Wayne State University Dissertations

January 2018

System Support For Stream Processing In Collaborative Cloud-Edge Environment

Quan Zhang
Wayne State University, fd7710@wayne.edu

SYSTEM SUPPORT FOR STREAM PROCESSING IN COLLABORATIVE CLOUD-EDGE ENVIRONMENT

by

QUAN ZHANG

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

2018

MAJOR: COMPUTER SCIENCE

Approved By:

_____

Advisor   Date

_____

_____

_____

_____

DEDICATION

To my beloved family.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to those who supported and encouraged me in one way or another during the last six years.

First of all, I would like to give the deepest gratitude to my advisor, Dr. Weisong Shi, who has provided me with valuable guidance and endless patience. Dr. Shi taught me how to find interesting research problems and how to come out solutions step by step. His guidance encouraged me throughout the tough time in the Ph.D pursuit, as well as writing of this dissertation. He is not only a knowledgeable professor in academia, but also a kind and wise elder who impacts my career a lot. I cannot succeed without his support and guidance.

I also want to extend my thanks to Dr. Nathan Fisher, Dr. Daniel Grosu, and Dr. Song Jiang for serving as my committee members. Their professional suggestions on the prospectus are very valuable for me to improve my work. They also showed me how to be an excellent researcher and educator.

My sincere thanks also goes for Dr. Fangzhe Chang, Dr. Yang Song and Dr. Ramani R. Routray, and Dr. Mu Qiao, who provided me opportunities to join their teams as summer intern. Without their precious support, it would not be possible to conduct these research. Moreover, I would like to thank former and present MIST and LAST group members that I have had the pleasure to work with or alongside of. With these brilliant people, we had many exciting, stimulating, and thoughtful discussions in the laboratory.

Last but not least, I deeply appreciate the support, encourage and love from my mother Haoxue Li, my father Zhigang Zhang, and my girl friend Zhiying Lin. They have been a constant source of strength and always there for me through the good time and bad time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

In the big data era, the volume of data is increasing at an unprecedented speed along with the fast development of information technology. The data itself has attracted plenty of attentions of stakeholders due to the underneath valuable information the data contains. Cloud computing has been arguably used as the de facto computing platform for big data processing by the researchers and practitioners for the last decade. As a shared pool of configurable computing resource, cloud computing provides ubiquitous and on-demand access to the provisioned resources via three major services, infrastructure as a service (a.k.a., IaaS), platform as a service (a.k.a., PaaS), and software as a service (a.k.a., SaaS). The key promise behind cloud computing is that the data should be already hold in or being transmitted to the cloud and eventually be processed in the cloud. Based on such centralized data processing model, a number of batch [97, 109, 90, 62, 38] and stream [110, 5, 70, 12, 22, 81, 85, 112] data processing platforms have been proposed and they suit the cloud services well so far.

At the same time, we are entering the era of Internet of Things (IoT). According to the estimation of Cisco [41, 7], 50 billions things will be connected to the network by 2020 and generate 507.5 zettabytes (ZB) data per year due to the increasing machine-to-machine connections, which is 49 times greater than the projected data center traffic (10.4 ZB) for 2019. Given that scale, current cloud computing platform is not economic enough to process all data in a centralized environment in terms of the network bandwidth cost and response latency requirement. Instead, the emerging edge computing (a.k.a., fog computing [28], cloudlet [91]) referring to "the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services" [95], is more efficient to process data at the proximity of data sources. Edge computing decentralizes the data storage and performs data processing at the edge of the network (e.g., micro-datacenter [48], cloudlet [91]). To leverage the computation resource in the cloud for IoT applications, recent industrial systems employ message brokers

[68, 3, 60, 54, 1] and streaming processing engines to build IoT analytics, including IoT in Google [11], AWS IoT [6], and Quarks [4].

Given such large scale of deployed sensors and actuators, the volume and speed of data being sent to data centers has exploded due to increasing number of intelligent devices that gather and generate data continuously. The ability of analyzing data as it arrives leads to the need for stream processing. It has become essential for organizations to be able to stream and analyze data in real time. In particular, the ability to provide low latency analytics on streaming data stimulates the development of distributed stream processing systems (DSPS), that are designed to provide fast, scalable and fault tolerant capabilities for stream processing. Based on how data is processed, there are two different types of stream processing models. Continuous operator model, that processes incoming data as records, is widely used in most DSPS systems [5, 102, 85, 81, 17, 70, 12], while recently proposed frameworks [25, 56, 22, 52, 110] adopt batch operator model that leverage Mapreduce [38] programming model and treat received data as continuous series of batch processing jobs. The continuous operator model leads to shorter response latency but also lower throughput compared to batch operator model.

Taking the large-scale video surveillance system in a city as an example, it is essential for large-scale cities' ecosystem and increasingly becomes a prominent component for the public safety [72, 86]. Figure 1.1 illustrates the hierarchical geo-distributed infrastructure for large-scale video surveillance system. The infrastructure includes private clouds (i.e., varying size clusters at campus and police department) and edge devices (cameras themselves and processing units placed close to the cameras, e.g., gateways/PCs at home, at traffic intersections, in vehicles, on bodies) with heterogeneous hardware to perform video analytics tasks, while also using the processing in public cloud clusters. Cameras are connected using wired or wireless link (Wi-Fi or LTE) to the edge device or the private cluster. The network bandwidth of these connections is a critical resource since in most scenarios the uplink bandwidth is in sufficient to support huge amount of video streams to the private or
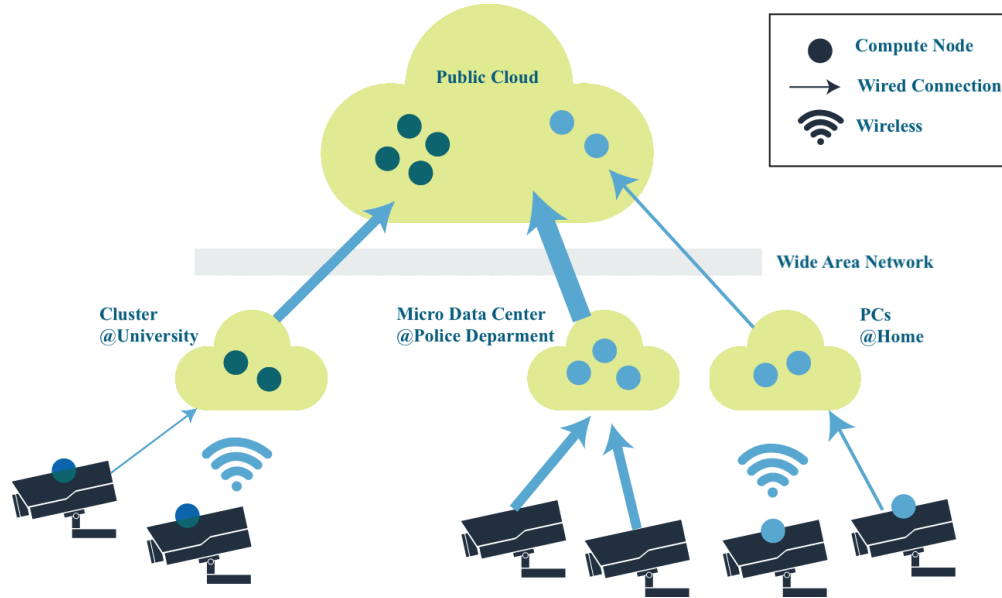
Figure 1.1: A high level overview of large-scale video surveillance system.

public cloud. In the era of smart and connected communities, a video surveillance system involves tens of thousand video cameras, such as Beijing and London have about one million cameras deployed [59], and provide video evidence and recognition for criminal and civil events [39]. Video cameras capture huge amounts of data in the manner of uninterruptedly operation for 7*24 hours. Intelligence approaches (e.g., face recognition [114], moving object detection/tracking [64] or behavior analytic [108]) are proposed to process video data at back-end in the cloud. For such a large-scale video surveillance system, there are two major challenges:

How to detect a failure camera in large-scale video surveillance system? In the current practice, a majority of failures such as blurred screen, artificial/nature occlusion, Rotary failure in pan/tilt) impacting on usefulness of video data, cannot be accurately identified and located in real time. When a failure occurs of edge cameras, the operation and maintenance team usually spends a lot of time to view these failures from video image, and then identify and locate the failure, which is manpower consuming and cannot meet real-time requirements

in a large-scale video surveillance system especially with the large number of geo-distributed edge cameras.

How to effectively and efficiently leverage the huge amount of video data? As aforementioned, in cloud computing the video data has to be transmitted to the cloud for further processing. However, given the zettabytes scale of video data, current cloud computing platform is not economic enough to process all data in a centralized environment in terms of the network bandwidth cost and response latency requirement. The emerging edge computing, that pushes data processing closely to the data source, provides an opportunity to migrate certain computing workload to edge devices and motivates cross-layer co-design for stream processing system in collaborative cloud-edge environment.

To explore and attack the challenges of stream processing system in such collaborative cloud-edge environment, we have proposed a series of empirical and theoretical research work. Figure 1.2 shows a high level overview of the collaborative cloud-edge environment, where the edge devices is connected to the gateways (i.e., edge cloud) and these edge clouds are connected to the cloud or the core of the Internet. Specifically, to automatically detect failure cameras in large-scale video surveillance system, we develop an hierarchical and hybrid anomaly detection model for multivariate time series. The performance metrics of an object forms multivariate time series and our anomaly detection model automatically selects the best model for different time series since the performance metrics exhibits different characteristics. To accelerate the anomaly detection process, we optimize the stream processing system (i.e., Spark Streaming) used in our detection model to reduce the end-to-end latency. To effectively and efficiently leverage the video data, we propose and implement a new computing framework, *Firework* that allows stakeholders to share and process data by leveraging both the cloud and the edge. We also expose programming interfaces for end users to facilitate the development of collaborative cloud-edge applications. A vision-based cloud-edge application is implemented to demonstrate the capabilities of *Firework*. These research topics are shown in Figure 1.3 given their corresponding relationship to the three
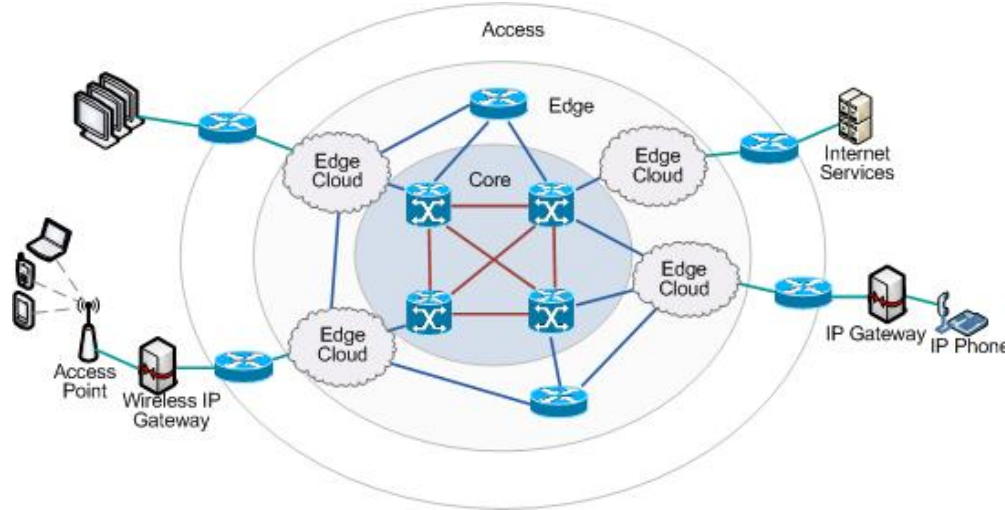
Figure 1.2: A high level overview of collaborative cloud-edge environment.

Figure 1.3: A high level overview of research topics in this dissertation.

layered service stack. In the following sections, we explain the motivation and challenges of each research component in Figure 1.3.

1.1 Outlier Detection for Multivariate Time Series

Outlier detection is widely used in real time stream analytics areas as fraud detection, financial analysis and system performance/health monitoring, network intrusion detection and etcetera. Many recent approaches detect outliers according to distance- or density-based measurements, which take only spatial feature into account. In this dissertation, we treat the data stream as multivaraite time series, which considers the temporal order as one of the significant factor for outlier detection.

Our motivation for exploring a hybrid and hierarchical method to detect anomalies in multivariate time series is multifold. First, the multivariate time series or variables exhibit different characteristics over time. While some variables show strong seasonality, others may not exhibit such temporal patterns. As a result, a single type of model is not appropriate to fit all the variables. Second, there are many candidate models for anomaly detection in multivariate time series. These models have different performance on different types of data. For instance, the seasonal-trend decomposition based method can be used to detect anomalies in time series exhibiting strong seasonality. However, it does not perform well when applied to time series without seasonality. Vector autoregression (VAR) is one of the most widely used models for multivariate time series analysis. It can be applied for anomaly detection when the predicted value is significantly different from the observation. But if the data does not exhibit time series characteristics, VAR cannot predict future values well because of the low goodness-of-fit to training data. On the other hand, distance based models, such as local outlier factor (LOF), may be able to better detect anomalies from such data, treating each data point as independent. Third, we are facing the problem of constructing a huge number of anomaly detection models for all the endpoints. Big data computing platforms, such as Apache Spark, make it scalable to construct hybrid models for a large amount of modeling objects in parallel and perform intensive model selection process.

1.2 Dynamic Sizing for Stream Processing System

Stream processing systems are also critical to supporting application that include faster and better business decisions, content filtering for social networks, outlier detection, and etcetera. Many streaming applications, such as monitoring metrics, campaigns, and customer behavior on Twitter or Facebook, require robustness and flexibility against fluctuating streaming workloads. Traditionally, continuous stream processing systems have managed such scenarios by i) dynamic resource management [53, 104], or ii) elastic operator fission (i.e., parallelism scaling in directed acyclic graph (DAG)) [62, 85, 90], or iii) selectively dropping part of the input data (i.e., load shedding) [36, 66, 101]. For batch based

stream processing system (i.e., Spark Streaming), dynamic batch sizing adapts the batch size according to operating conditions [37]. However current dynamic batch sizing method suffers long delay and overestimation for batch size prediction. In this dissertation, we focus on a batch-based stream processing system, Spark Streaming [110], which is one of the most popular batched stream processing systems, and minimize the end-to-end latency by tuning framework specified parameters in Spark Streaming.

Specifically in Spark Streaming, a batch size should guarantee that a batch could be processed before a new batch arrives, and this expected batch size varies with time-varying data rates and operating conditions. Moreover, depending on the workload and operating condition, a larger batch size leads to higher processing rate, but also increases the end-to-end latency. On the contrary, a smaller batch size may decrease the end-to-end latency while destabilize the system due to accumulated batch jobs, which means the data cannot be processed as fast as it is received. With the exception of batch size, the processing time of a batch job also significantly affects the end-to-end latency. With the same amount of available resources, less data processing parallelism (i.e., the number of blocks in a batch) may incur less execution overhead of task creation and communication but lower resource utilization, while massive parallel may dramatically increase the overheads even the resources may be fully utilized. By default, Spark Streaming adopts static batch size and execution parallelism (i.e., batch_interval/block_interval), which makes it possible that the system may involve any aforementioned issue. In this dissertation, we focus on both *batch interval* and *block interval* as they are the most important factors affecting the performance of Spark Streaming.

1.3 Data Processing and Sharing for Hybrid Cloud-Edge Analytics

Real-time video analytics is promising to significantly improve public safety, business intelligence, and healthcare&life science, among others. Video analytics from cameras installed on either fixed positions (e.g., traffic light, retail store) or mobile carriers (e.g., vehicles, smart phones) are used for traffic safety and planning, self-driving and smart cars,

surveillance and security, as well as user-centric applications including digital assistant and augmented reality [24]. We use object tracking in AMBER alert system as the motivation application to illustrate how edge computing could benefit the application and challenges of implementing the application in hybrid cloud-edge environment.

Conventionally, cloud-centric object tracking require that video streams must be preloaded to a centralized cluster or cloud, which suffers extremely high response latency and formidable cost of data transmission. The emerging edge computing can reduce the response latency and network bandwidth cost by analyzing video streams close to data sources. Figure 1.4 shows an example video analytics application of object tracking in AMBER alert system in hybrid cloud-edge environment. As shown in Figure 1.4, video streams are collected by various cameras and owned by different owners. The *video clipping* deployed at the edge scans video streams to selectively filter out frames with a license plate and send these frames to the cloud, so that compared to cloud-centric scenario, massive data transmission between the data sources and the cloud can be avoided. In the cloud, *license plate recognition & tracking* is used to recognize the text on the plate and tracking a target plate number. Note that the *license plate recognition & tracking* can also be collocated with *video clipping* at the edge, in which case the response latency and network cost can be further reduced by eliminating the communication between the edge and the cloud.

Although edge computing is promising for real-time video analytics, several barriers still prevent it from practical deployment. First, as shown in Figure 4.1, the data sources varies from on-body cameras to fixed security cameras so that it is hard to share the data in real-time manner with others due to privacy issue (e.g., faces captured by traffic/security cameras) and resource limitation (e.g., power/computation/network limitations of on-body or dash cameras). Second, services provided by the edge and the cloud (e.g., *video clipping* and *license plate recognition* in Figure 1.4) might be implemented in various platforms over geo-distributed computing nodes, which brings heavy overhead for developers to orchestrate data/services among different data sources and computing platforms. Third, programming
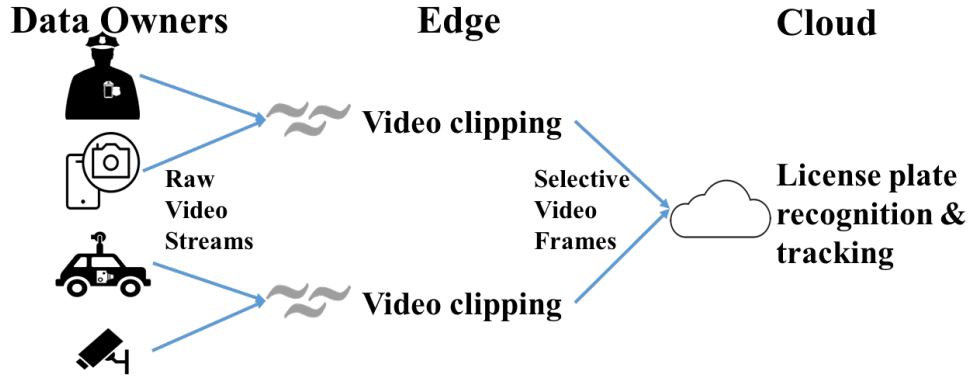
Figure 1.4: An example of real-time video analytics at hybrid cloud-edge.

in such cloud-edge environment is difficult considering the variant of data ownership, data format, APIs provided by data owners, and computing platforms. Therefore, in this dissertation we will focus on several major challenges of facilitating data sharing and processing in hybrid cloud-edge environment and summarize as following: 1) Applications in collaborative cloud-edge environment require fully control privileges over the data from multiple data sources/owners, which prevents data sharing among stakeholders due to privacy issue and resource limitation; 2) Although offloading from the cloud to the edge reduces response latency and network bandwidth cost, offloading of different applications depends on various polices and generates distinct intermediate data/service, which make the data/service orchestration much harder among multiple stakeholders and applications; and 3) Programming in edge computing is not as efficient as that in the cloud using MapReduce. The heterogeneity of edge devices increases the programming effort of developers and decreases the usability of exiting applications.

1.4   Edge Video Analysis for Public Safety

In the era of IoT, the widespread of cameras are deployed in either fixed locations (e.g., intersection, light pole, store) or mobile carriers (e.g., smart phones, vehicles), which makes video analytics an emerging technology. Cloud computing is not time- and cost-efficient for video analytics since it suffers long response latency and formidable cost of data transportation. Another important concern is privacy because video analytics requires

access to data owner's private data, which is unacceptable to send the data to the cloud. The proposed *Firework* leverages the resources of both cloud computing and edge computing to improve the performance of stream applications by shifting workload from the cloud to the edge. One goal of the proposed *Firework* is to optimize the response latency given the bandwidth and energy constrains. To show the potential benefits, we use EVAPS (Edge Video Analysis for Public Safety) in urban area as a showcase and to demonstrate the capability of *Firework*.

One potential use case of EVAPS is to find a lost object in the urban area, where all edge devices in EVAPS provide a video search interface. A user can spread the search request to all connected devices and do the search in real time fashion. When an object is reported as lost, it is very likely that this object is captured by cameras around at either fixed locations or mobile devices in the urban area. In cloud computing, the video data from the camera has to be uploaded to the cloud to locate the lost object. Data transmission is still costly, which makes it extremely difficult and inefficient to leverage the wide area video data. With *Firework* paradigm, a request of searching the lost object can be created at a user's device, where the feature extraction could be conducted. The feature vector then is distributed to all connected devices with cameras and each device performs the object searching using archived local data or real time video stream. We will implement an prototype of *Firework* framework and use the video analytics as a use case to show how *Firework* reduces the response latency and data transmission cost given the limited energy and network bandwidth constrains.

1.5   Objectives

Given the challenges and opportunities discussed above, this dissertation aims to provide system support for stream processing in collaborative cloud-edge environment. Specifically, this dissertation will accomplish the following objectives:

1. Explore a real-world stream processing workload that detects abnormal behaviors among millions multivariate time series that exhibit different characteristics (e.g., seasonal-

ity) over time. Extend the capability of outlier detection method for seasonal time series. Automate the model adaptation through a hierarchical selection process.

2. Explore the relationships of block interval and end-to-end latency, and batch interval and end-to-end latency in Spark Streaming. Develop an online heuristic algorithm to dynamically adjust the block and batch intervals so that the end-to-end latency is minimized.

3. Design a new computing framework, called *Firework* that: 1) allows stakeholders to share data without explicitly data transmission; and 2) leverages computing resources both in the cloud and at the edge to reduce the network bandwidth cost and response latency; and 3) provides programming interfaces for developers. Implement a prototype of the proposed computing framework and evaluate the prototype with a vision-based application.

4. Implement a vision-based hybrid cloud-edge application for public safety (i.e., vehicle tracking for AMBER alert system) using *Firework* framework. Demonstrate the capabilities of *Firework* by implementing customized tracking policies.

1.6 Our Approach

As shown in Figure 1.3, there are four major components to support stream processing in collaborative cloud-edge environment. In the following, we describe our approach for each component.

To handle multivariate time series exhibiting different characteristics of each variable, an outlier detection model should be able to fit variables with different models depending upon their time series characteristics. Therefore, we propose H_2O, which is a general hybrid and hierarchical outlier detection method for multivariate time series. For variables with significant seasonality, we develop a new seasonal-trend decomposition based anomaly detection method, considering the interactions among different variables in the form of common trends, which is more robust to outliers in the training data and can better detect true anomalies. For variables without seasonality, we select either time series based model to capture "global" patterns in the data, or distance based models which are more focused on "local" characteristics. H_2O selects the best models for variables following a "global to local"

fashion. Built on top of Apache Spark (i.e., SparkR and Spark Streaming), H_2O automates the model selection process and constructs millions of anomaly detection models in parallel.

To address aforementioned issues in Spark Streaming, we propose an online heuristic algorithm called DyBBS, which dynamically adjusts batch size and execution parallelism as the workload changes. This algorithm is developed based on two significant observations: i) the processing time of a batch is a monotonically increasing function of batch size, and ii) there is an optimal execution parallelism for each particular batch size to achieve the minimal latency. These two observations (addressed in Chapter 3.4) inspire us to develop our heuristic algorithm based on *Isotonic Regression* [31] to dynamically learn and adapt the batch size and execution parallelism in order to achieve low end-to-end latency while keeping system stability.

To attack aforementioned barriers of developing hybrid cloud-edge analytics, we propose *Firework* that: **i)** fuses data from multiple stakeholders as virtual shared data set that is a collection of data and predefined functions by data owners. The data privacy protection cloud be carried out by privacy preserving functions preventing data leakage by sharing sensitive knowledge only to intended users; **ii)** breaks down an application into subservices so that a user can directly subscribe intermediate data and compose new applications by leveraging existing subservices; and **iii)** provides an easy-to-use programming interface for both service providers and end users. By leveraging subservices deployed on both the cloud and the edge, *Firework* aims to reduce the response latency and network bandwidth cost for hybrid cloud-edge applications and enables data processing and sharing among multiple stakeholders. We implement a prototype of *Firework* and demonstrate the capabilities of reducing response latency and network bandwidth cost by using an edge video analytics application (i.e., vehicle tracking for AMBER alert system) developed on top of *Firework*.

1.7 Contributions

We summarize the contributions of this dissertation as following:

1. We propose and implement H_2O, a novel hybrid and hierarchical outlier detection method for multivariate time series. H_2O automatically selects the best anomaly detection models for each multivariate time series and reduces the false positives by considering the interactions among different variables. The experiments show H_2O outperforms the other detection method in both accuracy and recall.

2. We propose and implement DyBBS, an adaptive block and batch sizing algorithm on Spark Streaming to reduce the end-to-end latency. DyBBS is able to achieve lower latency compared to the state-of-the-art solution without any assumption of workload specific tuning or user program modification. The experiments also show DyBBS can adapt to time-varying workload and operating conditions.

3. We design and implement a prototype of *Firework*, a new big data processing framework for collaborative could-edge environment, which aims to reduce the response latency and data transition cost by leveraging the computing resources on the edge devices. We demonstrate the usage of the programming interface of *Firework* by implementing a vision-based hybrid cloud-edge application for public safety, which utilizes video analytics implemented on *Firework* to efficiently find a lost object in urban area.

1.8 Outline

The rest of this dissertation is organized as follows. Chapter 2 presents the related work of this dissertation. Chapter 3 introduces the H_2O of outlier detection for multivariate time series. The DyBBS will be discussed in Chapter 4. Chapter 5 describes the system design and prototype of our proposed *Firework* framework and a case study of a vision-based cloud-edge application. Finally, Chapter 6 concludes this dissertation and Chapter 7 describes our research plan for the future.

CHAPTER 2: RELATED WORK

In this chapter, we summarize research topics and prior works that are closely related to this dissertation. We categorize these prior works according to the hierarchy of this dissertation.

2.1 Outlier Detection on Time Series Streams

To detect outliers in time series, one of the most widely used method is to build statistical model for the historical data and then predict the value at future time point t [27]. An outlier is detected if the observed data is significantly different from the expected value. Several well known statistical models have been proposed to model time series, including autoregression (AR) for univariate time series, and vector autoregression (VAR) for multivariate time series [79]. [50] develop a multivariate autoregression method to detect anomalies in product and service ratings.

Seasonal-trend decomposition based method has been recently used to detect anomalies in time series with strong seasonality. A time series can be decomposed into the *trend*, *seasonal*, and *remainder* components [34]. Vallis *et al.* [103] propose a piecewise median decomposition method to detect anomalies in the cloud data at Twitter. Specifically, the trend component is estimated as a piecewise combination of short-term medians and the seasonal component is extracted by STL (Seasonal-trend decomposition based on Loess), a well known decomposition method in time series analysis [34]. The remainder is computed by subtracting the trend and seasonal components from the original data. A customized extreme studentized deviate test (ESD) [89] is finally applied to all the remainders to detect the top k anomalies. Decomposition based method has also been used in detecting anomalies at Airbnb's payment platform, where the seasonality is estimated by Fast Fourier Transform (FFT) and the trend is estimated by rolling median [78]. However, all the existing decomposition based anomaly detection methods are only focused on univariate time series. Greenaway-McGrevy [47] has made efforts in removing the seasonal component from multivariate time series in economics by estimating the trend component using a factor model,

which captures covariation and common changes among multiple time series. Motivated by the work in [47], in this dissertation, we develop a new decomposition based anomaly detection method for multivariate time series.

Another family of anomaly detection methods for multi-dimensional data is based on full dimensional distances to local neighborhoods. Local outlier factor (LOF) measures the local deviation of a given point with respect to its neighbors [30]. As variants of LOF, incremental LOF [84] is proposed to determine the LOF score instantly for new arriving data record and LoOP [69] scales the LOF score to [0, 1] which can be interpreted as the probability of a data point being an outlier. These methods can also be applied to multi-dimensional time series, which treat the time series at each time point as independent and do not consider their temporal characteristics. Instead of using full dimensional distances, some methods have been proposed to detect anomalies in subspaces to avoid the sparse nature of distance distributions in high dimensional spaces. The feature bagging method [74] randomly selects a set of dimensions as a subspace and then applies LOF under each subspace. Keller *et al.* [65] propose to compute and rank outlier scores in high contrast subspaces with a significant amount of conditional dependence among selected dimensions. Since anomaly scores are obtained from different subspaces, ensemble methods can be applied to combine the results and derive the final consensus [20, 116].

