scale the operator if it cannot find an alternative placement with the given parallelism or the adaptation overhead is higher than a specific threshold ($t_{adapt} > t_{max}$). However, WASP may limit the number of additional tasks to scale per iteration to prevent resource hoarding or over-allocation, and may further choose to re-evaluate the query plan if the parallelism has exceeded the threshold ($p' > p_{max}$).

3. **Overhead.** WASP estimates the adaptation overhead based on the slowest state migration time: $t_{adapt} = max(\frac{|state_x|}{B_x^y}), \forall x, \forall y$, where x and y are the source and the destination sites respectively. If the overhead is too high, WASP will scale out the operator from p to p'. This can reduce the overhead through state partitioning. Since most stream operators balance their workload among their tasks [**?**], the average state size per task after the scaling is $\frac{|state|}{p'} < \frac{|state|}{p}$, given that $p' > p$. Hence, t_{adapt} can typically be reduced as the size of the state that needs to be migrated is reduced. In practice, t_{max} can be set based on the frequency of the dynamics, so that $t_{max} <$ the frequency for the system to progress.

4. **Type of dynamics.** So far, we have focused on addressing short-term dynamics. However, WASP can also be extended to handle long-term dynamics (e.g., daily workload shift [**?**]). This type of dynamics usually follows a specific pattern

and can be predicted. Thus, WASP will handle this differently by periodically re-evaluating the query plan in the background. How to accurately model/profile the dynamics itself is out of the scope of this work.

4.7 Discussion & Assumptions

- **Re-optimize or degrade?** Data degradation has been widely used to handle bottlenecks in wide-area streaming analytics and video analytics [**?**, **?**, **?**, **?**]. We believe that this approach is complementary to our work, and can be used together with our re-optimization-based adaptation. For example, if maintaining high accuracy is a priority, the system may prefer re-optimizing the execution to degrading the data. On the other hand, if the query can tolerate a certain degree of inaccuracy, the system may first degrade the data and start re-optimizing the query once it has reached the minimum accuracy threshold. Operator scaling will also be preferable when recovering from failures since dropping a large number of events may result in significant quality loss. Furthermore, operator scaling can also handle misconfiguration. Thus, both techniques are complementary to each other and their applicability depends on the query itself.

- **Transient workload spikes.** In this work, we focus on dynamics that lass longer than a few seconds. Re-optimizing a query execution to handle very short workload spikes may leave the system unstable because the workload may have already changed when the query is adapted. Thus, WASP ignores transient workload fluctuations that happen only for a very short period.

- **Homogeneous compute power across slots.** WASP abstracts the computational resources using *computing slots*, similar to the approach adopted by many distributed stream processing systems [**?**, **?**, **?**]. Since, the performance of most wide-area streaming analytics queries is predominantly determined by the inter-site data transmission, we hide the heterogeneity across slots and only consider the heterogeneity across sites based on the number of available slots per site.

- **Balanced event partitioning.** A stream operator may partition its output to multiple downstream tasks. For clarity reason, we assume that the output

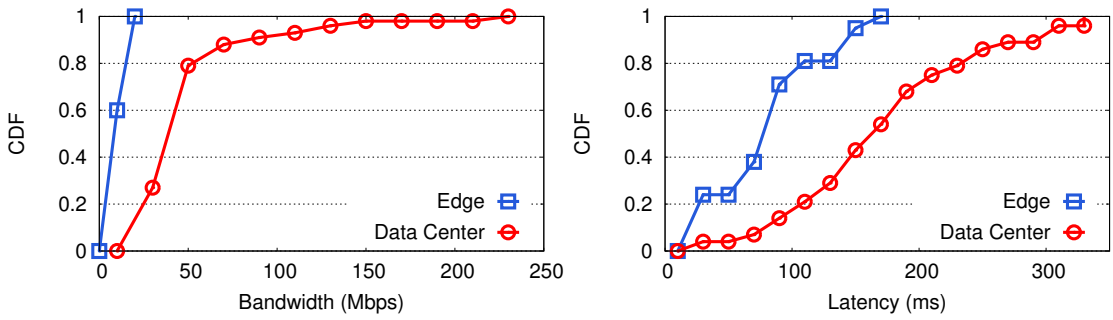
stream is evenly distributed across tasks, which is common for most operators [**?**]. However, our proposed techniques are not limited by this assumption and can be integrated with other stream partitioning schemes. Note that, although we consider even stream partitioning across tasks, we do not make any assumption on the input data distribution.

4.8 Implementation

We have implemented a `WASP` prototype on Apache Flink [**?**] by (1) implementing a network monitoring module (*WAN Monitor*), (2) incorporating WAN awareness in planning queries and scheduling tasks, and (3) implementing an adaptability module that periodically gathers tasks' runtime information every 10 seconds, diagnoses any unhealthy execution and wasteful resource utilization, and adapts them. We override the default Flink's scheduling algorithm by plugging in our WAN-aware task placement algorithm that solves the ILP problem using the Gurobi optimization tool [**?**]. We also generate multiple plans for each submitted query with different ordering of aggregation operators. For example, in the case of aggregating 3 data streams: A, B, and C, the system will generate 4 plans with different aggregation orders: $(A \oplus B \oplus C)$, $((A \oplus B) \oplus C)$, $(A \oplus (B \oplus C))$, and $(B \oplus (A \oplus C))$.

4.9 Experimental Evaluation

We evaluated `WASP` using an emulated testbed derived from a real wide-area system deployment. It consisted of 16 nodes: 8 edge nodes and 8 data center nodes. Each edge node was configured with 2-4 slots/node and each data center node was configured with 8 slots/node. A slot was configured with 1 CPU and 1GB of RAM. To mimic the actual wide-area system deployment, we limited the pair-wise network bandwidth between sites and introduced network delay using a `tc` limiter tool. The network connections of the data center nodes were initially configured based on the average of a 1-day measurement of network bandwidth between 8 Amazon EC2 data centers: Oregon, Ohio, Ireland, Frankfurt, Seoul, Singapore, Mumbai, and Sao Paulo. On the other hand, the



(a) Bandwidth distribution.

(b) Latency distribution.

Figure 4.7: Inter-site network distribution. Edge connections only consider nearby sites.

connections between the edge nodes were configured based on the actual public Internet network whose bandwidth varied from 7-14Mbps [**?**]. Figure 4.7 shows the network bandwidth and latency distributions between the nodes. We set $\beta = 0.8$, $p_{max} = 3$, and a monitoring interval of 40 seconds to allow any adapted query to stabilize.

Our experiments address the following questions:

- Are the proposed adaptation techniques (task re-assignment, operator scaling, and query re-planning) generally applicable for different types of queries (stateless and stateful) and dynamics (workload and network) (Section 4.9.2)?
- How do task re-assignment, operator scaling, and query re-planning compare with each other (Section 4.9.3)?
- How does `WASP` perform in a wild environment where workload variation, bandwidth changes, and failures may happen unpredictably (Section 4.9.4)?
- How does `WASP` mitigate the overhead of migrating/distributing state in wide-area settings (Section 4.9.5)?

4.9.1 Methodology

Query and Dataset. To demonstrate the generality of our technique, we evaluated `WASP` using the following 3 different queries (see Table 4.3 for details):

1. *Advertising Campaign* from Yahoo! Streaming Benchmark (YSB) [**?**] which monitors relevant advertisements related to specific campaigns every 10 seconds. To

Table 4.3: WASP query details.

Application	State Size	Operators	Dataset
Advertising Campaign	<10 MB	filter, map, window, join	Synthetic data
Top-K Topics	∼100 MB	filter, map, union, window, reduce	Twitter trace [**?**]
Event of Interests	0 MB	filter, union, project	Twitter trace

prevent potential bottleneck in Redis and Kafka (e.g., partition mismatch), we replace all I/O operations with in-memory operations and cache intermediate results using an in-memory data structures.

2. *Top-K Topic Detection* on a replayed Twitter trace. The query aggregates the top 10 most popular topics for each country over a period of 30 seconds. It consists of 2 different states: the source's offset and the intermediate aggregation results.

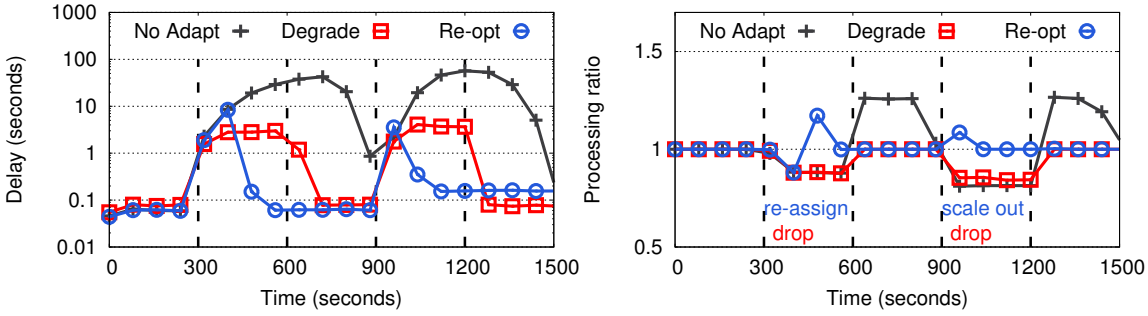
3. *Event of Interests* on a replayed Twitter trace. This query simply filters out events based on one or more attributes (e.g., language, topic, country of origin) and it does not maintain any internal state (stateless).

The YSB data were synthetically generated and distributed evenly across the 8 edge locations. On the other hand, the Twitter trace (collected from Twitter Streaming APIs [**?**]) was distributed based on the geographic information in each tweet. Thus, the latter covers the spatial and temporal distributions of actual events. Each operator was configured with $p = 1$ and we set a checkpointing interval of 30 seconds.

Metrics. We consider 2 main metrics in our experiments:

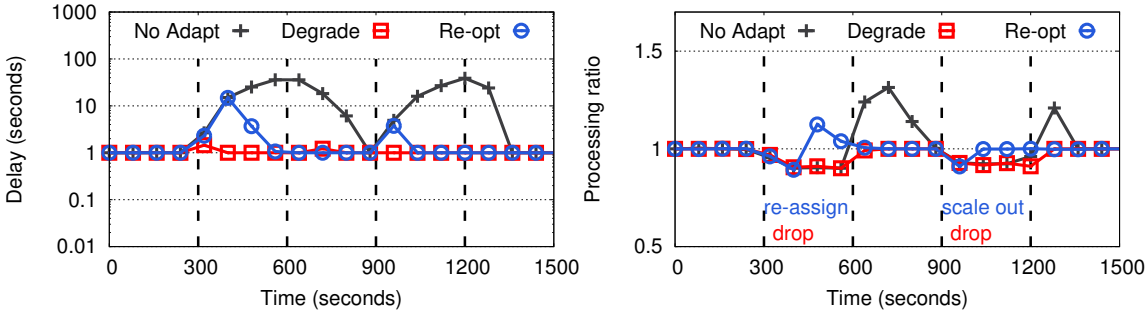
1. *Execution Delay.* The delay is measured as the average *event latency* which is the difference between the time an event is emitted at the sink and the time it was generated by the external source. In the case of windowed grouped aggregation, the event generation time is set to the maximum event time of all events within a particular window (the latest event within a window) [**?**].

2. *Processing Ratio.* The processing ratio is computed as the ratio between the observed processing rate and the expected processing rate. A ratio of 1 indicates that the query is able to process all the events, while a ratio of < 1 indicates that the query is constrained by the allocated resources. This is more general than an accuracy metric [**?**, **?**] since the latter is algorithm-specific [**?**].



(a) Average execution delay over time. (b) Processing ratio over time.

Figure 4.8: YSB execution under workload and bandwidth dynamics.



