THESIS

PREDICTION BASED SCALING IN A DISTRIBUTED STREAM PROCESSING CLUSTER

Submitted by

Kartik Khurana

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2020

Master's Committee:

Advisor: Sangmi Lee Pallickara

Shrideep Pallickara
Ellison Carter

Copyright by Kartik Khurana 2020

All Rights Reserved

ABSTRACT

PREDICTION BASED SCALING IN A DISTRIBUTED STREAM PROCESSING CLUSTER

Proliferation of IoT sensors and applications have enabled us to monitor and analyze scientific and social phenomena with continuously arriving voluminous data. To provide real-time processing capabilities over streaming data, distributed stream processing engines (DSPEs) such as Apache STORM and Apache FLINK have been widely deployed. These frameworks support computations over large-scale, high frequency streaming data. However, current on-demand auto-scaling features in these systems may result in an inefficient resource utilization which is closely related to cost effectiveness in popular cloud-based computing environments. We propose ARSTREAM, an auto-scaling computing environment that manages fluctuating throughputs for data from sensor networks, while ensuring efficient resource utilization. We have built an Artificial Neural Network model for predicting data processing queues and this model captures non-linear relationships between data arrival rates, resource utilization, and the size of data processing queue. If a bottleneck is predicted, ARSTREAM scales-out the current cluster automatically for current jobs without halting them at the user level. In addition, ARSTREAM incorporates threshold-based re-balancing to minimize data loss during extreme peak traffic that could not be predicted by our model. Our empirical benchmarks show that ARSTREAM forecasts data processing queue sizes with RMSE of 0.0429 when tested on real-time data.

ACKNOWLEDGEMENTS

Any achievement without effort is dull, and this thesis is an achievement that is not dull for me made possible because of the support of multiple people in my life. I want to thank my advisor Dr. Sangmi Lee Pallickara, for her help and sincere guidance that has resulted in the successful completion of my thesis. I would also like to thank my thesis committee members, Dr. Shrideep Pallickara and Dr. Ellison Carter, for their constructive suggestions and remarks.

I want to thank my family for believing in me and for allowing me to go for higher studies when it wasn't easy for them to send someone very close, far away in another country. I would also like to thank my colleague, Mr. Aaron Pereira, for all the discussions, suggestions, and contributions to this work.

Finally, I would like to thank the whole CS department and my fellow graduate students with whom I discussed the topic and helped me in some way or the other.

DEDICATION

I would like to dedicate this thesis to my family and my friends.

TABLE OF CONTENTS

ABSTRACT . . . ii
ACKNOWLEDGEMENTS . . . iii
DEDICATION . . . iv
LIST OF FIGURES . . . vii

Chapter 1 Introduction . . . 1
1.1 Research Questions . . . 2
1.2 Overview of Approach . . . 3
1.3 Research Contributions . . . 4
1.4 Organizations . . . 4

Chapter 2 Related Work . . . 5
2.1 Stream Computing Frameworks . . . 5
2.2 Auto-Scaling in Related Systems . . . 11
2.3 Machine Learning on Stream Processing Engines . . . 17

Chapter 3 Methodology . . . 18
3.1 Auto Scaling . . . 18
3.1.1 What is Auto-Scaling? . . . 18
3.1.2 Approach used to provide Auto-Scalability . . . 21
3.2 Task-Level Parallelism . . . 28
3.2.1 What is Task-Level Parallelism? . . . 28
3.2.2 Approach used to provide Task-Level Parallelism Updates . . . 29

Chapter 4 System Architecture . . . 31
4.1 System Components . . . 32
4.1.1 Metrics Fetch Module (MFM) . . . 32
4.1.2 Reactive Auto-Scaling Module (RASM) . . . 32
4.1.3 Proactive Auto-Scaling Module (PASM) . . . 32
4.1.4 Prediction Module (PM) . . . 33
4.1.5 Re-Scaling Module (RM) . . . 33
4.1.6 Machine Learning Module (MLM) . . . 34
4.2 Internal working Of ARSTREAM and System Design Constraints . . . 34

Chapter 5 Experiments And Empirical Evaluation . . . 36
5.1 Experimenting with Reactive Scaling Mode . . . 38
5.2 Experimenting with Proactive Scaling Mode . . . 46
5.3 Proactive Model Accuracy . . . 50

Chapter 6 Conclusions and Future Work . . . 57
6.1 Future Works . . . 57

Bibliography 60

LIST OF FIGURES

2.1 Architecture of STORM Engine 6

3.1 Rebalance command executed from CLI in STORM 22
3.2 Configurable parameter to specify the type of strategy to use 23
3.3 Scale-in and Scale-out thresholds for Reactive Scaling Mode 23
3.4 Reactive Scaling Methodology 24
3.5 An example of rotating window 25
3.6 Scale-in and Scale-out thresholds for Proactive Scaling Mode 25
3.7 Proactive Scaling Methodology 26
3.8 Artificial Neural Network model in Prediction Module 27
3.9 Task Level Parallelism Methodology 29

4.1 Architecture of ARSTREAM Engine 31

5.1 ARSTREAM's common system configuration for scaling 37
5.2 Reactive ARSTREAM versus STORM compared using arrival rate, emit rate and executor count for constant arrival rate 39
5.2 Reactive ARSTREAM versus STORM compared using throughput and executor count for constant arrival rate 40
5.3 Reactive ARSTREAM versus STORM compared using arrival rate, emit rate and executor count for varying arrival rate and 5 minutes sleep persist duration 41
5.3 Reactive ARSTREAM versus STORM compared using throughput and executor count for varying arrival rate and 5 minutes sleep persist duration 43
5.4 Reactive ARSTREAM versus STORM compared using arrival rate, emit rate and executor count for varying arrival rate and 1-minute sleep persist duration 44
5.4 Reactive ARSTREAM versus STORM compared using throughput and executor count for varying arrival rate and 1-minute sleep persist duration 45
5.5 Proactive ARSTREAM versus STORM compared using arrival rate, emit rate and executor count for constant arrival rate 47
5.5 Proactive ARSTREAM versus STORM compared using throughput and executor count for constant arrival rate 48
5.6 Proactive ARSTREAM versus STORM compared using arrival rate, emit rate and executor count for varying arrival rate and 5 minutes sleep persist duration 50
5.6 Proactive ARSTREAM versus STORM compared using throughput and executor count for varying arrival rate and 5 minutes sleep persist duration 51
5.7 Proactive ARSTREAM versus STORM compared using arrival rate, emit rate and executor count for varying arrival rate and 1-minute sleep persist duration 52
5.7 Proactive ARSTREAM versus STORM compared using throughput and executor count for varying arrival rate and 1-minute sleep persist duration 53
5.8 Throughput comparison for original STORM, Reactive ARSTREAM and Proactive ARSTREAM 54
5.9 R2 and MSE graph for the proactive model 54

5.10 Comparison of the accuracy measures between 5 minutes sleep persisted run, and 1-minute sleep persisted run 56

Chapter 1

Introduction

Given the amount of data that is being generated every day and the need to accumulate, operate, and interpret the data to provide useful insights and predictions resulted in the development of distributed batch processing frameworks. But the data generated at such a large scale usually take hours and days to interpret even with the popular batch processing frameworks like Apache SPARK [1–3] and most of the times, what we need is only the real-time insights from the data. Therefore, the need is of the system that can ingest data stream and produces useful insights instantaneously. Many stream processing engines were developed, which helped to parallelize the computations giving us the swifter results by providing more resources to the application and running the operators in the parallel. But still, the streaming platforms that are available in the industry lack the potential to perform machine learning on the streaming data in a distributed way. However, distributed machine learning is available in batch processing engines like SPARK's MLlib [1–3], but is limited in terms of algorithms it offers. Therefore, researchers developed their distributed versions of the machine learning algorithms, and they created a library to be used as a pluggable adapter on the available streaming engines to make available the online machine learning on DSPEs, the project called Apache SAMOA [4, 5].

With the advancement in artificial intelligence and machine learning, people are now interested in *deep learning* on the streaming data. As we know, deep learning is a resource-intensive application, and for the same reason, GPU comes handy. In our work, we show that current stream processing engines can also be used for deep learning as effectively as it is done on GPUs by parallelizing the tasks to run on the multiple workers and using the streaming data. The distributed stream processing engine we use is Apache STORM [6], which is a widely deployed stream computing engine.

When we run any resource-intensive application on the distributed platform, whether streaming or batch, one thing about which we can never be accurately sure is the resource requirements of

that application throughout its execution due to continuously changing workloads. Therefore, we thought of building ARSTREAM, which enables *auto-scaling* in STORM and thus helps an application by automatically providing resources during the times of need maintaining efficient resource utilization. Auto-Scaling is a feature that has been missing from most of the engine's official releases, whether batch or streaming, but most of them have provided a module to re-scale an application manually, which can be used only after a user sees an overloaded operator. This way of scaling has many downsides- first, it is not auto-scalable and the user will have to surveil the operator performance manually requiring a lot of human efforts and second, it is on-demand which will take action only after the bottleneck is seen by the user. People have over time researched best auto-scaling techniques and categorized them into *reactive* and *proactive* techniques. We will explore the use of both the techniques and implement them in ARSTREAM for our applications, which need auto-scaling.

When we configure an application in distributed stream processing engines, we configure the tasks which will eventually run in parallel in the system, and these tasks are fixed during the entire execution of the topology. In this work, we modify the approach used in Apache STORM, which limits the number of tasks to change at run-time and show that whenever we re-scale, the number of tasks also changes without any side effects.

1.1 Research Questions

Research questions that we explore in this study include the following:

- **[RQ1]** How can we detect excessive traffic in advance for proactive up-scaling within a stream computing cluster?

  To provide auto-scaling, critical system capabilities that must be supported include monitoring network traffic and detecting early signs of potential overload. (Chapter 3.1).

- **[RQ2]** How can we provide auto-scaling while minimizing data loss?

Real-world applications often encounter frequent network traffic fluctuations. The system should be able to scale even with abrupt peaks that might occur within the monitoring window. (Chapter 3.1).

- **[RQ3]** How does the system account for memory-intensive computing requirements at the streaming computing node?

  Given the computational complexity and memory intensity of streaming computations (e.g., machine learning models), computational workloads at each node must be factored into the auto-scaling process. (Chapter 3.2).

1.2 Overview of Approach

ARSTREAM is a distributed stream processing framework, just like other stream processing engines available. It provides an auto-scalable mechanism for applications and the option to select from different types of scaling techniques implemented. ARSTREAM is built on top of the latest version of STORM enabling users to write streaming applications and run on the top of it, effectively making use of cluster resources. STORM starts the user topology with the user-defined resources and, in no way, can react to the changing resource needs of an application over time owing to the different rates of data arrival. ARSTREAM fixes this problem by providing an auto-scaling mechanism that monitors the metrics sent to nimbus by various workers. There can be many workers in the topology where each worker hosts only a single executor. Since a task instance runs on an individual thread for maximum parallelism, each executor generates its own set of metrics to be sent to nimbus. Nimbus aggregates all the metrics sent to it by different workers and then decides to either *scale-in* or *scale-out*. There are two techniques by which we can scale an application in ARSTREAM - reactively and proactively, and the users can specify which technique they want to use in the configuration file. Whenever a scale-out decision is made, a new task is added in parallel, and therefore, a new executor is provided for each newly added task to run in parallel.

1.3 Research Contributions

This research presents our approach for providing an auto-scaling mechanism that is tightly integrated with the original STORM engine in the form of ARSTREAM to perform real-time analytics on the data coming at fast and varying speeds.

1. We have designed a module that gathers the metrics from the application and the system and then sends these metrics to nimbus for making scaling decisions.

2. There is another module in nimbus which runs every "x" seconds to analyze these metrics. Depending on the scaling strategy chosen by the user, it calls that scaling module and then issues the STORM's *re-balance* command if the scaling criteria are met.

3. We have also designed a Jupyter Notebook for training the model for proactive scaling mode. This model is application-specific and therefore requires new training for new applications. It also has the code to run tests on real-time data and give us the loss measures.

4. We used a deep learning application to test and evaluate ARSTREAM and also to compare it with STORM to show the limitations of the latter system for such applications. With our experiments, we will prove that our auto-scaling in ARSTREAM works perfectly with resource-intensive applications like deep learning.

1.4 Organizations

The rest of the document is organized as follows. Chapter 2 describes related work in the context. Chapter 3 describes our methodology i.e., what techniques we used to achieve auto-scalability and to provide *task-level parallelism* updates. Chapter 4 goes over the ARSTREAM's system architecture, which gives us a high-level view of the system and the internal workings. In Chapter 5, we go over experiments and evaluation of ARSTREAM for both reactive and proactive mode, and we also compare ARSTREAM with STORM. Finally, we talk about future scope and the conclusion in Chapter 6.