Several big data frameworks have been developed to detect anomalies in large scale data sources. Solaimani *et al.* [99] develop a Chi-square based method to detect anomalous time series in performance data at VMware based cloud data centers. The anomalies are time series, or data streams, instead of individual data points. The new arriving stream is compared with previous streams using Chi-square test [107], which compares if they follow the same distribution. Similarly, Rettig *et al.* [88] apply Kullback-Leibler divergence [71] and Pearson correlation to compare the distributions of two time series in order to detect anomalies over big data streams. Most recently, Laptev *et al.* [73] introduce EGADS, a generic and scalable framework for automated anomaly detection on large scale data at

Yahoo, which can detect three classes of anomalies: outliers, change points, and anomalous time series. Their framework automatically selects the best model for time series depending on their characteristics. All the aforementioned frameworks are only focused on detecting anomalies from univariate time series. H_2O, however, detects anomalies over large scale multivariate time series, considering the covariation and interactions among the variables. An ensemble of models are selected for all the variables in a multivariate time series. The model selection process is hierarchical, following a "global to local" fashion, which is first of its kind.

There has been much work on workload-aware adaption in stream processing systems. The work in [37] is the closest one to our work. In [37], the authors proposed adaptive batch sizing for batched stream processing system based on fix point iteration, which targets to minimize the end-to-end latency while keeping the system stable. It changes the batch interval based on the statistics of last two completed batches. This work is also designed for handling *Reduce-* and *Join*-like workloads, but it also handles *Window* workload. Compared to this work, our approach is different from it in two folds: i) Our approach dynamically adapts not only batch interval but also the block interval that is not considered in their work, and our block sizing method shows that it can further reduce the end-to-end latency compared than batch sizing only approach; and ii) For the batch interval prediction, our work utilizes all historical statistics so that the batch interval prediction is much accurate. Furthermore, our algorithm also eliminates the overestimation and control loop delay.

2.2 Stream Processing System

For continuous operator based stream processing system, DAG based task scheduling is widely used in systems including Dryad [62], TimeStream [85], MillWheel [22], Storm [102], and Heron [70]. One approach used to achieve elasticity in DAG is graph substitution. In DAG (or a sub graph of DAG), graph substitution requires that the new graph used to replace is equivalent to the one being replaced, and two graphs are equivalent only if they compute the same function. This approach is used in Dryad, TimeStream and Apache Tez

[90]. A similar elastic resource adjustment is based on the operators' process [45, 93, 23]. All of those methods identify the bottleneck of a system based on resource utilization. After that, the system increases the parallelism of the bottleneck nodes via either splitting (e.g. increase map operators) or repartition (e.g. increase shuffle partitions). In our work, we dynamically change the execution parallelism by block interval sizing, which requires no resource provisioning and partition functions. Load shedding discards part of received data when the system is overloaded [101, 36, 66], which is a lossy technique. Our approach does not lose any data as long as the system can handle the workload with certain batch interval while keeping the system stable. However, the presence of the load shedding technique is necessary when the processing load exceeds the capacity of system.

2.3   Data Sharing and Processing for Hybrid Cloud-Edge Analytics

In this section, we discuss the related work of edge computing in following primary areas, including service discovery in web service, stream processing system, mobile cloud, crowdsourcing, and edge computing.

Service Discovery: *Firework* leverages traditional techniques for service discovery, which are widely adopted in service-oriented architecture (SOA) [40] for web services. In SOA, a centralized service registry is used to hold service metadata for service providers and to lookup services for service consumers. To facilitate service metadata exchange, standardized XML or web service description language (WSDL) [29, 33, 32] are used to describe a web service. The data or service access is achieved by higher layer protocols, such as simple object access protocol (SOAP) [49] and RESTful interface. In addition, traditional SOA assumes the services are reachable and relatively stable (e.g., infrequently change of IP address) for consumers via directly requests using aforementioned access protocols. However, considering the heterogeneity of service providers (e.g., sensors and edge nodes) in *Firework*, it is rare that all of them define services with XML or WSDL and exchange data through SOAP or REST interface that are designed for small data access. A centralized service repository is also difficult to scale up in IoE scenario, where millions of service providers are geographically

distributed. Furthermore, SOA is dedicated for web service not for big data processing in distributed computing environment. To provide scalable service discovery, *Firework* adopts a layered distributed repository so that a service consumer can access remote services by only interacting with one service registry.

Stream Processing: To cope with unbounded continuous data, stream processing systems [102, 17, 110, 112] have been explored and employed massively. Storm [102] divides a streaming application into several stages and each stage is carried out by meshed and dedicated operators that adopt continuous operator model, which is the most similar to the data processing model in *Firework*. However, in most cases the operators in Storm are deployed in the same data facility running over homogeneous hardware platforms with centralized resource management, while *Firework* leverages heterogeneous resources so that operators could be implemented on different hardware and software platforms. *Firework* also provides layered distributed service management to support dynamic topology composition. Furthermore, the data in a Storm application cannot be shared with other applications at the operator level, while *Firework* achieves such data sharing by reusing exiting tasks. A recent work in [82] extends Storm so that it can be deployed at edge nodes that are close to the interacting objects (e.g., sensor, on-site database, backend database) of an operator, to reduce response latency. However, the proposed framework in [82] is application specific and the operators cannot be shared among multiple applications.

To provide comprehensive support for IoE applications, cloud-centric stream processing frameworks [11, 1, 6, 4] have been proposed. The shared underlying architecture is that the data generated by IoE devices are aggregated and delivered using data ingestion systems (e.g., Apache Kafka, MQTT, Amazon Kinesis Firehose) to analytics systems in the cloud. Analyzing all IoE data in the cloud suffers large data transmission latency and consumes a lot of network bandwidth. In *Firework*, the cloud can offload substantial amount of work to edge nodes to reduce both response latency and transmission cost.

Mobile Cloud: In mobile computing, several work have been studied to offload part of a task on mobile device to remote cloud. MAUI [35] proposes a fine-grained code offload with managed runtime management to improve energy efficiency at mobile devices. COSMOS [94] achieves high speedup by efficient resource allocation between mobile device and the cloud. Cuckoo [67] proactively executes methods at reachable cloud resources via intercepting the interprocess communication (IPC) in Android operating system and redirecting local method call to remote cloud, to improve performance and reduce energy consumption at smartphones. However, these systems cannot collaborate among multiple mobile devices.

Crowdsourcing: Mobile devices are widely used in crowdsourcing systems, which outsource a single task to a crowd of people for contributions. By leveraging online crowdsourcing platform, i.e., Amazon Mechanical Turk [2], a bunch of crowdsourcing tools have been proposed. TurKit [77] provides JavaScript-like programming language for Amazon Mechanical Turk to automatically schedule and price tasks, and accept or reject results. CrowdDB [44] extends SQL to embed human input into a query that only machines/databases can adequately answer. Since human labor is involved in corwdsourcing, different game theory based incentive mechanisms are studied in [106] and [115], which maximize a worker's profit, to raise the participation of more workers. In *Firework*, we focus on processing data without human efforts in loop but providing easy-to-use programming interface for users to interact with existing services available geographically.

Edge Computing: Inspired by low-latency analytics, edge computing [95] (a.k.a. fog computing [28], cloudlet [91]) is proposed to process data at the proximity of data sources. To leverage computing resources on edge nodes, mobile device cloud [42], femto clouds [51], mobile edge-clouds [43], and Foglets [92] have been proposed to orchestrate multiple edge devices for intensive applications that are difficult to run on a single device. Differencing from these systems, *Firework* leverages not only mobile devices and the cloud, but also edge nodes to complete big data processing task collaboratively, while aforementioned systems are not for large scale data processing and sharing among multiple stakeholders.

GigaSight [98] has been proposed as a reversed content distribution network using VM-based cloudlets for scalable crowd-sourcing of video from mobile devices. GigaSight collects personal video at the edges of Internet with denaturing for privacy that automatically applies contributor-specific privacy policy. The captured video in GigaSight is tagged for search and shared using network file system (NFS). However, GigaSight is designed to share video data and cannot apply video analytics functions. In contrast to GigaSight, *Firework* provides APIs for data owners to create customized video analytics functions, which can be used by a user to compose his/her IoE application.

Vigil [114] is a distributed wireless surveillance system that prioritizes video frames that are most relevant to the user's query and maximizes the number of query-specified objects while minimizing the wireless bandwidth cost. Vigil partitions video processing (e.g., object/face recognition or trajectory synthesis) between edge nodes and the cloud with a fixed configuration. Differencing from this prior work, *Firework* allows a user to define workload partitioning and deployment and provides dynamic workload migration (e.g., JVM migration) depending on the available resources on the edge nodes and the cloud.

Wang *et al.* [105] propose OpenFace that is an open-source face recognition framework to provide real-time face recognition/tracking by using edge computing (i.e., cloudlet [91]). Integrated with video stream denaturing, OpenFace selectively blurs faces depending on user-specific privacy policy. However, OpenFace leverages only edge computing whose computation is conducted on edge nodes. In contrast to OpenFace, *Firework* leverages both edge nodes and the cloud to reduce the response latency and network bandwidth cost. A programming interface is provided to manipulate data from multiple data sources.

Panoptes [63] presents a cloud-based view virtualization system to share steerable cameras among multiple applications by moving the cameras in a timely manner to the expected view for each application. A mobility-aware scheduler prioritizes virtualized views based on motion prediction to minimize the impact on application performance caused by camera moving and network latency. Zhang *et al.* [111] propose VideoStorm to support

real-time video streams analytics over large clusters. An offline profiler generates query-resource quality profile, and an online scheduler allocates resources to each query to maximize performance on quality and lag based on the quality profile. Resource demand for a query can be reduced by sacrificing the lag, accuracy, and quality of outputs. These prior works are orthogonal to *Firework*. Panoptes can be adopted by *Firework.View* (e.g., integrated with VS in license plate recognition) to provide customized camera views which are most relevant to user interests. The online scheduling algorithm of VideoStorm can also be used to adjust the resources allocated for a *Firework.View*. Furthermore, *Firework* can reduce impact on application performance (e.g., latency and network bandwidth cost) by carrying out the analytics on edge nodes, while both Panoptes and VideoStorm assume video data are preloaded in the cloud and clusters, which is infeasible given the scale of zettabytes data.

Ananthanarayanan *et al.* [24] present a geo-distributed framework for large-scale video analytics that can meet the strict requirements of real-time. The proposed framework in [24] leverages public cloud, private clusters, and edge nodes to carry out different computation modules of vision analytics. The prior works, Panoptes [63] and VideoStream [111], are integrated with the framework to optimize the resource allocation and minimize latency. *Firework* differs from [24] because our work expands data sharing along with attached computing modules (e.g., functions in a *Firework.View*) and provides programming interfaces for users to compose their IoE application.

2.4 Summary

In this chapter, we summarize related work for each proposed research areas of this dissertation, including anomaly detection methods, stream processing systems, and big data analytics systems in edge computing. In the following chapters, we will present the research results for each proposed research ares.

CHAPTER 3: OUTLIER DETECTION

Outlier detection is widely used in real time stream analytics areas as fraud detection, financial analysis and system performance/health monitoring, network intrusion detection and et cetera. In this chapter, we explore the outlier detection for data protection and propose a generalized outlier detection framework that can adapt to various detection objects. We believe that without loss of generality, the proposed framework can be also used for anomaly detection in large-scale video surveillance system.

As the most valuable asset for organizations, data is crucial to be carefully protected. Data protection or data backup[1] is the activity of creating a secondary copy of data for the purpose of recovery in case of accidental deletion of mission-critical files, application corruptions, hardware failures, and even natural disasters. In this big data era, with the rapid growth of data, every organization has a huge amount of objects to be protected, such as databases, applications, virtual machines, and systems. We refer to these objects as backup endpoints. According to a recent International Data Corporation (IDC) report [46], the worldwide data protection and recovery software market reached more than $6 billion in 2015. Many IT companies provide data protection services. They set up large scale storage infrastructures to manage backup services for many business clients, each of which may own hundreds of thousands of backup endpoints. Service providers are equipped with a single point of control and administration platform so that they can configure, schedule, and monitor all the backup jobs. It is the service provider's responsibility to make sure all the backups are operated under the service level agreement (SLA).

However, it is non-trivial to manage such a large and complex backup environment with a huge number of backup endpoints and millions of daily backup jobs. Backups have different types, such as full, differential, and incremental backups. Different endpoints have different backup schedules, for instance, some files need to be backed up hourly, some systems daily, and some applications maybe weekly. Clients may also request different backup

---

[1]we use the terms "data protection" and "data backup" interchangeably in this dissertation.

policies, for instance, certain applications need to have weekly full backups supplemented with daily differential backups. These endpoints exhibit quite different backup patterns and characteristics over time. Identifying abnormal backups from such a large and diverse set of backup jobs is very important to prevent data protection failures. For instance, some backup policies can be misconfigured by human during service update or transition process, which cause certain critical data to be accidentally excluded from the backup path. As a result, the total backup file size and job duration time can be greatly dropped. In another case, on backup servers with file expiration and versioning mechanism, if several backup versions are unexpectedly created within a very short time (e.g., significant increase of backup file count and size), the older backup version will expire and be deleted. If the old backup happens to contain important files which have not been backed up in the newly created versions, significant data protection failures can happen.

3.1 Preliminaries

In this section, we introduce the preliminaries of anomaly detection models used in our proposed H_2O method. Specifically, our models are from three categories: (a) seasonal-trend decomposition based anomaly detection, (b) vector autoregression (VAR) based anomaly detection, and (c) distance-based anomaly detection. Let $\mathbf{y}_t = (y_{1t}, y_{2t}, ..., y_{Kt})^t$ denote a time series at time t, which consists of K variables. The entire multivariate time series is denoted by $\mathbf{Y} = (\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_T)$, where T is the total number of time points.

3.1.1 Seasonal-Trend Decomposition based Anomaly Detection

Generally speaking, a time series comprises three components: trend, seasonal, and remainder. The trend component describes a long term non-stationary change in the data. The seasonal component describes if the series is influenced by seasonal factors or fixed and known period. The remainder represents the component in the time series after the seasonal and trend components are removed. Seasonal-trend decomposition has been used in detecting anomalies in univariate time series data. Suppose the vth time series, the trend component, the seasonal component, and the remainder component are denoted by $\mathbf{Y}_v$, $\mathbf{T}_v$,

$\mathbf{S}_v$, and $\mathbf{R}_v$, respectively, for $v = 1, ..., N$, where N is the total number of time series. We have

$$\mathbf{Y}_v = \mathbf{T}_v + \mathbf{S}_v + \mathbf{R}_v . \tag{3.1}$$

Many methods can be used for a seasonal-trend decomposition. One of the most well known methods is STL, a seasonal-trend decomposition procedure based on Loess (locally weighted scatterplot smoothing) [34]. After the decomposition, if the absolute value of the estimated remainder is significantly larger than the others, the corresponding data point is considered as an anomaly.

3.1.2 Vector Autoregression based Anomaly Detection

The vector autoregression (VAR) model is widely used in economics and finance for multivariate time series analysis. Each variable is a linear function of past lags of itself and the lags of the other variables. The VAR model of lag p, denoted by VAR(p), is defined as

$$\mathbf{y}_t = A_1 \mathbf{y}_{t-1} + ... + A_p \mathbf{y}_{t-p} + \mathbf{u}_t ,$$

where A_i are coefficient matrices for $i = 1, ..., p$ and $\mathbf{u}_t$ is the error term. A model selection method, such as Akaike information criterion (AIC), can be applied to select the optimal value of p. When VAR is used for anomaly detection, we can determine if the data point is abnormal on each variable by comparing its estimated value in $\mathbf{y}_t$ with the corresponding real observed value.

3.1.3 Local Outlier Factor

As a distance based anomaly detection method, local outlier factor (LOF) measures the deviation of a data point with respect to its density in the neighborhood formed by the k nearest neighbors [30]. Specifically, the local reachability density is proposed to estimate the degree of abnormality. The local reachability density of a given data point is defined as the inverse of the average reachability distance of its k nearest neighbors. Its outlier factor

is computed as the average of the ratios between the local reachability density of this data point and its k nearest neighbors. A data point is identified as an outlier if it has a higher outlier factor than its neighbors.

3.2 Proposed Method

In this section, we introduce H$_2$O, a hybrid and hierarchical method for outlier detection.

The three outlier detection methods introduced in Section 3.1 are representatives in their own categories. Given a time series, we first determine if there is seasonality in the data. Seasonal-trend decomposition based models can best handle such type of data. If the time series does not exhibit significant seasonality, we apply VAR to model the time series, which has shown flexibility in modeling multivariate time series without strong assumptions. If VAR cannot model the time series very well, as indicated by its fitted low R^2 values, we assume that the data does not have strong time series characteristics. Finally, LOF, a distance-based method focusing on local patterns, is applied to detect anomalies. Therefore, our model selection process is hierarchical, following a global to local fashion, and starts from checking the most strong pattern (i.e., seasonality) in time series. We explain the detailed steps in the following sections.

3.2.1 Determine periodicity or seasonality

We first determine the seasonality in each variable through a spectral analysis. After removing a linear trend from the series, the spectral density function is estimated from the best fitting autoregression model using Akaike information criterion (AIC). If there is a large maximum in the spectral density function at frequency f, the periodicity or seasonality will be $1/f$, which is rounded to the nearest integer.

3.2.2 Decomposition based Anomaly Detection (STL+DFA)

We apply seasonal-trend decomposition based method to detect anomalies on the variables with strong seasonality. One important issue in this type of anomaly detection model is the trend estimation. A sudden change in a time series is graduated over many time

periods. As shown in previous work [103], the trend estimation can be easily affected by such spikes in the training data, and therefore introduce artificial anomalies into the detection. In this dissertation, we develop a new seasonal-trend decomposition based anomaly detection model by considering the covariation and interactions among variables in the form of common trends. Specifically, we apply Dynamic Factor Analysis (DFA) to obtain the common trends among multiple variables. In practice, we find the trend estimated by DFA is more robust to the presence of outliers in the training data and less distorted by the sudden changes. Therefore, the number of false positives is significantly reduced.

DFA is a dimension reduction technique that aims to model a multivariate time series with K variables in terms of M common trends. The trend can be analyzed by univariate models by treating them as K separate trends. However, the interactions between variables are ignored. DFA aims to reduce the number of trends from K to M by considering the common changes in the variables. The general formulation for the dynamic factor model with M common trends is given by:

$$
\begin{aligned}
\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t &\sim MVN(0, \mathbf{Q}) \\
\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t &\sim MVN(0, \mathbf{R}) \\
\mathbf{x}_0 &\sim MVN(\pi, \Lambda) \,,
\end{aligned}
\tag{3.2}
$$

where $\mathbf{Z}$ contains the factor loadings, which is of dimension $N \times M$, and $\mathbf{x}_t$ is a vector containing the M common trends at time t. $\mathbf{w}_t$ and $\mathbf{v}_t$ are the error components at time t, following a multivariate normal distribution, with $\mathbf{Q}$ and $\mathbf{R}$ being the covariance matrix, respectively. $\mathbf{x}_0$ is the common trend at the initial time point, which also follows a multivariate normal distribution with mean π and covariance matrix Λ. The idea is that the response variables ($\mathbf{y}$) are modeled as a linear combination of common trend ($\mathbf{x}$) and factor loadings ($\mathbf{Z}$) plus some offsets ($\mathbf{a}$). An expectation-maximization (EM) algorithm can be applied to infer all the parameters [117]. The number of common trends can be determined by model selection,

such as AIC. Note that after the common trend $\mathbf{x}$ is estimated, it needs to be multiplied by the factor loadings $\mathbf{Z}$ to obtain the final trend for each individual variable.

After obtaining the trend components from DFA, we apply STL to extract the seasonal component of each variable. The remainder component in each variable is obtained by subtracting the seasonal and trend components from the original data. For each variable, we can determine if the data point is anomalous on that variable by examining its remainder component as introduced in Section 3.1.1. Since both STL and DFA are applied, we denote our new decomposition based anomaly detection method as STL+DFA.

3.2.3 VAR

For variables without strong seasonality, we apply vector autoregression (VAR) to model the multivariate time series, where the optimal lag value p is determined by AIC. We then obtain the fitted R^2 value for each variable. If the average value of all the R^2 values is smaller than a threshold, we assume that VAR does not fit the multivariate time series very well. Otherwise, we use the trained VAR model to detect anomalies as introduced in Section 3.1. Specifically, for each variable, we compare the estimated value and the real observation at time point t. If their absolute difference is above certain threshold, we consider the data point is abnormal on that variable.

3.2.4 LOF

Finally, for all the variables which have gone through the above steps, we apply the local outlier factor (LOF) model to identify anomalies. Specifically, LOF computes the distance using either multiple dimensions or a single dimension, depending on the number of variables left.

3.2.5 Ensemble Learning

We have selected the best outlier detection modes for a multivariate time series based on the exhibited time series characteristics from each variable. Specifically, if a group of variables exhibit seasonality, we apply STL+DFA to detect the outliers. If only one variable shows seasonality, STL is applied. Similarly, VAR is used for modeling multiple variables

and AR is used for modeling a single variable. Finally, we have an ensemble of selected models for each multivariate time series. Note that except LOF, all the other methods still detect anomalies on each variable although the covariation and interactions among multiple variables have been considered in the modeling process. A majority vote scheme is applied to finally determine if the multivariate time series at a certain time point is anomalous or not. Each variable has one vote in the ensemble learning. Since LOF can be applied to multiple variables using their full dimensional distance, if there are k variables in LOF, it will take k votes in the final voting. In Figure 3.1, we show one example of the entire model selection and ensemble learning process.
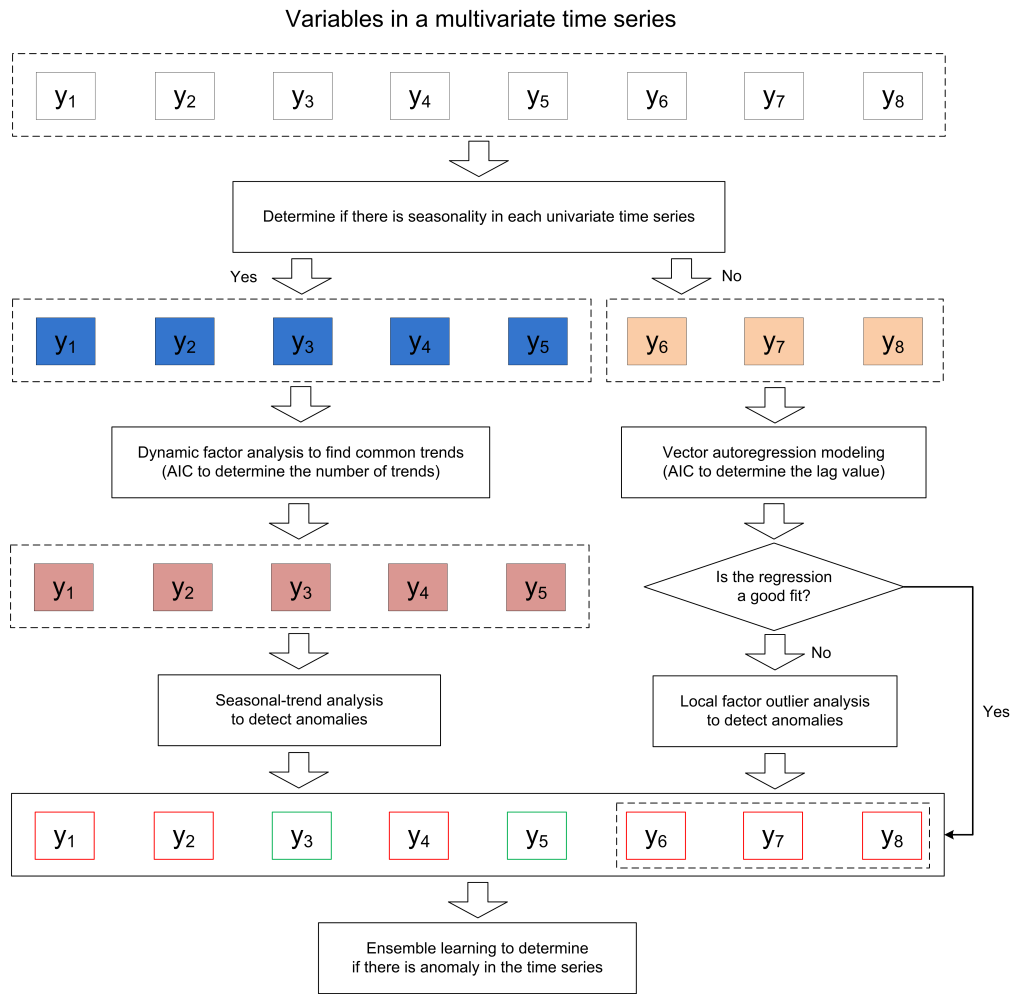
Figure 3.1: An H$_2$O running example

3.2.6 Apache Spark

Apache Spark has been widely used for large-scale data computation. Spark introduces an abstraction called resilient distributed datasets (RDDs), which is a fault-tolerant collection of elements that can be operated in parallel. Spark has an advanced Directed Acyclic Graph (DAG) execution engine that supports cyclic data flow and in-memory computing. As an extension of the core Spark engine, SparkR [87] provides an interface to execute R codes on Spark. To leverage the in-memory parallel data processing advantage of Spark, we implement H_2O in the R language and run it on Spark via the SparkR. Specifically, all the multivariate time series are abstracted as RDDs and assigned automatically to different compute nodes to construct large number of models in parallel. Figure 3.2 shows the overall framework of H_2O.
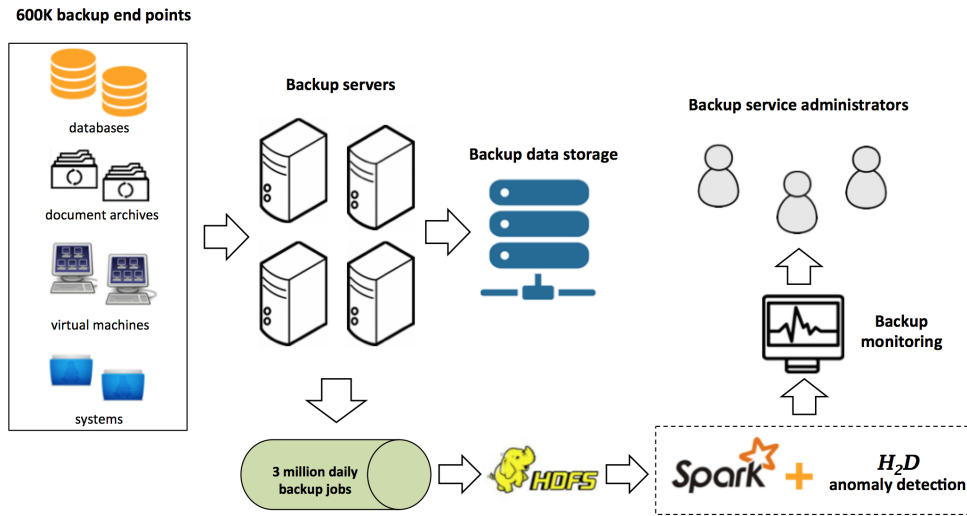
Figure 3.2: H_2O overall framework

3.3 Experiment

We evaluate H_2O on a real world data set from a large data protection service provider. We first introduce the data set and experimental setup. Second, we show that our new seasonal-trend decomposition based method using dynamic factor analysis can better detect true outliers and significantly reduce false positives, comparing with traditional decomposi-

Table 3.1: A backup job metadata record example

Attributes	BACKUP_ID	SERVER_ID	CLIENT_ID	TARGET_ID	LEVEL_ID	JobTimeLocal	ByteCount	ErrorCount	FileCount	JobDuration
Values	203399774	6221	1343910	/path/to/backup/location	Incremental	2016-01-01 16:26:18.01	3.854E10	0	17799	1395

tion based method. Third, we illustrate the importance of doing model selection and show that H_2O can automatically adapt to the dynamically changing characteristics of variables and always select the best models. We then compare the performance of H_2O with two baseline models through a simulation study. Lastly, we provide the distribution of selected models over time.