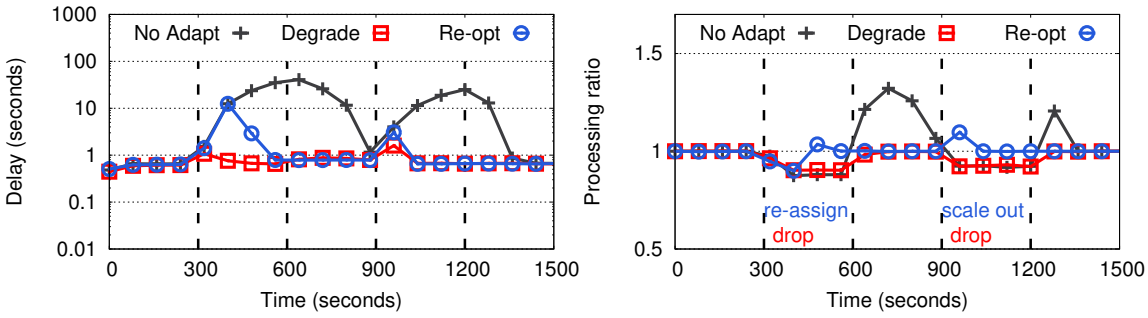
(a) Average execution delay over time. (b) Processing ratio over time.

Figure 4.9: Top-K execution under workload and bandwidth dynamics.

4.9.2 Adapting to Wide-area Dynamics

We initialized the input stream rate at each source to 10,000 events/second at $t = 0$ and introduced dynamics every 5 minutes. Specifically, we first increased the rate to 20,000 events/second at $t = 300$, and decreased it back to 10,000 events/second at $t = 600$. To see the effect of network bandwidth variation, we halved the bandwidth of every link at $t = 900$ and restore it at $t = 1200$. We compare our re-optimization-based approach, (1) `Re-opt`, against (2) `No Adapt` which did not adapt to dynamics, and (3) `Degrade` which dropped late events in case of insufficient resources. We set the SLO to 10 seconds for `Degrade`. Figure 4.8, Figure 4.9, and Figure 4.10 show the average delay and processing ratio of the 3 queries respectively.

1. [$300 \leq t < 600$]: We see from Figure 4.8(a), Figure 4.9(a), and Figure 4.10(a) that the delay of `No Adapt` increased continuously by up to 2-3 orders of magnitude as



(a) Average execution delay over time.

(b) Processing ratio over time.

Figure 4.10: Event Interest execution under workload and bandwidth dynamics.

the workload increased because some network links could not sustain the workload. This shows in the reduction of the processing ratio from 1 to ∼0.86 (Figure 4.8(b), Figure 4.9(b), and Figure 4.10(b)). The processing ratio of `Degrade` also dropped to ∼0.86 but it was able to maintain the delay within the SLO by dropping late events, which in practical scenario may affect the result's accuracy. In contrast, `Re-opt` was able to maintain low-latency processing without dropping any event (maintain the average processing ratio to ∼1) by *re-assigning* the bottleneck tasks to different locations at $t = 380$. The processing ratio of the YSB and Top-K momentarily dropped since the executions were suspended for approximately 2 and 10 seconds to migrate the states (Figure 4.8(b) and Figure 4.9(b)). Notice that the delay of `Degrade` increased to ∼8 seconds for the YSB case but it remained low for the Top-K case. This was because the key distribution of the former was much lower than the latter's, making it more sensitive to late events when measuring the event time of the windowed grouped aggregation operators.

2. [$600 \leq t < 900$]: When the workload reduced at $t = 600$, `Degrade` stopped dropping events and its processing ratio started to increase to 1. In the case of `No Adapt`, the processing ratio temporarily increased > 1 indicating that the system was consuming queued events from the previous interval.

3. [$900 \leq t \leq 1200$]: To see the effect of network bandwidth variation, we halved the bandwidth capacity of all links at $t = 900$. We can see that the delay and processing ratio have similar trends to the effect of increasing workload. However,

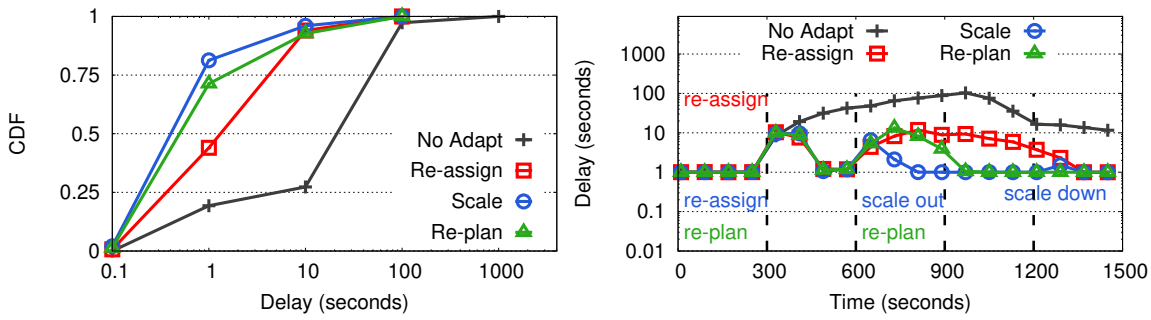
`Re-opt` took a different adaptation action by *scaling out* the bottleneck operator instead of *re-assigning* the tasks since the adaptation module could not find a single alternative link whose bandwidth was higher than the stream rate. We can also see that operator scaling resulted in a faster convergence, taking advantage of having more resources. Finally, the delay of the 3 queries dropped at $t = 1200$ when the bandwidth increased.

These results show that (1) the re-optimization-based adaptation can handle both workload and bandwidth variations, (2) this is generally applicable for both stateless and stateful executions, and (3) it can maintain low-latency execution without dropping any event. For the rests of the experiments, we used the stateful Top-K query as our workload since it is the best representation of an actual geo-distributed workload among the 3 queries, and they have a similar trend.

4.9.3 Re-Assign vs. Scale vs. Re-Plan

Next, we compared task re-assignment, operator scaling, and query re-planning, in handling a combination of workload and bandwidth variations independently. We introduced dynamics every 5 minutes by varying the workload and bandwidth by a factor of $\{1, 2, 2, 1, 1\}$ and $\{1, 1, 0.5, 0.5, 1\}$ respectively. We compared (1) `No Adapt`: which did not adapt to dynamics, (2) `Re-assign`: which only handled dynamics by re-assigning tasks, (3) `Scale`: which would first try to re-assign the tasks but might scale some operators if it could not find a solution, and (4) `Re-plan`: which re-evaluated the execution plan based on the observed workload and resource availability. Both `Re-assign` and `Re-plan` never changed the parallelism.

First, we can see from Figure 4.11(a) that all of the techniques that adapt the query resulted in lower delay compared to `No Adapt`, highlighting the importance of adaptability in handling dynamics. Secondly, `Scale` resulted in the lowest overall delay compared to `Re-plan` and `Re-assign`, and `Re-plan` resulted in a lower delay for the majority of the events ($<$ 93rd percentile) with respect to `Re-assign`. Figure 4.11(b) breaks down the delay of each technique for each interval. At $t = 300$, both `Re-assign` and `Scale` *re-assigned* the tasks while `Re-plan` switched to another plan. All of them could handle the the workload increase during this interval. However, when the available bandwidth



(a) Delay distribution.

(b) Average delay over time.

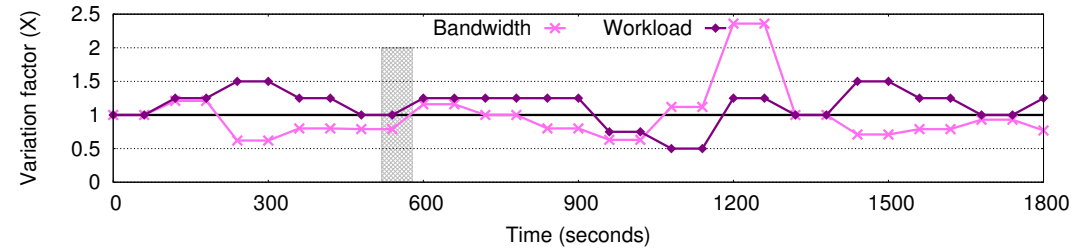
Figure 4.11: Comparison between the 3 techniques in handling dynamics individually.

decreased at $t = 600$, `Re-assign` was unable to find an alternative task placement since it was constrained by the initial parallelism. In contrast, `Scale` was able to handle this problem by acquiring 2 additional slots and *scaling out* the bottleneck operators. `Re-plan` was also able to handle the problem by *re-planning* the query. However, `Scale` resolved the bottleneck faster than `Re-plan`, taking advantage of the additional resources. Lastly, `Scale` decreased the parallelism when the bandwidth availability had increased at $t = 1200$ and some of the resources became underutilized.

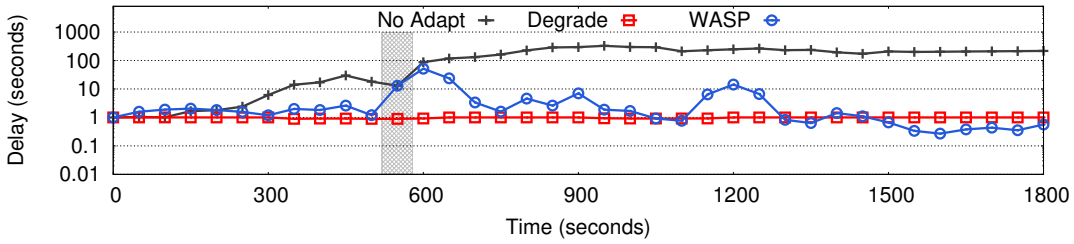
Comparing `Re-assign` and `Scale`, we can see that dynamically adapting operator parallelism can better handle bottlenecks with the expense of consuming more resources. We can also see that re-optimizing both the logical and physical executions (`Re-plan`) results in a more optimal adaptation than simply re-assigning tasks (`Re-assign`) with the same parallelism, and hence the former is preferable whenever possible.

4.9.4 WASP in a Live Environment

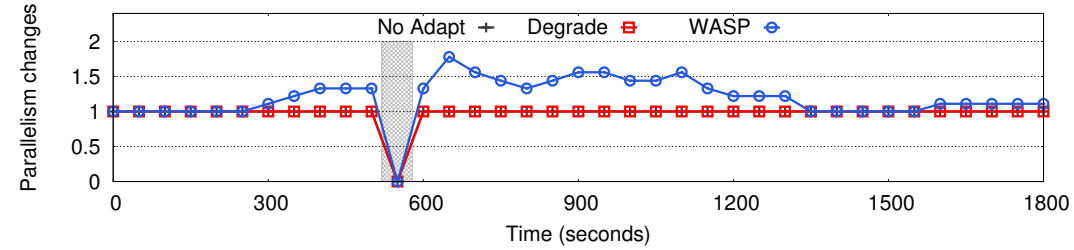
In this experiment, we evaluate `WASP` in a live, trace-driven environment where we introduced (1) network bandwidth dynamics based on a real pair-wise bandwidth variation trace between 8 Amazon EC2 data centers which ranged from 0.51 to 2.36, and (2) random workload patterns with a variation factor ranging from 0.8 to 2.4, and (3) a failure at $t = 540$ by revoking all the computational resources and re-allocating them after 60 seconds. We added failure in this experiment to see how `WASP` can *scale* a query to quickly process queued events. Figure 4.12(a) shows the bandwidth and workload variations. Here, we compared (1) `WASP` against (2) `No Adapt` and (3) `Degrade`. `WASP`



(a) Bandwidth and workload variations over time.



(b) Average delay over time.