Chapter 2

Related Work

The world we live in now is maturing in every aspect, with software turning much more efficient and hardware performing a lot better. The proliferation of IoT devices at households, streets, and on the machines is continuously generating voluminous data at unprecedented rates. With the advent of IoT and the emerging need to gain useful insights from the data, people developed engines to run analytics on the data stream. It is also possible to store voluminous data over a period of time and then use it later to gain insights, but the downside with that is the missed real-time benefits. The need is of the systems which can provide immediate benefits without waiting and as it is being generated. There are numerous distributed processing engines available, which are categorized into batch processing and stream processing, and it depends on the data characteristics which system will fit for the use-case. Our research is based on the streaming data, and the system we propose is auto-scalable, having the ability to run machine learning algorithms efficiently. The systems which are popular and are currently in use have the potential to run ML algorithms, but they are inefficient due to the lack of auto-scaling feature in their core package. The genuine need is of the systems which are auto-scalable and do not require re-deploying of the application and any manual intervention to scale-in or scale-out for achieving efficient resource utilization.

There are many different big data stream processing frameworks available, and in the next section, we will talk about them and about how they are different from the system we propose.

2.1 Stream Computing Frameworks

There are many different data streaming engines available in the industry and to name some – Apache STORM, SAMZA, FLINK, and SPARK Streaming. They have differences in how they do state management, fault tolerance, and scalability.

- **Apache STORM**

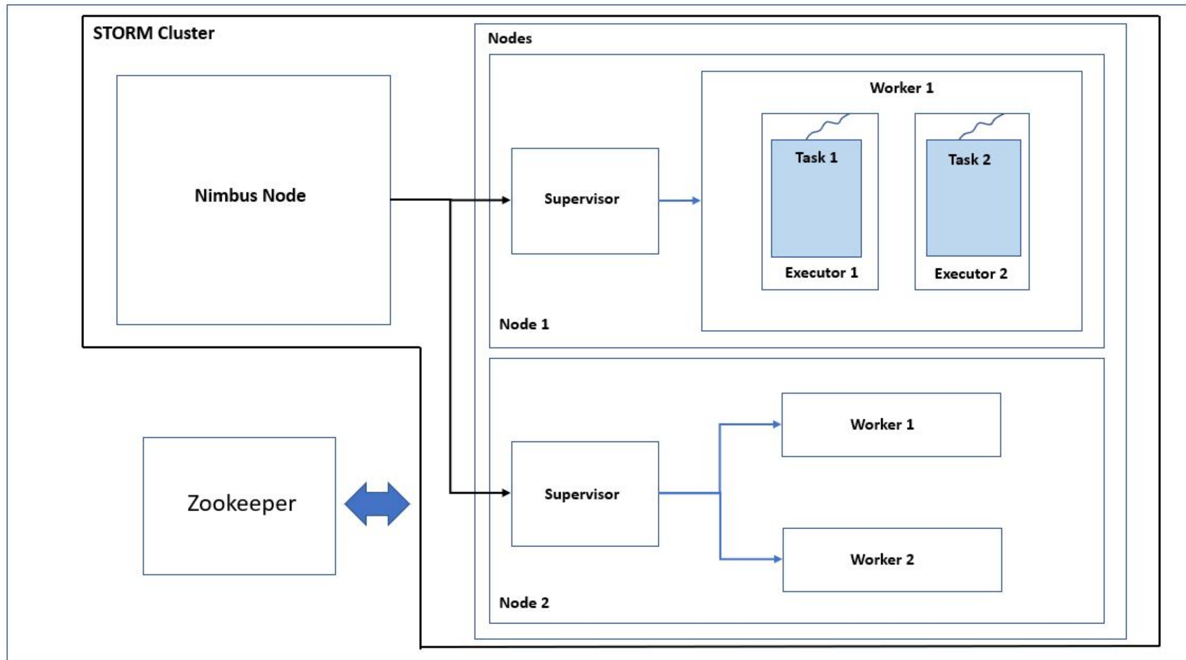


Figure 2.1: A High level view of the architecture of STORM engine

Apache STORM is one of the first distributed stream processing engines and is also an open-source framework for processing large structured, and unstructured data [7]. It can handle an unbounded stream of data with performing instantaneous processing on it. The APIs are simple to use, making it easier to write applications in STORM along with a relatively simple design that does not hide the complexity of the underlying system, unlike other stream processing engines like FLINK [8].

While writing applications, which essentially is a directed acyclic graph consisting of *spouts* and *bolts*, a user can configure as many executors, workers, and tasks as they want, but once the topology is started, it cannot be altered. By altering the topology, what we mean is that we cannot change the number of tasks defined initially; however, we can change the count of executors and workers by calling the re-balance command in STORM from CLI or by using WebUI [9]. We can set the number of tasks to a maximum at the start, but still, we will never be sure of how many tasks we need since the speed of the stream may change.

Data enters the topology using the spout, which can read data through any of external systems like Kafka etc. and the user's processing logic goes inside the bolt performing operations on the data. The spouts and the bolts are connected, which enables data movement between them. STORM did not support any state management by default in the previous versions; however, STORM offered the feature of Trident topologies, which is a high-level abstraction layer to accomplish state persistence. In the recent releases, they have included state management in STORM core. Since it is a true streaming engine, it can process a million messages per second per node in a fault-tolerant manner, which means that if a node or worker dies, STORM will automatically restart them. Another essential feature of STORM is its reliability – STORM guarantees at-least-once or exactly-once guarantee of message processing [10].

A developer who is writing a STORM application is aware of the fact that there is no perfect topology configuration (number of tasks, executors, and workers) that one can set while creating an application. The throughput/load is unpredictable, and hence, it is impossible to find the perfect configuration for the topology. If a user wants to change the topology configuration for increasing the parallelism, he can do so by calling the re-balance command, but if the number of executors is already set to a maximum value, the parallelism cannot be increased further. The only option available with the user is to restart the topology after increasing the number of tasks.

Some applications require machine learning for supervised or unsupervised learning, and it is easier when we have a batch data-set as compared to when data is available as streaming. This is because currently, there is relatively little support for machine learning on the streaming data. STORM does not support any machine learning algorithms in its use case. Therefore, if a user is interested in machine learning, it will require him to either write his version of a machine learning algorithm for streaming, or he would have to integrate STORM with any other library or project like Apache SAMOA [5]. As we all know that machine learning is a resource-intensive application, it will also have higher chances of cre-

ating a bottleneck in the cluster. STORM does not provide any mechanism by which it can manage and re-adjust the load automatically, and so, it will require human intervention. We can manually monitor the application load, monitor its resource needs, and then call the re-balance command, but it causes loss of data, and the level of parallelism that we can achieve by calling the re-balance command is limited by the number of tasks we configured at the beginning.

Hence, the lack of auto-scaling feature and the missing task-level parallelism updates makes STORM unsuited for many use cases.

- **Apache SAMZA** Apache SAMZA is also a distributed stream processing framework just like STORM and FLINK developed at LinkedIn, which allows you to process and analyze data in real-time [11]. It is highly scalable, fault-tolerant achieved using incremental checkpointing, and provides a unified API to write application irrespective of whether data is available as batch or streaming [12]. Although SAMZA is good at maintaining large states of information using RocksDb and Kafka logs, it lacks advanced streaming features like watermarks, sessions, and triggers. One of the most significant advantages of using SAMZA is that it manages the state correctly. It takes the snapshots of the stream processor states when required and restores the state from the snapshots taken when the processor is restarted [7, 13, 14].

  SAMZA is based on the concept of a publish/subscribe model where tasks listen to a data stream, processes it, and then outputs the result to another stream. To achieve scalability, a stream can be broken into multiple partitions, and a copy of the task will be spawned for each partition. In this way, multiple tasks can be executed in parallel to consume all the partitions in a stream simultaneously [14, 15]. The number of task instances is fixed determined by the number of input partitions at the time before starting the execution. During the execution, if a user decides to change the number of tasks to scale-in or scale-out, the only way possible is by stopping the job and restarting from scratch after making the desired change to the number of tasks. However, we can add or remove containers (1 container is equal to 1 CPU

core) while the job is running. As we scale-out or scale-in, the same state remains attached to each task instance. Tasks instances may be moved from one container to the other, and along with it, the state also moves; this allows the job to work correctly. We can only have as many containers as there are tasks; this is because we cannot keep a container without any process. Similarly, we need at least one container which will host the tasks, but to achieve maximum parallelism, we need a maximum of 1 task assigned to 1 container [16].

SAMZA is still young not only because it is one of the latest streaming engines but also because it lacks many important features and libraries – one of them is machine learning. Machine Learning is a must-have library, and because it is not inbuilt with SAMZA, an external library or framework is needed for machine learning capabilities. Users can also write their machine learning algorithms, but having these algorithms provided with the framework is always great.

Overall, *SAMZA is a framework that we can use for stream processing, but since it is relatively new, it lacks many essential features such as dynamic scalability, updating the number of partitions for scaling, and the missing Machine Learning library*.

- **Apache FLINK** Apache FLINK is another stream processing engine that is popular in the industry. It considers everything as a stream, including the batches, and is therefore considered as a true streaming platform for distributed stream processing. FLINK is better than many other stream processing engines as it provides exactly-once message processing guarantee, is fault-tolerant, offers a high throughput and provides high-level APIs (DataStream API and Dataset API) which makes it easy for the user to write applications [17]. It also provides a save-point mechanism that we can use to make changes to the running application. FLINK has simple data structures to define the processing pipelines, which hides most of the complexities of the underlying system and writing applications. Hiding these complexities under the API is useful for writing an application but also makes it difficult to use the system for research. Since it masks most of the complexities, it makes it difficult to understand the underlying scalability mechanism. Therefore, there are fewer papers and studies in which

Apache FLINK is used for the experiments. Even though users are provided with Dataset and DataStream APIs to write applications, these are the dataflow graphs that are executed by the FLINK's runtime engine, a layer that sits at the bottom and runs the graph. These dataflow graphs are made up of two components- operators and streams, where operators perform processing on the streams and generate new streams. To achieve parallelism, we can create multiple instances of the operator called subtasks, whereas streams can also be partitioned into one or more stream partitions. The number of parallel instances is what defines operator parallelism. A user is also required to set the max parallelism before an application is started, which cannot be changed in later stages. There is currently no way to change the max parallelism while an application is running, but we can use REST API and CLI to modify the parallelism of running application if it is within the limits of max parallelism and which is achieved by using save-point mechanism [18–21].

FLINK provides inbuilt support for machine learning through its machine learning library called FlinkML, which is a growing library of algorithms and whose main aim is to provide scalable machine learning algorithms along with making it easy to write an ML application in FLINK. However, there is no support for machine learning with streaming data, and if a user wants to create an ML application for streaming, he will have to use external frameworks like SAMOA for that purpose [22]. Though we can use REST API and CLI to change the parallelism of a running application in FLINK, the need to scale an application automatically is essential. If we are running a machine learning algorithm on the top of FLINK, it would have varying resource requirements during its execution, which means one will have to manually re-balance the configuration to provide it with enough resources. Secondly, if we want to change the max parallelism, we will have to restart the application from scratch with the brand-new state. Restarting an application from the previous save-point after changing the max parallelism is currently unavailable. Still, we can change the number of workers if we save point the current state and restart it after redistributing the states to the new con-

figuration. This max parallelism limits the parallelism we can achieve in the system for each operator.

Hence, the lack of auto-scaling feature and inability to change the max parallelism renders it unsuited for machine learning on FLINK, which works only for the batch-ed data-set. More importantly, the presence of high-level APIs available to users makes it difficult to understand the scalability mechanism by hiding the underlying complexities.

- **Apache SPARK** We may get Apache SPARK streaming in our minds too when talking about different streaming engines, but it is not a true streaming engine. Apache SPARK uses micro batching where it groups the incoming tuples into batches for processing at a specific time duration. Since the batches are processed at a particular time, it renders higher latency than true streaming engines, which are instance-based and handles each tuple as soon as they arrive in the system. The engines which are not truly streaming have a completely different mechanism of how they deal with processing, failures, and guarantees; therefore, they cannot be used for our research. Although Apache SPARK streaming comes inbuilt with the auto-scaling feature, it's high latency and low throughput as compared to other true streaming engines, and the on-demand scaling decisions rule it out from the game [2].

2.2 Auto-Scaling in Related Systems

There are numerous different stream processing engines available in the market today, and the choice of selecting one is always difficult. Researchers favor one, which is easier to tune and has minimum abstractions. The abstraction of the underlying system using high-level APIs makes it easier to use for the users but difficult for researchers. Many other factors drive the decision to choose a specific engine like performance measures (i.e., throughput and execute latency). We need a system that offers a high throughput and low latency, and STORM-2.0 is the first streaming engine to break the microsecond latency barrier as per the STORM's official page [23]. We decided to use Apache STORM for our research because of its modular design, which makes it easier to read and understand the code along with providing performance benefits and minimal abstraction,

making it easier to play with also. Many researchers have used STORM for their research and experiments for similar reasons. Ahmed et al. used Apache STORM for their experiment too [8].