3.3.1 Data Set and Experimental Setup

The data set contains the backup performance metrics for about 420,000 unique backup endpoints, which have 81 million backup jobs running on 2,700 servers over a time window of 210 days[2]. Note that one backup endpoint may have more than one backup type. We want to detect anomalies from backup jobs of the same type. Therefore, for each backup endpoint, we further separate all of its backup jobs according to its backup types. Eventually, we obtain about 700,000 unique backup endpoint and type pairs. We need to construct an anomaly detection model for each pair. Therefore, 700,000 anomaly detection models need to be constructed in total. Every backup job has a time stamped performance record. One such record is shown in table 3.1. Our modeling target has a sequence of backup jobs running at different time points, whose performance metrics compose a multivariate time series. In our experiment, we select four backup performance metrics: byte count, file count, error count, and job duration. We denote these four metrics as v_0, v_1, v_2, and v_3, respectively. Therefore, the multivariate time series has 4 variables. All the multivariate time series have 120 data points unless otherwise specified. We have deployed H_2O on an Apache Spark (version 1.6.1) cluster with 20 nodes, where each node has an Intel Xeon E3-1260L CPU of 2.4 GHZ, 8 cores, and 16GB memory. We have every worker node host 8 executors, which has 1 core and 2GB memory. All the algorithms are implemented in the R language, which are running using SparkR [87] on the cluster.

---

[2]It is a subset of all the backup jobs in our running environment

3.3.2 Decomposition based Anomaly Detection

We first determine the seasonality in each variable using the R package *forecast* [61].

For multivariate time series with seasonality, we develop a new seasonal-trend decomposition based anomaly detection method using STL and dynamic factor analysis (STL+DFA), which considers the covariation and interaction among multiple variables in the form of common trend. Specifically, dynamic factor analysis (DFA) is applied to estimate the common trend. The number of common trends is determined using the Akaike Information Criterion (AIC), which is readily parallelizable. In this experiment, we empirically set it to be one to reduce the running time as we find that there is no significant difference in the trend estimation between one and more trends due to the relatively small number of variables. The seasonal component is still extracted by STL. We leverage the R packages *MARSS* [55] and *forecast* [61] to perform DFA and STL. The remainder are finally obtained by subtracting the trend and seasonal components from the original data.

Figure 3.3 and Figure 3.4 shows the seasonal-trend decomposition results of applying STL+DFA and STL to a multivariate time series. We show the decomposition results on v_0 and v_3, since only these two variables exhibit seasonal pattern, which are of seasonality 7 and 6, respectively. The seasonal components of STL+DFA and STL are omitted in Figure 3.3 and Figure 3.4, since they are identical, extracted using the same procedure. The k-σ rule in Gaussian distribution is applied to the remainder component of each variable to detect outliers. Specifically, we set $k = 2$, that is, if the remainder component is two standard deviations away from 0, the corresponding data point on that dimension is considered as abnormal. The higher the value of k, the more stringent the outlier detection. In Figure 3.3 and Figure 3.4, the dash lines in the residual plot indicates the threshold for $\pm 2\sigma$. A data point is finally identified as an anomaly if it is abnormal on both variables. As we can see, the trend components extracted by DFA are relatively smooth and less affected by the spikes. On the other hand, STL introduces seven false positives (marked with blue triangles) due to local
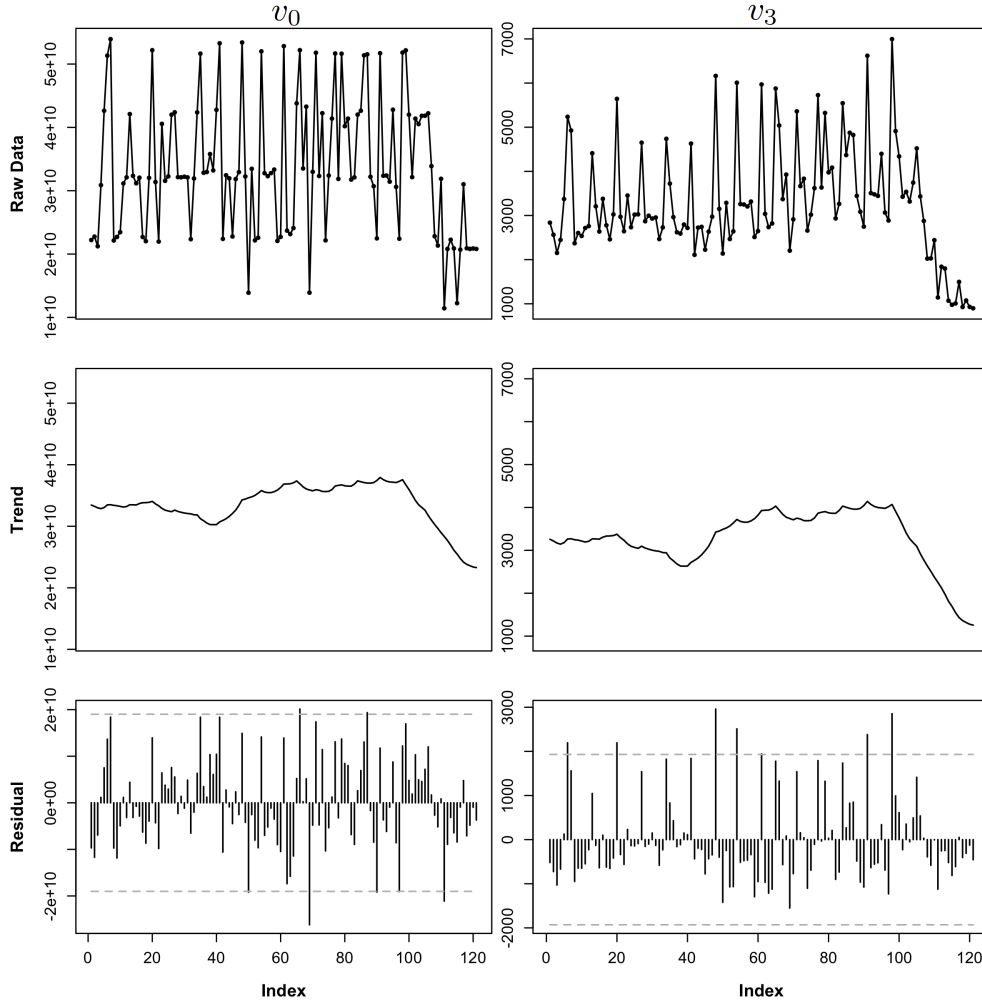
Figure 3.3: Seasonal-trend decomposition for multivariate time series using STL+DFA.

sudden changes in the trend estimation. Our new method therefore has better performance in detecting true positives than STL by considering the covariation and interaction among variables.

3.3.3 Model Selection

We illustrate in this section the importance of doing model selection. Specifically, we compare the performance of STL+DFA, VAR, and LOF on detecting anomalies in different time series. A sliding window method is applied, where all the models are trained on the historical data inside the window and used to predict whether the next observation is abnormal or not. Specifically, the window size is set to 45. The lag length in VAR is selected
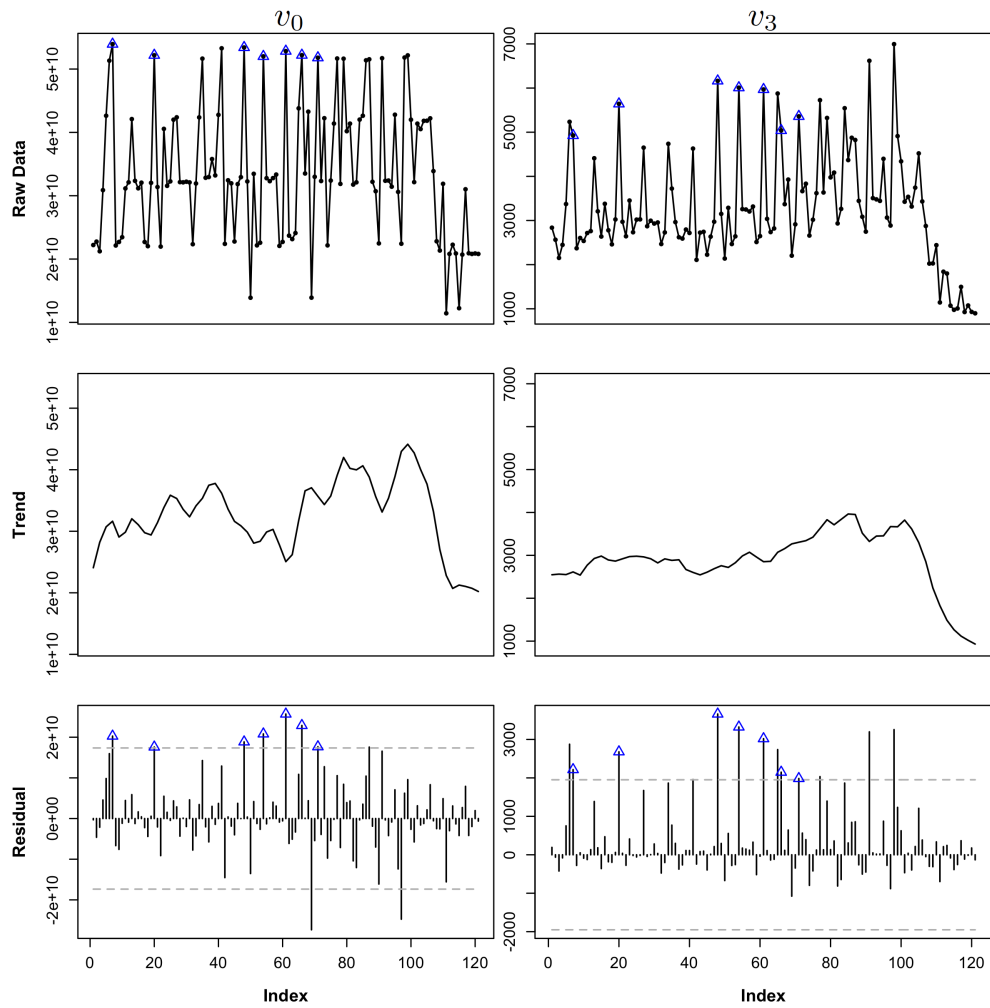
Figure 3.4: Seasonal-trend decomposition for multivariate time series using STL.

by AIC from a given range of values (the maximal lag length is set to be half of the length of the historical window). The goodness-of-fit of a VAR model is computed as the mean of the fitted R^2 values for each individual variables. We consider a VAR model is under-fitting if the goodness-of-fit is less than 0.5. LOF computes the neighborhood density using the full Euclidean distance between normalized data points. The number of nearest neighbors k in LOF is empirically set to be 10. We leverage the R package *vars* [83] for VAR modeling and the package *rlof* [57] for LOF. We use the same 2-σ rule to detect anomalies in VAR and LOF. Specifically, in VAR model, for each variable, we obtain the difference between the estimated value and the real observation at both historical and current time points. If the absolute difference at the current time point is significantly larger than the other values (more than 2σ away from the mean), we consider the data point is abnormal on that variable. Similar approach is applied to all the anomalous scores obtained by LOF.

Figure 3.5 shows the results of applying STL+DFA, VAR, and LOF to a multivariate time series with seasonality. The first 45 observations composes the first historical training window, which is indicated by the red vertical line. We only detect outliers from the observations on the right side of the red line. We consider the data point corresponding to the big spike in v_0 and v_3 as a true outlier. All the detected outliers are highlighted in Figure 3.5 only if they are abnormal on both variables. As we can see, STL+DFA successfully detects the true outlier (marked with red square) and no false positive is incurred. Although VAR and LOF detect the same true outlier, they introduce false positives as well. The reason that VAR introduces false positives is because its training is affected by the big spike in the historical data. LOF introduces false positives since it is a local anomalous measurement, and the data distribution in the local neighborhood may affect the result. Therefore, for multivariate time series with strong seasonality, STL+DFA outperforms VAR and LOF because it can better capture the seasonal pattern and is less affected by outliers in the training data.
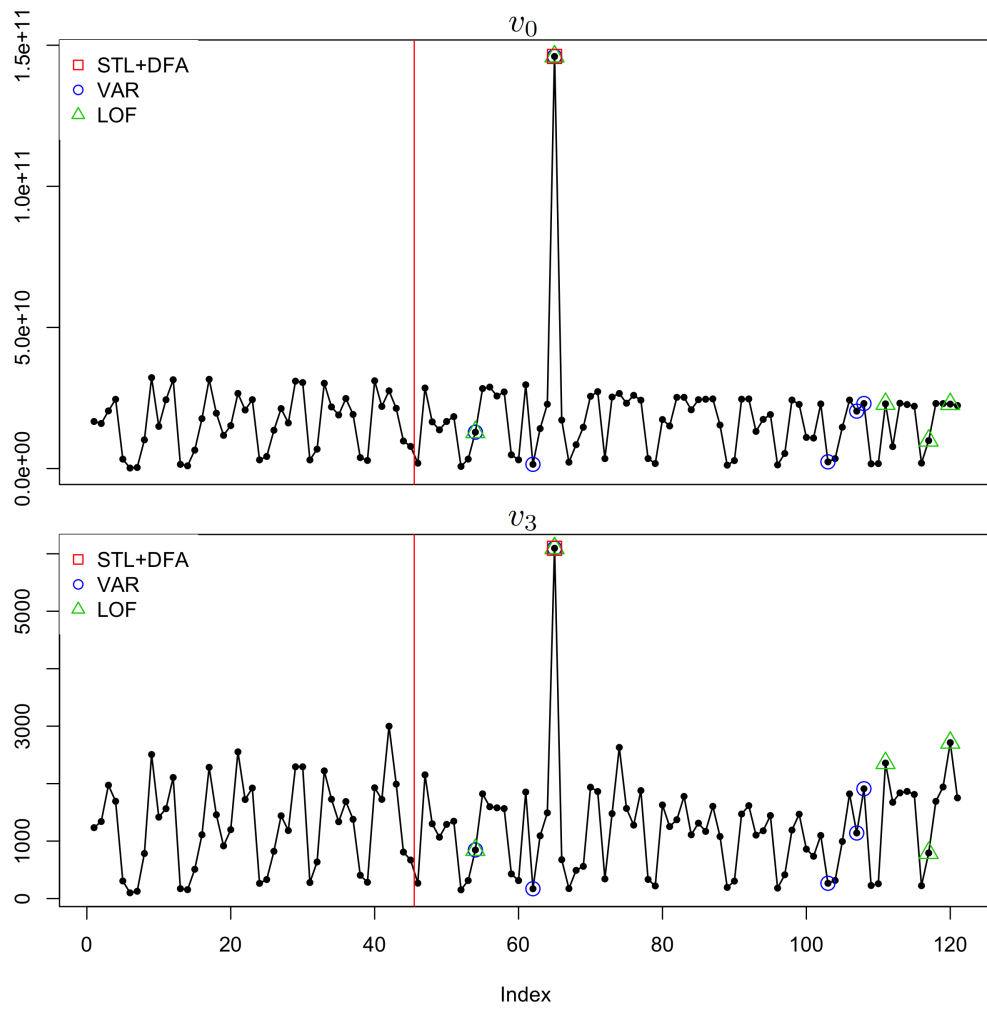
Figure 3.5: Outlier detection using STL+DFA, VAR, and LOF.

In Figure 3.6 and Figure 3.7, we compare the performance of VAR and LOF on multivariate time series without seasonality. The time series in Figure 3.6 has three non-seasonal variables, i.e., v_0, v_2 and v_3. We consider the spike after the first window (indicated by the red vertical line) as a true outlier. The VAR model fits the data well and detects the true outlier (marked with blue circle). LOF also detects the true outlier but introduces several false positives. On the other hand, Figure 3.7 shows one example where LOF outperforms VAR when VAR is under-fitting. We consider the data point corresponding to the big spike on v_0, v_2, and v_3 as a true outlier. Although the VAR model captures the true outlier, marked with the first green triangle from left in Figure 3.7, it also incurs four more false positives following the true outlier. LOF only introduces oner false positive, marked with the first green triangle from right in Figure 3.7, which is caused by the significant change in a single variable v_0.

As shown in Figure 3.5, Figure 3.6, and Figure 3.7, different methods have different performance depending on the time series characteristics. STL+DFA and VAR, as time series modeling based models, have good outlier detection performance when the data has strong time series characteristics while LOF is more focused on local patterns. Therefore, it is important to conduct model selection. In H_2O, we select the model in a hierarchical way, following a global to local fashion.

3.3.4   Model Evolution

H_2O selects models depending upon the exhibited characteristics of variables in multivariate time series. Since the variables have dynamic patterns, the selected models also vary over time. In Figure 3.8, we show one example of the dynamic evolving behavior of model selection in H_2O. Similar as before, a sliding window method is applied, where the models are selected based on the characteristics of 45 historical observations in the window. The red vertical lines indicate the time point where the selected model changes.

As shown in Figure 3.8, there is no strong seasonal pattern in all the variables in the beginning. VAR is selected first since it has shown high goodness-of-fit on all the variables.
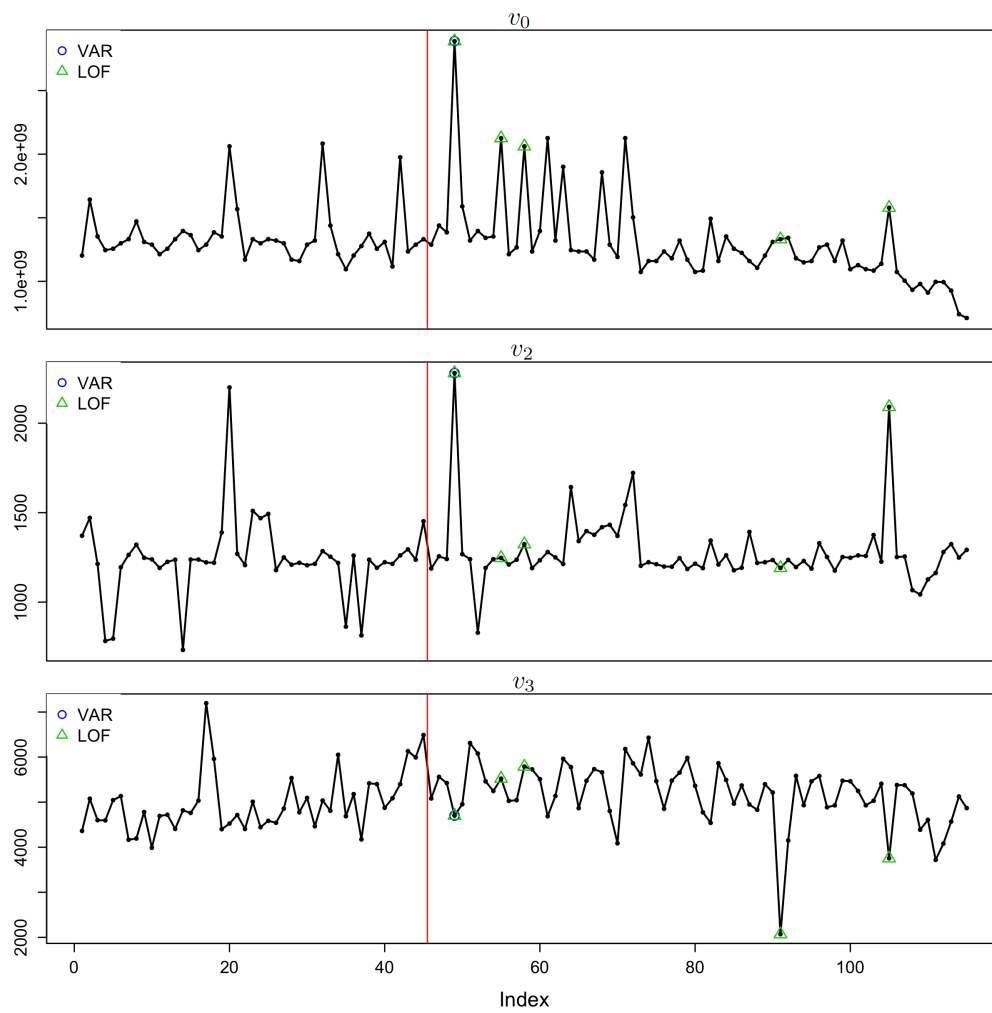
Figure 3.6: VAR outperforms LOF when VAR is well fitted.
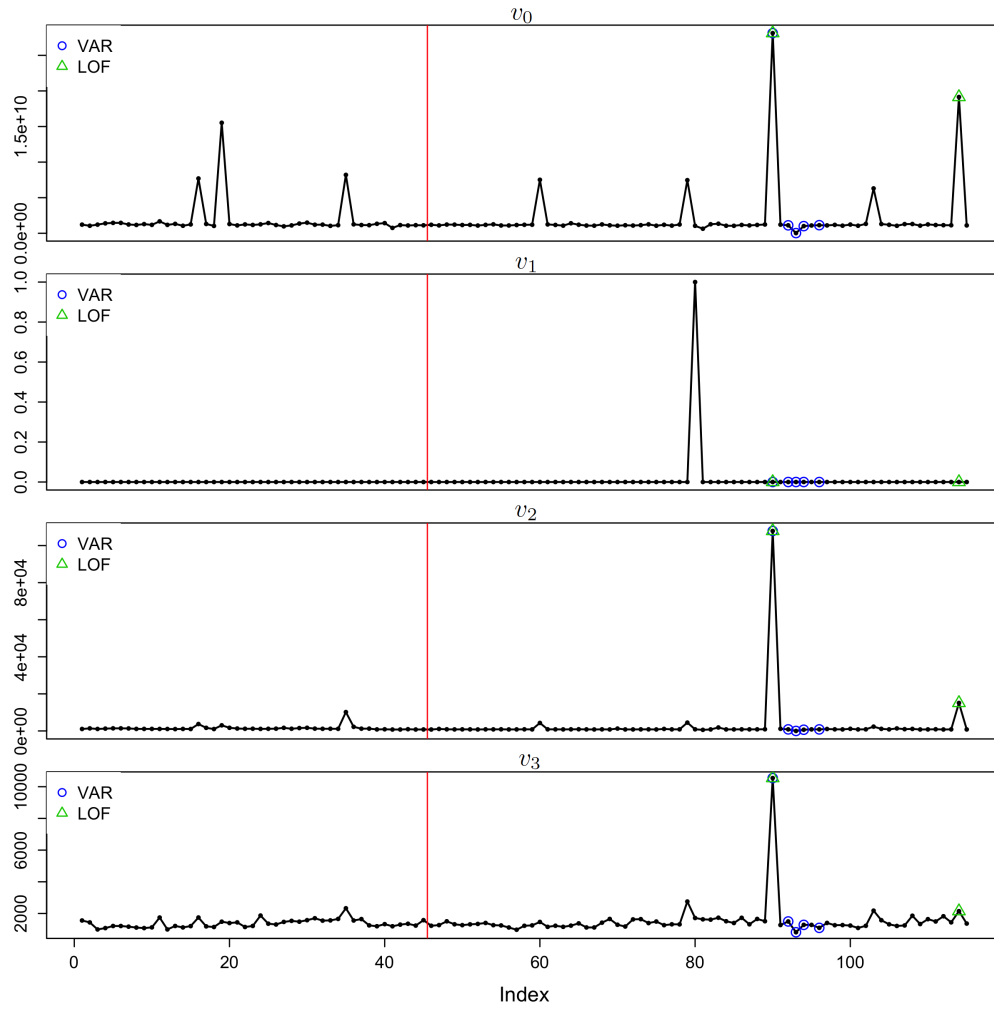
Figure 3.7: LOF outperforms VAR when VAR is not well fitted.
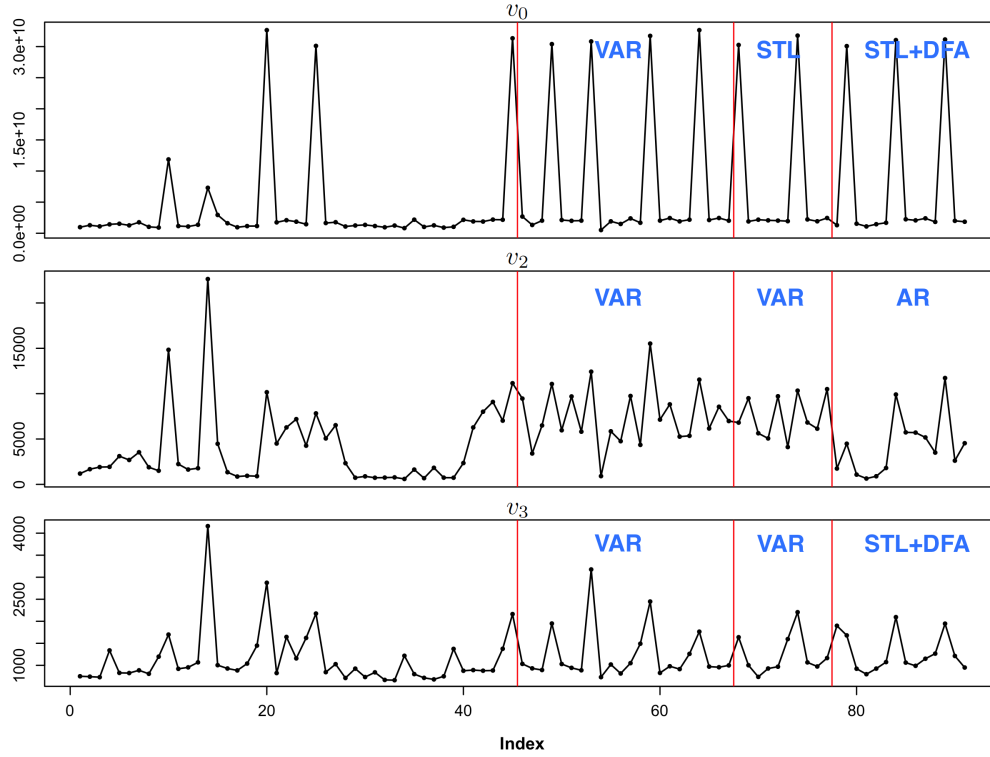
Figure 3.8: The evolution of selected model for a multivariate time series

As seasonality is detected on v_0 over time, H_2O automatically selects STL for v_0 since only variable exhibits seasonality. VAR remains selected for v_2 and v_3. Gradually, v_3 also exhibits seasonality. Therefore, STL+DFA is selected for v_0 and v_3 since two variables show seasonality. An AR model is selected for v_2 since only one variable left which does not show seasonality. Therefore, H_2O can automatically adapt to the dynamically changing characteristics of variables and always select the best models.

3.3.5 Simulation Study

We evaluate the performance of H_2O using a simulation study. We randomly select $100,000$ multivariate time series from the data set, each of which contains 46 data points. We use the first 45 data points as training data and predict if the $46th$ data point is an outlier. We artificially change the $46th$ data point to be either true outlier or noise. Specifically, a true outlier is defined as a data point which is abnormal on all variables of a multivariate time series (i.e., its value is significantly larger than the other data points), and a noise

is defined as a data point which is abnormal only on one variable. Among the $100,000$ time series, we randomly have one half of the $46th$ data points replaced by true outliers and the other half by noise. We compare the anomaly detection performance of H_2O as well as VAR and LOF on these time series. Note that the reason that we do not include a standalone seasonal-trend decomposition based method as comparison is because some variables do not shown seasonality in the multivariate time series. It does not make sense to enforce a decomposition on such variables. VAR uses the same majority voting schema as H_2O to determine the outlier while LOF can directly identify if a data point is an outlier by using the full dimensional distance. Figure 3.9 shows the performance of these three methods. As we can see, H_2O outperforms VAR and LOF on both precision and recall, since it models both seasonal and non-seasonal time series well via the hybrid and hierarchical model selection. The VAR model is applied to all the time series regardless of its goodness-of-fit. Therefore, the under-fitting VAR models induce both false positives and false negatives. The performance of LOF suffers from the mislabelling of noise, that is, introducing false positives. The dramatic change in one variable does not necessarily indicate an outlier in multivariate time series without considering the change in other variables. However, LOF may consider such noise as outliers due to the significant change in full dimensional distances. Overall, our proposed H_2O method improves the F1 score by 10% and 12%, comparing with LOF and VAR, respectively.