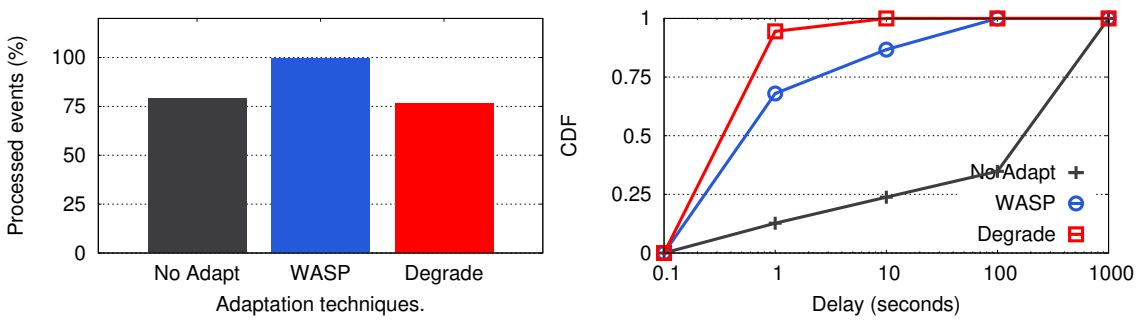


(c) Parallelism changes over time.

Figure 4.12: WASP's adaptations to dynamics and failures.

could use any of the adaptation techniques: task re-assignment, operator scaling, and query re-planning, depending on the condition discussed in Section 4.6.2.

Figure 4.12(b) and Figure 4.12(c) show the delay and parallelism changes over time. We make a few observations here. First, `WASP`'s processing delay stayed close to 1 second (similar to the unconstrained case) for most of the time except for some intervals. At $300 \leq t < 540$, there was a slight variation in the delay when `WASP` *scaled out* 2 of the tasks to handle workload increases and bandwidth drops. At $t = 640$, `WASP` was able to quickly handle the accumulated events after recovering from failure by *scaling out* the bottleneck operators. It then gradually *scaled down* some operators after the workload had reduced and the execution stabilized. Finally, it further *scaled down* the majority of the additional tasks at $900 \leq t < 1320$ when the available bandwidth had increased.



(a) Average processed events.

(b) Delay distribution.

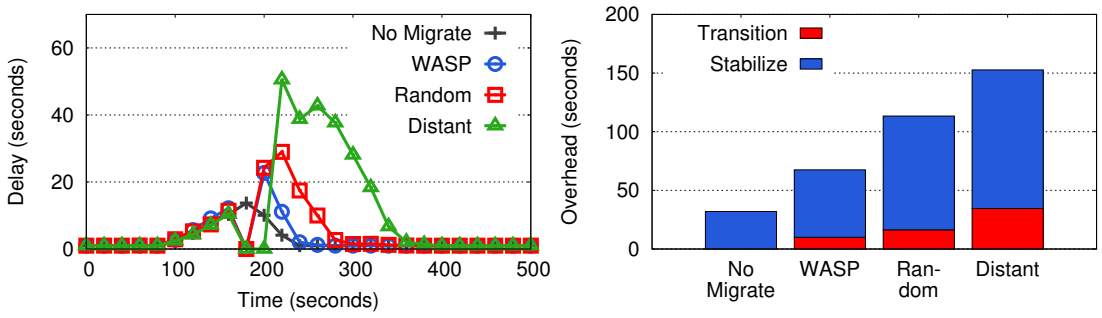
Figure 4.13: Quality vs. delay trade-offs.

In contrast to `WASP`, the delay of `No Adapt` increased by more than 2 orders of magnitude, especially after the execution recovered from failure since it was unable to handle the queued events. Although `Degrade` could maintain the average delay within 1 second for most of the time, it had to sacrifice up to ∼24% of the events. This resulted in an inaccurate result and hence, may not be tolerable for queries that require high accuracy. In contrast, `WASP` could process all of the events while maintaining the low-latency processing (Figure 4.13(a)). From Figure 4.13(b) we can see that `WASP` had a longer delay tail distribution compared to `Degrade`. We observed that the majority of the delay came from the monitoring process, the transitioning phase for migrating states, and the processing of queued events after `WASP` recovered from failure.

These results show that (1) `WASP` can handle real-world dynamics and failures without dropping any of the events, and (2) there is essentially a trade-off between the re-optimization and degradation-based adaptation techniques in maintaining the quality/accuracy of the results and maintaining the low-latency processing.

4.9.5 Mitigating Adaptation Overhead

In the last set of experiments, we see how `WASP` can reduce the overhead of adapting queries with large computation state. Specifically, we highlight the importance of network awareness and the benefit of state partitioning in reducing the overhead. We break down the overhead into 2 phases: (1) *transition time*: when the execution is suspended for state migration, and (2) *stabilizing time*: the time needed to consume all queued events that have been accumulating during the transition process. We highlight the



(a) Execution delay over time.

(b) Adaptation overhead.

Figure 4.14: Network-aware state migration.

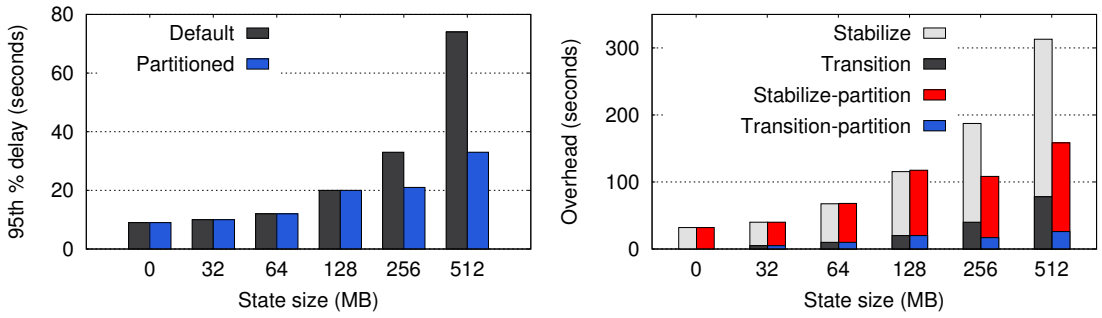
importance of network-aware state migration in Section 4.9.5 and show the benefit of state partitioning to further reduce the overhead in Section 4.9.5. In both experiments, we controlled the size of the state that needed to be migrated.

Network-aware State Migration

To ensure low-overhead adaptation, `WASP` estimates the task migration overhead based on the size of its state and the bandwidth availability between the initial site and the new site. Migrating a state over the WAN may incur high overhead if the bandwidth between the two sites is constrained. This is impractical for frequent dynamics that are common in wide-area settings. In this experiment, we compared (1) `WASP` against (2) `No Migrate`[3] which did not migrate the state (equivalent to adapting stateless operators), (3) `Random`: which ignored the bandwidth availability, and (4) `Distant`: which migrated a state to a site. In any case, the system ensured that the destination site had sufficient bandwidth to process the actual data stream and hence, the execution would eventually stabilize. We fixed the state size to 60MB.

Figure 4.14(a) compares the effect of different state migration techniques to the overall query execution delay. In any of the cases, the system started adapting the query at $t = 180$. Here, we can see that `No Migrate` could quickly reduce the delay without migrating the state. However, this resulted in an incorrect result or a loss in accuracy. Comparing the other 3 techniques that maintained the state, we can see that `WASP` resulted in the lowest delay during the adaptation phase.

[3] Ignoring the state will result in a loss of accuracy in the result.



(a) Execution delay distribution.

(b) Adaptation overhead.

Figure 4.15: Mitigating overhead through operator scaling and state partitioning.

Figure 4.14(b) shows the breakdown of the overhead. First, `No Migrate` incurred ∼0 transition time since it only redirected the data streams. There was a stabilizing time and a slight increase in delay despite not migrating the state (similar to adapting stateless operator) due to the queued events during diagnosis period. Reducing this period may reduce the transition time but makes the system more susceptible to spikes and miss-estimation. Secondly, `WASP` resulted in 41-56% lower overhead and 7-20 seconds lower 99th percentile delay compared to `Random` and `Distant` respectively. The reason is because the two WAN-agnostic approaches migrated state over constrained links, leading to a higher adaptation time. These results show the importance of WAN awareness in reducing the adaptation overhead in wide-area settings.

State Partitioning

In addition to network awareness, we observe how partitioning and distributing large states across multiple links can further reduce the adaptation overhead. In this experiment, we compared `Default`: which never partitioned the state, and `Partitioned`: which would force the adaptation module to find an alternative placement (may involve operator scaling) and partition the state whenever the estimated transition time exceeded a specific threshold. We varied the state size to {0, 32, 64, 128, 256, 512} MB and set the maximum threshold to 30 seconds.

Figure 4.15(a) shows the 95th percentile delay over different state size. We can see that the delay of `Default` increased as the state size increased. In contrast, `Partitioned` could reduce the delay in the case of large state (256MB and 512MB). This is due to the

reduction in the adaptation overhead. Figure 4.15(b) shows the breakdown of the overhead. In general, the overhead of adapting a query increased as the state size increased. However, `Partitioned` was able to reduce this overhead by scaling out the bottleneck operator and partitioning its state across multiple network links. This reduced the overhead by more than 120 seconds (Figure 4.15(b)) which subsequently reduced the average delay by 42 seconds (Figure 4.15(a)). These results highlight another benefit of operator scaling in reducing the adaptation overhead.

4.10 Related Work

Wide-Area Data Analytics Systems. Wide-area data analytics systems can be classified into two different groups based on their processing model: (1) batch analytics, and (2) streaming analytics. Early work in wide-area data analytics has focused on incorporating WAN awareness in scheduling jobs and placing tasks across multiple sites with the goal of minimizing query execution latency [**?**, **?**, **?**] or saving WAN bandwidth consumption [**?**]. However, they ignore the dynamic nature of wide-area environment. Tetrium [**?**] considers dynamic resource availability but does not consider re-optimizing jobs that have already been deployed. Turbo [**?**] has looked at dynamic query re-planning for batch analytics but its techniques cannot be applied directly for long-running streaming analytics queries.

Existing work has also looked at optimizing streaming analytics queries in wide-area settings [**?**, **?**, **?**, **?**, **?**]. Pietzuch et al. [**?**] address the problem of network-aware operator placement, and Sana [**?**] considers sharing common execution between queries. However, they do not address workload/resource dynamics. Others have considered the dynamic nature of a wide-area environment, but focused on trading latency, WAN traffic, and accuracy. Heintz et al. [**?**] propose a technique to trade accuracy and delay in the context of windowed grouped aggregation. Kumar et al. [**?**] proposes a TTL-based approach that trades delay and WAN traffic for windowed grouped aggregation. JetStream [**?**] allows users to specify different degradation policies with a data-cube model. AWStream [**?**] relies on a profiling technique to determine which degradation policy to take to ensure a certain degree of accuracy. We argue that these degradation approaches are application-specific and they are complementary to our techniques.

Adaptability in Distributed Stream Processing Systems. There have been a large body of work that address the importance of adaptability in cluster-based stream processing systems [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]. However, they have focused on addressing computational bottlenecks by scaling out tasks within a cluster. These techniques cannot be directly applied to handle dynamics in a wide-area environment due to the highly heterogeneous and limited network bandwidth.

Others have also looked at the importance of minimizing the adaptability overhead. Drizzle [?] reduces the synchronization overhead for *Bulk Synchronous Processing* model. Chi [?] relies on control mechanism to reduce the overhead of global synchronization. ChronoStream [?] partitions and distributes large states across multiple nodes to allow fast recovery. DS2 [?] predicts the scaling factor based on the expected processing rate of each operator for dataflow model. Although these techniques are related to our work, they do not account for network constraints. In wide-area environment, the overhead of migrating large states over WAN is significantly higher than the partitioning overhead, and hence our techniques focus on minimizing this overhead.