STORM-2.0 is recently released but is still missing the crucial auto-scaling feature. There is only one DSPE available in the market today, which has an auto-scaling feature already implemented (i.e., Apache SPARK), but it has the latency in seconds because of its mini-batch processing. People have shown to use dynamic scaling in Apache SPARK streaming, but there is no official mention by the community members and on SPARK's official website [24]. The auto-scaling is present in the STORM's JIRA issue for ages, but no action has been taken on it to date. Veen et al. also studied and experimented with auto-scaling in Apache STORM, and their work also relates to our work, but the approaches are different [25]. Cloud infrastructure is growing at a rapid pace, and people have been attracted to its "pay as you go" model in which any organization can rent the resources provided by cloud providers on-demand, and they pay as they go. The primary benefit for organizations using the cloud infrastructure is auto-scaling operations. Lorido-Botran et al. talked about elasticity in the cloud environment as the Monitoring, Analysis, Planning, and Execution process [26]. After following this process, the auto-scaler decides to scale-in or scale-out the application for efficient resource utilization. The Cloud environment is convenient but does not suit the needs of organizations that have their own data centers. Therefore, we need auto-scaling features in DSPEs itself rather than leveraging it from cloud environments.

A lot of efforts have been made in this field to achieve auto-scaling for both streaming and non-stream processing engines, but no one has ever shown auto-scaling for applications that run deep learning algorithms. A great deal of research has also been done in the field of machine learning for streaming data as we know of Apache SAMOA, which is a pluggable library compatible with most of the available DSPEs and providing a variety of ML algorithms for streaming data [5]. Today also, it is a hot topic for many machine learning enthusiasts' who want to do machine learning on the streaming data.

Auto-scaling is not a new topic in the area of real-time stream processing; many researchers have selected this topic for their research because of the benefits it offers. It saves us a lot of com-

putational resources, maintains high throughput for the application, and saves us a lot of manual hours. Two auto-scaling techniques – reactive and proactive, both have their advantages and disadvantages and are suited for different applications differently. These two broad categories are also discussed by Nikravesh et al. [27].

People have always argued about which technique is better. We propose using hybrid scaling in ARSTREAM for applications that use the proactive technique because there is always a risk that proactive scaling may suffer from wrong decisions and, as a result, causing missed re-scaling operation and the loss of data. But all these problems are application-specific and will depend on the application use case which technique will suit them best. There are various reactive and proactive auto-scaling techniques proposed in several kinds of research; one of them is using the Hidden Markov Model (HMM) discussed in a study by Nikravesh et al. [27].

Cooper et al. used time series analysis for workload modeling to predict SLA breaches for streaming data in cloud environments [28]. Ahmed et al. proposed a proactive framework to predict the future workloads of an application using the data regression algorithm provided in SAMOA and the adaptive model rules regressor algorithm using Apache STORM as the underlying system [8].

Amazon introduced predictive scaling to EC2 instances using recurrent neural network (RNN) that can forecast CPU utilization of the EC2 machines and expected traffic. They use tons of data from thousands of EC2 machines and at least one day of historical data based on customer's usage. One drawback of using RNN we believe is that it fails if the traffic becomes unpredictable, showing no signs of pattern in it. It may only be successful for the applications where there are patterns and hence, can forecast for the next 48 hours. The application we use for ARSTREAM assumes that there is no pattern in the data, and the arrival rate varies unpredictably [29]. Roy et al. used the ARMA method for workload modeling, but again this applies only to the applications where there is a strong correlation with the historical data [30].

Many things need to be taken care of while making a scaling decision such that throughput is always maintained. The decisions can be made on various metrics like – data processing queue load, latency, memory utilization, CPU utilization, pending tasks, etc. Researchers have used many

of these different types of metrics to make scaling decisions. There are many ways to gather metrics from the system, but we decided to write our code to fetch the system metrics in ARSTREAM. JCatascopia is a monitoring system that collects several kinds of parameters from system and application [31]. Our primary concern and the need was to maximize the system throughput, which is primarily affected by the current queue load making the queue population a critical metric to make the scaling decision.

Now, we will discuss some other systems which implemented auto-scaling and will see how those are different from what we are doing. Xu et al. presented a stream processing system called STELA, which supports scale-out and scale-in operations in an on-demand manner. They aim to optimize the post scaling throughput and minimize the interruption to the running applications while re-balancing is being carried out. STELA uses a metric called effective throughput percentage, which it uses to carefully select the nodes to prune for scale-in and operators to provide with more resources for scale-out with minimum interruption. STELA also doesn't change the topology at run-time, which means we cannot add more tasks to running topology resulting in limited scaling, and this technique is invoked on-demand, an external event triggers the scaling operation [32].

Dhalion was developed on the top of Twitter HERON [33] to overcome the limitation of HERON of not being able to self-regulate. It uses a rule-based policy for making scaling decisions and uses metrics like queue sizes, backpressure of the operators to learn about the system's health. It has symptom detectors, diagnosers, and resolvers, which does all the work to make the system auto-scalable. Symptom detectors collect the metrics from the engine and look for symptoms that may be proof of an issue. Detected symptoms are then passed to diagnosers, which try to find explanations for the symptoms using the hand-coded rules. Finally, the resolver reads the diagnoser report and explores the possible measures to rectify the situation and takes that action. It falls under the category of reactive scaling where the action is taken after the problem has occurred and is different from our work, where we also explore the predictive scaling technique. One thing they follow like us in their technique is the use of the cool-down period after every action

is taken; the system waits for a reasonable period before considering any further action. It does auto-scaling through a stepwise iteration, incrementally reacting to bottlenecks by adjusting the parallelism until the topology is stable [34].

Like Dhalion, IBM Streams also uses backpressure and the congestion in the operators to find bottlenecks but uses threshold-based policies to make the scaling decisions [35].

DS2 is another framework, low latency, and a robust controller for dynamic scaling of streaming applications, which can vary resources available to an application to handle varying workloads. This is an advancement over Dhalion for HERON [33], and they have shown improvements in auto-scaling when applied to HERON. Their focus is on accurately determining the parallelism for all operators within a single scaling decision, unlike systems that look at only one operator in one scaling action, just like ARSTREAM does. It works in the online and reactive setting, and based on real-time performance it identifies the optimal level of parallelism required for each operator. It considers each operator's true processing capabilities and studies the effect of scaling each operator on downstream operators [36].

Apache SPARK has its internal auto-scaling mechanism, which operates only with a fixed resource pool. After an application is started, it cannot allocate new resources to the cluster. It requires setting up a shuffle service on each worker node and independent of executors allowing executors to be removed without deleting the shuffle files that they write. So, the worker is kept alive while the executors are scaled-in or scaled-out, making it impossible to re-scale the cluster to take advantage of the elasticity [37]. For adding or removing the executors, it uses the number of pending tasks as the metric. It implies that the existing set of executors is insufficient to saturate simultaneously all the tasks that have been submitted but not yet finished. If that is true for a duration greater than the threshold, it adds more executors [38].

There are other private projects aimed towards finding a better auto-scaling mechanism for SPARK. One such example is Elastic spark streaming, which uses processing percentage (processing time and micro-batch interval ratio) as the scaling logic. People have argued that it is available only at the executors' level and that there is no information available for node or worker

scaling [39]. But in conclusion, it still uses a reactive technique that is threshold-based; it may be useful in the SPARK environment but also may not be the best.

These were significant researches in the field of streaming auto-scaling systems and are also related to what we are trying to achieve. With the flexibility we have, we can keep it easy with some basic metrics to make the scaling decisions. However, none of them worked on task-level parallelism updates in STORM, which makes it scalable to whatever degree we want, and also for most of them, the scaling criteria were reactive.

There were many other kinds of research in the field of streaming systems that tried to solve problems, overcome the limitations, and added useful features to the systems. The Granules DSPE [40, 41] incorporates support for stateful streaming computations with support for MapReduce. The Neptune suite of extensions to the Granules runtime incorporates support for high-velocity data streams generated in IoT and sensing environments [42]. This includes support for flow control, frugal creation of objects, and minimizing context switches during processing. Optimal stream scheduling is NP-Hard, the prediction rings algorithm leverages time-series analysis and accounts for interference between stream processing computations to alleviate throughput and latency constraints [43]. More importantly, the algorithm supports dynamic scaling maneuvers while accounting for resource utilization imbalances. DSPEs typically are backed by stream dissemination systems that may be deployed in contexts [44,45] other than stream processing. Stream processing in the context of resource-constrained settings such as edge devices attempts to minimize the amount of data in transit [46, 47].

Streams occur naturally in settings involving observation devices. Data management challenges in these settings have addresses issues such storage [48, 49], subsetting [50, 51], sketching to reduce data volumes [52, 53], alleviating I/O costs [54], anomaly detection [55]. A key objective in these systems is support streams of queries that must be evaluated at low latencies. Such streaming queries may be ad hoc [56], analytic [57], geometry constrained [58], or relate to visualization [59, 60].

2.3 Machine Learning on Stream Processing Engines

Since streaming in its essence means continuous flow of data, traditional machine learning algorithms that runs on batch data-sets cannot be applied. Researchers have been working for long to create machine learning algorithms that can run on the streaming data given the fact of how important it is now to study the data characteristics in real-time and make predictions. Apache STORM did not have any machine learning library at all when it was developed, Apache SPARK did have a library of machine learning classifiers, but it was meant only for batch data-sets [61]. Apache FLINK came late in the picture, and it also did not have any machine learning capabilities for streaming data [22].

Apache SAMOA, which is a distributed streaming machine learning framework, was developed, enabling the development of new ML algorithms without directly dealing with the complexities of the underlying streaming engines. It can be used as a pluggable library with most of the available DSPEs. Some of the algorithms available for use are Vertical Hoeffding Tree Classifier, Bagging and Boosting, and Distributed Stream Clustering [4]. Because of its ability to be used as an adapter with most of the available DSPEs, it is popular in the field.

Few of the other projects like Jubatus and StormMOA also belong to the same category of distributed streaming machine learning [62]. Jubatus is a framework that runs multiple models at the same time processing different sets of data, and in this way, it achieves horizontal parallelism. There is a tight coupling between the underlying SPE and ML library, which is the reason it cannot be used with other DSPEs [63]. Similarly, StormMOA also works specifically with STORM and cannot be used with other DSPEs. It also lacks some widely used algorithms making it the least preferable choice among the options we have [64]. Though SAMOA provides us with useful machine learning algorithms, what is missing from it is the deep learning capability in a distributed manner. In ARSTREAM, we chose to experiment using a federated learning approach to machine learning, which uses parameter server to prove that we can efficiently run deep learning models by providing enough resources it needs when the need arises and on the fly.

Chapter 3

Methodology

ARSTREAM is built on the top of Apache STORM. It uses STORM's re-balance API to scale an application, but the way re-balance is called by ARSTREAM is internal to nimbus and considers many feedbacks from supervisors. There is a master node called nimbus and the slave workers that reside on supervisor daemons. The coordination between master and supervisors is achieved by using Zookeeper, a cluster management framework. Supervisors that host one or more workers sends the queue loads and the predictions to nimbus for making the scaling decisions using a thrift interface for communication. Nimbus then decides whether to scale-in or scale-out. A more detailed discussion about ARSTREAM 's system architecture is available in Chapter 4.

3.1 Auto Scaling

3.1.1 What is Auto-Scaling?

Auto-Scaling is a process of automatically adding or removing the number of resources being used by an application based on its need at any point in time [65]. Given the fact that it is challenging to accurately predict the future load of an application even before we begin its execution, it makes it challenging to allocate the correct number of resources to an application from the cluster when it is started. The load on the application is reliant on many factors like – data ingestion rate, the complexity of the code, CPU utilization, etc. and therefore, we can only speculate the resource requirements before the start.

In general, it is difficult to scale an application on the fly, and not many systems provide this feature. There are situations where the user of an application would not want to stop the application to increase the resources, and the only way possible at that point to increase the resources is by stopping the application and restarting it after adding the additional resources in the configuration. It would result in loss of performance and the user data, both during the time application was

congested and when the application was halted for re-scaling, which the user would not want. When an application requires more resources, the system must be ready to provide additional resources without much delay, as one would encounter if monitoring manually to maintain the throughput of the application. STORM provides us the re-balancing API to increase/decrease the number of workers and executors on the fly, and other systems should also offer the same [9]. Although streaming engines provides flexibility to re-adjust the load of an application using REST APIs or CLI commands, these require human intervention wherein user of the application must monitor and raise the request for re-balance to the streaming engine. But in conditions where an application is running continuously for days, weeks, or months, it is difficult, and therefore, having an auto-scaling feature is a must, especially if an application runs continuously and with varying arrival rates. If there is any bottleneck or any idle task in the system, then auto-scaling does all provisioning and de-provisioning of resources for the computing job.

There had been several types of research in the field of auto-scaling of resources [8, 28, 30, 32, 34, 39], but still many popular and original SPEs lacks the ability of automatically adding resources when in need to process the streaming data. However, there had been many advancements in batch processing engines, and now they support auto-scaling (e.g., Apache SPARK). There are two different types of auto-scaling techniques – Reactive and Proactive [66] in the field of streaming data.

In the reactive technique, the system collects the metrics of resource usages by the application as well as the run-time stream properties. It then uses these metrics to make scaling decisions, whereas, in the proactive technique, the system is dynamic and predicts future bottlenecks. When it detects any future bottleneck, it makes scaling decisions before the bottleneck occurs, making room for more data to arrive and process in the system. Our efforts focus on both the reactive and the proactive techniques of auto-scaling and their feasibility of application in Apache STORM. In the reactive mode, the system reacts to an event; the event here is the chosen metric going over the set threshold, and the reaction is scaling decision. We collect the system metrics indicating the resource usages like – CPU utilization, memory utilization, network latency, data processing

queue loads, arrival rate, etc. If any of these, whichever chosen by the user exceeds a defined threshold, the system takes action and provides more resources to the running topology. People have the choice of using any system metric they want, or they can even make their metric just like STELA created, called ETP (Effective throughput percentage) [32].