3.3.6 Model Distribution

In this section, we show the distribution of selected models for a large number of multivariate time series. We employ H_2O to determine the best models for each time series at a given time point. As the selected models vary over time, we have a weekly snapshot of the distribution of selected models for all the time series (in total around $330,000$) over ten consecutive weeks. Similar as before, the training window size is 45. Figure 3.10 shows the ten snapshots, where all the models are categorized into six categories. *CONSTANT* indicates that all the variables of a multivariate time series are constant within the training
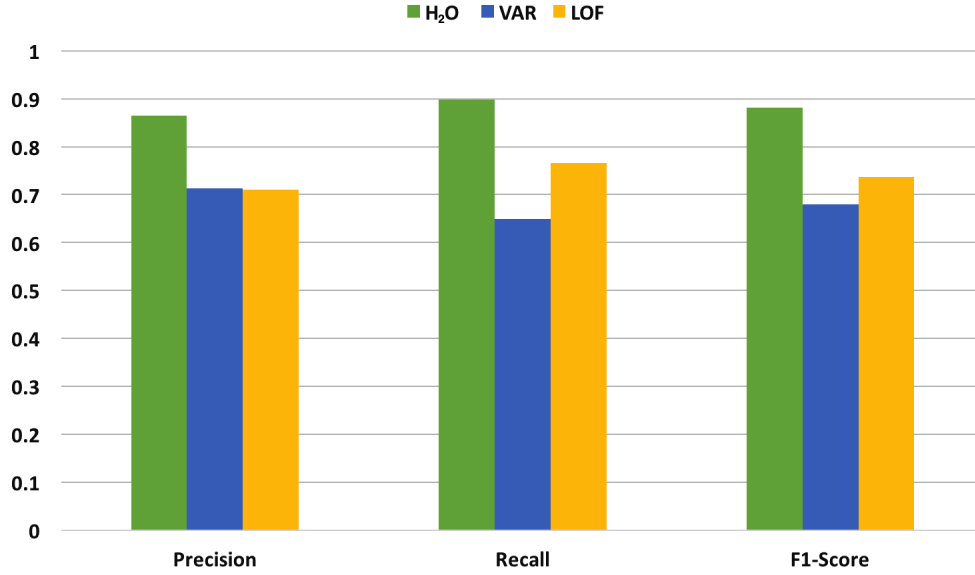
Figure 3.9: Outlier detection model performance in simulation study

window, where no anomaly detection is needed. If the seasonal-trend decomposition based method is selected for a time series, we label the selected model as STL+DFA, which also includes STL, for simplicity. Similarly, VAR indicates autoregression based outlier detection method, which also includes AR. *STL+DFA & VAR* indicates that both decomposition and autoregression based methods are selected. *STL+DFA & LOF* indicates both decomposition and distance based methods are selected. As we can see in Figure 3.10, the proportion of each category keeps almost the same with slight fluctuation over ten weeks. An ensemble of outlier detection methods from different categories, i.e., *STL+DFA & VAR* and *STL+DFA & LOF*, are selected for more than 40% of the multivariate time series.

3.4   Summary

In this chapter, we introduce H_2O, a hybrid and hierarchical outlier detection method for multivariate time series. Instead of fitting a single type of model on all the variables, we propose a hybrid method which employs an ensemble of models to capture the diverse patterns of variables. A hierarchical model selection process is applied to select the best anomaly detection models for variables based on their time series characteristics, following a
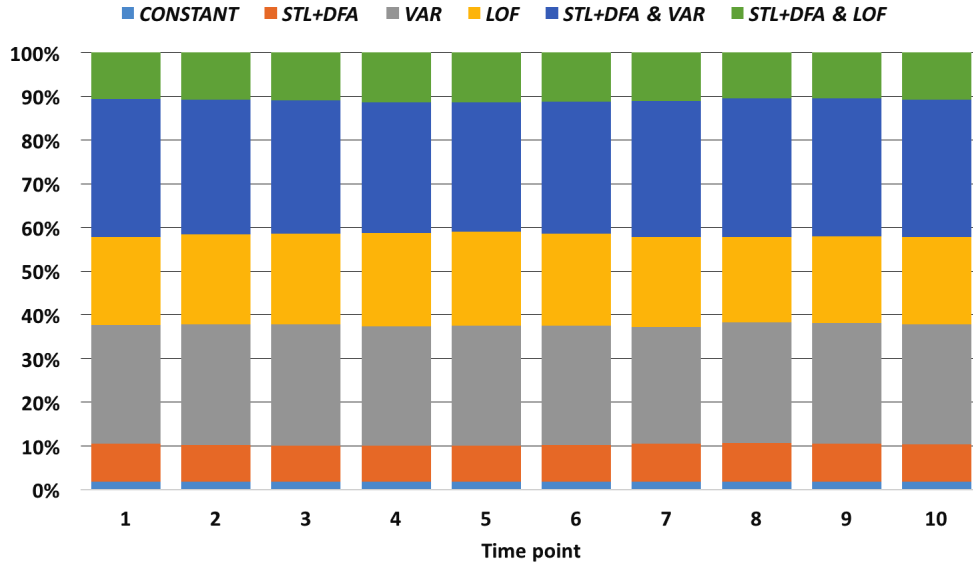
Figure 3.10: The distribution of selected models over ten consecutive weeks.

global to local fashion. We also develop a new seasonal-trend decomposition based detection method for multivariate time series, which considers the covariation and interactions among variables. Built on top of the Apache Spark, H_2O automatically selects and constructs a large number of anomaly detection models in parallel. Extensive experiments illustrate the robustness and superior performance of H_2O. Last but not the least, our H_2O method by its nature is very general and can be applied to detect anomalies over multivariate time series in many other domains, such as IT system health monitoring and fault detection.

To accelerate the detection speed, we further explore the stream processing system (i.e., Spark Streaming) used in H_2O and optimize it to reduce the end-to-end latency for each detection task. We will introduce the optimization of Spark Stream in the next chapter.

CHAPTER 4: DYNAMIC SIZING FOR STREAMING SYSTEM

The volume and speed of data being sent to data centers has exploded due to increasing number of intelligent devices that gather and generate data continuously. The ability of analyzing data as it arrives leads to the need for stream processing. Stream processing systems are critical to supporting application that include faster and better business decisions, content filtering for social networks, and intrusion detection for data centers. In particular, the ability to provide low latency analytics on streaming data stimulates the development of distributed stream processing systems (DSPS), that are designed to provide fast, scalable and fault tolerant capabilities for stream processing. Continuous operator model, that processes incoming data as records, is widely used in most DSPS systems [5, 102, 85, 81, 17, 70, 12], while recently proposed frameworks [25, 56, 22, 52, 110] adopt batch operator model that leverage Mapreduce [38] programming model and treat received data as continuous series of batch processing jobs.

In this dissertation, we focus on a batch-based stream processing system, Spark Streaming [110], that is one of the most popular batched stream processing systems, and minimize the end-to-end latency by tuning framework specified parameters in Spark Streaming. Ideally, a batch size in Spark Streaming should guarantee that a batch could be processed before a new batch arrives, and this expected batch size varies with time-varying data rates and operating conditions. Moreover, depending on the workload and operating condition, a larger batch size leads to higher processing rate, but also increases the end-to-end latency. On the contrary, a smaller batch size may decrease the end-to-end latency while destabilize the system due to accumulated batch jobs, which means the data cannot be processed as fast as it is received.

To address these issues in Spark Streaming, we propose an online heuristic algorithm called DyBBS, which dynamically adjusts batch size and execution parallelism as the workload changes. This algorithm is developed based on two significant observations: i) the processing time of a batch is a monotonically increasing function of batch size, and ii) there

is an optimal execution parallelism for each particular batch size to achieve the minimal latency. We develop our heuristic algorithm based on these two observations and leverage *Isotonic Regression* [31] to dynamically learn and adapt the batch size and execution parallelism in order to achieve low end-to-end latency while keeping system stability.

4.1 Spark Streaming Insights

For a batched stream processing system, there are several key factors affecting the performance, which include cluster size, execution parallelism, batch size, etc. In this dissertation, we explore adaptive methods that minimize the end-to-end latency while maintaining the stability of system. In this section, we first describe the system model of Spark Streaming, which is our target platform. Then we discuss the basic requirements to minimize the latency. Finally, we show the impact of batch sizing and execution parallelism tuning on the performance, and how these insights inspire our algorithm design.

4.1.1 System Model

Spark Streaming is a batched stream processing framework, which is a library extension on top of the large-scale data processing engine Apache Spark [109]. Figure 4.1 demonstrates a detail overview of Spark Streaming. In general, Spark Streaming divides the continuously input data stream into batches in discrete time intervals. To form a batch, two important parameters, *block interval* and *batch interval* are used to scatter the received data. First, the received data is split with a relatively small block interval to generate a *block*. Then after a batch interval, all the blocks in the block queue are wrapped into a *batch*. Finally, batches are put into a batch queue, and the spark engine processes them one by one. Basically, the batch interval determines the data size (i.e., number of data records), and the execution parallelism of a batch is decided by batch_interval/block_interval, which is exactly the number of blocks in that batch. The end-to-end latency of a data record consists of three parts: *Batching time*: The duration between the time a data record is received and the time that record is sent to batch queue; *Waiting time*: The time a data record waits in the batch queue, which depends on the relationship between batch interval and processing
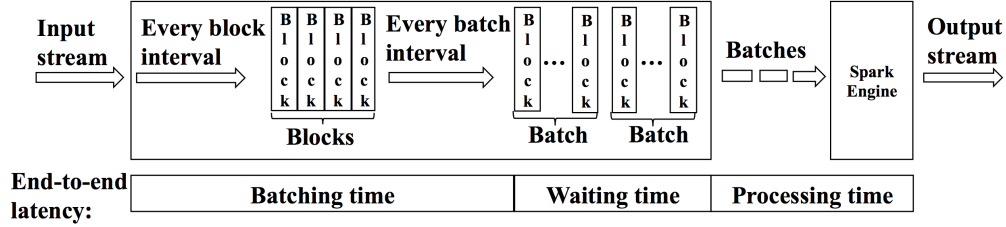
Figure 4.1: Stream processing model of Spark Streaming.

time; *Processing time*: The processing time of a batch, which depends on the batch interval and execution parallelism.

Note that the batching time is upper bounded by the batch interval. The waiting time could be infinitely large, and eventually the accumulated batches exhaust available resources (e.g. OutOfMemory), which causes the system destabilized. Moreover, the processing time depends on not only the batch interval and execution parallelism but also the available resources of the process engine.

4.1.2 Requirements for Minimizing End-to-End Latency

To make sure that the batched stream processing system handles all workloads under any operating situations (e.g., different cluster, unpredictable data surge), the desired interval needs to be chosen either by offline profiling, or by sufficient resource provisioning. However, offline profiling is vulnerable to any change of available resources and workloads. It is also hard to provision enough resources to handle unpredictable data surges, which is also not necessary for normal conditions that may dominate most of the running time. Based on the definition of end-to-end latency, it is clear that the batch interval and execution parallelism have a significant impact on the latency. To minimize the latency, a small batch interval is preferred since it means less data to process in the processing stage, which may also lead to less processing time. To avoid indefinitely increasing waiting time, the desired batch interval is also required to be larger than the processing time. Therefore, the first requirement for minimizing latency is keeping the stability of the system, which essentially means on average the processing time of a batch should be less than the batch interval. With a fixed batch interval, larger execution parallelism may incur less processing time and consequently reduce

the total latency, but it is needless to be true since larger parallelism also accompanies other overhead for task creation and communication. Thus, to minimize the latency, the second requirement is to identify how batch interval and execution parallelism affect the performance.

Static batch interval and execution parallelism cannot maintain the stability and minimize the latency. In this dissertation, we propose to dynamically adjust the batch interval and block interval such that the system is stable as well as the latency is minimized. To find a batch interval that satisfies the stability condition and the optimal execution parallelism that minimizing the latency, we need to identify the relationship between batch interval and processing time as well as the relationship between block interval and processing time. Thus, in the following two sections, we will discuss how the batch interval and block interval affect the latency in the following section.

4.1.3 Effect of Batch and Block Interval Sizing on Latency

In this section we first show the relationships between block interval and processing time for two representative streaming workloads. Then we further explore how block interval affects the processing time for the same workloads.

The Case for Batch Interval Sizing: Intuitively, the processing time should be a monotonically increasing function of batch interval. Moreover, the exact relationship could be any monotonically functions (e.g., linear, super-liner, exponential), that depends on the characteristics of workload, input data rate, and the processing engine. Thus, we choose two representative streaming workloads in the real world to explore the relationship between batch interval and processing time. The first workload is *Reduce*, which aggregates the received data based on keys. In the following sections of this chapter, we used networked word count as an example of *Reduce* workload. The second one is *Join*, which joins two different data streams.

Figure 4.2 illustrates the relationship between batch interval and processing time for *Reduce* and *Join* workloads with different data ingestion rates, in which case the block
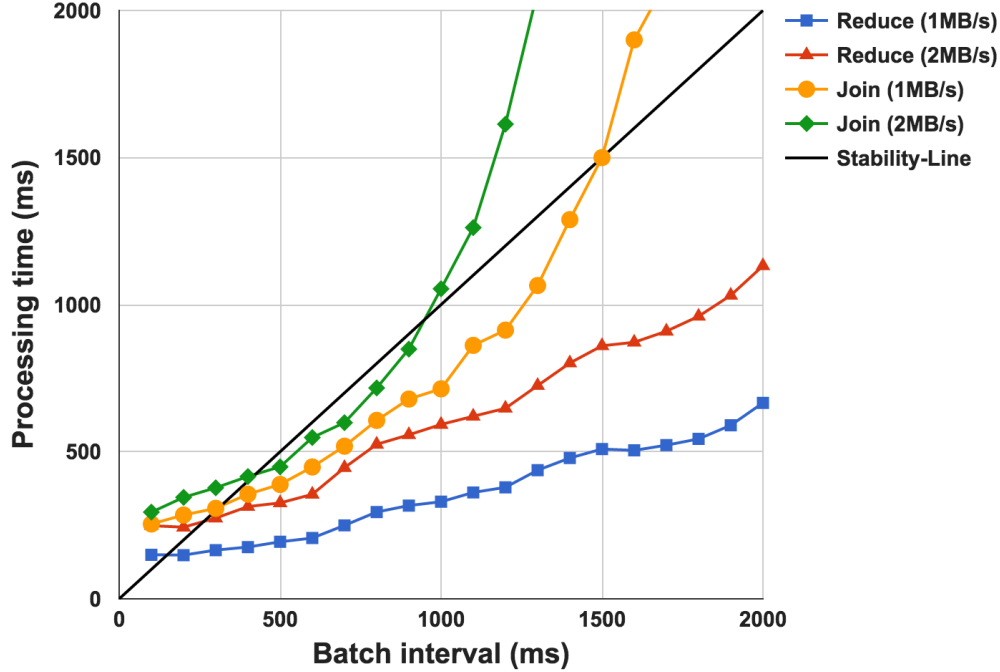
Figure 4.2: The relationships between processing time and batch interval.

interval is set to 100 ms. Basically, it is the linear relationship between batch interval and processing time for *Reduce* workload and superlinear for *Join* workload. The area below the *stability-line* is the stable zone where the batch interval is larger than processing time. Note that with a longer batch interval, *Reduce* has more stable operating status (i.e., batch interval is much larger than processing time), while the stable operating condition for *Join* is limited as the larger batch interval leads to unstable operating zone. For both workloads, the ideal batch interval is the smallest batch interval that meets the stability conditions. For linear workload (i.e., *Reduce*), there will be only one point of intersection, while for superlinear workload (i.e., *Join*), multiple intersection points may exist. For superlinear case, we expect our adaptive algorithm is able to find the lowest intersection point.

The Case for Block Interval Sizing: As we discussed, the batch interval and block interval determine the execution parallelism, which significantly affects the performance on processing time. Figure 4.3 and Figure 4.4 illustrates the effect of block sizing on processing time for *Reduce* workload. Figure 4.3 shows the relationship between block interval and processing time for different batch intervals. We applied quadratic regression, and the op-
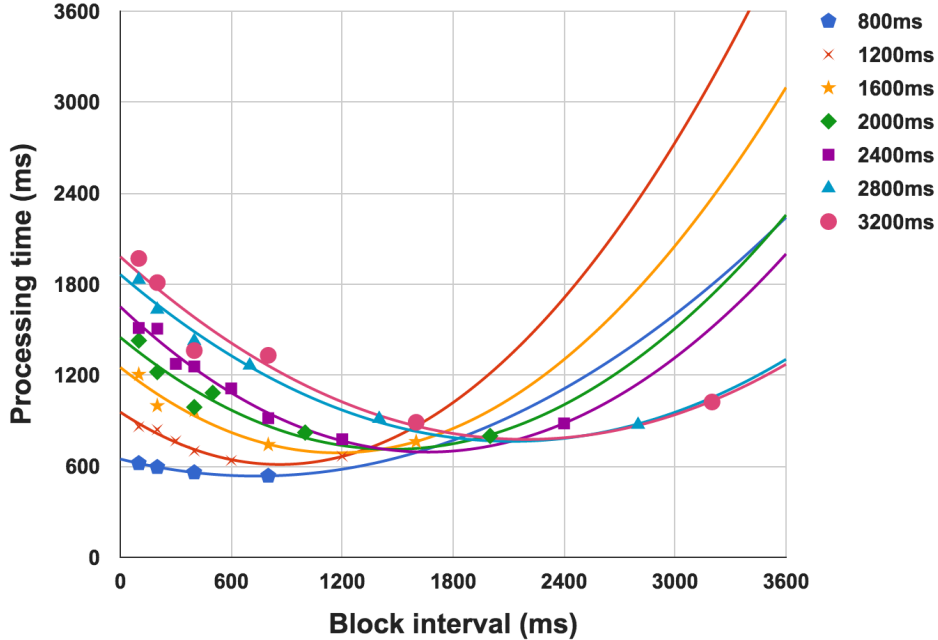
Figure 4.3: The relationship between block interval and processing time.

timal block interval is achieved around the extreme point of that parabola. For the same batch interval, the optimal block interval with different data ingestion rate may be the same (i.e., 3200ms-6MB/s and 3200ms-4MB/s) or different (2800ms-6MB/s and 2800ms-4MB/s) as shown in Figure 4.4. Note that we do not claim that the relationship between block interval and processing time is a quadratic function. Undoubtedly, the optimal block interval varies along with batch interval and data rate. For *Join* workload, it has similar relationships as *Reduce* workload. Being aware of above observations, for a given batch interval that meets the stability condition, we can further reduce the end-to-end latency by using a block interval that minimize the processing time.

To this point, we have explored how batch and block sizing affect the processing and consequently the end-to-end latency. In the next section, we introduce our online adaptive batch- and block-sizing algorithm that is designed according to these insights.

4.2 DyBBS: Dynamic Block and Batch Sizing

In this section, we first address the problem statement that describes the goal of our control algorithm and remaining issues in existing solutions. Then we introduce how
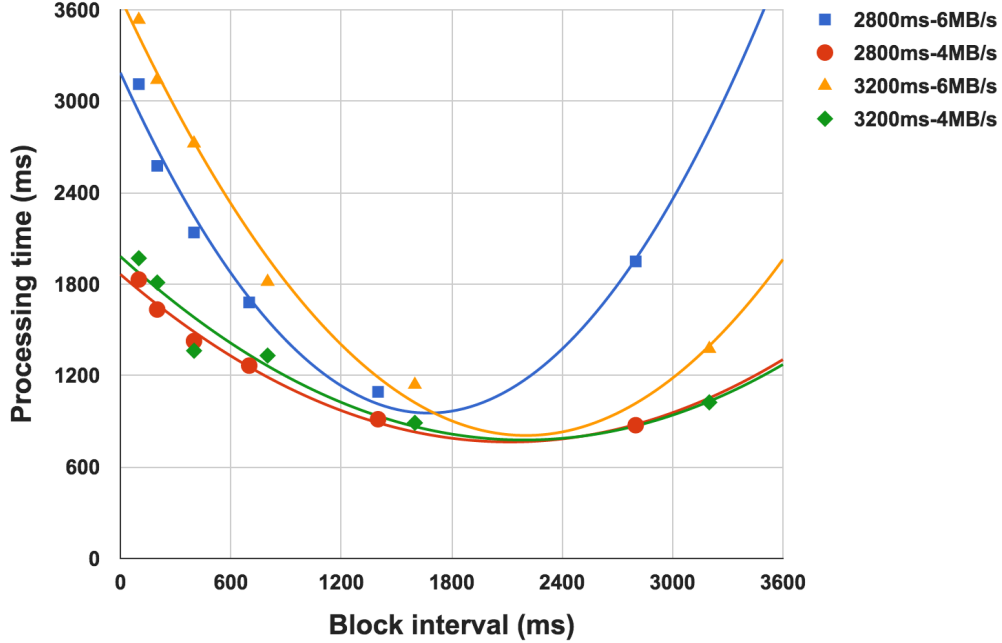
Figure 4.4: The optimal block interval varies with batch interval and data rate.

we achieve the goal and address the issues with isotonic regression and heuristic approach. Finally, we present the overall algorithm.

4.2.1 Problem Statement

The goal of our control algorithm is to minimize the end-to-end latency by batch and block sizing while ensuring the system stability. The algorithm should be able to quickly converge to the desired batch and block interval and continuously adapt batch and block intervals based on time-varying data rates and other operating conditions. We also assume the algorithm has no prior knowledge of workload characteristics. Compared to the works in literature, we adapt both batch interval and block interval simultaneously, which is the most challenging part in the algorithm design. In addition to this major goal, we also wish to solve several issues in current solutions: *Overestimation*: Generally, a configurable ratio (e.g. $\rho < 1$) is used to relax the convergence requirements to cope with noise. In this case, a control algorithm converges when the processing time is $\rho \times batch\_interval$, and the gap between batch interval and processing time (i.e., $(1 - \rho) \times batch\_interval$ increases linearly along with the batch interval. This overestimation induces non-negligible increment

on latency; *Delay response*: In ideal case, the waiting time is small and the algorithm adapts the workload variations in near real-time manner. However, when the waiting time is large (e.g., few times of the batch interval), the statistics of the latest completed batch used to predict new batch interval is already out of date, which usually cannot reflect the immediate work load conditions. This long loop delay may temporarily enlarge the waiting time and latency;

To achieve the major goal and address the issues, we introduce batch sizing using isotonic regression and block sizing with heuristic approach, which are explained in detail in following two sections, respectively. Moreover, we concentrate the algorithm design for *Reduce* and *Join* workloads. The case for other workloads is discussed in Section VI.

4.2.2   Batch Sizing with Isotonic Regression

First we look at online control algorithms that model and modify a system and at the same time suffer the trade-off between learning speed and control accuracy. The more information the algorithm learns, the higher accuracy it achieves but also requires longer convergence time. We chose an algorithm with slow convergence time in exchange for high accuracy. The reason here is that compared to record-based stream processing system, batched stream system has relatively loose constraints in terms of convergence speed since the time duration between two consecutive batches is at least the minimal batch interval that is available in that system. The control algorithm can learn and modify the system during that short duration.

Given that a time-consuming regression based algorithm can be used, an intuitive way to model the system is to directly use linear or superlinear function to fit a curve with statistics of completed batches. However, it requires prior knowledge of workload which violates our intention. As shown in Section II, the processing time is a monotonic increasing function of batch interval for both *Reduce* and *Join*. We chose a well-known regression technique, Isotonic Regression [31], which is designed to fit a curve where the direction of the trend is strictly increasing. A benefit of isotonic regression is that it does not assume

any form for the target function, such as linearity assumed by linear regression. This is also what we expect that using a single regression model to handle both *Reduce* and *Join* workloads. Another benefit of using regression model is that the overestimation can be eliminated since we can find the exact lowest intersection point as long as the fitting curve is accurate enough. However, in reality noisy statistics may affect the accuracy of the fitting curve, and thus we still need to cope with noisy behavior. Compared to ratio based (e.g., $processing\_time < \rho \times batch\_interval$) constraint relaxing, we use a constant (e.g., c) to relax the constraint, that is $processing\_time + c < batch\_interval$. With a constant difference between processing time and batch interval, the algorithm can converge as close as possible to the optimal point even with a large batch interval, in which case the ratio based method introduces a large overestimation.

The optimal point is achieved at the lowest point of intersection between stability-line and workload specified line. We first fit a curve with isotonic regression using the gathered statistics (i.e., a batch's interval and its corresponding processing time) of completed batches that have the same block interval. Suppose the regression curve is $IsoR(x)$, where x is the batch interval, and $IsoR(x)$ is the processing time estimated with the regression model. Then we identify the lowest point of intersection by finding the smallest x such that $IsoR(x) + c < x$. Note that we do not need to examine all data ingestion rate to find the intersection point. We only fit the curve for the immediate data ingestion rate that is the same rate in the latest batch statistics as we assume the data ingestion rate keeps the same in the near future (within the next batch interval).

4.2.3   Block Sizing with Heuristic Approach

The block interval affects the end-to-end latency via its impact on execution parallelism. As shown in Section II, with a fix block interval, the best case a control algorithm can reach is actually a local optimal case in the entire solution space, which is far from the global optimal case that can significantly further reduce the end-to-end latency. Therefore, we wish our algorithm were able to explore all solution space through block sizing. As

aforementioned, the relationship between block interval and processing time is not quadratic although we use it as the trend line in above results. Even though this relationship is quadratic, our control algorithm still needs to enumerate all possible block intervals to find the global optimal case, which is the same as brute-force solution. Therefore, we propose a heuristic approach performing the block sizing to avoid enumeration and frequent regression computation.

Similar to the batch sizing, with different data ingestion rates, the curves of processing time and block interval are also identical. Thus, we use the same strategy used in batch sizing, which is that for each specific data rate we use the same heuristic approach to find the global optimal point. Basically, this heuristic approach starts with the minimal block interval and gradually increase the block interval size until we cannot benefit from larger block interval. The detailed heuristic approach is described as following: i) Starting with the minimal batch interval, we use isotonic regression to find the local optimal point within that block interval (i.e., only applying batch sizing with a fix block interval); ii) Then increase the block interval by a configurable step size and apply the batch sizing method until it converges; and iii) If the end-to-end latency can be reduced with new block interval, then repeat step ii; otherwise the algorithm reaches the global optimal point. Besides that, we also track all local optimal points for all block intervals that have been tried. Note that with the same data ingestion rate and operating condition, the global optimal point is stable. When the operation conditions change, the global optimal point may shift, in which case the algorithm needs to adapt to the new condition by repeating the above three steps. However, restarting from scratch slows down the convergence time, thus we choose to restart the above heuristic approach from the best suboptimal solution among all local optimal points to handle operating condition changes.

Up to this point, we have addressed batch sizing that leverages isotonic regression and block sizing with a heuristic approach. The next step is to combine these two parts to form a practical control algorithm, which is discussed in next section.

4.2.4   Our Solution - DyBBS

Here we introduce our control algorithm - *Dynamic Block and Batch Sizing* (DyBBS) that integrates the discussed isotonic regression based batch sizing and heuristic approach for block sizing. DyBBS uses statistics of completed batches to predict the block interval and batch interval of the next batch to be received. Before we explain the algorithm, we first introduce two important data structures that are used to track the statistics and current status of the system. First, we use a historical table (denoted as *Stats* in the rest of this chapter) with entries in terms of a quadruple of ($data\_rate, block\_intvl, batch\_intvl, proc\_time$) to record the statistics, where the $data\_rate$ is the data ingestion rate of a batch, $block\_intvl$ is the block interval of a batch, $batch\_intvl$ is the batch interval of a batch, and $proc\_time$ is the processing time of a batch. The $proc\_time$ is updated using a weighted sliding average on all processing times of batches that have the same $data\_rate$, $block\_intvl$, and $batch\_intvl$. Second, we track the current block interval (denoted as $curr\_block\_intvl$ in the remaining paper) for a specific data ingestion rate by mapping $data\_rate$ to its corresponding $curr_block\_intvl$, which indicates the latest status of block sizing for that specific data rate, and this mapping function is denoted as DR_to_BL.

Algorithm 1 presents the core function that calculates the next block and batch interval. The input of DyBBS is the $data\_rate$ that is the data ingestion rate observed in the last batch. First, the algorithm gets the block interval for the $data\_rate$. Secondly, all the entries with the same $data\_rate$ and $curr\_block\_intvl$ are extracted, which are used for function $IsoR$ to calculate the optimal batch interval, as shown on Line 3 and 4. Thirdly, the algorithm compares the estimated new_batch_intvl and the sliding averaged $proc\_time$. We treat the isotonic regression result converges as if the condition on Line 6 is true and then the algorithm will try different block interval based on operating condition. If the $curr\_block\_intvl$ is the same as that of the global optimal point stated on Line 7, then increase the block interval by certain step size. Otherwise, we treat it as if the operating condition changing and choose the block interval of the suboptimal point. If the condition on

---

Algorithm 1 *DyBBS* : Dynamic Block and Batch Sizing Algorithm

---

Require: *data_rate*: data ingestion rate of last batch
Require: *DR_to_BL*: Map<data_rate, current_optimal_block_interval>
Require: *Stats*: List<data_rate, block_interval, batch_interval, process_time>
 1: **Function** *DyBBS(data_rate)*
 2:   current_block_interval = DR_to_BL(*data_rate*)
 3:   Pair(batch_interval, process_time)[] pairs = Stats.Get(*data_rate, current_block_interval*)
 4:   new_batch_interval = IsoR(pairs)
 5:   **if** (IsoR is converged) **then**
 6:     new_block_interval = current_block_interval + block_step_size
 7:   **else**
 8:     new_block_interval = current_block_interval
 9:   **endif**
10:   Set(*new_block_interval, new_batch_interval*)
11: **EndFunction**

Line 6 is not satisfied (Line 12), it means the *IsoR* for *curr_block_intvl* has not converged and keep the block interval unchanged. Finally, DyBBS updates the new block interval and batch interval. Note that the *new_batch_intvl* is always rounded to a multiplier of the *new_block_intvl*, which is omitted in the above pseudo code. In addition, when the *IsoR* is called, if there is less than five samples points (i.e., the number of unique batch intervals is less than five), it will return a new batch interval using slow-start algorithm [100].