4.11 Conclusion

In this work, we rethink the adaptability property of wide-area stream processing systems and propose a WAN-aware adaptation framework, `WASP`, that allows queries to handle dynamics without compromising quality. `WASP` adapts queries through a combination of multiple techniques: task re-assignment, operator scaling, and query re-planning. `WASP` can automatically determine which adaptability action to take depending on the type of queries, dynamics, and goals. `WASP` further incorporates network awareness to mitigate the overhead of adapting queries in wide-area settings. Experimental evaluation shows that `WASP` is able to handle wide-area dynamics with low overhead and without sacrificing quality.

Chapter 5

Multi-Query Adaptation in Wide-Area Streaming Systems

5.1 Introduction

Adaptability is an important property of stream processing systems to maintain the low-latency and high-throughput execution of long-running streaming analytics queries in the face of dynamics. In wide-area settings, dynamics are common not only because of the workload variability but also the nature of wide-area network (WAN) bandwidth that frequently changes. Furthermore, job arrivals/completions, stragglers, and failures are inevitable in large-scale distributed systems.

Existing work has made a strong case for the importance of adaptability in wide-area stream processing systems, but has mainly focused on answering the question of *how to adapt a query?* [**?**, **?**, **?**, **?**, **?**] Specifically, their focus is on the trade-off between query execution delay, accuracy, and WAN bandwidth consumption. As modern distributed stream processing systems support multiple query deployments, concurrent query executions may compete for common limited resources, leading to a resource contention problem. In such a condition, *all* of the queries will typically be marked as *unhealthy*, i.e., they do not have sufficient resources to handle their workload, and they need to be adapted. However, adapting all of the queries will result in multiple adaptation waves that not only take a long time to complete but also incur large performance perturbation. Instead, adapting only a subset of queries may be sufficient in most cases, as we

will show. In this case, the key questions are *which queries* need to be adapted and *how to adapt* them. Answering these questions is challenging as different queries may have different characteristics and requirements, and different adaptations may result in different performance and resource consumption while incurring different overhead.

To address the above challenges, we propose `Nako`: a multi-query adaptation module for wide-area streaming systems. `Nako` continuously monitors the resource requirements of each query and identifies bottlenecks in the system. Instead of adapting each query independently, `Nako` accounts for any resource contention among queries. It selectively determines *how* to resolve the bottleneck and *which* queries need to be adapted. `Nako` uses a notion of *adaptation cost* to make these decisions. The adaptation cost is computed as a linear combination of an *overhead cost* and a *resource consumption cost*. The overhead cost is computed based on the time required to adapt an execution, while the resource consumption cost is computed based on the cost of acquiring additional computing resources and the increase in network bandwidth consumption. Our motivation is based on the observation that different adaptation actions may incur different overhead and result in different resource consumption.

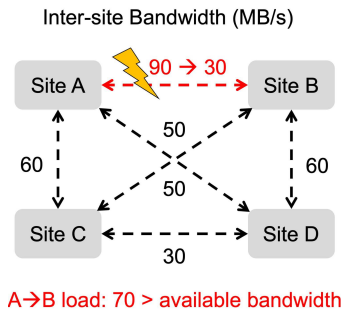
Since multiple queries may share common execution, recent work has shown the benefit of applying multi-query optimization to eliminate redundant data processing and duplicate data transmission over the network [**?**, **?**, **?**]. Yet, adapting an execution that is shared by multiple queries may affect the execution performance of all of the queries. Thus, `Nako` carefully considers the impact of adapting such an execution by computing the overhead and resource consumption costs differently. It will then decide whether to (1) adapt any other alternative query execution, (2) adapt the shared execution while maintaining its sharing property, or (3) split the shared execution.

We have implemented `Nako` by extending the adaptation module in `WASP` (Chapter 4) to consider multiple query executions in handling bottlenecks. We deploy `Nako` over an emulated testbed that profiles real geo-distributed Amazon EC2 sites and evaluate `Nako` using 8 streaming analytics queries over real geo-tagged Twitter trace. Experimental evaluation shows that `Nako` can identifies a small set of queries to be adapted, resulting in up to 2.1× lower average query execution delay and 2× faster adaptation in handling bottlenecks compared to an existing technique that adapts queries independently.

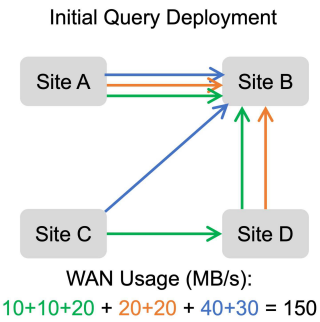
Per Query Input Stream

Query	Source	Rate (MB/s)
1	A	10
	B	40
	C	10
	D	20
2	A	20
	B	20
	D	20
3	A	40
	B	40
	C	30

(a) Workload.



(b) Environment.



(c) Initial deployment.

Figure 5.1: Initial deployment and workload of Query 1, 2, and 3.

5.2 Motivation

Recent work has addressed the importance of adaptability in wide-area stream processing systems [?, ?, ?, ?]. However, much of the work has focused on adapting each query individually irrespective to the other queries that may be competing for the same resources. In this work, we consider the query adaptation techniques as discussed in Chapter 4. We motivate through an example to show that adapting *one-query-at-a-time* independently may result in a sub-optimal adaptation, incur unnecessarily high overhead, or lead to wasteful resource consumption.

Figure 5.1 shows an example of three queries that are deployed over 4 different sites (A, B, C, and D). The sites are connected by heterogeneous network bandwidth. For clarity reason, we consider the inbound and outbound bandwidth between any two sites to be equal (in an actual wide-area system deployment, the inbound and outbound bandwidth capacities may vary). Here, the initial deployment of the three queries consumes network bandwidth with a rate of 150MB/s (Figure 5.1(c)). Suppose the network bandwidth from Site A to Site B that is shared by the three queries has dropped from 90MB/s to 30MB/s, which causes a network bandwidth contention among them. Upon identifying the network bottleneck, the system needs to determine which queries need to be adapted and how to adapt them.

Figure 5.2 shows three different adaptation options that may be taken by the system. Suppose the system decides to first adapt Query 1 by migrating its tasks that are located at Site B to Site D in order to avoid the congested network link (Option 1). This will

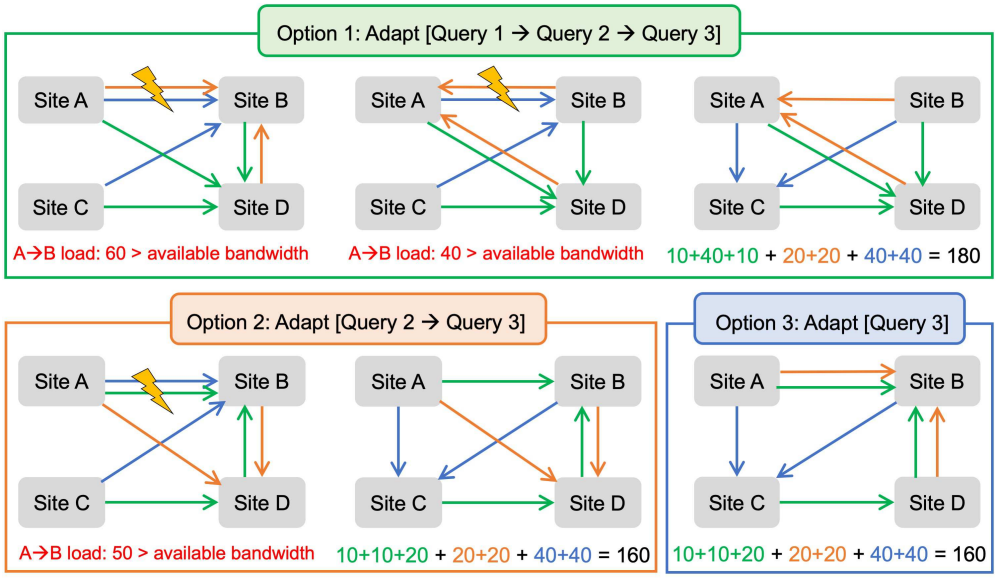


Figure 5.2: Different adaptations result in different network consumption and overhead.

resolve the network bottleneck for Query 1. However, the system will observe that the executions of Query 2 and Query 3 are still constrained and they need to be adapted. Suppose, the system decides to adapt Query 2 next. This will resolve the network bottleneck for Query 2. Yet, the system still needs to further adapt Query 3 whose execution is still constrained by the bandwidth availability between Site A and Site B. Thus, Option 1 will adapt all the three queries in order to resolve the network bottleneck. This adaptation option also increases the overall network bandwidth consumption of the three queries by 20% (see the final deployment of Option 1 in Figure 5.2).

Alternatively, the system may decide to adapt Query 2 first, e.g., because it has the highest SLO violation among the three queries (Option 2). After the adaptation, the system will still diagnose Query 1 and Query 3 to be unhealthy since their executions are still constrained by the available bandwidth. Thus, the system will further adapt Query 3 and finally observe that Query 1 is no longer constrained. In this case, Option 2 results in one less adaptation while consuming less network bandwidth compared to Option 1. Yet, if the system is aware of the network bandwidth contention among the three queries, it may instead adapt Query 3 first (Option 3). In this case, we can see that adapting only Query 3 is sufficient to resolve the bottleneck while consuming the same

overall network bandwidth as Option 2. Thus, the system no longer needs to adapt the other two queries. If the overhead of adapting each of the queries is the same, Option 2 unnecessarily doubles the overhead while Option 3 triples the overhead. We later generalize the policy to handle cases where different queries have different adaptation overheads. This example illustrate that adapting queries independently may result in unnecessarily high overhead and network bandwidth consumption.

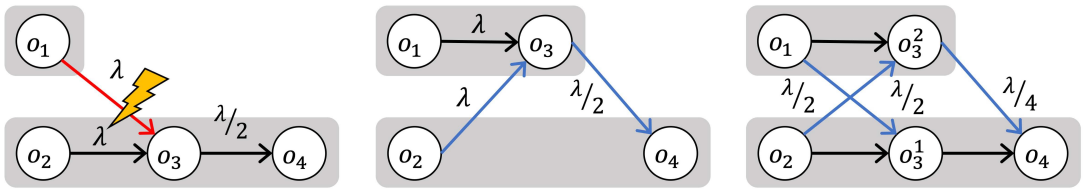
5.3 Adaptation Cost

In this section, we propose a notion of *adaptation cost*, that is used by `Nako` to determine *which* queries need to be adapted to handle resource contention bottlenecks among multiple query executions. We discuss `Nako`'s adaptation mechanism and policy later in Section 5.4. Adapting the execution deployment of an operator may have a drawback of increasing the overall resource consumption and incurring high adaptation overhead. Here, we define the overhead as the time required to transition from one execution to another execution which includes the time for (1) suspending the current execution and checkpointing any of its computation states, (2) migrating/distributing the states to different locations for the new execution, and (3) starting the new execution. In a practical scenario, increasing the resource consumption of a query typically corresponds to increasing the overall analysis cost (in terms of monetary cost [**?**, **?**]) while higher overhead may result in a stale or inaccurate result. Thus, it is critical to ensure low overhead and efficient resource utilization in adapting a query execution.

More formally, the adaptation cost (C_{adapt}) is computed as a linear combination of the resource consumption cost (C_R), and the overhead cost (C_O):

$$C_{adapt} = \alpha C_R + (1 - \alpha) C_O$$

Here, α is a weight factor that can be set based on the relative importance of the two cost factors. Setting $\alpha = 1$ focuses on the resource consumption cost and ignores the adaptation overhead, while setting $\alpha = 0$ solely focuses on the overhead cost and it ignores the resource consumption cost. We set $\alpha = 0.5$ in our deployment which balances the resource consumption and the adaptation overhead. We evaluate the effects of varying α later in Section 5.5.



(a) Constrained execution. (b) Option 1: re-assign o_3. (c) Option 2: scale o_3.

Figure 5.3: Different adaptation actions may result in different resource consumption.