Since we are building a system that is scalable and tries to maximize throughput during the execution, the queue load plays an important role, and that is what we chose as the metric for re-scaling. In the stream processing engines, every executor maintains the sender and the receive queues where they store the outgoing and incoming messages, respectively, and they have their ways to identify the bottlenecks in the downstream operators. In STORM, if spout sees any downstream bolt as being overloaded with data, it will stop sending the messages to the bolt by activating its throttling mechanism [67]. It periodically checks for the operator load and propagates only limited data to downstream bolts if the bottleneck is found. This is a situation where an operator is performing as fast as it can to process the messages in the queue and decrease its size, but the speed at which data arrives is even more than the processing speed of the bolt, and hence, queue size never decreases. Since messages stay in the queue while the operator is performing its operation, it results in throughput loss as throughput would have increased if there was an extra task working in parallel to ingest the pending message from the queue. We aim to create a system that tries to maximize the throughput and which keeps up with the arrival rate of the stream.

The proactive technique is an advanced method that makes use of machine learning to make the scaling decision. An artificial neural network model is created as the first step, which learns about different system performance characteristics and stream properties to detect any early signs of the bottleneck [RQ1]. We do not want the bottleneck to occur as it would result in the loss of data, so we try to predict future bottlenecks and then provide extra resources to the application before it happens, resulting in increased throughput. There are many ways to make predictions like using the historical data in machine learning, but not all of them are applicable in our case. Some of the conventional machine learning techniques are Linear Regression, Random Forests, and using Artificial Neural Networks for identifying the linear/ non-linear patterns in the data. Researchers

have used many of these techniques to predict future workloads and bottlenecks in popular SPEs, and we will do the same. Cooper et al. applied time series analysis to forecast SLA breaches for streaming data in cloud environments [28]. In contrast, Ahmed et al. proposed a proactive framework to predict future workloads using the streaming data regression algorithm provided in Apache SAMOA [8].

We use a deep learning application to execute on our system, which is computing-intensive, unlike the applications which were used in the previous decade and characterized relatively low complexity and whose data stream rate had a direct correlation with the bottleneck. For such resource-intensive applications, simple workload modeling to predict future bottlenecks cannot be used. Instead, we need to incorporate computational workload and the network traffic by means of predicting queue sizes for each running task. Queue size prediction will accurately take into account the computational complexity and the workload characteristics to predict the bottleneck and protecting us from its consequences [RQ3].

Since the stream of data is boundless, dynamic, and unforeseeable, traditional auto-scaling techniques are preferred, but they result in loss of data. The data loss here occurs when tuples residing in data processing queues are dropped during auto-scaling operation. We can use proactive mode and reduce data loss, but that comes with its drawbacks too, like wrong predictions (i.e., high error rate). In this work, we aim to generate a predictive model that will provide reasonable accuracy to be trustworthy while minimizing the data loss and including a call to an on-demand reactive mode for dealing with missed re-scaling operations (i.e., false-negatives).

3.1.2 Approach used to provide Auto-Scalability

STORM does not provide any auto-scaling feature in its core package where the user can automatically add nodes or more threads while the topology is running. It requires manually monitoring the resource usages in the system and administer the "re-balance topology" command with the revised configuration of resources. When the re-balance command is executed, STORM inter-

nally kills the running topology, modifies the resource configuration, finds the best assignments of components, and starts it again.

As discussed previously, the process of manually monitoring the system and issuing the re-balance command takes a lot of effort. We need a monitoring capability or an administration that monitors the system while an application is running and issues the command for us. The re-balance command STORM provides uses two different flags and also to note, there is no flag to change the number of tasks–

1) -n for the number of revised workers

2) -e for number of revised executors for a specific component

```
$ storm rebalance mytopology -n 5 -e blue-spout=3 -e yellow-bolt=10
```

Figure 3.1: Rebalance command executed from CLI in STORM

This command is executed from CLI and modifies the topology resource configuration for us [9]. We can also use REST API to instruct STORM to re-balance the current topology. It, either way, invokes the same operation and re-balances the topology for us.

ARSTREAM has an automatically re-scaling module, which is called by both of the scaling techniques when the bottleneck is detected. Just like STORM which gives users the option to select the number of workers, tasks, and executors to be any number in the topology configuration, ARSTREAM also allows that but for our experiments, we start with one worker, executor and task to show the scalability in best possible way. When ARSTREAM calls the re-scaling module, it adds a new bolt task to the topology and deploys it on a newly spawned executor who is deployed on the freshly spawned worker. In this way, we make sure that we achieve true parallelism and provide extra room for increasing the throughput using a new worker. Currently, we have restricted the scaling to just one operator, which the user specifies, and all other components are skipped for monitoring.

The two types of auto-scaling techniques are available to the user using a configurable parameter. Based on the needs of the application and how important the task is, a user can use any of the

techniques. We use both of these techniques in our experiments to show the results. The configuration file contains a field for the type of technique and the field for the thresholds for each of the techniques, which is set by the user to specify what scaling technique they want to use and on what queue load threshold the system should re-scale. One important thing to note is that ARSTREAM performs both scale-in and scale-out operation depending on the queue utilization of each bolt executor; therefore, we need to keep a cool-down period before taking in a new re-scaling request to prevent aggressive re-scaling. It is necessary because it takes time for the system to get stable when the re-scaling has just occurred.

```
topology.rebalance.strategy: "Reactive"  ##Proactive or Reactive
```

Figure 3.2: Configurable parameter to specify the type of strategy to use

Reactive Scaling Mode

Reactive scaling is a scaling technique in which master nimbus takes action only after seeing the bottleneck in the system. The bottleneck here is decided by looking at the current queue utilization of the bolt's tasks, which is calculated as the count of messages in queue divided by the maximum queue capacity. Similar to the "re-balance strategy" configuration parameter, there is a parameter for specifying the thresholds-

```
rebalance.reactive.min.threshold: 0.05
rebalance.reactive.max.threshold: 0.20
```

Figure 3.3: Scale-in and Scale-out thresholds for Reactive Scaling Mode

When the nimbus master gets queue loads from the workers, it checks all the loads, and if any of these loads is greater than the set threshold, it initiates the scale-out operation by calling the re-scaling module. This process of checking the loads is repeated after every configurable number

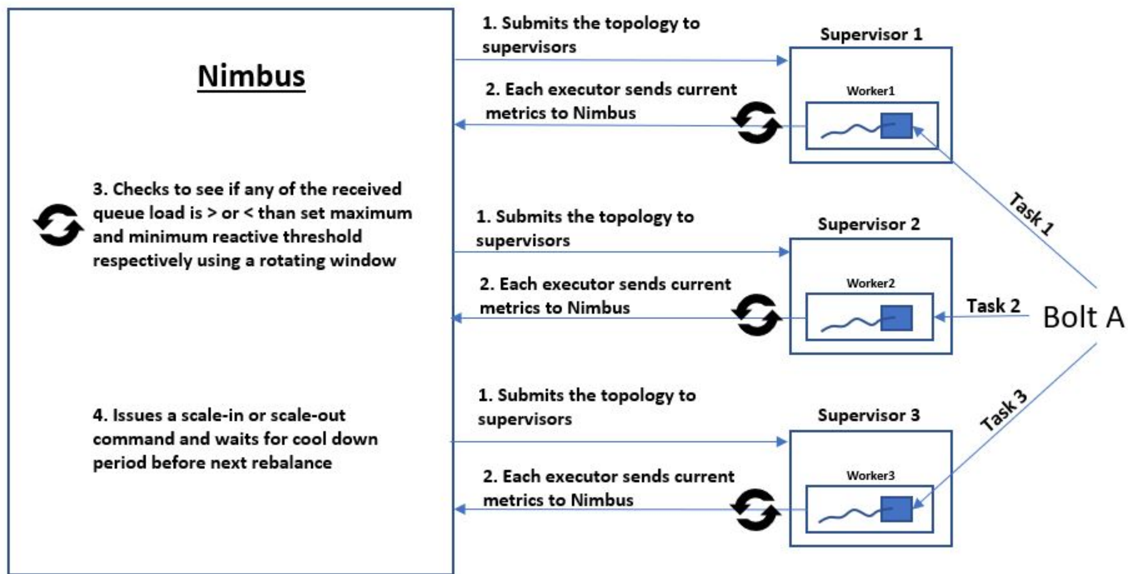


Figure 3.4: Steps during reactive re-scaling mode in ARSTREAM

of seconds. Similarly to scaling-out, it also performs the scaling-in operation by checking the loads lesser than the set minimum threshold.

When ARSTREAM receives queue utilization of each task for one iteration, it finds the maximum load among them. If the maximum load exceeds the threshold, it stores it in a rotating window of size 3 to keep a check against the unnecessary re-scaling needs. Since nimbus checks for bottleneck every 2 (configurable) seconds, it takes 6 seconds to fill up the window, and only after that, re-scaling is done. The rotating window respects some properties like- it should only contain values in increasing order for scale-out and decreasing order for scale-in. If it receives a value that is out of order, then the window is cleared, and the rest of the tasks are skipped. When we have three values in the rotating window, and all of them violate the threshold, the re-scaling module is called. ARSTREAM checks the last re-scaling time to make sure that it has served the cool-down period, and if it has, only then re-scaling is allowed. If it is still cooling down, the window is rotated, and a new value is inserted into the window.

```
TopologyId: federated-learning-1-1577081563, TopologyName: federated-learning, Bolt Id: BoltAA, Capacity: 0.680, queueLoad: 0.82516479
Prev Load: 0.74310302734375, Current Load: 0.825164794921875, AvgLoad: 0.0, checkIncreasingValues: 1, maxLoadInWindowFlag: 0.0
Values of load inside window: [0.74310302734375, 0.825164794921875]
```

Figure 3.5: An example of rotating window

When the re-scaling module is called, it will either scale-in or scale-out depending on the decision made by the reactive scaling module. We will discuss more what happens in the re-scaling module in Chapter 4.

Proactive Scaling Mode

Proactive scaling enables scaling wherein we predict the future load of an operator and then take the appropriate action (i.e., scale-in, scale-out, or do-nothing). It is better than reactive mode in terms of minimizing the loss of data because we take action before bottleneck happens and hence, saving some good number of the tuples from getting dropped during the queue flushing-out step at the start of re-balancing given the fact that actual queue size will be much lesser when re-balance command is initiated. We use a trained model in the background of each bolt task which predicts the future queue load and then these loads are sent to nimbus for taking the scaling action. This technique also has the user-configurable thresholds for scaling-in and scaling-out.

```
rebalance.proactive.min.threshold: 0.05
rebalance.proactive.max.threshold: 0.20
```

Figure 3.6: Scale-in and Scale-out thresholds for Proactive Scaling Mode

When nimbus gets the future loads of all bolt tasks, it finds the maximum load and stores it in a rotating window if the load is greater than the threshold defined by the user. It also follows all the properties defined for the rotating window, similar to what is done in the reactive module. The predictive scaling is also combined with the reactive mode to deal with missed scaling operations (i.e., false-negatives). Even if the previous model execution could not predict future load correctly, ARSTREAM will detect the excessive workload by applying reactive methodology and taking suitable scaling action [RQ2].

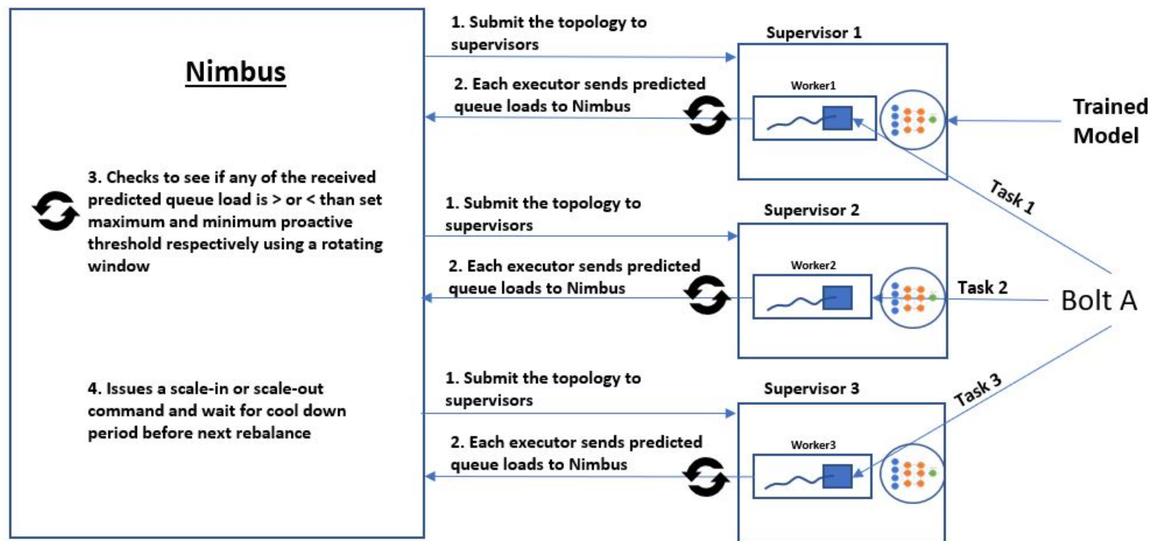


Figure 3.7: Steps during proactive re-scaling mode in ARSTREAM

In this effort, we chose to predict the queue loads that are 1 minute ahead in the future, given the fact that it takes on average 1 minute for the re-balance command to execute and restart the workers. More the seconds in future we predict, smaller the current queue utilization will be and hence, lesser the data loss. There is another form of data loss that occurs when the topology is halted during the re-balance stage and is not ingesting any data into the system. Therefore we want to minimize the re-balance time also which is possible by not aggressively re-scaling.