In this section, we have introduced our adaptive block and batch sizing control algorithm and showed that it can handle the overestimation and long delay issues. We will discuss the implementation of DyBBS on Spark Streaming.

4.3   Implementation

We implement our dynamic block- and batch-sizing algorithm in Spark Streaming (version 1.4.0). Figure 4.5 illustrates the high-level architecture of the customized Spark Streaming system. We modified the *Block Generator* and *Batch Generator* modules and introduced the new *DyBBS* module such that Spark Streaming can generate different size blocks and batches based on the control results. First, the *Block Generator* generates blocks with block interval, and the blocks are handed off to the *Batch Generator*. Then, blocks are wrapped into batches, and the batches are put in the batch queue. Finally, the batches are processed in the *Batch Processor*. The *DyBBS* module leverages the statistics to predict the block interval and batch interval for next batch.
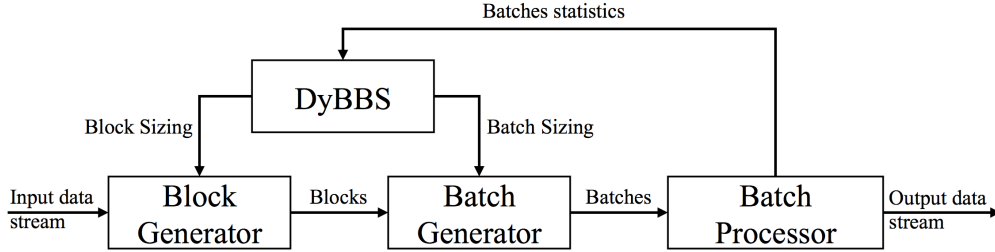
Figure 4.5: High-level overview of our system that employs DyBBS.

There are three configurable parameters used in our algorithm. The first one is constant c used to relax the stability condition. A larger c keeps the system more stable and more robust to noisy, but it also leads to a larger latency. The second parameter is a block interval incremental step size. A larger step size incurs fast convergence speed on block sizing procedure. However, it may also never converge to the global optimal case due to too large of a step size. The last parameter is used for data ingestion rate discretization. In DyBBS, the isotonic regression is conducted on batch interval and processing time for a specific data rate. In our implementation, we choose 50ms as the constant value c, 100ms for the block step size, and 0.2MB/s for the data rate discretization.

In summary, our modifications do not require any changes to the user's program or programming interfaces. Although our implementation is Spark Streaming specified, it is easy to implement our control algorithm on other batched stream processing systems that have similar design principle of block and batch.

4.4 Evaluation

We evaluate the DyBBS with *Reduce* and *Join* streaming workloads under various combinations of data rates and operating conditions. Our results show that DyBBS is able to achieve latencies comparable to the optimal case. The comparison with the other two practical approaches illustrates that our algorithm achieves the lowest latency. We demonstrate that the convergence speed our algorithm is slower than the state-of-art only for the first time a data rate appears, and for the second and following appearance of the

same data rate our algorithm is much faster. Finally, we also show our algorithm can adapt to resource variation.

4.4.1 Experiment Setup

In the evaluation, we used a small cluster with four physical nodes. Each node can hold four Spark executor instances (1 core with 2 GB memory) at most. We ran two workloads as mentioned-*Reduce* and *Join*, with two different time-vary data rate patterns. The first data rate pattern is sinusoidal fashion of which the data rate varies between 1.5 MB/s to 6.5 MB/s for *Reduce*, and 1 MB/s to 4 MB/s for *Join*, with periodicity of 60 seconds. The other one is Markov chain [13] fashion of which the data rate changes every 15 seconds within the same rate duration as sinusoidal fashion. For the input data, we generated eight bytes long of random strings as the key for reducing and join operations, and the number of unique key is around $32,000$. The output is written out to a HDFS file [97].

As comparison, we implemented three other solutions that are as follow:

FPI[37]: the state-of-the-art solution, that adopts fix-point iteration (FPI) to dynamically adapt batch interval and is the only practical one in Spark Streaming.

FixBI: We compare our algorithm with this unpractical hand-tuning method. It is the best case (i.e., with the lowest average latency over entire running duration) by enumerating all possible pair of block and batch intervals. However, it still uses static setting over the running time and cannot change either block or batch interval at runtime.

OPT: This is the oracle case, which we dynamically change the block and batch intervals at the runtime based on prior knowledge.

For the performance experiments, we implemented two different versions for all methods: i) *Fixed Block Interval* that changes the batch interval while it keeps the block interval fixed (i.e., 100ms); and ii) *Dynamic Block Interval* that adapts both block and batch intervals. For the FPI that is not designed for block sizing, we set a block interval based on prior knowledge such that it achieves local optimal with the batch interval FPI estimates. To implement the *Fixed Block Interval* algorithm in DyBBS, we disable the block sizing
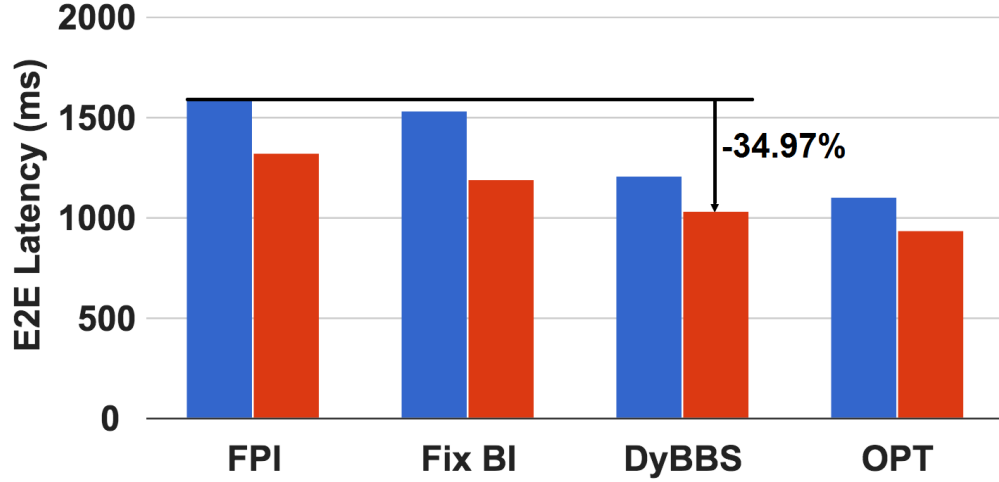
Figure 4.6: Comparison for *Reduce* workload with sinusoidal input.

and set block interval to 100ms. The end-to-end latency of a batch is defined as the sum of *batch interval*, *waiting time*, and *processing time*, and we used the average end-to-end latencies over entire experiment duration as the criteria for performance comparison.

4.4.2 Performance on Minimizing Average End-to-End Latency

We compared the average end-to-end latency over 10 minutes for *Reduce* and *Join* workloads with sinusoidal and Markov chain input rates. Figure 4.6-4.9 shows that the DyBBS achieves the lowest latencies that are comparable with the oracle case (OPT). In general, by introducing block sizing, the latency is significantly reduced compared to that only applies batch sizing. As Figure 4.6-4.9 showed, our DyBBS outperforms the FPI and FixBI in all cases. Specifically, compared with FPI with fixed block interval, DyBBS with dynamic block interval reduced the latencies by 34.97% and 63.28% for *Reduce* and *Join* workloads with sinusoidal input rate (the cases in Figure 4.6 and 4.7), respectively. For Markov chain input rates, DyBBS with dynamic block interval reduced the latencies by 48.02% and 67.51% for *Reduce* and *Join* workloads (the cases in Figure 4.8 and 4.9) respectively, compared to FPI with fixed block interval.

In terms of real-time performance, Figure 4.10 shows the dynamic behaviors including block interval, batch interval, waiting time, processing time, and end-to-end latency for
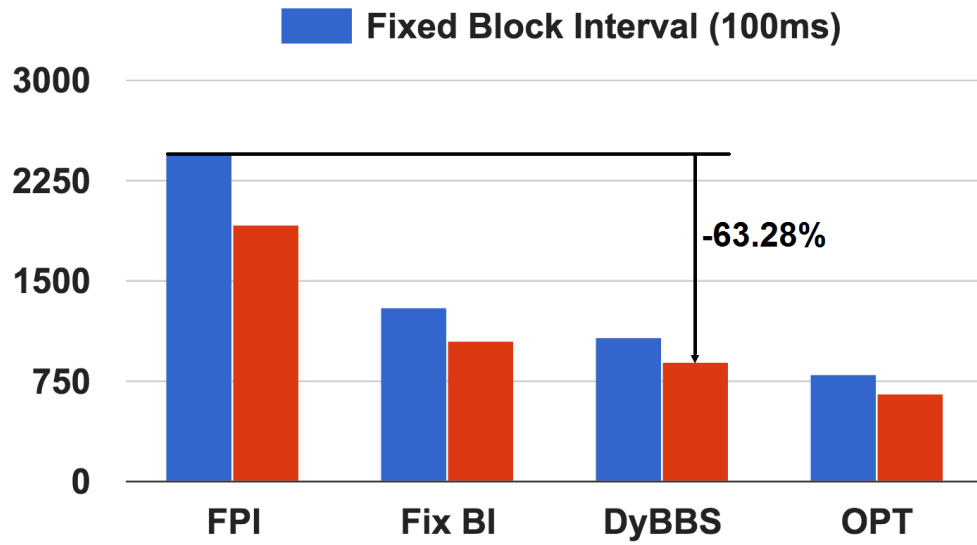
Figure 4.7: Comparison for *Join* workload with sinusoidal input.
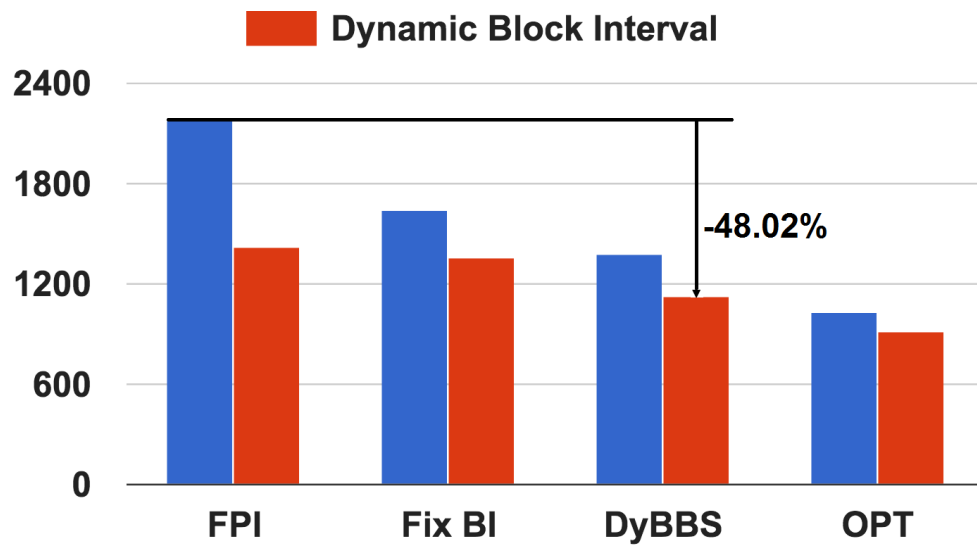
Figure 4.8: Comparison for *Reduce* workload with Markov chain input.
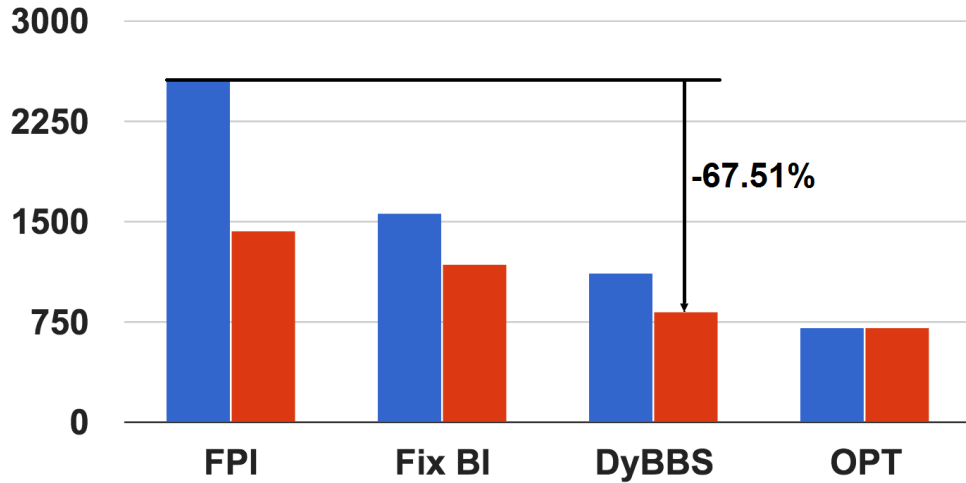
Figure 4.9: Comparison for *Join* workload with Markov chain Input.

Reduce workload with sinusoidal input, for DyBBS. During the first three minutes (the first three sine waves), our algorithm was learning the workload characteristics, and hence the latency is relatively high. After our algorithm converged (from the fourth minute), the latency is relatively low and maintained the same for the rest of time. In Figure 4.11, we compared the batch interval and end-to-end latency of FPI and DyBBS. During the first 60 seconds, the end-to-end latency of DyBBS is higher than FPI due to imprecise isotonic regression model. In the next two cycles, the end-to-end latency of DyBBS reduces since the isotonic regression model is becoming more and more accurate.

4.4.3 Convergence Speed

We first compared the convergence speed of *Reduce* workload for FPI and DyBBS with a step changing data rate. Since FPI does not support block sizing, we only present the behavior of batch interval along with the data rate in Figure 4.12. Before the star time, we run FPI and DyBBS long enough such that both of them have converged with the 1MB/s data rate. At the 5th second, the data rate changes to 3MB/s and that is the first time the rate appears during entire experiment. The FPI first converges roughly after the 11th second (the red vertical line on the left), while DyBBS converges until the 18th seconds (the blue vertical line on the left). This is caused by the longer learning time of isotonic
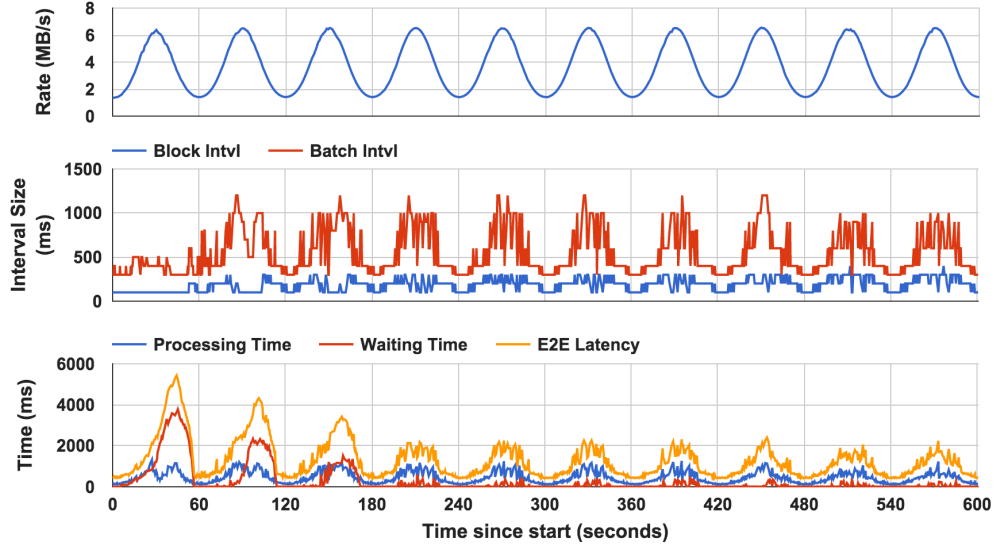
Figure 4.10: Real-time behavior for *Reduce* workload.

regression used in DyBBS. When the second time that the data rate changes to 3MB/s, in contrary to the first time, DyBBS converges at the 47th second (the blue vertical line on the right), while FPI converges seven seconds later (the red vertical line on the right). Since FPI only uses the statistics of last two batches that means it cannot leverage historical statistics, hence FPI has to re-converge every time the data rate changes. Thus, for a specific data rate, DyBBS has long convergence time for the first time that rate appears and relatively small convergence time for the second and rest times the same data rate appears.

With block sizing enabled in DyBBS, the convergence time is longer than that when block sizing is disabled. Figure 4.13 and 4.14 shows the convergence time for *Reduce* workload with two different constant data rates. With larger data ingestion rate, our algorithm spent more time to search for the optimal point since there are more potential candidates. Although our algorithm spends tens of seconds to converge for large data ingestion rate, this convergence time is relatively small compared to the execution time (hours, days, even months) of long run streaming application. Therefore, we believe that our algorithm is able to handle large data ingestion rate for long run applications.
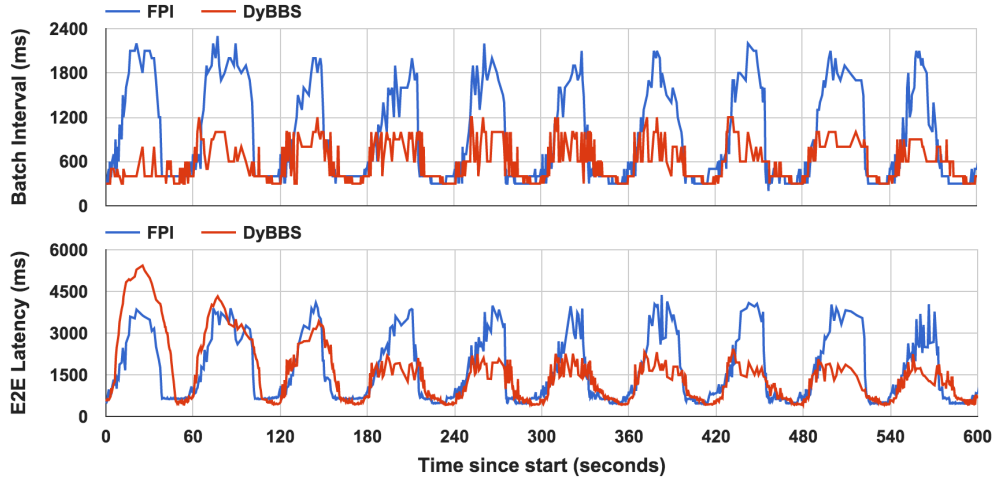
Figure 4.11: Comparison between FPI and DyBBS for *Reduce* workload.
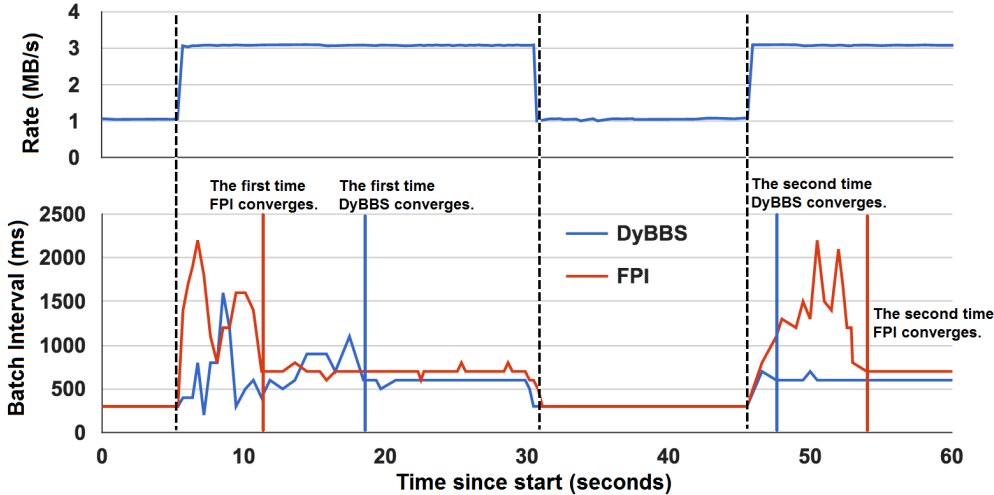
Figure 4.12: Timeline of data input rate and batch interval for *Reduce* workload.

4.4.4 Adaptation on Resource Variation

In above experiments, we have showed that our algorithm is able to adapt to various workload variation. When the operating condition changes, our algorithm should detect the resource variation and adjust the block and batch interval consequently. To emulate the resource reduction, we run a background job on each node such that one core on each node is fully occupied by the background job. Figure 4.15 illustrates that our algorithm can adapt the resource reduction (at 60th second) by increasing the batch interval and end-to-end latency.
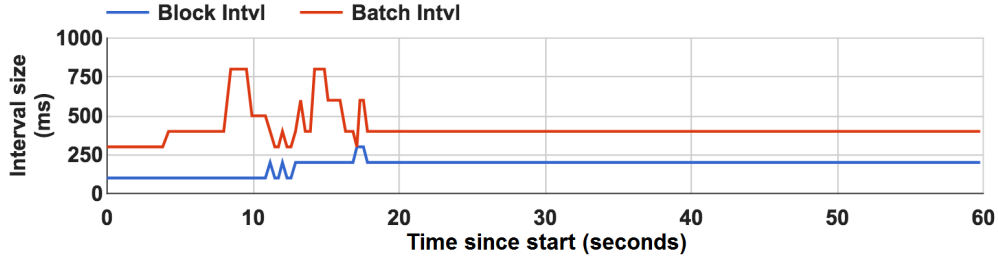
Figure 4.13: Timeline for *Reduce* workload with 3MB/s data ingestion rate.
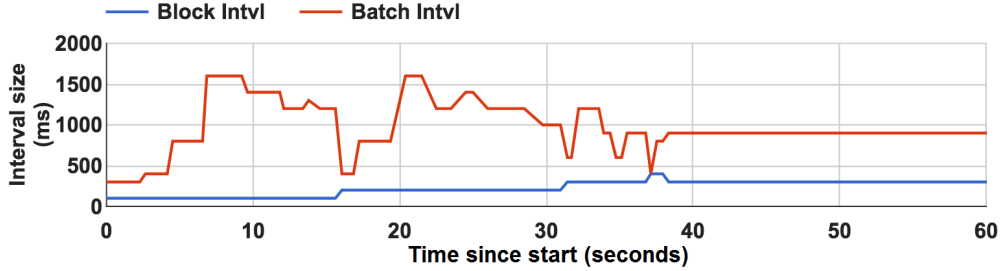
Figure 4.14: Timeline for *Reduce* workload with 6MB/s data ingestion rate.

4.5   Discussion

In this dissertation, we narrowed down the scope of our algorithm to two workloads with simple relationships (i.e., linear for *Reduce* workload and superlinear for *Join* workload). There may be a workload with more complex characteristics than *Reduce* and *Join* workloads. An example of such workload is *Window* operation, which aggregates the partial results of all batches within a sliding window. This issue is also addressed in [37], and the *Window* operation is handled by using small mini-batch with 100ms interval. In our approach, there are two issues for handling *Window* workload. One is that it is possible that we cannot find a serial of consecutive batches such that the total time interval exactly equals the window duration. If we always choose the next batch interval such that the previous condition is satisfied, then we lose the opportunity to optimize that batch, as well as all the batches in the future. Another problem is that our approach dynamically changes the block interval, which means our approach cannot directly employ the method used in [37]. Therefore, we leave the work to support *Window* workload for the future. Another limitation is that the scale of our experiment test bed is relatively small. For large scale environment, the solution
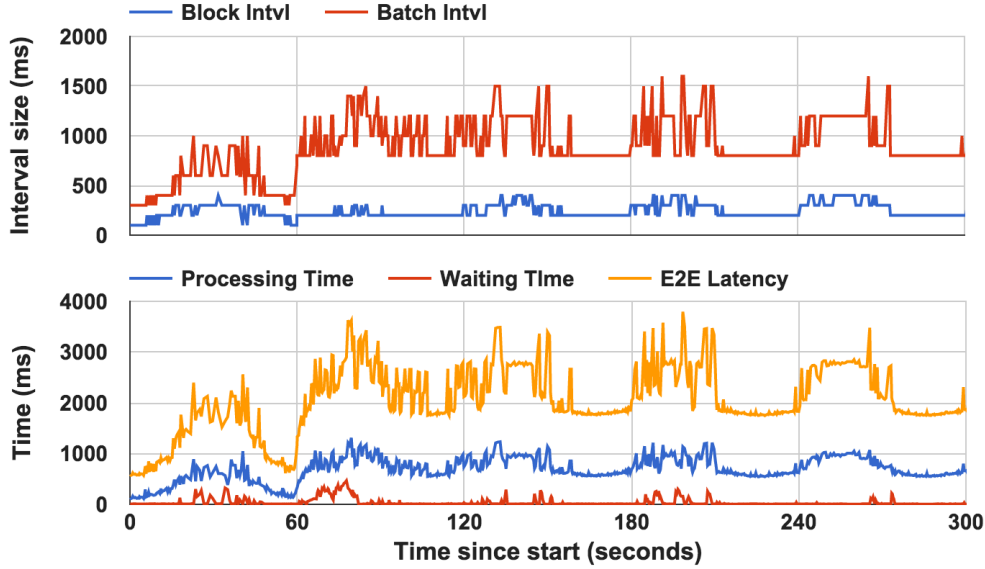
Figure 4.15: Timeline for *Reduce* workload with resource variation.

space is much larger, which takes longer time to converge. In addition, large scale cluster can handle more workload (e.g., larger data ingestion rate), which needs to adjust the granularity of data rate discretization. Next step we plan to explore these features of DyBBS algorithm in large cluster environment.

4.6 Summary

In this chapter, we have illustrated an adaptive control algorithm for batched processing system by leveraging dynamic block and batch interval sizing. Our algorithm is able to achieve low latency without any workload specific prior knowledge, which is comparable to the oracle case due to our accurate batch interval estimation and novel execution parallelism tuning. We have shown that compared with the state-of-the-art solution, our algorithm can reduce the latency by at least 34.97% and 63.28% for *Reduce* and *Join* workloads, respectively. We have also presented the abilities of DyBBS to adapt to various workloads and operating conditions.

Up to this point, we have introduced H_2O and DyBBS to solve the challenge of efficiently managing the large-scales video surveillance system, which can be used to detect abnormal hardware and software behavior even given millions of various objects. In the next

chapter, we will present our research that solves the challenge of effectively and efficiently utilizing video data collected from cameras.

CHAPTER 5: FIREWORK: DATA ANALYTICS AT EDGE

Cloud computing and edge computing are the core of future computing facilities and adopted in most data processing scenarios, however, an important or fundamental assumption behind them is that data is owned by a single stakeholder, where the user or owner has fully control privileges over the data. As we mentioned, cloud computing requires the data to be preloaded in data centers before a user runs its applications in the cloud [26], while edge computing processes data at the edge of the network but requires closely control of the data producers and consumers. Data owned by multiple stakeholders is rarely shared due to various reasons, such as security concern (e.g., data across border), conflict of interest (e.g., data from competitors), privacy issue (e.g., data of health care), and resource limitation (e.g., extremely large and long network distance data transportation) and etcetera.

Taking the cooperation in connected health as an example, the health records of patients hosted by hospitals and customer records owned by insurance companies are highly private to the patients and customers and rarely shared. If an insurance company has the access to its customers' health records, the insurance company could initiate personalized health insurance policies for its customers based on their health records. Another example is "find the lost" in city [96], where video steams from multiple data owners across the city are used to find a lost object. It is common that the police department manually collects video data from surveillance cameras on the streets, retailer shops, individual smart phones, or car video recorders in order to identify a specific lost object. If all these data could be shared seamlessly, it can save huge amount of human work and identify an object in real-time fashion. Furthermore, simply replicating data or running analyzing application provided by third party on stakeholders' data may break the privacy and security restricts. Unfortunately, none of the aforementioned can be easily achieved by leveraging cloud computing or edge computing individually.