5.3.1 Resource Consumption Cost

In this work, we consider 2 different types of resource consumption: (1) the number of computing slots used by a query, and (2) the total WAN bandwidth consumption. Adapting a query execution in wide-area settings may consume additional computing slots and increase the overall WAN bandwidth consumption. In a Cloud environment, this may incur additional monetary cost as most Cloud providers charge users for both the computing resource usage and any data transmission over the WAN.

Figure 5.3 shows an example that illustrates the case. If the bandwidth between operators o_1 and o_3 is constrained (Figure 5.3(a)), the system may (1) re-assign o_3 by co-locating it with o_1 to avoid competing for the constrained network link (Figure 5.3(b)) or (2) scale o_3 to reduce the rate of the data streams transmitted over the constrained network link (Figure 5.3(c)). Here, Option 1 and Option 2 will increase the overall network bandwidth consumption by $1\frac{1}{2}\lambda$ and $1\frac{1}{4}\lambda$ respectively. Although Option 2 (scale out case) results in lower network bandwidth consumption, it incurs additional computing cost for acquiring more computing resources.

In this work, we consider the resource consumption cost as a combination of both the computing resource cost and the WAN bandwidth consumption cost. More formally, the resource consumption cost (C_R) is computed as follow:

$$C_R = \Big(\Delta p'[s] \cdot \frac{N[s]}{A[s]}\Big) + \Big(\frac{\Delta\hat{\lambda}^{in}_{s\neq u}}{\hat{\lambda}^{in}_{s\neq u}} + \frac{\Delta\hat{\lambda}^{out}_{s\neq d}}{\hat{\lambda}^{out}_{s\neq d}}\Big), \forall s, \forall u, \forall d$$

Here, $\Delta p'[s]$ is the change to the number of operator's tasks deployed at site s, $N[s]$ is the total number of slots at site s, $A[s]$ is the number of available slots at site s, and $\hat{\lambda}^{(in|out)}$ is the inbound/outbound network bandwidth consumption from/to the

upstream/downstream operators (u/d). Here, the cost of acquiring additional computing resources at each site is weighted based on the popularity/scarcity of the slots. Intuitively, acquiring popular/scarce resources will incur higher cost. However, other cost models (e.g., fixed cost based on the instance type of a slot) may also be incorporated. On the other hand, the network consumption cost is computed based on the increase/decrease in the overall network bandwidth consumption ($\Delta\hat{\lambda}$) from/to its upstream/downstream operators that are deployed on remote sites. Here, the network cost can also be weighted based on the cost of transmitting data over the WAN if minimizing the analysis expense is the priority.

5.3.2 Overhead Cost

Minimizing adaptation overhead in streaming analytics is critical as it may incur high processing delay. Although the overhead of adapting an operator can be considered as a one-time cost (in contrast to the resource consumption cost that is continuous), high processing delay can significantly affect the timeliness and quality of the results. Thus, it is imperative to ensure a low adaptation overhead, especially for frequent dynamics. In wide-area settings, the main overhead of adapting an execution is typically in migrating an intermediate processing state over the WAN since the state maintained by a *stateful* operator can be very large in practice [**?**, **?**] while WAN bandwidth is scarce. Thus, we consider the state migration time as the major factor to the adaptation overhead. Specifically, the overhead cost (C_O) is computed as follow:

$$C_O = \frac{t_{adapt}}{t_{freq}}$$

Here, t_{adapt} is the total time required to adapt a query (including the suspension time and state migration time) and t_{freq} is the adaptation frequency of a query, estimated based on its historical statistics. Intuitively, normalizing the adaptation time with the adaptation frequency indicates the usefulness of the adaptation. Furthermore, frequently adapting an execution will significantly affect its performance and result's quality. This can also be used as a consideration whether to adapt a query for short-term dynamics if the overhead is too high. Similarly, this justifies the importance of adapting a query execution with high overhead to handle long-term dynamics (e.g., daily workload shift [**?**, **?**]).

The adaptation time itself is computed as the slowest state migration time (if any) since an operator can resume its execution only after all of its states have been migrated to the sites where the new tasks have been deployed to:

$$t_{adapt} = max\Big(\frac{|state|_{s1}^{s2}}{B_{s1}^{s2}}\Big), \forall s1, \forall s2, s1 \neq s2$$

Here, $s1$ and $s2$ correspond to the initial site and the new site respectively, $|state|_{s1}^{s2}$ is the size of the state that needs to be migrated from $s1$ to $s2$, and B_{s1}^{s2} is the bandwidth availability between the two sites.

5.4 Multiple Query Adaptation

This section presents how `Nako` handles resource contention among multiple query executions by considering the cost of adapting each of the queries. Although the computational resources are typically exclusively allocated to a particular query based on the number of computing slots[1] , the network bandwidth is typically shared by multiple queries. Thus, network bottleneck may arise if the aggregated bandwidth demand of a particular network link is higher than its available bandwidth. In this case, most runtime monitoring systems will diagnose that *all* of the executions are unhealthy and they need to be adapted. Instead of adapting all of the constrained executions, `Nako` will try to adapt only a subset of the executions. This is often sufficient to resolve the bottleneck. The main challenge here is to determine *which* queries need to be adapted.

5.4.1 Adaptation Flow

Figure 5.4 shows `Nako`'s adaptation process. Each operator periodically reports its runtime performance metrics (e.g., processing rate, output rate, and backpressure status) to the *Metric Reporting* module. A *Resource Monitoring* module continuously monitors the available network bandwidth between sites and the number of available computing slots at each site. Both the *Metric Reporting* and the *Resource Monitoring* modules report any update to the *Runtime Diagnosis*, which in turn uses this information to identify any bottleneck based on the execution model described in Chapter 4.4.

[1] This limits the number of tasks that can be run on a particular machine, although the actual hardware scheduling itself cannot be guaranteed.

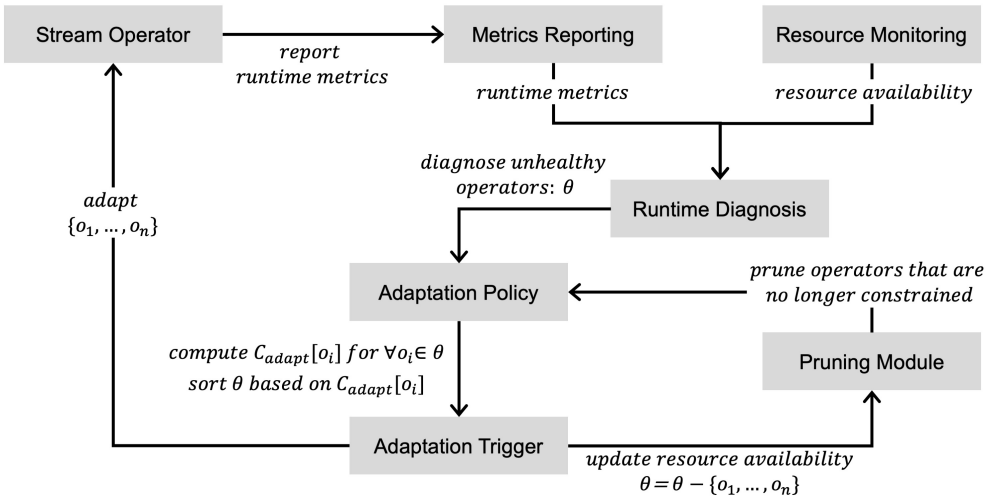


Figure 5.4: Nako's adaptation flows.

Which Queries to Adapt?

When the *Runtime Diagnosis* identifies a set of unhealthy operators θ, the *Adaptation Policy* will determine *how* to adapt each unhealthy operator execution, whether to scale the operator or simply re-assign the operator instances. It computes the cost of adapting each operator ($C_{adapt}[o_i], \forall o_i \in \theta$) using the formulation presented in Section 5.3. It will then sort the operators based on their adaptation cost values in ascending order and let the *Adaptation Trigger* module adapt the operators. The *Adaptation Trigger* adapts the operators in a greedy manner starting from the operator that incurs the lowest adaptation cost ($o_1 \leftarrow \theta.poll()$).

After adapting an operator, the *Adaptation Trigger* will re-evaluate the adaptation decisions for any other bottleneck operators that share resources with o_1 since the decisions were computed based on the availability of the resources before adapting o_1, and hence the pre-computed adaptation decisions may no longer be valid or necessary. However, any other operators that do not share resources with o_1 can be adapted in the same iteration. Note that, adapting o_1 will not cause any computational or network bottleneck to the other executions since there is no slot sharing across different operators and any adaptation decision, whether task re-assignment (Section 4.5.1), operator scaling (Section 4.5.2), or query re-planning (Section 4.5.3), ensures no bandwidth contention.

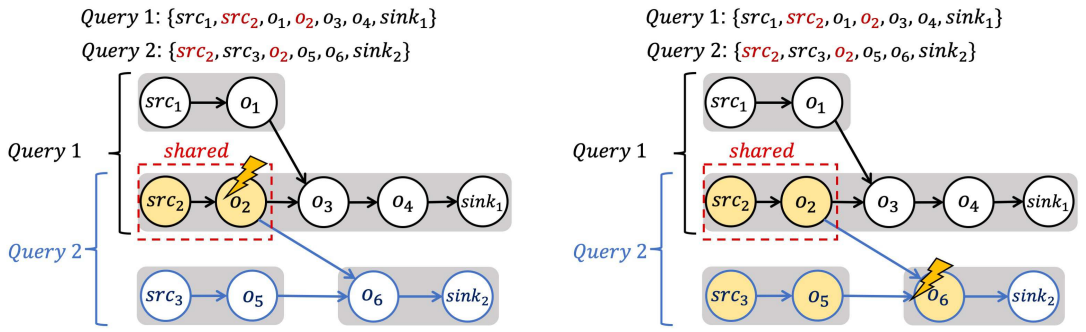
Pruning Module

After adapting some of the bottleneck operators $\{o_1, \cdots, o_n\}$, `Nako` will update the availability of the resources and check whether there are other operators still need to be adapted ($\theta - \{o_1, \cdots, o_n\}$). Some of the operators may no longer need to be adapted if any of the prior adaptations have released the previously contended resources. For example, if both o_1 and o_2 are competing for the same network bandwidth, B, with a rate of $\lambda[o_1] = 0.6B$ and $\lambda[o_2] = 0.7B$ respectively, both queries may be identified as unhealthy by the *Runtime Diagnosis* since the aggregate resource demand is higher than the network bandwidth availability ($0.6B + 0.7B = 1.3B > B$). However, if o_1 is migrated to a different site, o_2 no longer needs to be adapted as migrating o_1 has freed up $0.6B$ of the bandwidth. In this case, the *Pruning Module* will remove o_2 from θ. The *Adaptation Policy* will re-compute the adaptation action for the remaining operators, following the same process.

5.4.2 Adapting Shared Execution

In some cases, multiple queries may exhibit a common sub-execution, whether in consuming common input data streams or even performing similar operations [**?**, **?**, **?**]. This is especially common for queries that are submitted by different groups within the same organization that analyze the same data for different purposes. For example, user-access logs from multiple CDN servers may be analyzed for billing, advertisement, and security purposes. Chapter 3 presents a multi-query optimization technique that allows queries to share their common operations in order to reduce the overall resource consumption of processing and transmitting duplicate data in the context of wide-area streaming analytics. Yet, this introduces new challenges to the adaptation policy since adapting an execution that is shared by multiple queries may affect the execution performance of all of the queries.

Adapting a directly shared operator. Figure 5.5 shows an example where the execution of Query 1 is shared with Query 2. When a bottleneck happens directly on the operator that is shared by the two queries (o_2), this will affect the execution of both queries. There are several adaptation options to resolve the bottleneck (Figure 5.5(a)). First, the adaptation module can simply adapt the operator without considering its



(a) Bottleneck on a directly shared operator.