We use an artificial neural network to create the model which runs in the background of bolt we want to monitor. Each bolt task will produce its prediction and will send it to nimbus since each task run on its executor and has its queue. For training the model, we use Keras Deep Learning framework with TensorFlow as the back-end [68], and we also use many other techniques in feature engineering and PCA to create a model which gives us reliable predictions of the queue load. From the metrics we get from supervisors, we initially identified six different features based on workload characteristics and system resource utilization. After applying PCA, we settled with only four important features which give us maximum explained variance and they are:

- Arrival Rate of the Stream

- CPU Utilization
- Emit Rate of tuples from the bolt
- Current queue utilization

We created a network of six total layers with the ReLu activation function in the hidden layers and sigmoid in the output layer. We lag the output data column to include the value of the queue load, which is 1 minute ahead in the future. Once the model is trained, we store the model in the H5 file format and write it to ARSTREAM directory from where it is read inside the bolt task when the task is initialized. For reading the model and making predictions in real-time, we use the DeepLearning4J framework [69]. The tasks related to creating the ANN model are done externally of ARSTREAM and require the developer to generate the data before training the model, which is discussed further in Chapter 4. The data is generated for different arrival rates separately and is then programmatically combined for using it in model training.

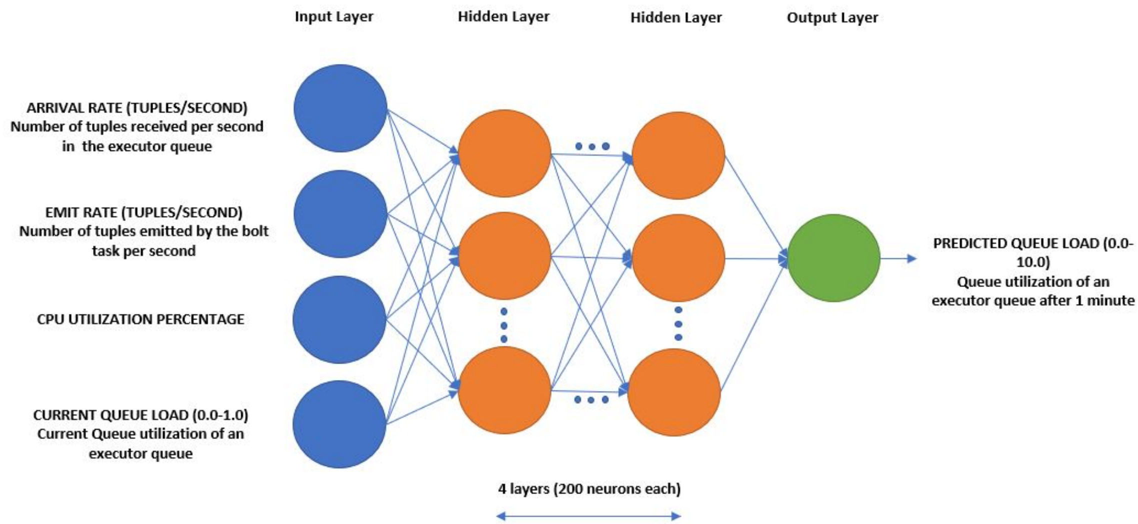


Figure 3.8: Artificial Neural Network model in Prediction Module

3.2 Task-Level Parallelism

3.2.1 What is Task-Level Parallelism?

Apache STORM consists of nodes that represent the physical machines in the cluster on which everything runs, workers that are JVM processes and spawns on the nodes, executors are the Java threads and runs inside worker processes and lastly the tasks that are instances of spouts and bolts where the user code runs. Why do we want to increase parallelism? Because we want to expand or at least maintain the throughput of the topology on the task it is executing. This motivates us to change the parallelism of an operator in the running topology for smooth execution.

STORM allows the user to change the level of parallelism at the worker level, executor level, task level, and even at the machine level. Each machine runs one or more worker processes to execute the topology, and each machine can have only as many worker processes as there is the number of slots specified in the configuration. Each worker process is assigned one or more threads, and each thread is assigned one or more tasks. The parallelism for a bolt/operator that we can achieve with any configuration is defined by the number of executors that are running and is used as "parallelism-hint". The number of tasks of a component in the topology is the task-level parallelism, and this number of tasks limits the executors that will be spawned for an operator.
We can only create as many executors as there are tasks to execute since a thread cannot be spawned for no task. STORM requires the user to specify the number of tasks for a component in the topology before even starting the topology. Therefore, the number of tasks is the limit to the number of executors that we can spawn dynamically. When STORM runs out of tasks to increase the parallelism further, parallelism can only be increased by restarting the topology with a greater number of tasks.

ARSTREAM implements a way to increase the number of tasks at run-time during re-scaling without stopping the topology. Hence, making it possible to increase the number of executors to as many as we want, assuming that we have available the sufficient amount of machines in our cluster to spawn that many executors.

3.2.2 Approach used to provide Task-Level Parallelism Updates

Creating hundreds or thousands of tasks while creating the topology to make them available for dynamically spawned executors during re-scaling is not a good idea because these tasks are executed serially if there are more tasks than the executors configured. What we want is to create these tasks on the fly only when we need them. STORM does not allow to change the topology once it is started. The reason being if the tasks maintain some local state and the inputs to the tasks are keyed on a field, then when we change the number of tasks in the topology on the fly, there is no guarantee that same key will be mapped to the same task which is what field grouping is and hence, corrupting the local state. ARSTREAM is not designed to work with field grouping, and so we can safely increase the number of tasks at run-time [70].

When ARSTREAM decides to scale-out, before calling the re-balance command, it modifies the

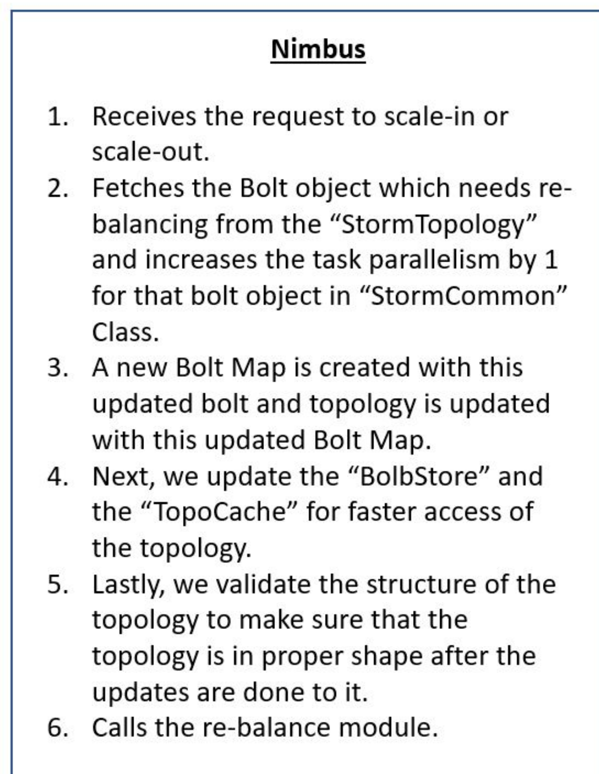


Figure 3.9: Steps followed for providing task-level parallelism updates in ARSTREAM

topology to increase or decrease the count of the tasks. After doing that, it needs to update multiple internal storage where the topology is stored (e.g., Blobstore, TopoCache, Configs, etc.). When the topology is updated and reflected to contain the updated task count for a component, ARSTREAM calls another module where the resource configuration is updated. It configures an additional worker and an additional executor to host that newly created task. As a result, when the topology is started again, the stream will be split equally among all the available tasks residing on their executors.

Chapter 4

System Architecture

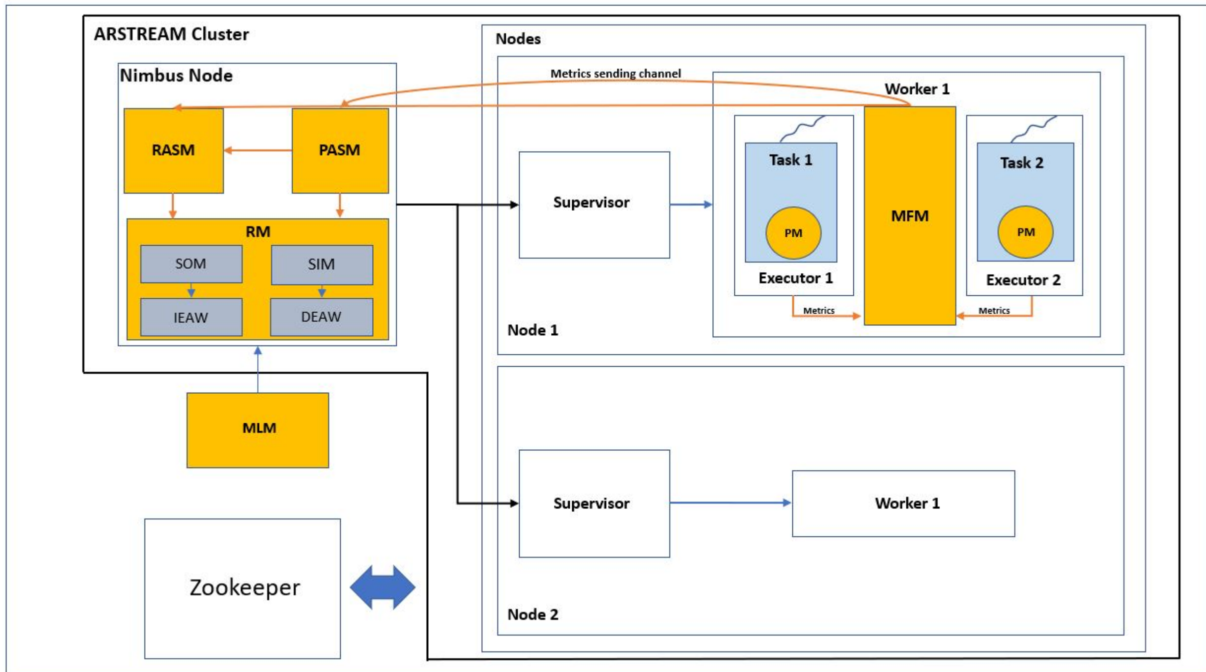


Figure 4.1: A High level view of the architecture of ARSTREAM engine built on the top of Apache STORM

The overview of ARSTREAM architecture is depicted in the Figure 4.1. ARSTREAM is built on the top of Apache STORM engine and it includes a Metric Fetch Module (MFM), Prediction Module (PM), Reactive Auto-Scaling Module (RASM), Proactive Auto-Scaling Module (PASM) and a Re-Scaling Module (RM) which helps in Monitoring, Sending, Analyzing, Planning and Executing the re-balance command either to scale-in or scale-out. All the components of ARSTREAM are tightly integrated with the STORM engine to not allow for any leakage, which could hamper its proper functioning except for one module (i.e., Machine Learning Module (MLM)), which works outside the ARSTREAM cluster.

4.1 System Components

4.1.1 Metrics Fetch Module (MFM)

There is a Metrics Fetch Module that runs in the background of every bolt task in the executor inside worker, which sends the system's real-time performance metrics data and the stream properties to nimbus for analyzing. These metrics are either fetched from the underlying system or created in the beginning and computed on the fly. This module works towards the monitoring phase of the five-step process described in the beginning for properly re-scaling the topology. PM also uses these metrics for generating the real-time predictions of the queue load, which are then sent to nimbus to be used by PASM [RQ1]. The metrics generated by MFM are also used by MLM for creating the machine learning model, which is deployed in PM in the background of each task.

4.1.2 Reactive Auto-Scaling Module (RASM)

The reactive module works reactively for dealing with the bottlenecks and calls the re-scaling module only after the bottleneck is spotted. This module analyzes the current queue load of each bolt task fetched by the MFM during the monitoring phase and which are sent to nimbus at regular intervals during the sending period. The analyzing phase consists of comparing the current queue load metric with the set threshold configured by the user in the configuration file, after which the planning phase starts. In the planning stage, ARSTREAM's RASM decides either to scale-in or to scale-out, which then calls the re-scaling module (RM) on the decision made during the planning phase.