To attack aforementioned barriers, *Firework* **i)** fuses data from multiple stakeholders as virtual shared data set that is a collection of data and predefined functions by data

owners. The data privacy protection cloud be carried out by privacy preserving functions preventing data leakage by sharing sensitive knowledge only to intended users; **ii)** breaks down an application into subservices so that a user can directly subscribe intermediate data and compose new applications by leveraging existing subservices; and **iii)** provides an easy-to-use programming interface for both service providers and end users. By leveraging subservices deployed on both the cloud and the edge, *Firework* aims to reduce the response latency and network bandwidth cost for hybrid cloud-edge applications and enables data processing and sharing among multiple stakeholders. We implement a prototype of *Firework* and demonstrate the capabilities of reducing response latency and network bandwidth cost by using an edge video analytics application developed on top of *Firework*.

5.1   System Design

Firework is a framework for big data processing and sharing among multiple stakeholders in hybrid cloud-edge environment. Considering the amount of data generated by edge devices, it is promising to process the data at the edge of the network to reduce response latency and network bandwidth cost. To simplify the development of collaborative cloud-edge applications, *Firework* provides a uniform programming interface to develop IoE applications. To deploy an application, *Firework* creates service stubs on available computing nodes based on a predefined deployment plan. To leverage existing services, a user implements a driver program and *Firework* automatically invokes the corresponding subservices via integrated service discovery. In this section, we will introduce the detailed design of *Firework*, including terminologies, system architecture, and programmability, to illustrate how *Firework* facilitates the data processing and sharing in collaborative cloud-edge environment.

5.1.1   Terminologies

We first introduce the terminologies that describe abstraction concepts in *Firework*. Based on the existing definitions of the terminologies in our previous work [113], we extend and enrich their meanings and summarize them as following:

Distributed Shared Data (DSD): Data generated by edge devices and historical data stored in the cloud can be part of the shared data. DSD provides a virtual view of the entire shared data. It is worth noting that stakeholders might have different views of DSD.

Firework.View: Inspired by the success of object oriented programming, a combination of *dataset* and *functions* is defined as a *Firework.View*. The *dataset* describes shared data and the *functions* define applicable operations upon the dataset. A *Firework.View* can be adopted by multiple data owners who implement the same functions on the same type of dataset. To protect the privacy of data owners, the *functions* can be carried out by privacy preserving functions that share sensitive data only to intended users [21].

Firework.Node: A device that generates data or implements *Firework.View*s, is a *Firework.Node*. As data producers, such as sensors and mobile devices, *Firework.Node*s publish sensing data. As data consumers, *Firework.Node*s inherit and extend *Firework.View*s by adding functions to them, and the new *Firework.View*s could be further extended by other *Firework.Node*s. An example application could be the city-wide temperature data, in which scenario sensor data is owned by multiple stakeholders and each of them provides public portals for data accessing. A user could reach all temperature data as if he/she operates on a single centralized data set. A sensor publishes a base *Firework.View* containing temperature data and read function, and a *Firework.Node* can provide a new *Firework.View* that returns highest regional temperature by extending the base *Firework.View*.

Firework.Manager: First, it provides centralized service management, where *Firework.View*s are registered. It also manages the deployed services built on top of these views. Second, it serves as the job tracker that dispatches tasks to *Firework.Node*s and optimizes running services by dynamically scaling and balancing among *Firework.Node*s depending on their resource utilizations. Third, it allocates computation resources including CPU, memory, network, and (optional) battery resources to running services. Fourth, it exposes available services to users so that they can leverage existing services to compose new applications.
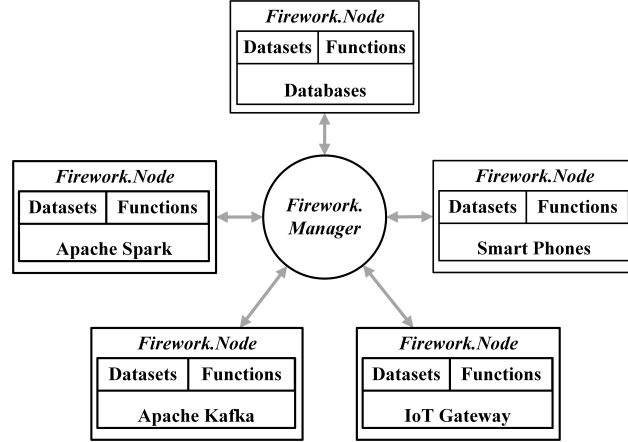
Figure 5.1: A *Firework* instance consisting of heterogeneous computing platforms.

Firework: It is an operational instance of *Firework* paradigm. A *Firework* instance might include multiple *Firework.Node*s and *Firework.Manager*s, depending on the topology. Figure 5.1 shows an example of *Firework* instance consisting of five *Firework.Node*s employing heterogeneous computing platforms. If all *Firework.Node*s adopt homogeneous computing platform, such a *Firework* instance will be similar to cloud computing and edge computing.

5.1.2 Architecture

As a major concept of *Firework, Firework.View* is abstracted as a "class-like" object, which can be easily extended. *Firework.Node* can be implemented by numerous heterogeneous computing infrastructures, ranging from big data computation engines (e.g., Apache Spark [109], Hadoop [97], databases) and distributed message queues (e.g., Apache Kafka [68], MQTT [60], ZeroMQ [54], RabbitMQ [16]) in the cloud, to edge devices of smart phones and IoE gateways (e.g., Intel Edison, Raspberry Pi). *Firework.Manager* is the access point of a *Firework* instance that allows users to deploy and execute their services. Both *Firework.Node* and *Firework.Manager* can be deployed on the same computing node, where an edge node acts as not only a data consumer in the point of view of actuators, but also a data producer in the point of view of the cloud.

Firework
Service Management
Discovery	Deployment
Job Management
Task Reuse	Task Dispatch	Task Optimization
Executor Management
CPU	MEM	Network
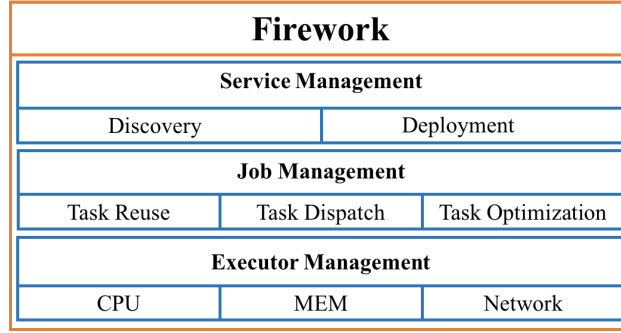
Figure 5.2: An abstraction overview of *Firework*.

To realize aforementioned abstractive concepts, we generalize them as a layered abstraction as shown in Figure 5.2, which consists of *Service Management*, *Job Management*, and *Executor Management*. The *Service Management* layer performs service discovery and deployment, and the *Job Management* layer manages tasks running on a computing node. The combination of *Service Management* and *Job Management* fulfills the responsibilities of a *Firework.Manager*. The *Executor Management* layer, representing a *Firework.Node*, manages computing resources. In the following paragraphs, we will describe each layer in detail.

Service Management: To deploy a service on *Firework*, a user has to implement at least one *Firework.View* which defines the shared data and functions, and a deployment plan, which describes how computing nodes are connected and how services are assigned to the computing nodes. Note that the application defined deployment topology might be different from the underlying network topology. The reasons of providing customizable deployment plan are to avoid redundant data processing and facilitate application defined data aggregation. In a cloud-centric application, data is uploaded to the cloud based on a predefined topology, where a developer cannot customize the data collection and aggregation topology. However, in IoE applications, sensors/actuators (e.g., smart phones, on-vehicle cameras) change the network topology frequently, which requires the application deployment to be adapted depending on available resource, network topology, and geographical location. Furthermore, *Firework.View* leverages multiple data sources to form a virtual shared data set,

where the data sources can be dynamically added or removed according to the deployment plan of the IoE application.

Upon a service (i.e., *Firework.View*) registration, *Firework.Manager* creates a service stub for that service (note that the same service registered by multiple nodes shares the same service stub entry), which contains the metadata to access the service, such as the network address, functions' entries, input parameters and etcetera. A service provider can create a *Firework.View* via extending a registered service and register the new *Firework.View* to another *Firework.Manager*. By chaining up all these *Firework.View*s, a complex application can be composed. Depending on the deployment plan, a service (i.e., *Firework.View*) can be registered to multiple *Firework.Manager*s and the same service can be deployed on more than one computing nodes. An application developer can implement services and corresponding deployment plans via the programming interfaces provided by *Firework*. We will show more details through a concrete example (i.e., VideoAnalytics implemented on *Firework*) in Section 5.1.3.

To take advantages of existing services, a user retrieves the list of available services by querying *Firework.Manager*. Then the user implements a driver program to invoke the service. Upon receiving the request, *Firework* filters out the computing nodes that implement the requested services. Afterwards, *Firework* creates a new local job and dispatches the request to these computing nodes. Details about job creation and dispatch are explained in Section 5.1.2. By repeating this procedure, *Firework* instantiates a requested service by automatically creating a computation stream. The computation stream implements an application by leveraging computing resources along the data propagation path, which might include the edge devices and the cloud.

Considering the mobility and operational environment of edge devices, it is common that they may fail or change the network condition. To deal with failure or varying network condition, *Firework* assigns a time-to-live interval to registered services and checks the liveness via heartbeat message periodically. A node will re-register its services after a failover

or network condition change. When a node acting as *Firework.Manager* fails, it recovers all service stubs depending on persistent logs. Specifically, it rebuilds the connections based on the out-of-date service stubs and updates these service stubs if the connections are restored successfully, otherwise the service stubs are removed.

Job Management: A user can send *Firework.Manager* a request to start a service. Upon receiving the invocation, a local job is created by the *Job Management* layer, which initializes the service locally. For each job, a dedicated communication port is assigned for exchanging control messages. Note that this port is not used by executors for data communication. Next *Firework.Manager* forwards the request to available *Firework.Node*s that implement the *Firework.View* of the requested service. Lastly, the local job is added to task queue waiting for execution. When a job is terminated by the user, the *Job Management* layer stops executors and releases the dedicated port of that job.

   Firework provides elasticity of computing resource scaling via task reuse. In *Firework*, all services are public for all users, which potentially means that two different users could request the same service. In such situation, *Firework* reuses the same running task by dynamically adding an output stream to the task. It is worth nothing that the input streams of a service might come from different sources. Extra computing resources are allocated to a task if the resource utilization of an executor exceeds a threshold and vice versa. To reduce the I/O overhead of a task brought by communicating with multiple remote nodes, *Firework* uses a separate I/O manager, which will be introduced in Section 5.2, to perform data transmission so that a running service subscribes/publishes the input/output data from the I/O manager. In addition to the resource scaling, *Firework* also optimizes the workload among multiple nodes. A *Firework* node can inherit a base service without extending it. In this case, two consecutive nodes provide exactly the same service. If the node closer to data sources is overloaded, it can delay and offload computation to the other node, which might be less loaded. The offload decision aims to minimize the response latency of the service, which depends on the resource utilizations (e.g., CPU, memory, and network bandwidth).

```java
/* FWView: an implementation of Firework.View */
public class FWView implements Runnable {
  protected String serviceName;
  protected FWInputStream[] inputStreams;
  protected FWOutputStream[] outputStreams;
  public FWView(String serviceName) {
    this.serviceName = serviceName;
    this.getFWInputStream();
    this.getFWOutputStream();
  }
  /* Get input streams from JobManager. */
  public void getFWInputStream() {
    inputStreams = JobManager.getInputStream(serviceName);
  }
  /* Get output streams from JobManager. */
  public void getFWOutputStream() {
    outputStreams = JobManager.getOutputStream();
  }
  /* Receive data from the input streams. */
  public Data[] read() {
    inputData = inputStreams.read();
  }
  /* Process the input data. */
  public Data[] compute(Data[] inputData) {
    // Do nothing by default.
    return inputData;
  }
  /* Send the processed data to other nodes. */
  public void write(Data[] outputData) {
    outputStream.write(outputData);
  }
  /* Run the computation procedure. */
  public void run() {
    while(true) {
      Data[] inputData = read();
      Data[] outputData = compute(inputData);
      write(outputData);
    }
  }
}
```

Listing 5.1: The Java code of FWView.

Executor Management: A task in *Firework* runs on an executor that has dedicated CPU, memory, and network resources. *Firework* nodes leverage heterogeneous computing platforms and consequently adopt different resource management approaches. Therefore, the *Executor Management* layer serves as an adapter that allocates computing resources to tasks. Specifically, some *Firework* nodes like smart phones or IoE gateways may adopt JVM or Docker [8], while some nodes like commodity servers may employ OpenStack [15] or VMWare, to host an executor. The executor management is fulfilled by the *Job Management* layer and operated by the *Executor Management* layer.

```java
/* DeployPlan: the application defined topology. */
public class DeployPlan {
  private List<Rule> rules;
  /* Add a new rule to the deployment plan. */
  public void addRule(Rule rule) {
    rules.add(rule);
  }
}
/* FWDriver: Firework application driver. */
public class FWDriver {
  /* FWContext: creates a session between user and Firework.Manager */
  private FWContext fwContext;
  private DeployPlan deployPlan;
  private UUID uuid;
  private String serviceName;
  public FWDriver(String serviceName, DeployPlan deployPlan) {
    this.serviceName = serviceName;
    this.deployPlan = deployPlan;
    this.fwContext = new FWContext();
  }
  /* Deploy an application and get an UUID back. */
  public void deploy() {
    uuid = fwContext.configService(serviceName, deployPlan);
  }
  /* Launch a deployed service by uuid. */
  public void start(Parameter[] params) {
    fwContext.startService(uuid, params);
  }
  /* Stop an application by uuid. */
  public void stop() {
    fwContext.stopService(uuid);
  }
  /* Get the final results from Firework.Manager. */
  public Data[] retrieveResult() {
    return fwContext.retrieveResult(uuid);
  }
}
```

Listing 5.2: The Java code of FWDriver.

5.1.3 Programmability

Firework provides an easy-to-use programming interface for both developers and users so that they can focus on programming the user defined functions. An application on *Firework* includes two major parts: programs implementing *Firework.View*s and a driver program to deploy and interact with the application. A developer can decompose an application into subservices, such as data collecting on sensors, data preprocessing on the edge, and data aggregation in the cloud. Each subservice can be abstracted as a *Firework.View* and deployed on one or more computing nodes. By organizing them with a driver program, a user can achieve real-time analytics in collaborative cloud-edge environment.

Specifically, *Firework* implements two basic programmable components, **FWView** and **FWDriver** that represent *Firework.View* and driver program, respectively. The *FWView* adopts a continuous execution model, in which a *FWView* continuously receives data from the input streams, processes the data, and sends out the data to other nodes. Listing 5.1 shows the Java code of *FWView*. As mentioned in Section 5.1.2, when a *Firework.View* is registered, a service stub is created, which contains the metadata information. Thus, the *FWView* could retrieve input and output streams from the *Job Management* layer, as shown by the *getFWInputStream()* and *getFWOutputStream()* functions in Listing 5.1. Note that, the input streams of a service depend on the base services that provide input data for that service. The most important function of *FWView* is the *compute()*, in which a user implements the payload. By calling the *run()* function, a *Firework* node repeats the actions of data receiving (via *read()*), processing (via *compute()*), and sending (via *write()*). Since an application on *Firework* is decomposed into several subservices, a developer needs to implement multiple *FWView*s, which perform different functionalities.

The other basic programmable component of *Firework* is the *FWDriver*, which provides the capabilities to deploy and launch an application. In Listing 5.2, we illustrate the basic functionalities of an application driver. The *deploy()*, *start()*, and *stop()* functions allow users to manage their applications, and the *retrieveResult()* function pulls final outcomes from a *Firework.Manager*. All these functions are conducted by *FWContext*, which maintains a session between a user and a *Firework* instance. The *DeployPlan* is a supplemental component of *FWDriver*, which describes an application defined topology. Without providing a deployment plan, *Firework* uses the network topology as the default one, which might lead to redundant computation. A user can define a rule-based deployment plan to compose subservices as needed. An example of deployment plan could be grouping sensors by regional areas so that a single *Firework* node processes all data in the same region, which is straightforward for certain application scenarios, especially when all sensors are owned by the same stakeholder. However, when a user employs subservices owned by multiple stake-

```
/* VideoStream: implements data collection on a camera. */
public class VideoStream extends FWView {
  private SensorInputStream[] sensors;
  public VideoStream(String serviceName) {
    super(serviceName);
    this.getFWInputStream();
  }
  /* Get the input stream directly from sensors. */
  @Override
  public void getFWInputStream() {
    sensors = getSensorInputStream();
  }
  /* Read data from sensor instead of FWInputStream. */
  @Override
  public Data[] read() {
    Data[] inputData = new ArrayList<Data>();
    for (SensorInputStream aSensor : sensors) {
      inputData.add(aSensor.read());
    }
    return inputData;
  }
}
```

Listing 5.3: An example of VideoStream based on FWView.

holders, the underlying network topology might not be able to aggregate all data to the user. Therefore, *Firework* provides the *DeployPlan* for users to customize the data propagation routes.

We use edge video analytics (i.e., search a targeted license plate) as an example to demonstrate how to program on *Firework*. In edge video analytics, we simplify the scenario and assume there are three nodes, including a camera, an edge node, and a cloud node. We also split the video search application into three subservices, of which the camera collects video data and sends it to the edge node; the edge node detects and recognizes a license plate from the video data and sends corresponding result to the cloud node if the targeted license plate is found; and the cloud node serves as a *Firework.Manager* and interacts with an remote user. We implement the entire application with three *FWView*-based services and a *FWDriver*-based client program. Listing 5.3 illustrates the VideoStream service on the camera, which performs data collecting. We rewrite the *getFWInputStream()* and *read()* functions since the data collection does not rely on other *Firework* service, in which case the camera directly collects data from onboard sensors and sends the data out without further computation (recall that by default *compute()* does not manipulate the data). On

```
/* VideoSearch: implements license plate recognition. */
public class VideoSearch extends FWView {
public VideoSearch(String serviceName) {
super(serviceName);
}
/* Process data using user defined function. */
@Override
public Data[] compute(Data[] inputData) {
Data[] outputData = LicensePlateDetectionAndRecognition(inputData);
return outputData;
}
}
```

Listing 5.4: An example of VideoSearch based on FWView.

```
/* VideoAnalyticsDriver: a program to exploit the service on the cloud node. */
public class VideoAnalyticsDriver {
public static void main(String[] args) {
DeployPlan deployPlan = new DeployPlan();
FWDriver fwDriver = new FWDriver("VideoAnalytics", deployPlan);
fwDriver.deploy();
Parameter[] params = Parameter.parse(args);
fwDriver.start(params);
Data[] results = fwDriver.retrieveResult();
fwDriver.stop();
}
}
```

Listing 5.5: An example of application driver.

the edge node, we implement *VideoSearch* service as shown in Listing 5.4. We rewrite *compute()* function to perform the license plate detection and recognition. We implement *VideoAnalytics* service on the cloud node and omit the code since it uses default *FWView* implementation to forward the results to a user. As a user, a driver program is needed to invoke the service. Listing 5.5 shows an example of an application driver program that interacts with *VideoAnalytics* on the cloud node.

Through the above example of real-time video analytics (i.e., license plate recognition), we design a programming interface for *Firework* to make the framework easy to use. By separating the implementation of service and driver program, *Firework* allows a third party to leverage existing services by only providing a driver program. A user can also interact with an intermediate node (e.g., the edge node in above example) to leverage the semi-finished data to build his/her own application.
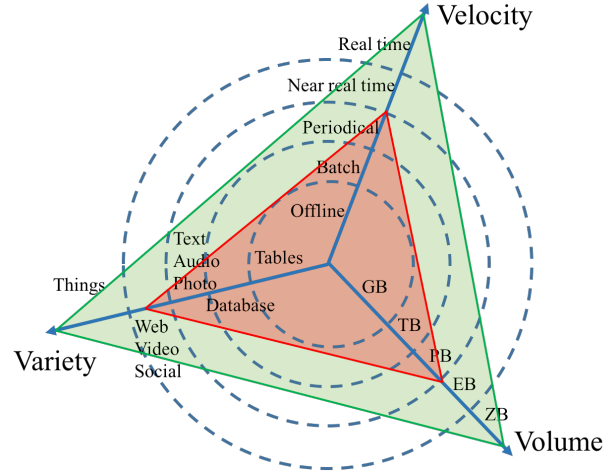
Figure 5.3: The comparison of cloud computing and edge computing.

5.1.4 Execution Model Comparison

We compare *Firework* with *Cloud Computing* and *Edge Computing* in terms of volume, variety, and velocity. As shown in Figure 5.3, *Firework* extends the capabilities of cloud computing by leveraging edge computing, where the data volume is expanded due to the data generated by IoE devices and the low latency computation is achieved by pushing computation to data sources. Specifically, *Firework* distinguishes from cloud computing and edge computing in the following aspects: i) *Firework* provides virtual data sharing among multiple stakeholders and data processing across the edge and the cloud. In contrast to *Firework*, cloud computing focuses on centralized computation resource sharing and data processing, and edge computing focuses manipulate local data with low latency and network bandwidth cost; ii) *Firework* allows data owners to define the functions that can be performed on their own data and shared with other stakeholders. The cloud computing collects data from users and defines the functions/services by the owners of clouds; iii) *Firework* reduces the network bandwidth cost by performing the computation at data sources; and iv) *Firework* leverages the cloud, as well as edge devices (and processing units placed close to the edge devices) so that the latency and network bandwidth cost can be reduced.

Up to this point, we have introduced the system design of *Firework* and illustrated the programmability of *Firework* via walking through the implementation of a potential IoE application on *Firework*, and compared with exiting computing paradigms. In the next section, we will explain the details of prototyping *Firework*.

5.2   Implementation

We implement a prototype of *Firework* using Java. Figure 5.4 shows an example architecture of our prototype system, which includes four *Firework* nodes and one *Firework* client. A *Firework* node fulfills the three-layered system design and a *Firework* client delegates an end user to communicate with *Firework* instance.

In the service management layer of a *Firework* node, the service registration is performed by the *Service Stub Manager* (shown in Figure 5.4) built on *etcd* [9], which is a key/value store accessible through RESTful interface. When a service is registered on a *Firework.Manager*, the service access portal (e.g., service name and its IP address and port number) is stored in *etcd* for persistent storage, which will also be used for recovering from a failure. *Firework* maintains an in-memory copy of all the key/value pairs to reduce performance degradation caused by querying the *etcd* with REST requests. For the same service registered by multiple *Firework* nodes, we use the same *etcd* entry to store all the service access portals. To obtain the liveness of a registered service, *Firework* periodically sends heartbeat messages to all the portals and refreshes the time-to-live attributes and the list of live portals for the corresponding *etcd* entry. Another reason we choose *etcd* is that it provides RESTful interface for a user to query available services. It is worth nothing that users can query any *Firework.Manager* to retrieve available services and compose their own applications. Another component in service management layer is the *Deployment Manager* (shown in Figure 5.4), which decides if a *Firework* node satisfies the application defined deployment plan and informs job management layer to launch services.

In the middle layer of a *Firework* node, the *Job Manager* (shown in Figure 5.4) is responsible for task decomposition, scheduling, and optimization. First, a job request is
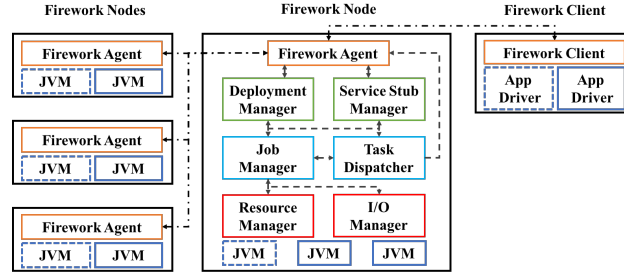
Figure 5.4: An architecture overview of *Firework* prototype.

analyzed to determine its dependencies (i.e., the services relying on), and each dependency service is notified through the *Task Dispatcher* (optional) after applying the rules in deployment plan. Then a local task is created and added to task queue. In current implementation, we use a first-come, first-serve queue for task scheduling. Finally, a task is submitted to an executor for execution. When a service is requested by multiple users, *Firework* reuses exiting running task by adding output streams to the task, where a centralized I/O manager is used in the executor management layer (explained in next paragraph) for the stream reusing.

The bottom layer is the executor management layer, where we implement the *Resource Manager* and *I/O Manager* (shown in Figure 5.4). When a task is scheduled to run, an executor (i.e., a JVM in our implementation) is allocated for it. It is worth nothing that we can extend the *Resource Manager* to be compatible with other resource virtualization tools (e.g., Docker [8], OpenStack [15]) by adding corresponding adapter. The input and output of executors are carried out by the centralized *I/O Manager*, which is implemented as message queues. An executor subscribes multiple queues as the input and output streams. The reasons we use a separated I/O manager are multifold. First, it is more efficient to manage the data transmission of an executor by dynamically adding or removing data streams to the message queue of the executor, which can be easily employed for task reuse. Second, by splitting the I/O management out from an executor, it reduces the programming efforts of developers so that they can focus on the functionalities. Third, such a design make it easy to leverage third party message queuing systems (e.g., Apache Kafka [68] and MQTT [60]). When there are huge number of sensors reporting to a single aggregation node, it makes
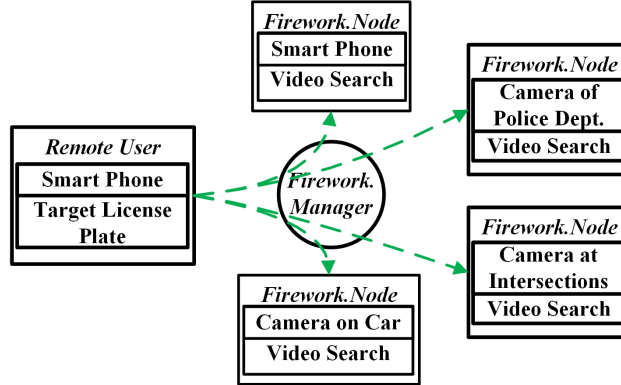
Figure 5.5: A *Firework* instance for searching a target license plate.

Firework more scalable by simply adding more aggregation nodes and subscribing from the queuing systems that guarantee exactly-once data processing semantics. Fourth, a unified system level security protection can be applied on top of the I/O communication to guarantee data integrity and fidelity. Therefore, a separated I/O manager is used in *Firework*.

By deploying on multiple computing nodes, an instance of *Firework* system can be materialized. As shown in Figure 5.4, multiple *Firework* nodes communicate with each other via the *Firework Agent* and form different topology based on application defined deployment plan. Note that we use star topology in Figure 5.4 as an example topology of a *Firework* instance. A user can interact with *Firework* using the utilities provided by the *Firework Client* and deploy multiple applications (the solid-line rectangles and dashed-line rectangles in Figure 5.4) on the same *Firework* instance.

Up to this point, we have introduced the implementation details of the prototype system of *Firework*. In the next section, we will show a case study of an edge video analytics on *Firework*.

5.3   Case Study: AMBER Alert Assistant

AMBER Alert system is used to alert the public of worrying or life-threatening disappearances of child(ren), and different countries have similar systems [18]. When a kidnapping occurs, an alert message is immediately sent to cell phones around the scene area, which includes description about the criminal, such as time, location, suspect vehicle license plate

number, et cetera. The witnesses can provide related information to the public safety authorities. However, it is inefficient to track the suspect/vehicle since it heavily depends on the reports of witnesses. On the other hand, video cameras is widely deployed in urban areas, including security cameras, traffic cameras, on-dash cameras, and even mobile cameras, which can be used for object tracking. This provides the opportunity to leverage existing video cameras/data to improve the efficiency of suspect/vehicle tracking for *AMBER Alert system* . Using automatic number plate recognition technique, video surveillance system is of great improvement for vehicle tracking. Coped with the privacy issues, more cameras (e.g., on-dash car cameras, smartphone cameras) might participate the system.