(b) Bottleneck on an indirectly shared operator.

Figure 5.5: Bottlenecks on a shared query execution.

shared property. This, however, will disrupt the execution of all of the queries that are sharing the operator. Furthermore, this may affect the execution performance of some of the queries. For example, if the bottleneck in Figure 5.5(a) is caused by the limited network bandwidth between o_2 and o_6, migrating o_2 to a different site may increase the execution latency of Query 1 despite its individual execution is not being constrained by the resources. Alternatively, the adaptation module may split the shared execution by instantiating a new instance of o_2 for Query 2. However, this will increase the overall resource consumption, both for acquiring additional computing resources and increasing the network bandwidth consumption for transmitting duplicate data streams. Thus, there is an overhead vs. resource consumption trade-off between maintaining the shared property of an execution and splitting the execution.

Adapting an indirectly shared operator. Bottlenecks may also happen on an execution that is indirectly shared by multiple queries. Figure 5.5(b) shows an example where bottleneck occurs on one of the Query 2's operators (o_6) whose upstream operators are partially shared with Query 1. This will affect the execution of the shared operators as o_6 will trigger a backpressure to its upstream operators to reduce the workload. The backpressure control messages will be propagated to the source operators. In this case, both src_2 and o_2 will observe the backpressure from o_6 and reduce their workload. Hence, this will affect the overall execution of Query 1. Although adapting o_6 should not affect the performance nor the resource consumption of Query 1, adapting o_6 requires suspending the execution of all of its upstream operators, including those that are

shared by Query 1. Thus, the execution of Query 1 will need to be suspended despite its execution is not being constrained.

Cost of Adapting a Shared Execution

Based on the above observation, we can see that adapting a shared execution while maintaining its sharing property may affect the runtime executions of all queries that are sharing the execution. Specifically, it needs to suspend the executions of all of the queries. Thus, the overhead cost (C_O) of adapting a shared execution should be computed as the aggregate overhead of adapting *all* of the sharing queries Q. Furthermore, adapting a directly-shared operator may also increase the overall network bandwidth consumption (C_R) of all of the sharing queries. Thus, the costs of adapting a shared operator are computed as follows:

$$C_O = \sum_{q \in Q} C_O[q] \qquad\qquad C_R = \sum_{q \in Q} C_R[q]$$

Alternatively, the adaptation module may determine to split the shared execution. This will reduce the overall overhead cost (C_O) as it does not need to suspend the executions of the other queries, but it tends to increase the resource consumption cost (C_R) both for acquiring additional computing resources and transmitting duplicate data over the network. Thus, `Nako` carefully determines whether to maintain the sharing property of a bottleneck execution or split the execution based on the overall adaptation cost of the two adaptation options. Specifically, it chooses the adaptation action that incurs the lowest overall cost:

$$min\Big(C_{adapt}^{share}, C_{adapt}^{split}\Big)$$

In general, the overhead cost will dominate the overall cost of adapting a shared execution while the resource consumption cost will dominate the overall cost of splitting a shared execution. Thus, adapting a shared execution typically incurs higher adaptation cost compared to adapting a non-shared execution. Thus, `Nako` often prioritizes adapting non-shared executions whenever possible. Intuitively, this reduces the number of executions that needs to be suspended while ensuring efficient resource consumption.

5.5 Experimental Evaluation

Environment and System Setup. We evaluated `Nako` using an emulated testbed derived from a real wide-area system deployment. It consisted of 16 nodes: 8 edge nodes and 8 data center nodes. Each edge node was configured with 2-4 slots/node and each data center node was configured with 8 slots/node. Here, a slot was configured with 1 CPU and 1GB of RAM. We capped the pair-wise network bandwidth between the nodes and introduced network delay using a `tc` limiter tool based on the actual measurement of inter-site network connections among the Amazon EC2 data centers.

We have implemented `Nako` as an extension to `WASP`'s adaptation module on Apache Flink (Chapter 4). While `WASP` directly updates any unhealthy execution, `Nako` determines whether adapting only a subset of the queries is sufficient. It relies on the query adaptation cost to prioritize *which* queries need to be adapted. In this experiment, we enabled 30 seconds monitoring interval for the *Runtime Diagnosis* to determine whether a particular execution is constrained. We also set $\alpha = 0.5$ in all of the experiments, unless explicitly specified. This balances the overhead cost and the resource consumption cost in computing the adaptation cost.

Dataset and Queries. We have implemented 8 location-based streaming analytics queries over real geo-tagged Twitter trace that was collected from Twitter Streaming APIs in December 2015. We scaled the playback rate to approximately 12,000 tweets/second to reflect the actual tweet rate. The tweets were distributed across the 8 edge nodes based on the geographic information embedded in each tweet. Thus, the trace covers the actual distribution of real geo-distributed workload. Table 5.1 shows the details of the queries. Each query keeps track of the *source offsets* of all of its input stream sources, which indexes the last event that has been successfully consumed by the query. Any processing state (e.g., source offsets and aggregation results of windowed grouped aggregation operators) are periodically checkpointed to a local file system every 30 seconds.

In addition to the data stream itself, some of the queries also rely on data dictionaries that are statically stored in a specific site. Examples of such data dictionaries include user demographic information, stopword list, and sentiment dictionary. The user demographic information was randomly generated and stored in one of the data center

Table 5.1: Nako query details.

ID	Description	Operators
Q1	Get user demographic information	map, window, join, project
Q2	Filter language specific tweets	map, filter, union, project
Q3	Find top 10 popular events in each country	map, filter, union, window, reduce
Q4	Get the user sentiment in each country	map, filter, union, window, join, reduce
Q5	% of Christmas-related events per country	map, filter, union, window, reduce
Q6	Monitor the workload of each site	map, window, reduce
Q7	Find top 10 popular topics per age group	map, filter, union, window, join, reduce
Q8	Find top 10 popular topics per gender	map, filter, union, window, join, reduce

nodes. The stopword dictionary was distributed across the data center nodes based on the commonality of the language in each location. Lastly, the sentiment dictionary was collected from the SenticNet [**?**] dataset, and it was used for sentiment analysis query. Each query subscribed to 4 input stream sources located at the edge nodes and stored its final result locally.

Metrics. We consider 3 main metrics in our experiments:

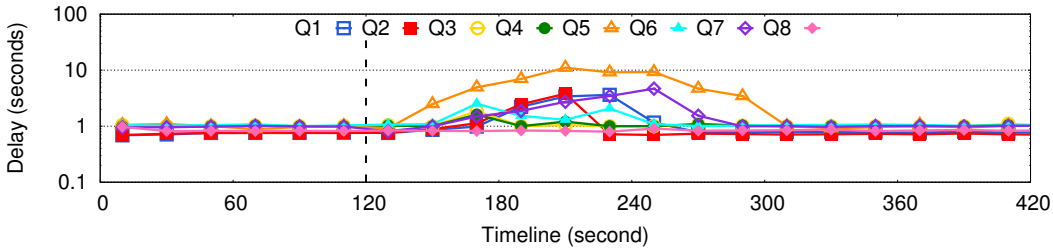
1. *Execution Delay.* The delay is measured as the average *event latency* which is the difference between the time an event is emitted at the sink and the time it was generated by the external source. In the case of windowed grouped aggregation, the event generation time is set to the maximum event time of all events within a particular window (the latest event within a window) [**?**].

2. *Constrained Time.* The total amount of time a query spends under a constrained condition, i.e., its execution latency is higher than its average execution latency when the execution is not constrained.

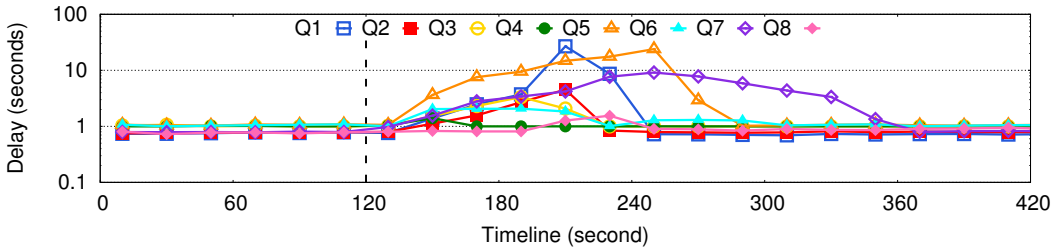
3. *WAN Bandwidth Consumption.* The total rate of inter-site network bandwidth consumed by a query execution.

5.5.1 SLO vs. Cost-based Adaptation

We first evaluated `Nako`'s adaptation policy in determining *which* queries to adapt based on the adaptation cost metric. As the baseline, we compared `Nako` against an SLO-based adaptation policy (`SLO` for short) that was proposed in Henge [**?**]. It prioritizes adapting queries based on an SLO violation metric. In this experiment, we defined the SLO metric



(a) Nako's query execution delay. Q2 and Q7 were adapted.



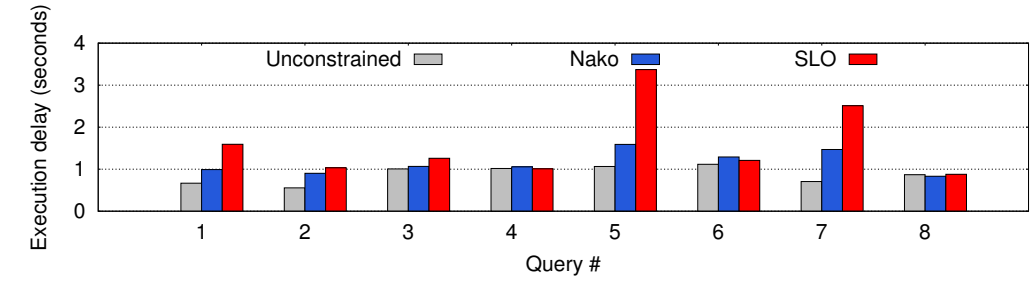
(b) SLO's query execution delay. Q1 and Q5 were adapted.

Figure 5.6: Per query execution delay over time.

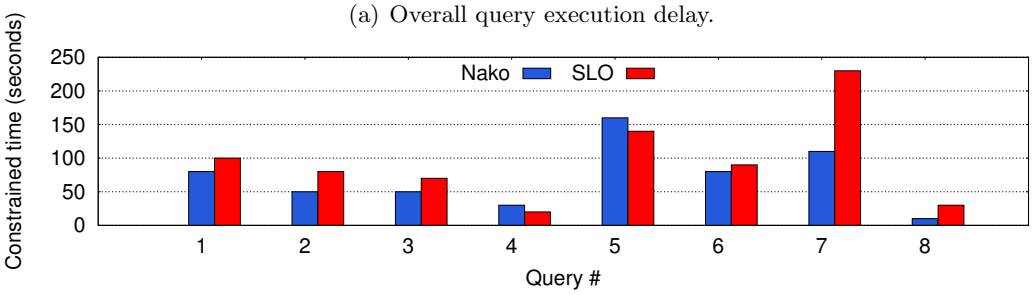
based on the relative increase in the overall query execution delay with respect to the average query execution delay when the execution was not constrained. Thus, queries that had higher increase in delay would have higher priority for being adapted first. For fairness reason, we enabled the pruning module of both approaches, allowing them to adapt only a subset of the bottleneck queries.

In this experiment, we introduced network bottleneck by increasing the event rate of each input stream source by 25% at $t = 120$, which caused some of the network links to be contended by the queries. Figure 5.6(a) and Figure 5.6(b) compare the overall delay increase per query between `Nako` and `SLO` respectively. We made a few observations here. First, we observed that both `Nako` and `SLO` adapted only two queries although there were six queries that were marked as unhealthy by the system. Specifically, `Nako` adapted Q2 and Q7 while `SLO` adapted Q1 and Q5. Thus, adapting only a subset of the bottleneck executions was sufficient to resolve the bottleneck.