4.1.3 Proactive Auto-Scaling Module (PASM)

The second technique for auto-scaling (i.e., Proactive scaling) predicts the future load of an operator and takes action before the bottleneck occurs. This module analyzes the future predicted queue load of each bolt task, which is sent to nimbus by the MFM at regular intervals. After the analysis phase, PASM decides to either scale-in or scale-out the resources to prepare for the future bottleneck, which then calls the re-scaling module (RM) for scaling the application. This module

also calls RASM to deal with the false negatives that may occur during the execution of the process due to frequent network traffic fluctuations [RQ2].

4.1.4 Prediction Module (PM)

This module is responsible for generating the predictions, which are 1 minute ahead in the future, of the queue load of each bolt task to detect any signs of potential overload [RQ1]. There is a trained artificial neural network model that runs in the background of each task as they are spawned, and it takes in a set of features generated for that task by MFM to output the prediction result. These are the predictions that are sent to nimbus by MFM for use in PASM. The model training is done by MLM before setting up ARSTREAM and is placed in the ARSTREAM directory to be read by the task.

4.1.5 Re-Scaling Module (RM)

The re-scaling module is the module where the actual scaling action is performed. First, both the RASM and PASM decides which scaling operation they want to execute, and then they request this module to execute that operation (i.e., either scaling-in or scaling-out). Both the scaling operations are written as separate modules, and only one of them is called at a time. Scaling-out module (SOM) calls "IncreaseExecutorsAndWorkers (IEAW)" whereas scaling-in module (SIM) calls "DecreaseExecutorAndWorkers (DEAW)" which are the modules inside RM. The re-balance command that is called inside RM, is called in IEAW and DEAW. Both SOM and SIM updates the topology to increase or decrease the number of tasks and then update the storage and cache to reflect updated topology, which is how we achieve task-level parallelism updates. Once this task is performed, it calls IEAW or DEAW. IEAW is responsible for checking if there are free slots to spawn a new worker. It issues a re-balance command with an increased executor and an increased worker only if there are free slots. DEAW is also responsible for checking if there are used slots, and if there are any, it checks the current count of executors. It issues a re-balance command with the decreased count of worker and executor only if the count of executor is more than one.

4.1.6 Machine Learning Module (MLM)

This module is also a part of ARSTREAM but lies outside the bounds of tight coupling with STORM. This module is responsible for creating and writing an artificial neural network model in an H5 format file, which is then read by the PM for making real-time predictions inside the running task at regular intervals. This module is written in Jupyter Notebook and comprises of reading the data, feature engineering for preparing the data, scaling and normalizing the data, creating and training an artificial neural network model, and then writing the trained model to a file. The data that is read in the first step is first written into a CSV file by nimbus; Nimbus receives the features from MFM, which are periodically stored in a data structure and, in the end, writes that to a CSV file. Out of all the features we receive from MFM, we keep only the selected ones, and that decision is made using Principal Component Analysis (PCA). PCA has been consistent in giving four total features that proved to be useful in making predictions.

4.2 Internal working Of ARSTREAM and System Design Constraints

When the re-balance command is issued, there is a sequence of steps that are performed for accurately re-balancing the topology. The user submits the request either through CLI or WebUI in STORM. In contrast, in ARSTREAM, we call the re-scaling module, which internally calls the re-balance command with the updated configuration of workers and executors. As soon as the operation is called, the topology is stopped for a while, all the components are deactivated, spout stops ingesting new data, and the status is set to `REBALANCING`. While the topology is in `REBALANCE` state, all the pending messages inside the queues of executors are flushed out, all the assignments of workers and executors are cleaned up in the cluster state, and finally, the worker and executors are killed to make them ready for updated assignments with new tasks, workers and executors. The new configuration is updated in the storm-base, which is a thrift object that maintains the information of running topologies to be used by the client. This storm-base is updated in the state storage,

which maintains the state across all nodes. The topology is also updated in blob-store, which is a cache for topologies and topology configs. Once these steps are completed, nimbus computes new assignments for workers, executors, and tasks and starts all of them. These are the steps that are performed when the re-balance command is issued inside the re-scaling module.

In ARSTREAM, there is a module that nimbus calls after every configurable number of seconds to check the system bottleneck. Each running executor maintains a queue for incoming data and a transfer queue to send messages to other workers. When we talk about queue load, it is the current incoming queue size divided by the total capacity; it tells us what percentage of the queue is filled, and this is the queue that drops all the tuples during re-balancing. Each executor has its queue, and there can be multiple tasks for one executor in STORM, but we limited it to one task per executor in ARSTREAM to achieve maximum parallelism. Since there is only one task per executor, each executor sends its current queue utilization to ARSTREAM's scaling module inside nimbus using a thrift service, and all of this happens inside the MFM module. Similarly, executors also send queue predictions to the ARSTREAM's scaling module.

Workers those are JVM processes run the executor inside them, and they can be configured in the topology in the same way as executors. But there is a constraint to it; there can only be as many workers on one supervisor as there are slots configured, and there can only be one supervisor on a machine. Since ARSTREAM has been designed to run resource-intensive applications like deep learning, each worker is configured to use maximum memory from the supervisor. We try to limit the number of workers on one supervisor so that we can provide maximum resources to one worker from one machine, which hosts a component. The other important thing to note is about the limit to the number of executors on one worker; we limit the number of executors to only one per worker and spawn a new worker for each new executor. Having more workers will give room for more data to enter the system since each worker has its TCP Socket, which is essential for maximizing throughput.

Chapter 5

Experiments And Empirical Evaluation

The experiments to compare STORM with auto-scalable ARSTREAM were conducted on a cluster consisting of 20 nodes with an 8 core HP-Z420-XeonE5-2650v2 CPU @ 2.6GHz processor, 32GM RAM and 1TB disk. Each machine in the cluster is connected to a Gigabit Ethernet switch and runs Fedora 28 operating system. STORM version 2.0 is installed in the file system, which is shared and is configured to use 28,672MB worker heap memory, 32,768MB supervisor memory, and 7,168MB topology component resources on-heap memory from the cluster. ARSTREAM uses the same cluster with the exact same configuration for the experiments.

The application we use to test auto-scalability in ARSTREAM and to compare with STORM is the federated learning approach for machine learning with streaming data, and it uses parameter server as an external system to store the weights and parameters. The MNIST data-set was used for our experiments, which is an easy and straightforward data-set of handwritten digits. Still, it can easily overload the system and, therefore, becomes the right candidate for our experiments [71].

This federated learning application can be used as a topology for STORM and ARSTREAM in which we can also configure the number of workers, executors, and tasks before submitting the topology. We configured the topology to start with minimal resources at the beginning assuming that we do not know the best configuration of different components in the topology. When the topology is executed, the STORM will not automatically re-scale the topology to adjust for changing workload conditions. However, ARSTREAM automatically adapts to the changing needs of the topology by adding/removing additional parallel tasks to consume the extra tuples that the system is ingesting and which the bolt was not able to consume initially.

Inside the spout, we read a directory path where images are stored, which is sent to downstream bolt as a tuple. We also modified the code in spout to simulate for different arrival speeds by putting a delay after sending every tuple to the downstream bolt tasks. The value for the sleep/delay is generated randomly from a gaussian distribution. We also persisted the value of sleep for some

time, and the value is changed only after that set period has passed. We chose two different periods (sleep persist duration) for our experiments to see how the system performs in each case. Changing the value of sleep after every tuple is sent to downstream bolt tasks is a random case where there is no pattern in the data arrival and is also a limitation for proactive mode. However, reactive mode reacts promptly to the changing stream speed and, therefore, works well for all the cases but with data loss. It depends on the application use case if the stream it uses has any pattern or seasonality in it or not, but for our experiments, we assume that the arrival rate of the stream does not follow any pattern and will be persisted for some time before getting changed again.

The two ways of scaling an application automatically reactively and proactively work differently, and therefore, we plot the results for both the techniques separately. The ARSTREAM configuration that is common to both of the techniques is shown below-

```
topology.rebalance.time.scaleout: 120000 # In milli-seconds
topology.rebalance.time.scalein: 420000  # In milli-seconds
check.capacity.rebalance.recur.secs: 2   # In seconds
check.capacity.rebalance.delay.secs: 60  # In seconds
```

Figure 5.1: ARSTREAM's common system configuration for scaling

We compare the reactive and proactive auto-scaling mode of ARSTREAM with not auto-scalable STORM using following comparisons-

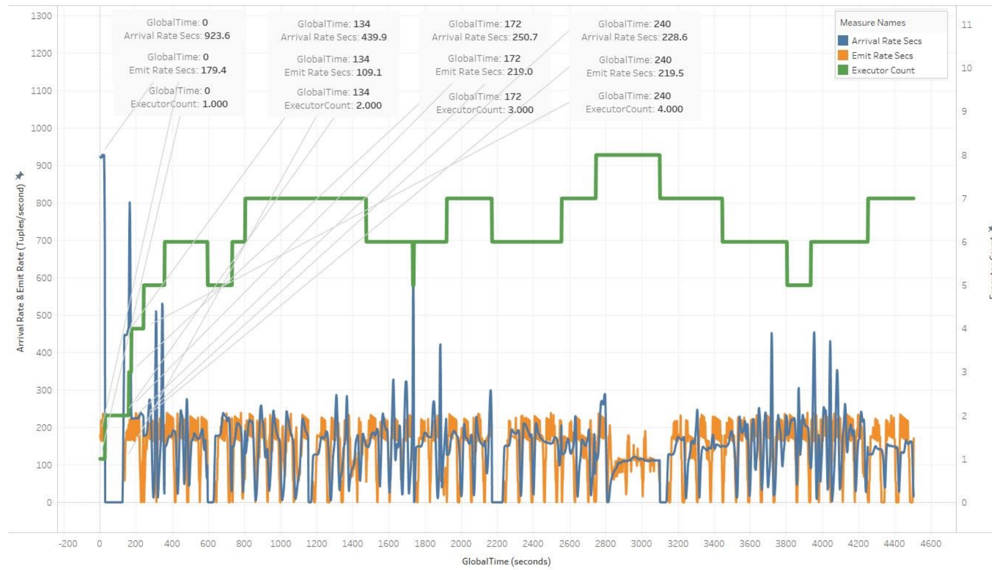
- **The first set of experiments compares ARSTREAM and STORM with a constant arrival rate of the stream throughout the execution of topology. To contrast these systems, we observe the effect of changes in executor count on the arrival rate and the throughput.**
- **Our second experiment is designed to evaluate the effect of varying arrival rates (5 minutes persist duration) of the data stream on the executor count and throughput while topology is executed.**
- **Finally, we contrast ARSTREAM with STORM with a frequently fluctuating (1 minute persist duration) data arrival rate.**

5.1 Experimenting with Reactive Scaling Mode

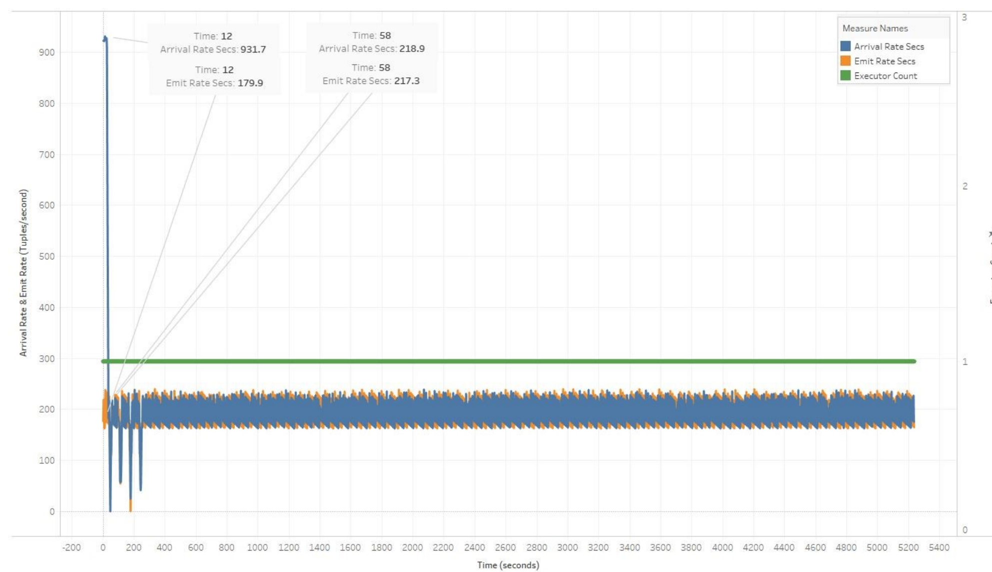
In our experiments with reactive mode, we kept the threshold for re-scaling to be 20 percent of the queue size for scaling-out, 5 percent of the queue size for scaling-in, and we kept two different durations for persisting sleep (i.e., 5 minutes and 1 minute). We plot the throughput graphs and compare ARSTREAM 's performance with that of STORM. The throughput in all of the experiments is calculated as the total number of tuples emitted in a 30-second window divided by 30, which gives us throughput per second, and this is how STORM also calculates throughput of their system.