In Figure 5.5, we illustrate a high level overview of a *Firework* instance for searching a target license plate in the urban area, which is common in the AMBER alter system. In such scenario, when a license plate is wanted, it is very likely that this object is captured by cameras around located at either fixed locations or mobile carriers in the urban area. In cloud computing, the video data captured by the cameras has to be uploaded to the cloud to identify the target license plate. However, the data transmission is still costly, which makes it extremely difficult and inefficient to leverage the wide area video data. With *Firework* paradigm, a request of searching the target license plate is created at a remote user's device. Then the target license plate number is distributed to connected devices with cameras and each device performs the license plate searching using archived local data or real-time video stream. The requester gathers the results from other *Firework.Node*s to locate the object. With *Firework*, the video data is processed at the edge and the response latency and network bandwidth cost will be significantly reduced. An extension of this video analytics could be real-time object tracking, or event detection, which is common in public safety applications, where the GPS information (e.g., smart phones and vehicles) can be used for multiple purposes.

In this case study, we propose and implement AMBER Alert Assistant system, which aims to improve the efficiency of target tracking (i.e., suspect vehicle tracking) of AMBER

alert system on top of our *Firework* framework. We demonstrate the capabilities of *Firework* using the edge video analytics of real-time license plate recognition. We deploy *Firework* in a testbed, implement the AMBER alert assistant application with *Firework*'s the programming interfaces, and compare the performances in terms of response latency and network bandwidth cost between *Firework* with different deployment plans and a cloud-centric solution.

5.3.1 Experimental Setup

In this case study, we implement the AMBER alert assistant service in Java using FFmpeg [10], OpenCV [19], and modified Openalpr [14]. Specifically, we split the AMBER alert assistant service into three subservices: *VideoStream*, *PlateDetection*, and *PlateRecognition*. The *VideoStream* (VS) sends the video data from a camera to an edge node. The *PlateDetection* (PD) detects if a license plate appears in a video frame and crops the area of the license plate in the video frame. The *PlateRecognition* (PR) reads the numbers and alphabets from the images generated by the *PlateDetection*. Then, these subservices are packaged into JAR files and deployed to different computing nodes. A remote user invokes the AMBER alert assistant service and *Firework* automatically starts the subservices and returns the results. The response latency in our experiments is defined as the time duration between a video frame is sent out by a camera and a remote user receives the decision if this video frame contains the target license plate.

As a baseline, we implement a cloud-centric video analytics system, where a camera directly pushes video data to a cloud node and the cloud node detects and recognizes license plate for each video frame and sends the result to a remote user.

Figure 5.6 shows the network topology of our testbed. To simulate live video captured by cameras, we replay a prerecorded traffic video clip collected in a large-scale campus with twenty-five thousand students. The video clip is transformed into four different resolution qualities including 1280×720 (720P), 1920×1080 (1080P), 2560×1440 (1440P), and 3840×2160 (2160P), and the frames per second is set to 30. The video data is encoded
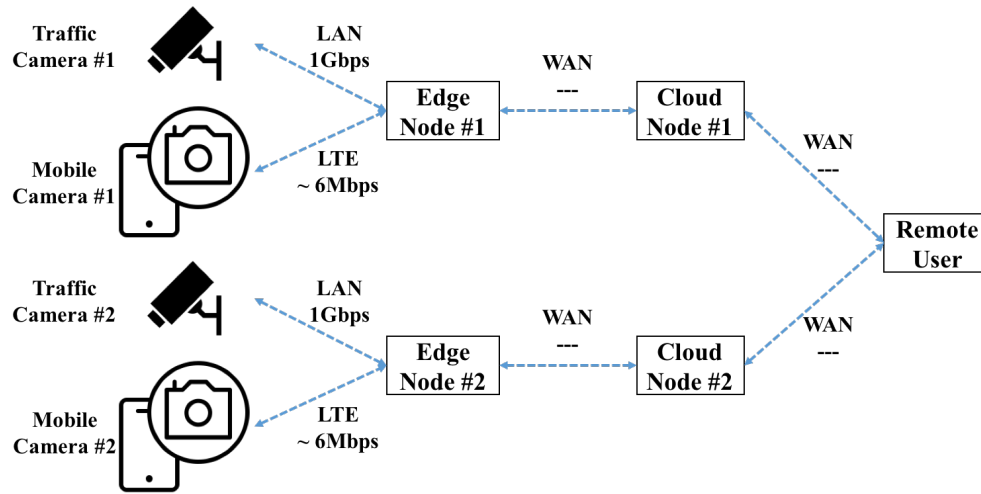
Figure 5.6: Network topology of with available upload bandwidths.

in H.264 format with baseline profile and we configure that one intra-frame (IFrame) is followed by fifty-nine predictive-frames (PFrames) without bi-directional frame (BFrame), because we simulate a live video stream and cannot compute the differences between the current frame and the next frame. The data is sent to edge nodes using real-time transport protocol (RTP) over UDP/IP network. In our experiments, we calculate the average size of one video frame for 720P, 1080P, 1440P, and 2160P individually and the sizes are 15.08KB, 29.38KB, 51.63KB, and 95.44KB, respectively. Note that these sizes might vary in different runs.

As shown in Figure 5.6, the cameras are connected to two edge nodes using LAN (the upload bandwidth is 1Gbps) and LTE (the upload bandwidth is around 6Mbps based on our speed test). The bandwidth of LAN is shared with other users and the available bandwidth for our experiments might vary over time. The edge nodes in Figure 5.6 are located at a computer lab within Wayne State University campus, which are two servers with the same hardware configuration, i.e., Intel E5620@2.4GHz (8 cores, 16 threads) and 16GB memory. The cloud nodes in Figure 5.6 are two m4.4xlarge virtual machine instances in Amazon EC2 data center located at US East (N. Virginia). The available network bandwidths of WANs are missing since they depend on the network traffic and certain service providers (e.g., Amazon

EC2). Each node shown in Figure 5.6 runs *Firework* and hosts one or multiple subservices depending on the deployment plan. Note that, the cameras implement *Firework.View*s (i.e., VS) and run as *Firework.Node* that registers VS as a subservice to the edge nodes. The edge nodes act as the *Firework.Manager* from the cameras' perspective, and at the same time they are also *Firework.Node*s because PD is another *Firework.View* extended from VS and registered to the cloud nodes. The cloud node acts as the *Firework.Manager* of the edge nodes.

As aforementioned, subservices can be loaded on sensors, edge nodes and cloud nodes according to the deployment plan. In our experiments, we deploy three subservices (i.e., VS, PD, and PR) to cameras, edge nodes and cloud nodes with various policies. We summarize all cases in Table 5.2. Given the network topology in Figure 5.6, the VS is always on cameras, but the PD and PR can be deployed to edge nodes or cloud nodes. In Case#1, cameras upload video data to edge nodes. The edge nodes detect and recognize license plates and send the results first to cloud nodes even though the data is not manipulated on the cloud nodes. Finally the cloud nodes forward the results to a remote user. Similarly, the edge nodes in Case#3 only forward the video data to cloud nodes and the cloud nodes perform the entire computation (i.e., PD and PR). Note that Case#3 is similar to the cloud-centric baseline solution, but in the baseline a camera directly pushes video data to cloud nodes. In Case#2, the edge nodes detect if a video frame contains a license plate and only send the cropped frames with license plates to cloud nodes. Otherwise, video frames without license plates are dropped to reduce network transmission cost.

5.3.2 Collaboration of Edge Nodes

To profile the resource requirement for real-time license plate recognition, we implement PD and PR on a single computing node with different number of workers and collect the response latency. As an example, we use 720P video clip as the input video stream. Figure 5.7 shows the response latency for a 720P video stream when using different number of workers for PD and PR. Generally, the less resource (i.e., number of workers), the higher
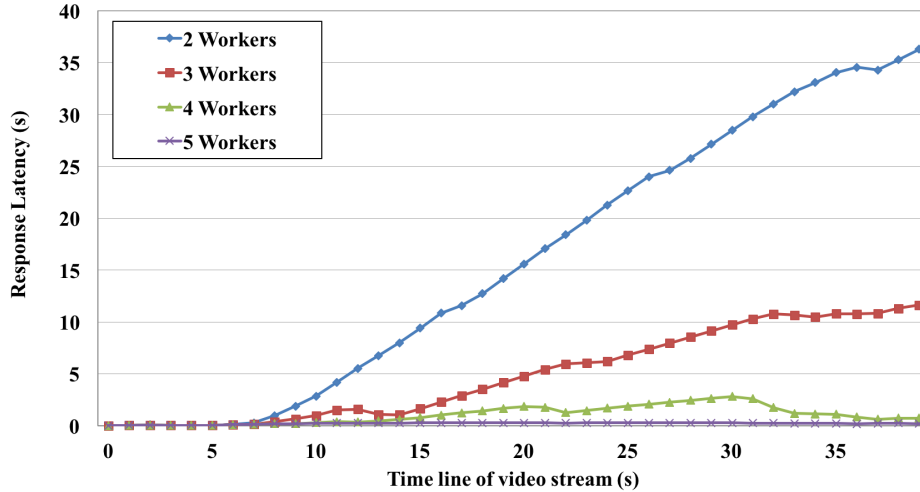
Figure 5.7: Response latency for a 720P video stream using different number of workers.
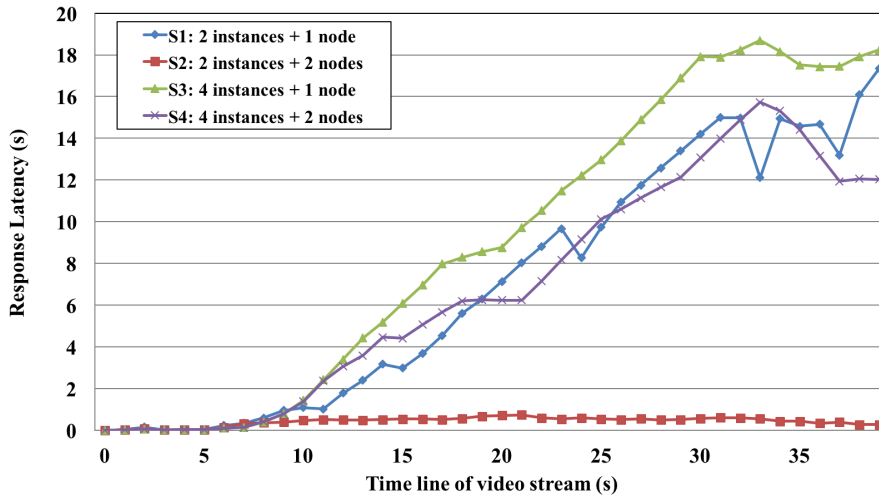
Figure 5.8: Response latency for different scenarios using collaborative edge nodes.

response latency. For a 720P video stream, five workers are least required to guarantee real-time processing. Note that for higher quality video streams (e.g., 1080P, 1440P, 2160P), the minimum number of workers to deliver real-time processing might be different due to larger size of input data.

To demonstrate the collaboration among edge nodes, we use two edge nodes in the same LAN to collaboratively analyze video streams. The video steam is 720P and we limit the maximum number of available workers on each edge node to five according to aforementioned experimental results. As comparison, we evaluate the performance in four different scenarios.

We run different number of instances for license plate recognition on one or two edge nodes, including *S1: 2 instance + 1 node*, *S2: 2 instances + 2 nodes*, *S3: 4 instances + 1 node*, and *S4: 4 instances + 2 nodes*.

Figure 5.8 shows the results for these four scenarios. For all scenarios, initially we run one instance on one node and allocate more instances at the fifth second. Thus, for first five seconds, the response latency of all four scenarios are similar and relatively low. For scenario S1, one instance is allocated at the fifth second. As we mentioned, the maximum number of workers on each node is five, and there is not enough resources for two instances running on one node. Thus, the response latency gradually increases. Similarly in scenario S3, three more instances are assigned at the fifth second and more severe resource contention leads to higher latency than the scenario S1.

For the scenario S2, when a new instance is allocated, the node with initial instance requests for workload migration due to limited resources, and the new instance is migrated to the other node. Therefore, the response latency remains low during the entire experiment since there is enough resources on two nodes for running two instances. Note that when the new instance is migrated to the second node, the video stream is also forwarded to the second node. Thus, the response latency for the new instance is relative higher than the initial instance due to the overhead of video stream forwarding. For the scenario S4, although two of the three new instances are migrated to the second node, there is still not enough resources to handle four instances and thus the response latency increases gradually.

5.3.3 Response Latency

We compare the response latencies under different deployment plans and network conditions. We collect the response latencies for $20,000$ video frames that contain license plates in each experiment explained below.

Figure 5.9-5.12 shows the response latencies regarding to distinct video resolutions using LAN connection. In general, the more workload is offloaded to edge nodes, the lower response latency is achieved, for any given video resolution. Compared to the baseline, the

Table 5.2: Subservice deployment plans

	Cameras	Edge Nodes	Cloud Nodes
Case#1	VS	PD&PR	–
Case#2	VS	PD	PR
Case#3	VS	–	PD&PR
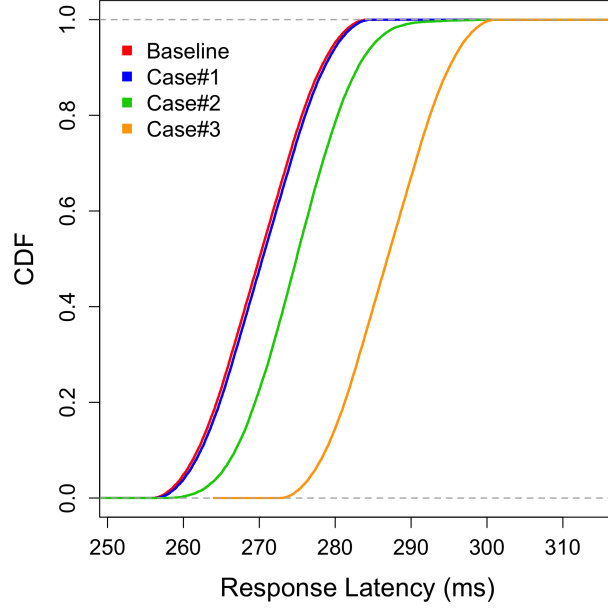
Figure 5.9: Response latencies regarding to 720P video stream using LAN connection.

higher video resolution, the lower response latency is achieved. More specifically, when the video resolution is relatively low (i.e., 720P shown in Figure 5.9), Case#1 achieves similar response latency as the baseline. Case#3 has higher latency than the baseline but only increased by 17ms on average. For video stream of 1080P shown in Figure 5.10, Case#1 outperforms the baseline and Case#2 achieves similar performance as the baseline. When the video resolution is relatively high (i.e., 1440P and 2160P in Figure 5.11 and 5.12), Case#1 and Case#2 have lower response latencies than the baseline. The response latencies are reduced by up to 6.42% and 11.62% for 1440P and 2160P, respectively. Case#3 has the highest response latency in all scenarios since the video data is first pushed to the edge nodes and then forwarded to the cloud nodes so that the data transmission time is much higher than the other cases.

Figure 5.10: Response latencies regarding to 1080P video stream using LAN connection.

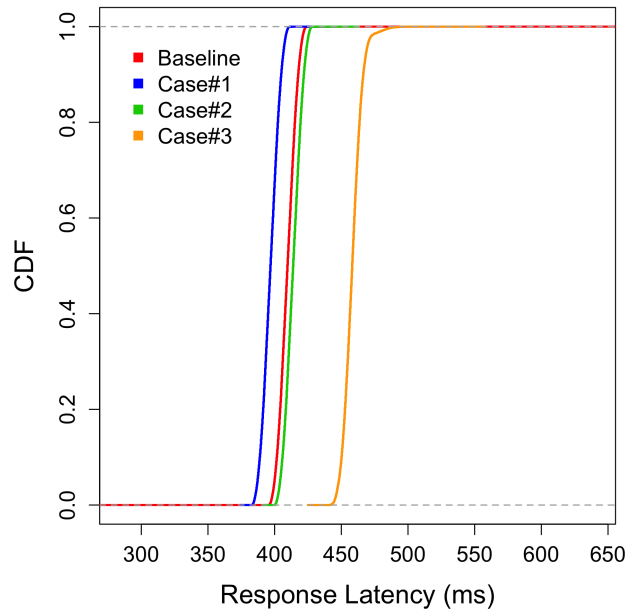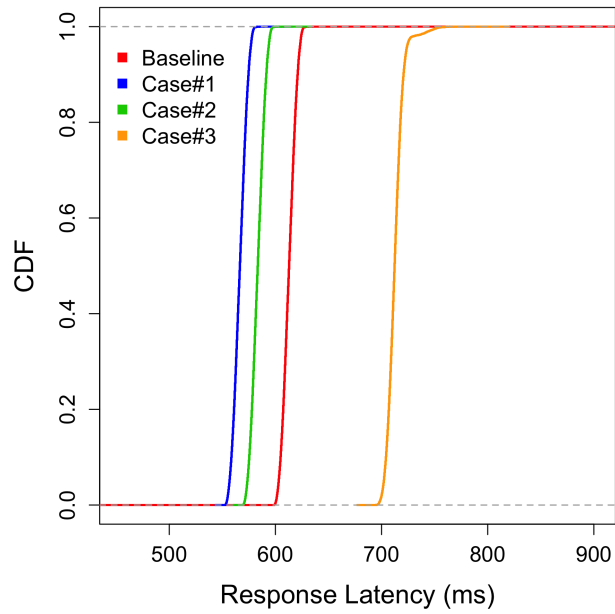Figure 5.11: Response latencies regarding to 1440P video stream using LAN connection.
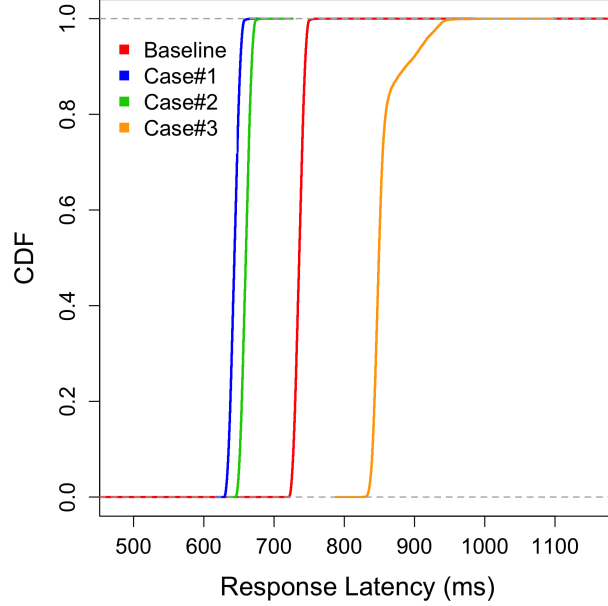
Figure 5.12: Response latencies regarding to 2160P video stream using LAN connection.

When cameras are connected with LTE connection, the video data can be uploaded to either edge nodes or cloud nodes only when the video resolution is low (i.e., 720P and 1080P). Transmitting video data with higher resolutions requires higher upload network bandwidth, which is limited with LTE connection. The LTE connection also suffers significantly high frame loss rate so that the edge nodes and cloud nodes cannot detect and recognize a license plate with incomplete frame data (explained in Section 5.3.4). Therefore, we only show the response latencies for 720P and 1080P video streams when using LTE connection.

In Figure 5.13 and 5.14, we show the response latency when cameras are physically fixed in one location and connected to a static LTE connection, which means there is no cellular base station switching during the experiments. The response latencies are reduced by up to 19.52% and 8.15% for 720P and 1080P video streams, respectively. Case#3 achieves higher latencies than the baseline for both 720P and 1080P video streams since the video data is transmitted to the cloud nodes through the edge nodes.

We also conduct the experiments in driving condition to simulate a vehicle video analytics scenario, where the LTE connection is dynamically changing and involves cellular
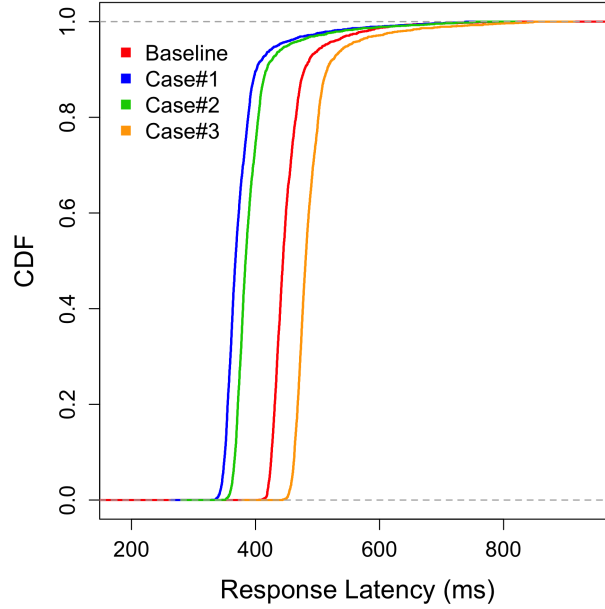
Figure 5.13:   Response latencies of 720P video stream using **static** LTE connection.

base station switching. We denote this as *dynamic LTE connection* in the rest of this chapter. We upload video data on a vehicle driving at 35 miles per hour circling around urban area of Detroit.   Figure 5.15 and 5.16 illustrates the response latencies when using dynamic LTE connection. Similarly, Case#1 also outperforms the baseline solution by reducing up to 11.12% and 7.9% of response latencies for 720P and 1080P video streams, respectively. Differentiating from the case of static LTE connection, dynamic LTE connection suffers from long tail latencies and higher variation due to varying signal strength in a moving vehicle and cellular base station switching.

We further break down the response latency into three parts, including transmission time, encoding/decoding time, and detection/recognition time. Figure 5.17 shows the average response latency under different scenarios. Generally, the detection and recognition time contributes the majority of response latency with LAN connection, while the data transmission time contributes the majority with LTE connection. Case#3 has the highest transmission time with any video resolution for both LAN and LTE connections since the edge nodes receive and forward video data to cloud nodes instead of directly uploading the
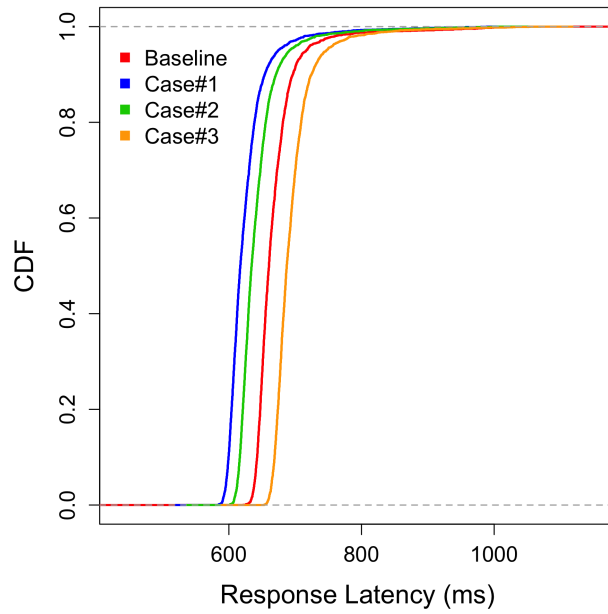
Figure 5.14: Response latencies of 1080P video stream using **static** LTE connection.
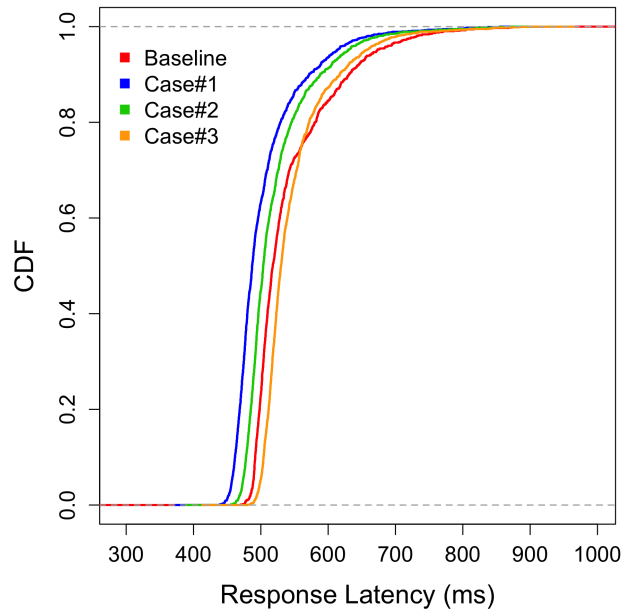
Figure 5.15: Response latencies of 720P video stream using **dynamic** LTE connection.
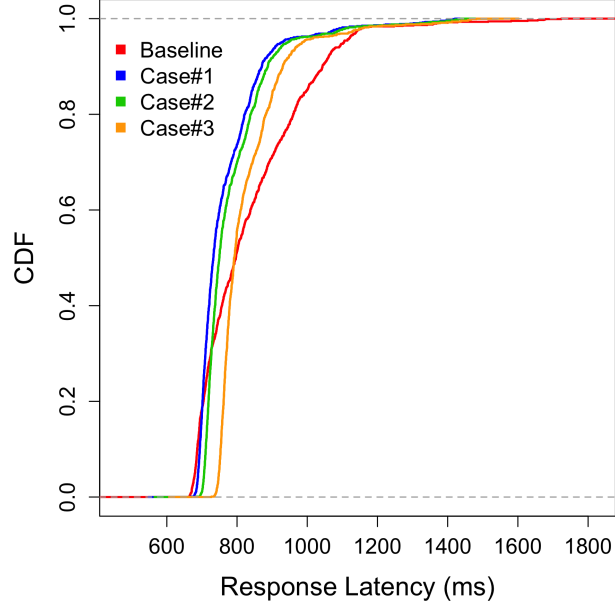
Figure 5.16: Response latencies of 1080P video stream using ***dynamic*** LTE connection.

video data to cloud nodes from cameras, which also leads to higher response latency as explained in Figure 5.9-5.12.

More specifically, as shown in Figure 5.17, the transmission time and encoding/decoding time increase as the video resolution enhances using either LAN or LTE connection. When the video resolution is high (i.e., 1440P or 2160P) with LAN connection, Case#1 and Case#2 reduce the transmission time since the video data is only uploaded to edge nodes that are closer to the cameras in terms of network distance compared to cloud nodes. When using LTE connection, the transmission time of Case#1 is reduced significantly compared to the baseline even with relatively low video resolutions. However, the average response latencies for both 720P and 1080P video streams are increased when using LTE connection compared to the cases with LAN connection because the bandwidth of LTE connection is much less than that of LAN connection.

5.3.4 Frame Loss

In addition to the response latency, we count the number of lost frames under LAN and LTE connections during the experiments. When using LAN connection, the frame loss
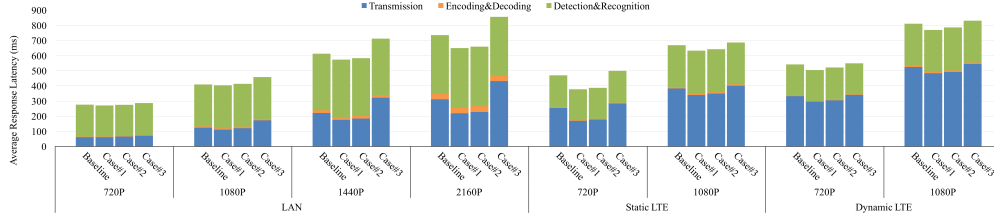
Figure 5.17: The time breakdown of average response latencies.

Table 5.3: Network bandwidth cost reductions compared to baseline

	Case#1	Case#2	Case#3
720P	∼100%	73.47%	0%
1080P	∼100%	72.77%	0%
1440P	∼100%	78.69%	0%
2160P	∼100%	86.38%	0%

rate is less than 0.9% for all four different video resolutions. However, when using LTE connection, the frame loss rates of 720P and 1080P are less than 2.73%, while for 1440P and 2160P, the frame loss rates are more than 63.71% due to the limited network bandwidth, which make it impossible to decode a valid frame base on incomplete IFrames or PFrames. Note that the network package loss rate of LTE connection when uploading 1440P and 2160P is less than 9.02%, which is much lower than the frame loss rate. Given the size of one video frame, it is common that the data of one video frame is transmitted using multiple network packages. We count one frame loss if at least one of the packages of the frame is lost. Moreover, one IFrame loss incurs the loss of the following fifty-nine PFrames since each video frame is decoded by using both IFrame and PFrame. Therefore, the frame loss rate is much higher than network package loss rate.