Secondly, we observed the effect of adapting different queries. The adaptation decisions made by `Nako` were based on the estimated time required to migrate intermediate states across different sites, which in this case took ∼0 and 6 seconds for Q2 and Q7



(a) Overall query execution delay.



(b) Time spent under constrained conditions.

Figure 5.7: Per query delay and constrained time comparison between Nako and SLO.

respectively. Here, adapting Q2 incurred 0 adaptation overhead as its execution is stateless: Q2 only needed to filter out language specific tweets and hence did not need to store any intermediate processing state. On the other hand, Q7 maintained an intermediate processing of the total tweet counts per topic per age group. In contrast to `Nako`, `SLO` instead adapted Q1 and Q5 since they had the highest increase in delay among the contending queries. In this case, adapting Q1 and Q5 incurred 7 and 13 second state migration time respectively. Although adapting Q1 and Q5 eventually resolved the bottleneck, it significantly increased the overall execution delay of Q5. Specifically, it resulted in 2.1× higher overall query execution delay compared to the execution delay of Q5 in `Nako` (Figure 5.7(a)).

Thirdly, comparing the execution performance of the other non-adapted queries between `Nako` and `SLO`, we can see from Figure 5.7(a) that `Nako` was able to maintain the overall delay closer to the unconstrained condition whereas the `SLO` resulted in a higher overall execution delay (especially for Q5). Figure 5.7(b) breaks down the time spent by each query under the constrained condition. Here, we can see that `Nako` resulted in a lower overall constrained time even for queries that were not adapted, explaining the lower overall execution delay it achieved in Figure 5.7(a).

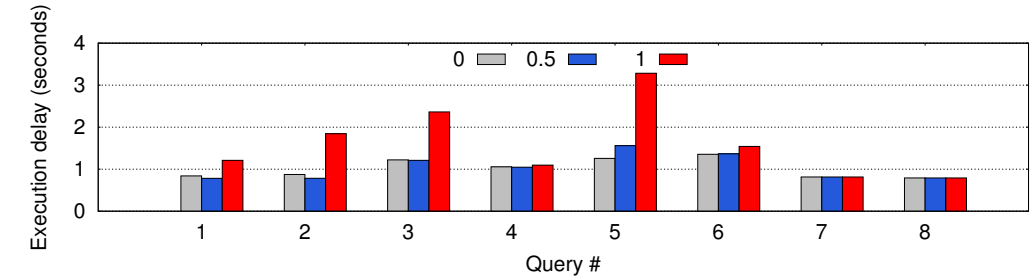
These results show that (1) adapting only a subset of queries can be sufficient to handle bottlenecks caused by resource contention among multiple query executions, and (2) the decision on *which* queries need to be adapted can affect the overall query execution performance. Specifically, we show that `Nako` can handle bottlenecks more efficiently compared to the SLO-based adaptation, resulting in a lower overall delay and lower constrained time.

5.5.2 Resource Consumption vs. Overhead Trade-off

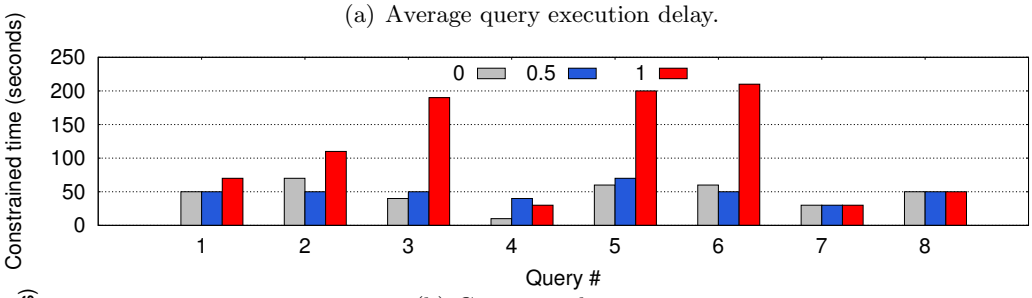
We next evaluated the overhead vs. resource consumption trade-off in adapting a query execution. Specifically, we varied the α value in estimating the adaptation cost (Section 5.3) from 0, 0.5, and 1. At one extreme, setting $\alpha = 0$ would minimize the overhead cost without considering the resource consumption cost, while $\alpha = 1$ would minimize the resource consumption cost regardless of the overhead. In general, setting $\alpha = 0.5$ balances the resource consumption and the overhead cost factors in adapting a query execution. In this experiment, we deployed all of the 8 queries concurrently and introduced dynamics by randomly varying (1) the bandwidth capacity by a factor of 0.7 to 0.9, and (2) the workload by 0 to 20% every 2-5 minutes.

Figure 5.8 compares the overall execution performance and the network bandwidth consumption of different α values. First, we can see from Figure 5.8(a) and Figure 5.8(b) that $\alpha = 0$ resulted in a lower delay and lower constrained time compared to $\alpha = 1$. However, we can also see from Figure 5.8(c) that $\alpha = 0$ consumed higher network bandwidth that $\alpha = 1$, especially for Q3 and Q4 (27% and 25% additional bandwidth respectively), whose operators were scaled in $\alpha = 0$. Comparing $\alpha = 0.5$ with the two extreme cases, we can see that $\alpha = 0.5$ followed the same adaptation decision as $\alpha = 0$ for Q3 but, followed the adaptation decision made by $\alpha = 1$ for Q4. In the former case, $\alpha = 0.5$ sacrificed higher network bandwidth consumption for query execution performance due to the high overhead of adapting Q3 which would have resulted in a similar increase in query execution delay and constrained time with $\alpha = 1$. On the other hand, it decided not to scale Q4 since the resource consumption cost of adapting Q4 was higher than the overhead cost.

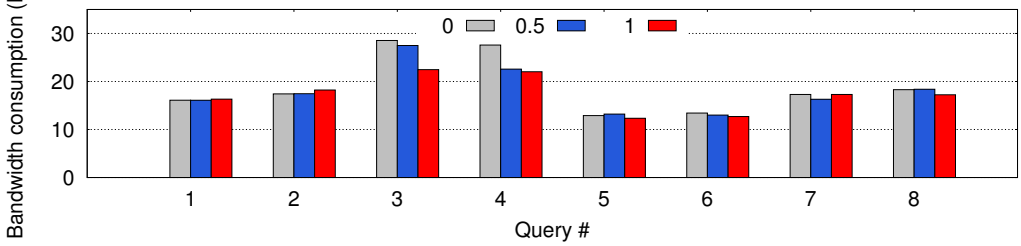
From these results, we can see that there is a clear trade-off between minimizing the overhead cost and the resource consumption cost. The decision on *which* cost to



(a) Average query execution delay.



(b) Constrained time.



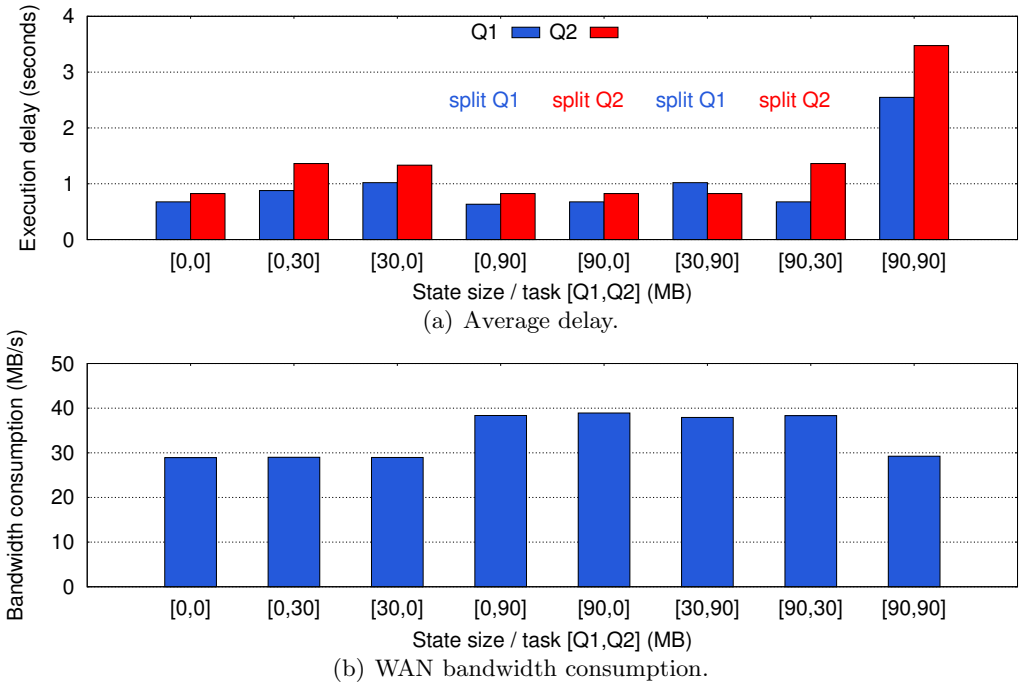
(c) WAN bandwidth consumption.

Figure 5.8: Overhead vs WAN bandwidth consumption trade-off.

minimize depends on the optimization goals. For example, if ensuring the timeliness result of a query is critical, the adaptation module should give a higher weight for the overhead cost. On the other hand, if the resource consumption incurs high monetary cost or the analyst is constrained by her resource budget, she may give a higher weight on the resource consumption cost. In general, balancing the weight between the two cost factors may be desirable in most cases since it considers the benefits between the two optimization goals.

5.5.3 Shared Query Adaptation

In the following set of experiments, we evaluate `Nako`'s adaptation policy in adapting an execution that is shared by multiple queries. Specifically, we observe how `Nako` decides



(a) Average delay.

(b) WAN bandwidth consumption.

Figure 5.9: Adapting/splitting shared execution over different combination of state size.

whether to maintain the sharing property of a shared execution or split the execution based on the overhead and the additional resource consumption trade-off between the two adaptation decisions. Here, we considered two queries: Q1 and Q2 that partially shared 2 out of 4 of their input stream sources. We fixed the size of the state maintained by each task of the two queries to 0, 30, and 90 MB. We first deployed the queries to run for 5 minutes and then introduced network bottleneck by halving the outbound network bandwidth used by the shared operators.

Figure 5.9(a) and Figure 5.9(b) show the overall execution delay and the network bandwidth consumption rate of the two queries over different combinations of state sizes. We observed that if the two executions were stateless, i.e., adapting the shared execution did not require migrating any intermediate processing state, `Nako` would not split the shared execution, as doing so would not affect the overhead cost but increased the resource consumption cost from transmitting duplicate data stream. In this experiment, we can also see that `Nako` did not split the shared execution for the [0,0], [0,30], [30,0], and [90,90] cases since the resource consumption cost of splitting the execution

over-weighted the overhead cost. In the case of [90,90] splitting the execution would only benefit one of the queries but it would increase the overall resource consumption cost from acquiring additional computing slot and increasing the network bandwidth consumption for transmitting duplicate data streams for the two executions. On the other hand, `Nako` split the shared execution for the other cases, resulting in approximately 30% higher bandwidth consumption compared to the initial shared execution (Figure 5.9(b)). However, `Nako` was able to maintain the low query execution delay by adapting only one of the queries that incurred the lower adaptation overhead, without disrupting the execution of the other query. For example, in the case of [30, 90], `Nako` split Q1 instead of Q2 as the former incurred lower overhead.

These results show that (1) there is an overhead vs. resource consumption trade-off between maintaining and splitting a shared execution, (2) `Nako`'s decision on whether to maintain/split a shared execution depends on the overhead differences among all of the sharing queries, and (3) `Nako` is able to selectively determine which query to adapt when splitting a shared execution that minimizes the overall adaptation overhead, resulting in a lower overall query execution delay.