Figure 5.2a and Figure 5.2b compares ARSTREAM and STORM by observing the effect of increase in executors on the arrival rate when the topology is started with constant stream speed. We see that when the number of scale-out operations is performed in ARSTREAM, the arrival rate decreases at each addition of a new executor. The reason for the decrease in arrival rate is that the stream gets split among all the executors of a component resulting in the decrease in arrival rate. This arrival rate is for a single task, and this scale-out operation is performed until the arrival rate becomes equal to the emit rate by continuous splitting at each step. Figure 5.2c and Figure 5.2d compares the throughput of reactive scalable ARSTREAM engine with that of STORM for same arrival rate i.e. with same delay between each tuple. As compared to STORM, when more executors are added in case of ARSTREAM for the same arrival rate, throughput increases at each step and when arrival rate for each task becomes equal to the emit rate, no more executors are added, and the topology becomes stable. The reason for this increase in throughput is that the additional tasks emit more tuples. In STORM, we observe that with one executor task configured for a bolt, throughput never goes above 198 tuples/second when the arrival rate is around 1000 tuples/second. In contrast, in ARSTREAM, we achieved a throughput of 1000 tuples/second, which is the maximum we could get at this arrival rate.

If an operator cannot emit tuples with the speed at which it is getting data, it causes a bottleneck in the topology and resulting in spout throttling. ARSTREAM alleviates bottlenecks by



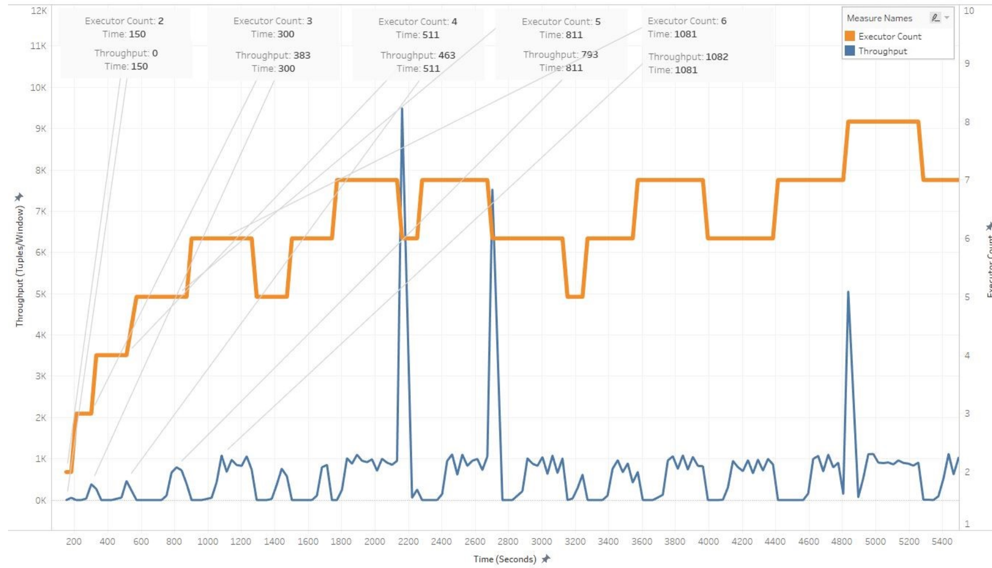
(a) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for constant arrival rate in ARSTREAM



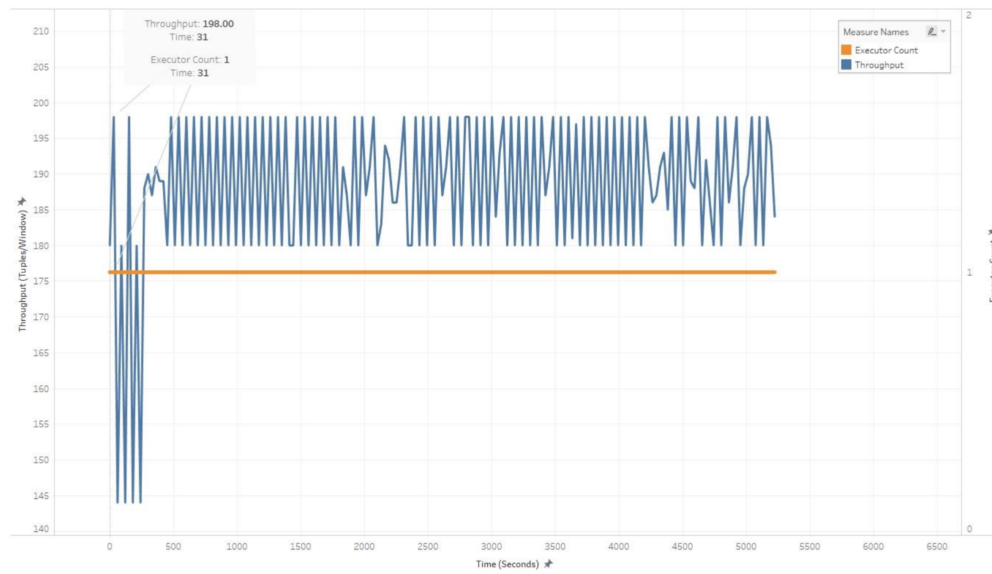
(b) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for constant arrival rate in STORM

Figure 5.2: A comparison between ARSTREAM and STORM executed with constant arrival rate, compared by observing the relation between arrival rate, emit rate and executor count, using Reactive Scaling Mode.

continuously spawning new tasks to run in parallel until the arrival rate becomes equal to the processing speed of each bolt task (i.e., emit rate of the bolt task), by splitting the stream.



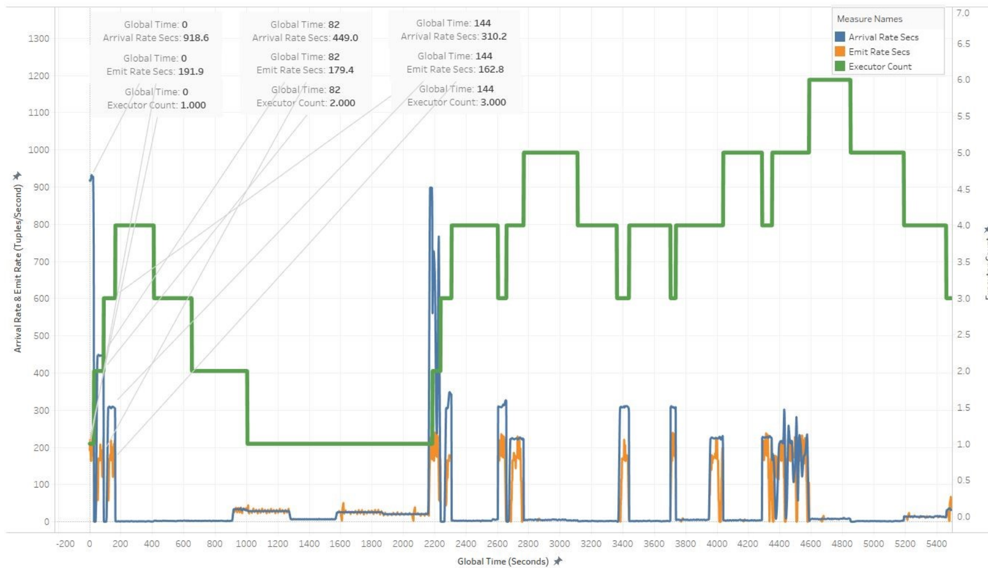
(c) Graph showing the relation between Throughput, Executor Count and Time for constant arrival rate in ARSTREAM



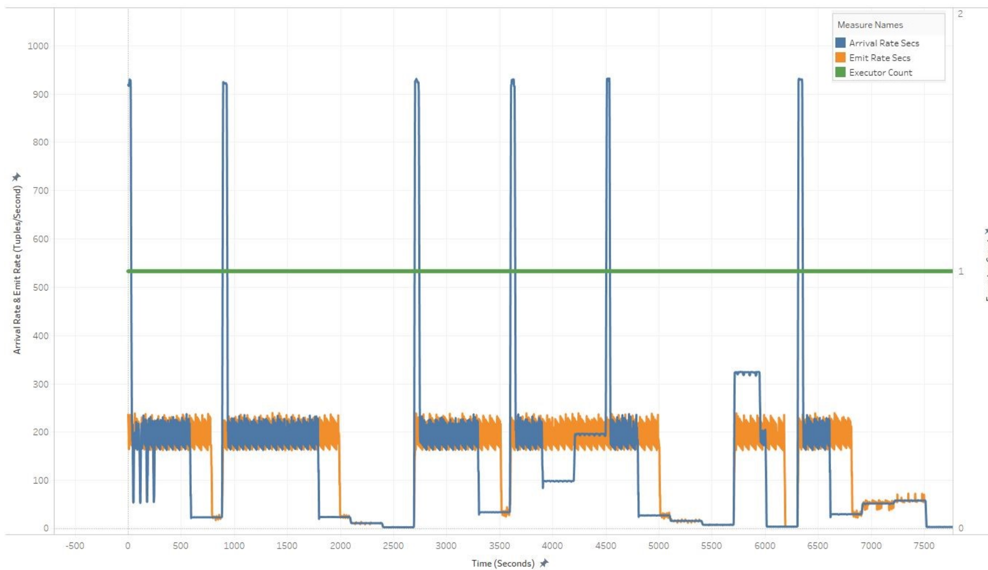
(d) Graph showing the relation between Throughput, Executor Count and Time for constant arrival rate in STORM

Figure 5.2: A comparison between ARSTREAM and STORM executed with constant arrival rate, compared by observing the relation between executor count and throughput, using Reactive Scaling Mode.

The next set of graphs depicts the results of the evaluation with variable arrival rates to simulate the real world condition. We chose two different periods to persist the arrival rate (i.e., 5 minutes and 1 minute), with 1 minute being closest to the utterly random case having no pattern at all.



(a) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for varying arrival rate in ARSTREAM



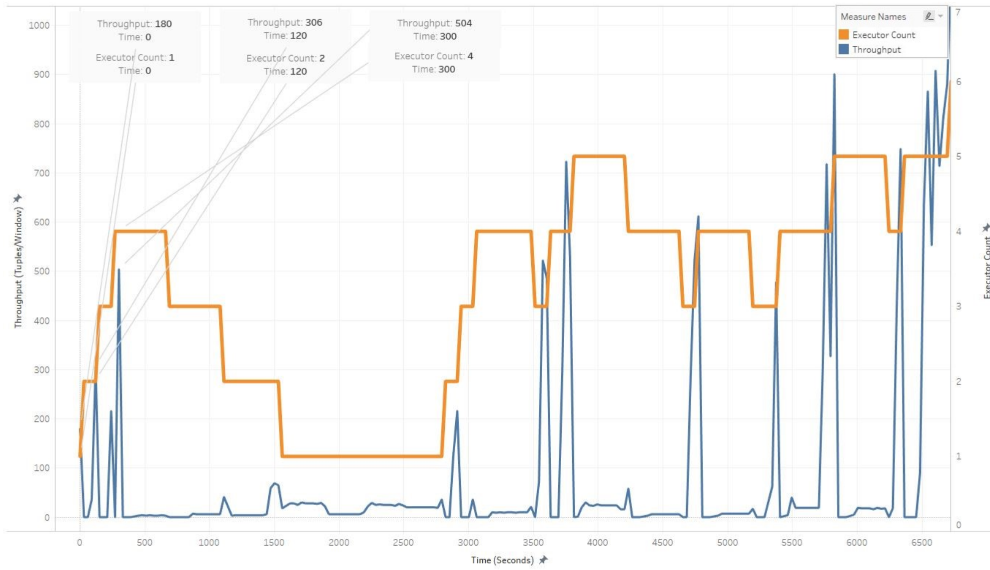
(b) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for varying arrival rate in STORM

Figure 5.3: A comparison between ARSTREAM and STORM executed with varying arrival rate and sleep persist duration of 5 minutes, compared by observing the relation between arrival rate, emit rate and executor count, using Reactive Scaling Mode.