5.3.5   Network Bandwidth Cost

With respect to the network bandwidth cost, *Firework* leverages edge nodes to perform part of the computation so that the amount of data transmitted over edge-to-cloud connections (i.e., the connections between EdgeNode and CloudNode in Figure 5.6) is reduced. Table 5.3 summarizes the bandwidth cost reductions of the edge-to-cloud connections compared to the baseline solution for the three cases in Table 5.2, regarding to four different

of video resolutions. In Case#1, the bandwidth cost is almost eliminated since all computation (i.e., PD and PR) is performed on the edge nodes and only negligible data (i.e., the recognition result) is sent to the cloud nodes. Thus, compared to the baseline in which the entire video data is transmitted to the cloud nodes, the bandwidth cost reduction is close to 100%. On the contrary, in Case#3 the edge nodes transmit the entire video data to the cloud nodes, thus the network bandwidth cost is the same as the baseline, which means 0% reduction.

For Case#2, the reduction rates vary over video resolutions. In Case#2, data transmitted over the edge-to-cloud connections is the extracted area of license plate in a video frame. The reduction rate is also affected by the size of the area of a license plate, which varies a lot over different video resolutions and the distance between a camera and an object (e.g., a license plate). Therefore, we take the average size of the output images generated by *PlateDetection* on the edge nodes to compute the reduction rate. To calculate the reduction rate for Case#2, we assume that every video frame contains exactly one license plate, of which the bandwidth cost reduction is the lower bound because a video frame without a license plate is dropped. In extremely case the reduction rate is close to 100% (i.e., no license plate appears in any video frame). Given the aforementioned assumption for Case#2, Table 5.3 shows that the higher video resolution quality, the larger reduction rate is achieved because most area of a video frame is cropped except the license plate so that the network bandwidth reduces significantly when video resolution is higher.

5.4 Discussion

In this dissertation, we narrow down the scope of *Firework* to prototyping and programming interface implementation. In this section, we discuss potential issues and limitations of *Firework*, in terms of system design and performance optimization

Privacy: Data captured by IoE devices can contain private information, e.g., GPS data, streams of video or audio, which might be used for complex analytics at somewhere other than where the data is generated. Thus, it is critical that, only data that is privacy compliant

is sent to the edge or the cloud for further analysis. As we mentioned, *Firework* supports privacy preserving function, which can be adopted by implementing a function, such as face blurring of video frames in [98, 105], as predefined function of a *Firework.View*. Since privacy preserving function is attached to the shared subserives of each service owner, it is feasible for a downstream subservice to apply different privacy policies by extending existing subservies (i.e., extending a *Firework.View* to add/override exiting privacy preserving functions). In addition, *Firework* manages data communication using a separate I/O controller, where an easy security enhancement can be added by using secure communication protocols.

Fault Tolerant: In the prototype of *Firework*, the *Job Manager* (shown in Figure 3.8) tries to restart a job when the job fails due to software failure (e.g., out of memory, uncaught exceptions). However, the *Job Manager* cannot restart a job when the underly hardware fails. In our license plate recognition example, if all cameras fail and the VS is unavailable, the PD and PR are still running on the edge nodes and/or the cloud but a user cannot get any output. In such case, *Firework* restores the VS whenever a camera is restored. Since *Firework* leverages computing resource that are owned by stakeholders and not controlled by *Firework*, there is no guarantee that an unavailable subservice would be available in the near future. Thus, the fault tolerance in *Firework* depends on the underly fault tolerant mechanisms of stakeholders that might be very different. Thus, we leave the fault/failure detection in *Firework* as a future work.

Optimization: To simplify the scenario, we assume that an application can be decomposed and represented by a sequence of n functions, and m computing nodes are connected in a line. The goal of optimizing computation offload is to minimize the end-to-end latency by optimizing the allocation of n functions over m nodes, where the functions have to be allocated sequentially. In current implementation of *Firework*, we use a simple deployment policy, in which the optimization target is a weighted sum of response latency and network bandwidth cost. Using the default deployment policy, it is possible to assign all subservices on one edge node (e.g., a smart phone), in which case the response latency (e.g., only including the time

used for license plate detection and recognition) and network bandwidth cost(e.g., there is no data transmitted through network since all data are consumed locally on the smart phone) are minimized. However, it leads to high power consumption, which is infeasible and leads to short battery life. Furthermore, automatic functionality decomposition of an application increases the difficult to optimize the function placement because the optimization goal of function decomposition might be contrast to that of function placement. Therefore, we leave the automatic functionality decomposition and workload placement/migration as a future work so that *Firework* provides efficient algorithm that co-optimizes these goals with little user intervention.

5.5 Summary

Real-time video analytics becomes more and more important to IoE applications due to the richness of video content and the huge potential of unanticipated value. To undertake the barriers of deploying IoE applications, we introduce a new computing framework called *Firework* that is a data processing and sharing platform for hybrid cloud-edge analytics. We illustrate the system design and implementation and demonstrate the programmability of *Firework* so that users are able to compose and deploy their IoE applications over various computing resources at the edge of the network and in the cloud. The evaluation of an edge video analytics application shows that *Firework* reduces response latencies and network bandwidth cost when using either LAN or LTE connection, compared to a cloud-centric solution. For the future work, we will explore automatic service/functionality decomposition so that *Firework* could dynamically optimize the subservice deployment according to the usage of computing, network, and storage resources on computing nodes.

Up to this point, we have presented the *Firework* framework that allows stakeholders and users to effectively and efficiently share and develop cloud-edge analytics applications. Combining with the H_2O, DyBBS and *Firework*, we implement a series of systems that improve the performance of stream processing systems and facilitate the development of big

data analytics in cloud-edge environment. In the next chapter, we will conclude our research results by summarizing the contributions of this dissertation.

CHAPTER 6: CONCLUSION

Stream processing is a critical technique to process huge amount of data in real-time manner. Cloud computing has been used for stream processing due to its unlimited computation resources. At the same time, we are entering the era of Internet of Everything (IoE). The emerging edge computing benefits low-latency applications by leveraging computation resources at the proximity of data sources. Billions of sensors and actuators are being deployed worldwide and huge amount of data generated by things are immersed in our daily life. It has become essential for organizations to be able to stream and analyze data, and provide low-latency analytics on streaming data. However, cloud computing is inefficient to process all data in a centralized environment in terms of the network bandwidth cost and response latency. Although edge computing offloads computation from the cloud to the edge of the Internet, there is not a data sharing and processing framework that efficiently utilizes computation resources in the cloud and the edge. Furthermore, the heterogeneity of edge devices brings more difficulty to the development of collaborative cloud-edge applications. In this dissertation, we explore a scenario of large-scale video surveillance for public safety and design and implement a series of systems to support vision-based stream processing applications in hybrid cloud-edge environment.

First, we introduce H_2O, a hybrid and hierarchical outlier detection method for multivariate time series. Instead of fitting a single type of model on all the variables, we propose a hybrid method which employs an ensemble of models to capture the diverse patterns of variables. A hierarchical model selection process is applied to select the best anomaly detection models for variables based on their time series characteristics, following a global to local fashion. We also develop a new seasonal-trend decomposition based detection method for multivariate time series, which considers the covariation and interactions among variables.

Second, we optimize one of the stream processing system (i.e., Spark Streaming) to reduce the end-to-end latency. We illustrate an adaptive control algorithm for batched processing system by leveraging dynamic block and batch interval sizing. Our algorithm is able

to achieve low latency without any workload specific prior knowledge, which is comparable to the oracle case due to our accurate batch interval estimation and novel execution parallelism tuning.

ThirdïijŇto facilitate the development of collaborative cloud-edge applications, we propose and implement a new computing framework, *Firework* that allows stakeholders to share and process data by leveraging both the cloud and the edge. We illustrate the system design and implementation and demonstrate the programmability of *Firework* so that users are able to compose and deploy their IoE applications over various computing resources at the edge of the network and in the cloud.

Last, we implement a vision-based cloud-edge application A3 for AMBER alert system, which is a real-time vehicle tracking application for public safety. Built on our *Firework* framework, A3 can analyze the video streams from traffic cameras in real time by collaborating among local edge nodes, and select participant cameras efficiently depending on customizable tracking policy.

In the next chapter, we will present potential research topics related to this dissertation for the future.

CHAPTER 7: FUTURE WORK

In the future, it will be beneficial to investigate how to extend and improve our approaches in this dissertation to other streaming applications (e.g., apply H_2O to healthcare scenario) and further improve the performance of Spark Streaming and *Firework*.

First, H_2O is designed for multivariate time series and we assume that the behavior pattern of these time series are static with in the detection window. False alerts occurs if the behavior pattern of time series changes within the detection window. Thus, it is important for H_2O to handle such pattern change, which is one of our future work to detection change point of multivariate time series to further reduce false alerts. On the other hand, H_2O method by its nature is very general and can be applied to detect anomalies over multivariate time series in many other domains, such as IT system health monitoring and fault detection. For example, in the IT system health monitoring, disk failure detection [58] could be a scenario where H_2O can directly be applied to find abnormal data point to avoid disk/server down time. For healthcare case, the ECG and EEG [76, 75, 80] signals are also treated as multivariate time series so that H_2O can be used to detection abnormal behaviors of these signals.

Second, in this dissertation, we narrowed down the scope of DyBBS to two workloads with simple relationships (i.e., linear for *Reduce* workload and superlinear for *Join* workload). However, DyBBS cannot handle complex workload, such as *Window* operation, which aggregates the partial results of all batches within a sliding window. In our approach, there are two issues for handling *Window* workload. One is that it is possible that we cannot find a serial of consecutive batches such that the total time interval exactly equals the window duration. If we always choose the next batch interval such that the previous condition is satisfied, then we lose the opportunity to optimize that batch, as well as all the batches in the future. Another problem is that our approach dynamically changes the block interval, which means our approach cannot directly employ the method used in [37]. Therefore, we leave the work to support *Window* workload for the future.

Third, we will explore automatic task decomposition, which plays an important role in *Firework* framework, because we aim at providing easy-to-use framework, reducing the effort a user must put into developing an edge application as much as possible. Task decomposition is obviously an important but difficult problem. The reason lies in the fact that it is very hard to determine the boundaries of the computational pieces for each edge application, and the task decomposition strategies are entangled with the task scheduling strategies. Splitting edge applications with carelessly chosen boundaries will make efficient task scheduling a mission impossible and eventually lead to poor system performance. Therefore, we plan to explore and propose efficient decomposition algorithms so that users can customize their applications with little effort, informing the framework about how the applications should be split into computational pieces, and our framework could precisely figure out the user hint from the invocations and tune the task decomposition decision accordingly if possible.

REFERENCES

[1] "Amazon Kinesis," https://aws.amazon.com/kinesis/, [Online; accessed Sep. 1st, 2016].

[2] "Amazon Mechanical Turk," https://www.mturk.com/mturk/welcome, [Online; accessed Sep. 1st, 2016].

[3] "Apache flume," https://flume.apache.org/, [Online; accessed April 20, 2016].

[4] "Apache Quarks," http://quarks.incubator.apache.org/, [Online; accessed Sep. 1st, 2016].

[5] "Apache storm," https://storm.apache.org/, [Online; accessed April 20, 2016].

[6] "Aws iot," https://aws.amazon.com/iot/, [Online; accessed April 20, 2016].

[7] "Cisco global cloud index: Forecast and methodology 2014âĂŞ2019 white paper," http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf, [Online; accessed April 20, 2016].

[8] "Docker," https://www.docker.com/, [Online; accessed Sep. 1st, 2016].

[9] "etcd," https://github.com/coreos/etcd, [Online; accessed Sep. 1st, 2016].

[10] "FFmpeg," https://ffmpeg.org/, [Online; accessed Dec. 1st, 2016].

[11] "Google cloud platform: Iot solution," https://cloud.google.com/solutions/iot/, [Online; accessed April 20, 2016].

[12] "IBM Infosphere streams," www.ibm.com/software/products/en/infosphere-streams, [Online; accessed Sep. 1st, 2016].

[13] "Markov chain," https://en.wikipedia.org/wiki/Markov_chain.

[14] "Openalpr," https://github.com/openalpr, [Online; accessed Dec. 1st, 2016].

[15] "Openstack," https://www.openstack.org/, [Online; accessed Sep. 1st, 2016].

[16] "RabbitMQ," https://www.rabbitmq.com/, [Online; accessed Dec. 1st, 2016].

[17] "Samza," http://samza.apache.org/, [Online; accessed Sep. 1st, 2016].

[18] "AMBER alert," March 2017. [Online]. Available: https://en.wikipedia.org/wiki/AMBER_Alert

[19] "OpenCV," March 2017. [Online]. Available: http://www.opencv.org/

[20] C. C. Aggarwal, "Outlier ensembles: position paper," *ACM SIGKDD Explorations Newsletter*, vol. 14, no. 2, pp. 49–58, 2013.

[21] R. Agrawal and R. Srikant, "Privacy-preserving data mining," in *ACM SIGMOD Record*, vol. 29, no. 2.   ACM, 2000, pp. 439–450.

[22] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[23] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure, "Adaptive control of extreme-scale stream processing systems," in *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on.*   IEEE, 2006, pp. 71–71.

[24] G. Ananthanarayanan, P. Bahl, P. Bodik, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *Computer*, vol. 50, no. 10, pp. 58–67, 2017.

[25] Apache, "Apache storm trident," https://storm.apache.org/documentation/Trident-tutorial.html, [Online; accessed June 12, 2015].

[26] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[27] V. Bamnett and T. Lewis, "Outliers in statistical data," 1994.

[28] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the Mobile Cloud Computing.*   ACM, 2012, pp. 13–16.

[29] D. Booth and C. K. Liu, "Web services description language (wsdl) version 2.0 part 0: Primer," *W3C Recommendation*, vol. 26, 2007.

[30] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: identifying density-based local outliers," in *ACM sigmod record*, vol. 29, no. 2.   ACM, 2000, pp. 93–104.

[31] H. Brunk, R. Barlow, D. Bartholomew, and J. Bremner, "Statistical inference under order restrictions.(the theory and application of isotonic regression)," DTIC Document, Tech. Rep., 1972.

[32] R. Chinnici, H. Haas, A. A. Lewis, J.-J. Moreau, D. Orchard, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 2: Adjuncts," *W3C Recommendation*, vol. 6, 2007.

[33] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 1: Core language," *W3C Recommendation*, vol. 26, 2007.

[34] R. B. Cleveland, W. S. Cleveland, J. E. McRae, and I. Terpenning, "Stl: A seasonal-trend decomposition procedure based on loess," *Journal of Official Statistics*, vol. 6, no. 1, pp. 3–73, 1990.

[35] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*. ACM, 2010, pp. 49–62.

[36] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons *et al.*, "Exploiting bounded staleness to speed up big data analytics," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 37–48.

[37] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive stream processing using dynamic batch sizing," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–13.

[38] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[39] G. Edmond and M. San Roque, "Justicia's gaze: surveillance, evidence and the criminal trial," *Surveillance & Society*, vol. 11, no. 3, p. 252, 2013.

[40] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[41] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, vol. 1, pp. 1–11, 2011.

[42] A. Fahim, A. Mtibaa, and K. A. Harras, "Making the case for computational offloading in mobile device clouds," in *Proceedings of the Conference on Mobile Computing and Networking.* ACM, 2013, pp. 203–205.

[43] N. Fernando, S. W. Loke, and W. Rahayu, "Computing with nearby mobile devices: a work sharing algorithm for mobile edge-clouds," *IEEE Transaction on Cloud Computing*, 2016.

[44] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "Crowddb: answering queries with crowdsourcing," in *Proceedings of the International Conference on Management of Data.* ACM, 2011, pp. 61–72.

[45] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1447–1463, 2014.

[46] P. Goodwin and L. Conner, "Worldwide data protection and recovery software market shares," https://www.idc.com/getdoc.jsp?containerId=US41573316.

[47] R. Greenaway-McGrevy, "A multivariate approach to seasonal adjustment," 2013.

[48] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 68–73, 2008.

[49] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "Simple object access protocol (soap) 1.2," *World Wide Web Consortium*, 2003.

[50] N. Günnemann, S. Günnemann, and C. Faloutsos, "Robust multivariate autoregression for anomaly detection in dynamic product ratings," in *Proceedings of the 23rd international conference on World wide web.* ACM, 2014, pp. 361–372.

[51] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," in *Proceedings of the International Conference on Cloud Computing.* IEEE, 2015, pp. 9–16.

[52] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," in *Proceedings of the 1st ACM symposium on Cloud computing.* ACM, 2010, pp. 63–74.

[53] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, 2011, pp. 22–22.

[54] P. Hintjens, *ZeroMQ: Messaging for Many Applications.* "O'Reilly Media, Inc.", 2013.

[55] E. Holmes, E. Ward, and K. Wills, "Package marss," https://cran.r-project.org/web/packages/MARSS/MARSS.pdf.

[56] L. Hu, K. Schwan, H. Amur, and X. Chen, "Elf: efficient lightweight fast stream processing at scale," in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference.* USENIX Association, 2014, pp. 25–36.

[57] Y. Hu, W. Murray, and Y. Shan, "Package rlof," https://cran.r-project.org/web/packages/Rlof/Rlof.pdf.

[58] S. Huang, S. Fu, Q. Zhang, and W. Shi, "Characterizing disk failures with quantified disk degradation signatures: An early experience," in *IEEE International Symposium on Workload Characterization.* IEEE, 2015, pp. 150–159.

[59] T. Huang, "Surveillance video: The biggest big data," *Computing Now*, vol. 7, no. 2, pp. 82–91, 2014.

[60] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "MQTT-S: A publish/subscribe protocol for wireless sensor networks," in *Proceedings of the International Conference*

on Communication Systems Software and Middleware and Workshops. IEEE, 2008, pp. 791–798.

[61] R. Hyndman, "Package forecast," https://cran.r-project.org/web/packages/forecast/forecast.pdf.

[62] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41. ACM, 2007, pp. 59–72.

[63] S. Jain, V. Nguyen, M. Gruteser, and P. Bahl, "Panoptes: servicing multiple applications simultaneously using steerable cameras," in *Proceedings of the International Conference on Information Processing in Sensor Networks*, 2017, pp. 119–130.

[64] K. A. Joshi and D. G. Thakore, "A survey on moving object detection and tracking in video surveillance system," *International Journal of Soft Computing and Engineering*, vol. 2, no. 3, pp. 44–48, 2012.

[65] F. Keller, E. Muller, and K. Bohm, "Hics: high contrast subspaces for density-based outlier ranking," in *2012 IEEE 28th International Conference on Data Engineering.* IEEE, 2012, pp. 1037–1048.

[66] J. Kelley, C. Stewart, N. Morris, D. Tiwari, Y. He, and S. Elnikety, "Measuring and managing answer quality for online data-intensive services," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on.* IEEE, 2015, pp. 167–176.

[67] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *Proceedings of the International Conference on Mobile Computing, Applications, and Services.* Springer, 2010, pp. 59–79.

[68] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[69] H.-P. Kriegel, P. Kröger, E. Schubert, and A. Zimek, "Loop: local outlier probabilities," in *Proceedings of the 18th ACM conference on Information and knowledge management.* ACM, 2009, pp. 1649–1652.

[70] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the International Conference on Management of Data.* ACM, 2015, pp. 239–250.

[71] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[72] A. Kumbhar, F. Koohifar, I. Güvenç, and B. Mueller, "A survey on legacy and emerging technologies for public safety communications," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 1, pp. 97–124, 2017.

[73] N. Laptev, S. Amizadeh, and I. Flint, "Generic and scalable framework for automated time-series anomaly detection," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 2015, pp. 1939–1947.

[74] A. Lazarevic and V. Kumar, "Feature bagging for outlier detection," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining.* ACM, 2005, pp. 157–166.

[75] Q. Li, C. Rajagopalan, and G. D. Clifford, "A machine learning approach to multi-level ecg signal quality classification," *Computer Methods and Programs in Biomedicine*, vol. 117, no. 3, pp. 435–447, 2014.

[76] ——, "Ventricular fibrillation and tachycardia classification using a machine learning approach," *IEEE Transactions on Biomedical Engineering*, vol. 61, no. 6, pp. 1607–1613, 2014.

[77] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "Turkit: Human computation algorithms on mechanical turk," in *Proceedings of the Symposium on User Interface Software and Technology.* ACM, 2010, pp. 57–66.

[78] J. Lu, "Anomaly detection for airbnb's payment platform," http://nerds.airbnb.com/anomaly-detection/.

[79] H. Lütkepohl, *New introduction to multiple time series analysis*. Springer Science & Business Media, 2005.

[80] I. Mporas, V. Tsirka, E. I. Zacharaki, M. Koutroumanidis, M. Richardson, and V. Megalooikonomou, "Seizure detection using eeg and ecg signals for computer-based monitoring, analysis and management of epileptic patients," *Expert Systems with Applications*, vol. 42, no. 6, pp. 3227–3233, 2015.

[81] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the International Conference on Data Mining Workshops*. IEEE, 2010, pp. 170–177.

[82] A. Papageorgiou, E. Poormohammady, and B. Cheng, "Edge-computing-aware deployment of stream processing tasks based on topology-external information: Model, algorithms, and a storm-based prototype," in *Proceedings of the International Congress on Big Data*. IEEE, 2016, pp. 259–266.

[83] B. Pfaff and M. Stigler, "Package vars," https://cran.r-project.org/web/packages/vars/vars.pdf.

[84] D. Pokrajac, A. Lazarevic, and L. J. Latecki, "Incremental local outlier detection for data streams," in *Computational Intelligence and Data Mining, 2007. CIDM 2007. IEEE Symposium on*. IEEE, 2007, pp. 504–515.

[85] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "Timestream: Reliable stream computation in the cloud," in *Proceedings of the European Conference on Computer Systems*. ACM, 2013, pp. 1–14.

[86] T. D. Räty, "Survey on contemporary remote surveillance systems for public safety," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 5, pp. 493–515, 2010.

[87] Apache Spark, "Sparkr (r on spark)," http://spark.apache.org/docs/1.6.2/sparkr.html.

[88] L. Rettig, M. Khayati, P. Cudré-Mauroux, and M. Piórkowski, "Online anomaly detection over big data streams," in *Big Data (Big Data), 2015 IEEE International Conference on.* IEEE, 2015, pp. 1113–1122.

[89] B. Rosner, "Percentage points for a generalized esd many-outlier procedure," *Technometrics*, vol. 25, no. 2, pp. 165–172, 1983.

[90] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the International Conference on Management of Data.* ACM, 2015, pp. 1357–1369.

[91] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[92] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, "Incremental deployment and migration of geo-distributed situation awareness applications in the fog," in *Proceedings of the International Conference on Distributed and Event-based Systems.* ACM, 2016, pp. 258–269.

[93] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic scaling of data parallel operators in stream processing," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 1–12.

[94] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "Cosmos: computation offloading as a service for mobile devices," in *Proceedings of the International Symposium on Mobile Ad-hoc Networking and Computing.* ACM, 2014, pp. 287–296.

[95] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[96] W. Shi and S. Dustdar, "The promise of edge computing," *IEEE Computer Magazine*, vol. 29, no. 5, pp. 78–81, 2016.

[97] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on.* IEEE, 2010, pp. 1–10.

[98] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan, "Scalable crowd-sourcing of video from mobile devices," in *Proceedings of the International Conference on Mobile systems, Applications, and Services.* ACM, 2013, pp. 139–152.

[99] M. Solaimani, M. Iftekhar, L. Khan, and B. Thuraisingham, "Statistical technique for online anomaly detection using spark over heterogeneous data from multi-source vmware performance data," in *Big Data (Big Data), 2014 IEEE International Conference on.* IEEE, 2014, pp. 1086–1094.

[100] W. R. Stevens, "Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," 1997.

[101] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *Proceedings of the 33rd international conference on Very large data bases.* VLDB Endowment, 2007, pp. 159–170.

[102] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the International Conference on Management of Data.* ACM, 2014, pp. 147–156.

[103] O. Vallis, J. Hochenbaum, and A. Kejariwal, "A novel technique for long-term anomaly detection in the cloud," in *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, 2014.

[104] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing.* ACM, 2013, p. 5.

[105] J. Wang, B. Amos, A. Das, P. Pillai, N. Sadeh, and M. Satyanarayanan, "A scalable and privacy-aware iot service for live video analytics," in *Proceedings of the Multimedia Systems Conference.* ACM, 2017, pp. 38–49.

[106] D. Yang, G. Xue, X. Fang, and J. Tang, "Crowdsourcing to smartphones: Incentive mechanism design for mobile phone sensing," in *Proceedings of the Annual International Conference on Mobile Computing and Networking.* ACM, 2012, pp. 173–184.

[107] F. Yates, "Contingency tables involving small numbers and the $\chi 2$ test," *Supplement to the Journal of the Royal Statistical Society*, vol. 1, no. 2, pp. 217–235, 1934.

[108] B. Yogameena and K. S. Priya, "Synoptic video based human crowd behavior analysis for forensic video surveillance," in *Advances in Pattern Recognition (ICAPR), 2015 Eighth International Conference on.* IEEE, 2015, pp. 1–6.

[109] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing*, vol. 10, 2010, p. 10.

[110] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Symposium on Operating Systems Principles.* ACM, 2013, pp. 423–438.

[111] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2017, pp. 377–392.

[112] Q. Zhang, Y. Song, R. R. Routray, and W. Shi, "Adaptive block and batch sizing for batched stream processing system," in *Proceedings of the International Conference on Autonomic Computing.* IEEE, July 2016, pp. 35–44.

[113] Q. Zhang, X. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: big data sharing and processing in collaborative edge environment," in *Proceedings of the Workshop on Hot Topics in Web Systems and Technologies*, Oct 2016.

[114] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *Proceedings of the International Conference on Mobile Computing and Networking.* ACM, 2015, pp. 426–438.

[115] Y. Zhang and M. van der Schaar, "Reputation-based incentive protocols in crowd-sourcing applications," in *Proceedings of the International Conference on Computer Communications.* IEEE, 2012, pp. 2140–2148.

[116] A. Zimek, R. J. Campello, and J. Sander, "Ensembles for unsupervised outlier detection: challenges and research questions a position paper," *Acm Sigkdd Explorations Newsletter*, vol. 15, no. 1, pp. 11–22, 2014.

[117] A. F. Zuur, R. Fryer, I. Jolliffe, R. Dekker, and J. Beukema, "Estimating common trends in multivariate time series using dynamic factor analysis," *Environmetrics*, vol. 14, no. 7, pp. 665–685, 2003.

ABSTRACT

SYSTEM SUPPORT FOR STREAM PROCESSING IN COLLABORATIVE CLOUD-EDGE ENVIRONMENT

by

QUAN ZHANG

MAY 2018

Advisor: Dr. Weisong Shi

Major: Computer Science

Degree: Doctor of Philosophy

Stream processing is a critical technique to process huge amount of data in real-time manner. Cloud computing has been used for stream processing due to its unlimited computation resources. At the same time, we are entering the era of Internet of Everything (IoE). The emerging edge computing benefits low-latency applications by leveraging computation resources at the proximity of data sources. Billions of sensors and actuators are being deployed worldwide and huge amount of data generated by things are immersed in our daily life. It has become essential for organizations to be able to stream and analyze data, and provide low-latency analytics on streaming data. However, cloud computing is inefficient to process all data in a centralized environment in terms of the network bandwidth cost and response latency. Although edge computing offloads computation from the cloud to the edge of the Internet, there is not a data sharing and processing framework that efficiently utilizes computation resources in the cloud and the edge. Furthermore, the heterogeneity of edge devices brings more difficulty to the development of collaborative cloud-edge applications.

To explore and attack the challenges of stream processing system in collaborative cloud-edge environment, in this dissertation we design and develop a series of systems to support stream processing applications in hybrid cloud-edge analytics. Specifically, we develop an hierarchical and hybrid outlier detection model for multivariate time series streams that automatically selects the best model for different time series. We optimize one of the stream processing system (i.e., Spark Streaming) to reduce the end-to-end latency. To

facilitate the development of collaborative cloud-edge applications, we propose and implement a new computing framework, *Firework* that allows stakeholders to share and process data by leveraging both the cloud and the edge. A vision-based cloud-edge application is implemented to demonstrate the capabilities of *Firework*. By combining all these studies, we provide comprehensive system support for stream processing in collaborative cloud-edge environment.

AUTOBIOGRAPHICAL STATEMENT

Quan Zhang is a Ph.D. candidate in the Department of Computer Science at Wayne State University. He joined the Ph.D. program in Sep 2012. He received his Masters degree in Computer Science at Wayne State University in May 2016 and received his Bachelor degree in Computer Science at Tongji University in Aug 2011. His research interests include Cloud Computing, Edge Computing, Real-time Stream Processing, and Energy-efficient systems. He has published several papers in workshops, conferences and journals, such as SEC, ICAC, IEEE BigData, SUSCOM, and TPDS. He has also served as a peer reviewer for journals and conferences.