5.6 Related Work

Wide-area Streaming Analytics. Recent work has addressed the importance of adaptability in wide-area streaming analytics. Heintz et al. [**?**] proposed a technique that considers the trade-off between accuracy and timeliness in the context of windowed grouped aggregation. Kumar et al. [**?**] focus on delay vs. WAN consumption trade-off by using a TTL-based approach for a windowed grouped aggregation. JetStream [**?**] allows users to specify different degradation policies with a data-cube model. AWStream [**?**] relies on a profiling technique to determine which degradation policy to take to ensure a certain degree of accuracy. Lastly, Jonathan et al. [**?**] propose an adaptation technique that adapts the execution itself to maintain the accuracy/quality of the result. Although they are related to this work, they have mainly focused on a single query adaptation. In contrast, we focus on multiple query adaptation, specifically in determining *which* queries that need to be adapted in the presence of bottlenecks. Hence, their work is orthogonal and complimentary to our work.

Adaptability in Distributed Stream Processing Systems. There have been a large body of work that address the importance of adaptability in cluster-based stream processing systems [?, ?, ?, ?, ?, ?, ?]. However, they have focused on addressing computational bottlenecks in a cluster-based distributed stream processing systems. These techniques cannot be directly applied to handle dynamics in a wide-area environment due to the highly heterogeneous and limited network bandwidth. Furthermore, they have mainly focused on adapting a single query execution. Recently, Henge [?] was proposed to address the problem of adapting streaming analytics queries in a multi-tenant environment. However, they have focused on maintaining the execution within SLO threshold and left the SLO definition to the analysts. In this work, we show that the system should also consider the overhead and resource consumption overhead when adapting a query, which is critical in a resource-constrained environment.

Others have also looked at the importance of minimizing the adaptability overhead. Drizzle [?] reduces the synchronization overhead for *Bulk Synchronous Processing* model. Chi [?] relies on control mechanism to reduce the overhead of global synchronization. ChronoStream [?] partitions and distributes large states across multiple nodes to allow fast recovery. DS2 [?] predicts the scaling factor based on the expected processing rate of each operator for dataflow model. Although these techniques are related to our work, they do not account for network constraints. In wide-area environment, the overhead of migrating large states over WAN is significantly higher than the partitioning overhead, and hence our techniques focus on minimizing this overhead.

5.7 Conclusion

In this chapter, we propose `Nako`: an adaptation technique for multi-query executions in the context of wide-area streaming analytics. `Nako` can selectively adapt only a small number queries to resolve bottlenecks. It uses an adaptation cost metric to determine which queries to adapt, which is computed based on the overhead and the changes in resource consumption of an adaptation. We have implemented `Nako` by extending `WASP`'s adaptation module. Experimental evaluation shows that `Nako` can identifies a small set of queries to be adapted, resulting in 2.1× lower overall query execution delay in handling bottlenecks compared to a technique that adapts queries independently.

Chapter 6

Future Research Directions

6.1 Machine Learning for Data Analytics Systems

Resource management, cluster scheduling, and query optimization in data analytics systems are imperative to the overall resource utilization and execution performance of data analytics queries. The majority of these techniques, however, often result in various trade-offs. For example, most cluster scheduling techniques have been designed focusing on their generality but may sacrifice the optimality of the outcomes. In the context of geo-distributed data analytics, most job scheduling and adaptation policies often involve a trade-off between query execution time, the accuracy/quality of the results, and the overall resource consumption. Improving the outcome of these policies for a specific application or workload often requires various parameter tuning that may be cumbersome in practice.

Recent attempts have proposed the idea of incorporating machine learning techniques to improve the outcome of general-purpose techniques for specific applications, whether in the context of cluster scheduling [**?**, **?**, **?**] or data management systems [**?**, **?**, **?**]. We believe that a similar approach should be considered in geo-distributed data analytics systems. For example, to handle runtime dynamics in the context of wide-area streaming analytics, the adaptation module may rely on the historical workload profile and the resource-accuracy profile of each query to further improve the adaptation policy that is tailored for a specific query. Another, more specific application of applying machine learning techniques to our work is to automatically learn the best adaptation

method and adjusting the overhead-resource consumption trade-off in computing the adaptation cost. This can be applied on a per-query basis as different queries may have different characteristics, workload profiles, and SLOs. Thus, incorporating machine learning techniques to geo-distributed data analytics systems can further improve the overall query execution performance and system resource utilization.

6.2 Pushing Data Analytics Further to the Edge

The emergence of Internet of Things (IoT) applications in recent years has led to the recent developments of Edge Computing comprising of small edge devices located at the edge of the network [**?**, **?**, **?**, **?**, **?**, **?**]. In turn, this trend has resulted in recent attempts for pushing data analytics further to the edge for localized processing, whether to edge machines/Cloudlets, mobile/IoT devices, or a combination between them [**?**, **?**]. This not only improves the timeliness of the result but also improves privacy. However, deploying data analytics queries in such a dispersed environment imposes additional challenges due to the unique characteristics of the environments:

- **Heterogeneous computational resources.** The available compute capacity across edge devices is typically very limited. They typically consist of only a handful number of processing units as opposed to hundreds or thousands of computing machines in large cluster/data center environment [**?**, **?**]. Furthermore, the processing hardware across edge devices is even more heterogeneous. For example, some edge devices may not be equipped with GPUs that can significantly improve the performance of image/video processing. Thus, deploying analytics queries in a dispersed edge environment should account for the availability and heterogeneity of computational resources since they can significantly impact the overall query execution performance.

- **Public Internet connectivity.** The network connections between edge nodes and public Cloud or other edge nodes typically use public Internet that has even more constrained bandwidth than the inter-data-center connectivity. Recent reports from Akamai and Microsoft have shown that the public Internet connections between edge nodes or private Clouds to public Clouds have an average of less

than 10Mbps [**?**, **?**]. Thus, deploying analytical jobs in this environment should account for the strict network limitation.

- **Co-location between data analytics jobs and user-facing applications.** While analysts can typically provision a large number of resources in large clusters for their analytical jobs to run in isolation, this may not be feasible in a dispersed edge environment due to the limited availability of the resources at the edge. Furthermore, these resources are typically used by other user-facing services that directly interact with end-users. Thus, the system should treat the user-facing services as a *first class citizen* and ensure that any analytical job does not disrupt these services. Furthermore, the co-location of these applications may introduce additional dynamics as Internet workload and user-oriented services are highly dynamic and unpredictable in practice.

In the future we plan to address the above challenges and consider adapting our multi-query optimization (Chapter 3) and adaptability technique (Chapter 4) to support data analytics at the edge. Applying multi-query optimization that merges common executions between queries can be beneficial in such a resource-constrained environment since it will reduce the overall resource requirements by removing any redundant data processing and duplicate data transmission over low-bandwidth network links. However, sharing common execution between queries should not violate any of the queries' SLOs and it has to preserve the privacy of the data. Since most edge clusters/devices have limited resources, any query execution running on such devices is prone to dynamics such as workload variation. Yet, most edge analytics applications are latency- and accuracy-sensitive. Thus, it is critical for edge analytics systems to be adaptive in order to preserve the timeliness and quality requirements of the applications regardless of dynamics.

Chapter 7

Conclusion

Recent years have seen an increasing amount of data that are generated in a geo-distributed fashion. These data vary from user-generated information (e.g., tweets and photo/video uploads), sensor readings from IoT applications, and distributed log files (e.g., transaction logs and CDN server logs from multiple geo-distributed servers). Collectively analyzing these geo-distributed data is crucial for many operational tasks. Researchers from industries and academia have proposed a system model, called *geo-distributed data analytics system*, to efficiently analyze geo-distributed data. It comprises multiple computing machines/nodes that are distributed across multiple sites (data centers or edge clusters) and they are connected by wide-area network (WAN). The main goal of such systems is to provide a low-latency processing while ensuring a stable and reliable query execution.

Achieving a high-performance execution of various geo-distributed data analytics applications/queries is challenging for a few reasons. First, different applications may have different processing models and optimization goals. In this case, the system should be able to handle various types of applications while satisfying each application's requirements. Secondly, wide-area resources, both computing capacities across sites and wide-area network bandwidth, are scarce and highly heterogeneous [**?**, **?**]. Thus, the system should account for these limitations in optimizing and scheduling queries in order to achieve high-performance query execution while ensuring efficient resource utilization. Lastly, wide-area environment is highly dynamic [**?**]. This dynamism includes unpredictable workload variation, network bandwidth fluctuations, occurrence of stragglers,

and failures that are inevitable in large-scale distributed systems. Thus, geo-distributed systems should gracefully handle these dynamics in order to ensure a stable and reliable query execution.

This thesis addresses the above challenges faced by geo-distributed data analytics systems. We highlight our contributions as follows:

- Firstly, we propose a resource management system, called `Awan`, that addresses the problem of resource sharing between multiple data analytics frameworks in a wide-area environment. The goal of `Awan` is to elastically adapt the resource allocation of each framework while allowing each framework to schedule its jobs with high locality. We propose a lease-based resource sharing model and a variant of a delay scheduling technique that allow a framework scheduler to improve its locality scheduling by providing the future availability of computational resources. Experimental evaluation using a real geo-distributed system deployment shows that `Awan` outperforms existing cluster-based resource sharing techniques by increasing the number of tasks that can be scheduled locally by up to 28%, which improves the overall query execution time.

- Secondly, we explore the benefits of incorporating multi-query optimization in the context of wide-area streaming analytics. Our goal is to efficiently utilize the limited wide-area resources while ensuring high performance query execution. We propose a network-aware multi-query optimization technique, called `Sana`, that allows queries to share their common executions and eliminate any redundant data processing. Since most streaming analytics queries are long-running, our proposed technique optimizes multiple queries in an *online* manner by allowing new queries to share their executions *incrementally* without disrupting existing query executions. We further highlight the importance of WAN awareness in applying multi-query optimization, both in planning and scheduling multiple queries. We show that applying traditional multi-query optimization without WAN awareness may degrade the overall query execution performance. Experimental evaluation using a real wide-area system deployment across geo-distributed EC2 data centers shows that `Sana` results in 21% higher throughput while saving WAN bandwidth utilization by 33% compared to a WAN-aware, sharing-agnostic system.

- Thirdly, we address the importance of adaptability in wide-area streaming analytics to handle various wide-area dynamics. Such dynamics include unpredictable workload variations, network bandwidth fluctuations, stragglers, and failures. Our goal is to maintain a stable, reliable, and low-latency query execution while providing efficient resource utilization. We propose a WAN-aware adaptation framework, `WASP`, that allows queries to handle bottlenecks without compromising quality. `WASP` adapts queries through a combination of multiple techniques: (1) Task re-assignment: which re-assign the placement of operator instances to avoid network bandwidth limitation, (2) Operator scaling: which dynamically scales bottleneck operators within a site and across sites to handle computational and network bottleneck respectively, and (3) Query re-planning: which further re-evaluates the execution plan of a query. `WASP` can automatically determine which adaptation action to take depending on the types of queries, dynamics, and optimization goals. Experimental evaluation shows that `WASP` is able to handle wide-area dynamics with low overhead and without sacrificing quality.

- Finally, we extend the adaptation policy of `WASP` into a multi-query environment where multiple queries may compete for common resources. We demonstrate that adapting each query independently without considering the deployment of the other queries may lead to a sub-optimal adaptation which results in wasteful resource consumption and/or degrade the overall execution performance of the queries. We propose `Nako`: an adaptation module for multi-query executions in the context of wide-area streaming analytics. `Nako` uses an adaptation cost metric to determine which queries need to be adapted. The adaptation cost is computed based on the overhead and the resource consumption of adapting a query. `Nako` can selectively adapt only a small number queries to resolve common execution bottlenecks. Experimental evaluation shows that `Nako`'s adaptation policy results in a more efficient adaptation compared to an existing technique that adapts queries independently.