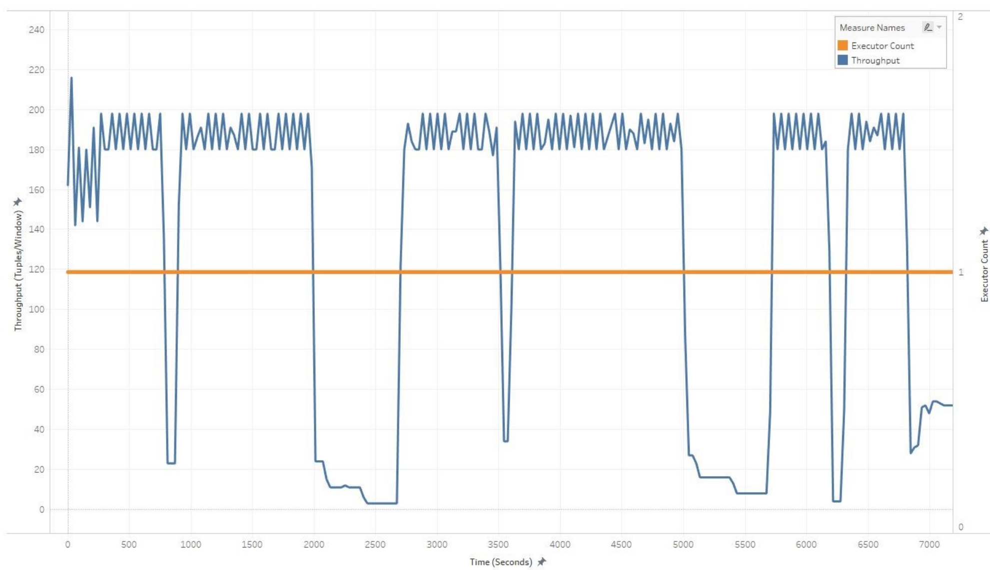
Figure 5.3a is a graph of varying arrival rates in which a new arrival rate is generated every 300 seconds. We plot the variables on three different axes to show the effect of varying arrival rates on the emit rate and the executor count. The execution starts with the stream speed of 980 tuples/second, which the topology tries to process using just one executor resulting in the queue getting filled up very quickly. As soon as the queue load reaches the threshold, topology is re-scaled, and a new executor is added to split the load equally between two executors. The process of re-scaling and splitting the stream is done until the arrival rate becomes equal to the emit rate of the task. After 300 seconds, a new arrival speed is generated, and scale-in operations are performed to adapt to the workload. Hence, ARSTREAM successfully adapts to the changing speed of the stream reactively as it happens, but STORM in Figure 5.3b is not able to adapt to the stream speed resulting in spout throttling getting activated whenever a task gets the stream which has arrival rate more than its processing speed. Figure 5.3c and Figure 5.3d compares the throughput of ARSTREAM and STORM in case of varying speed of the stream. ARSTREAM shows an increase in throughput as new executors are added, but STORM never emits more than 200 tuples/second since there is only one executor to process the data. Although we can keep the number of tasks to the maximum, which will process all the data that we expect to come inside the STORM system, it will waste the resources when the speed is much lesser than the total processing capacity of the bolt.

Figure 5.4a, Figure 5.4b, Figure 5.4c and Figure 5.4d are for experiment with varying stream speed but with 1 minute of stream speed persist duration before a new speed is generated. We observe that ARSTREAM re-scales whenever it sees an increase in the arrival speed which is greater than the emit rate of the bolt task i.e. 200 tuples/second. At all the points in the graph of Figure 5.4a where blue point lies above the orange point, system is scaled-out whereas in case of STORM in Figure 5.4b, arrival rate varies but the peaks that comes down is not because of splitting up of the stream; it is because of spout throttling. Figure 5.4c shows an increase in throughput as expected and hence, ARSTREAM works fine for 1 minute persist also.

Overall, Reactive Scaling Mode effectively manages the arrival rate regardless of the sleep persist duration. The advantages of the reactive scaling mode include the following aspects. First, it

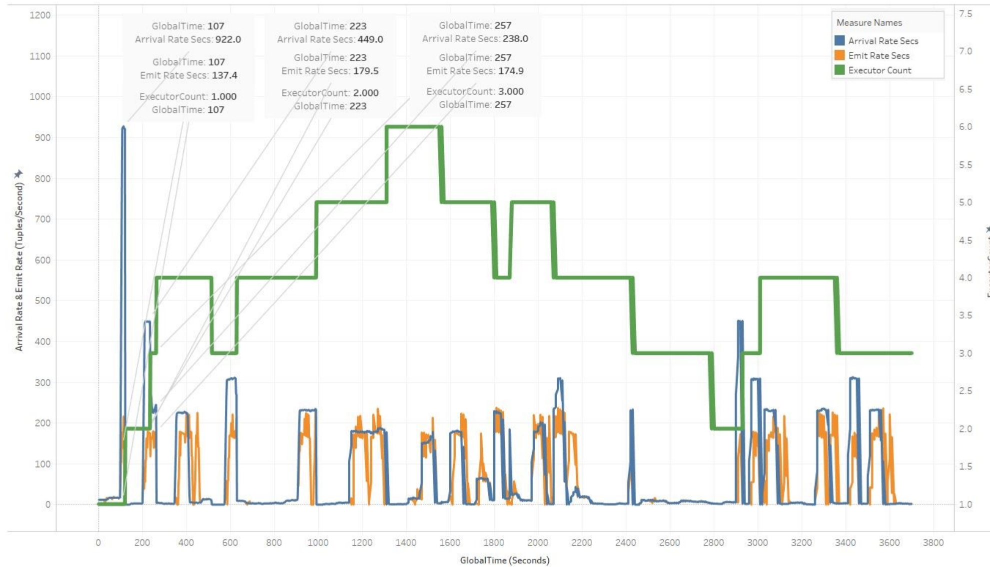


(c) Graph showing the relation between Throughput, Executor Count and Time for varying arrival rate in ARSTREAM

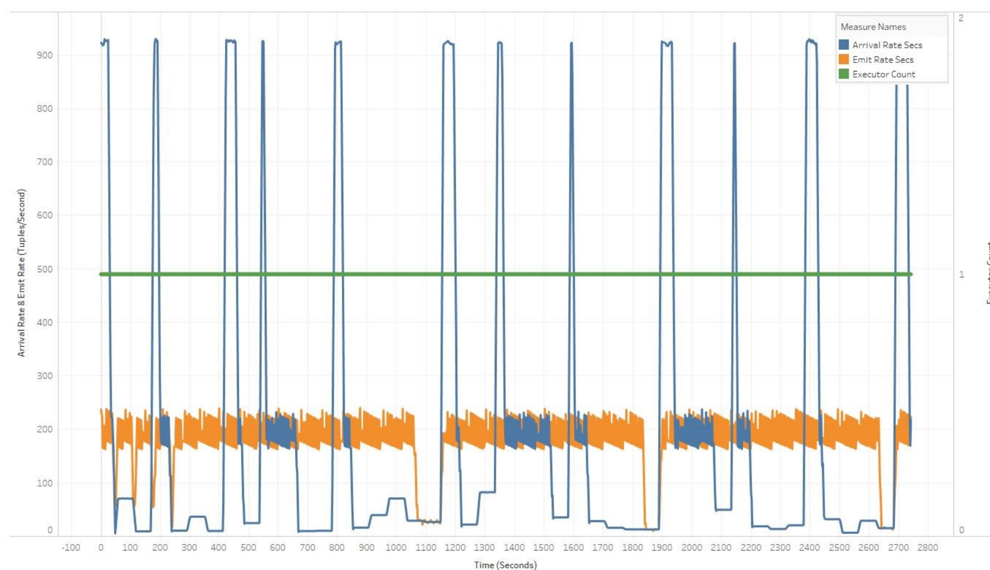


(d) Graph showing the relation between Throughput, Executor Count and Time for varying arrival rate in STORM

Figure 5.3: A comparison between ARSTREAM and STORM executed with varying arrival rate and sleep persist duration of 5 minutes, compared by observing the relation between executor count and throughput, using Reactive Scaling Mode.



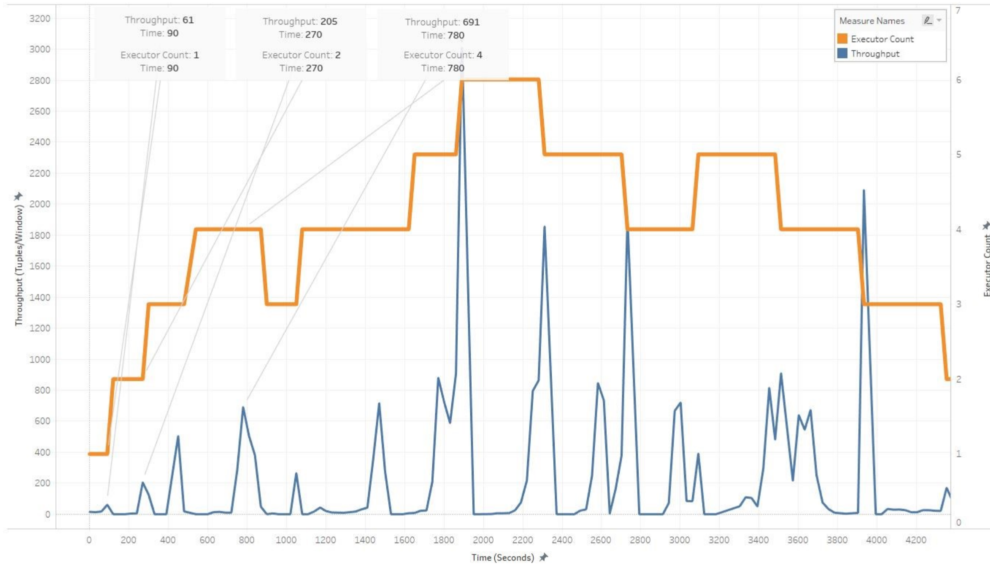
(a) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for varying arrival rate in ARSTREAM



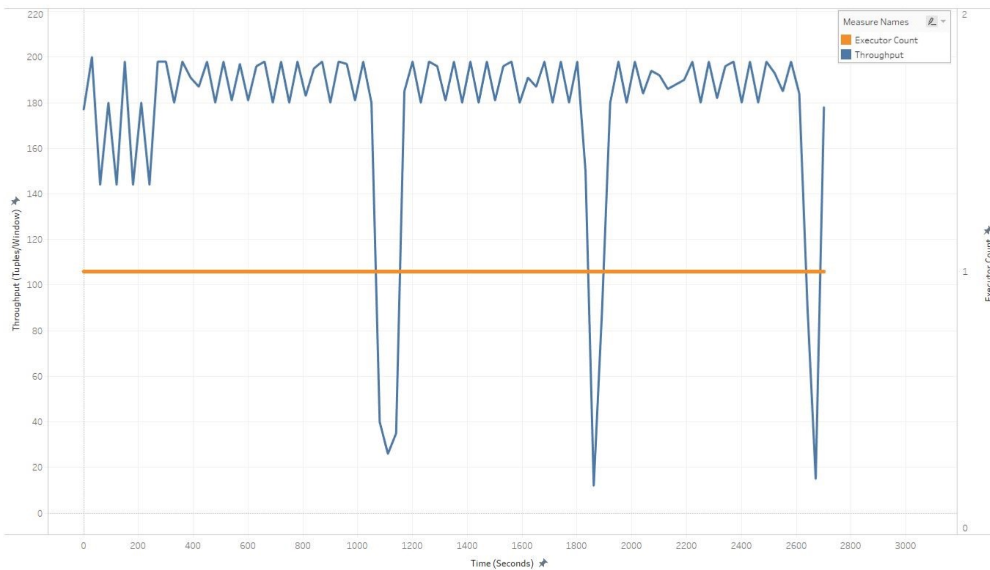
(b) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for varying arrival rate in STORM

Figure 5.4: A comparison between ARSTREAM and STORM executed with varying arrival rate and sleep persist duration of 1 minute, compared by observing the relation between arrival rate, emit rate and executor count, using Reactive Scaling Mode.

is easy to apply if the users are familiar with their computations and the cluster capacity. Second, overhead from the required computation is negligible. However, we anticipate a set of disadvan-



(c) Graph showing the relation between Throughput, Executor Count and Time for varying arrival rate in ARSTREAM



(d) Graph showing the relation between Throughput, Executor Count and Time for varying arrival rate in STORM

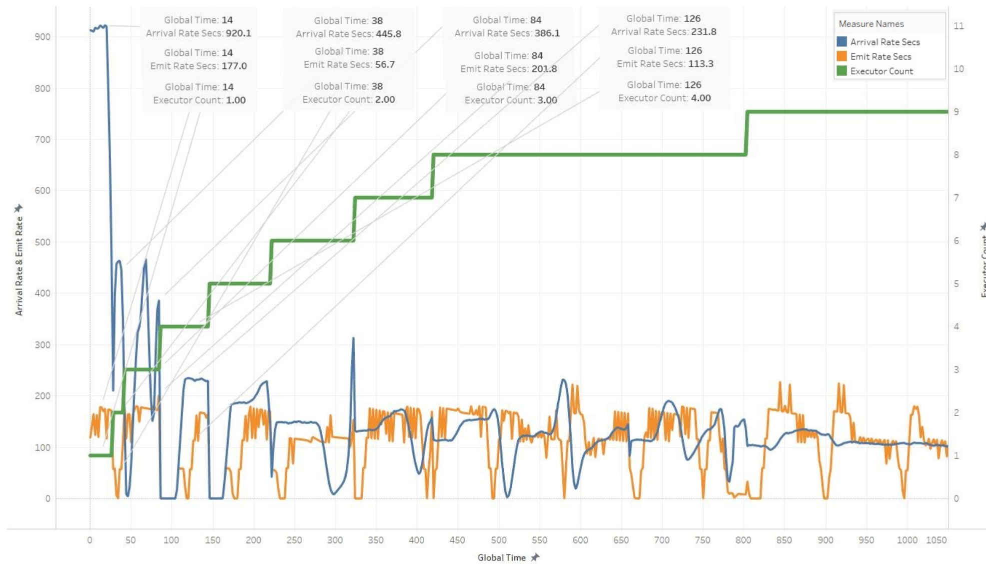
Figure 5.4: A comparison between ARSTREAM and STORM executed with varying arrival rate and sleep persist duration of 1 minute, compared by observing the relation between executor count and throughput, using Reactive Scaling Mode.

tages, such as the potential data loss from the queues during scaling operation, which the proactive mode tries to minimize.

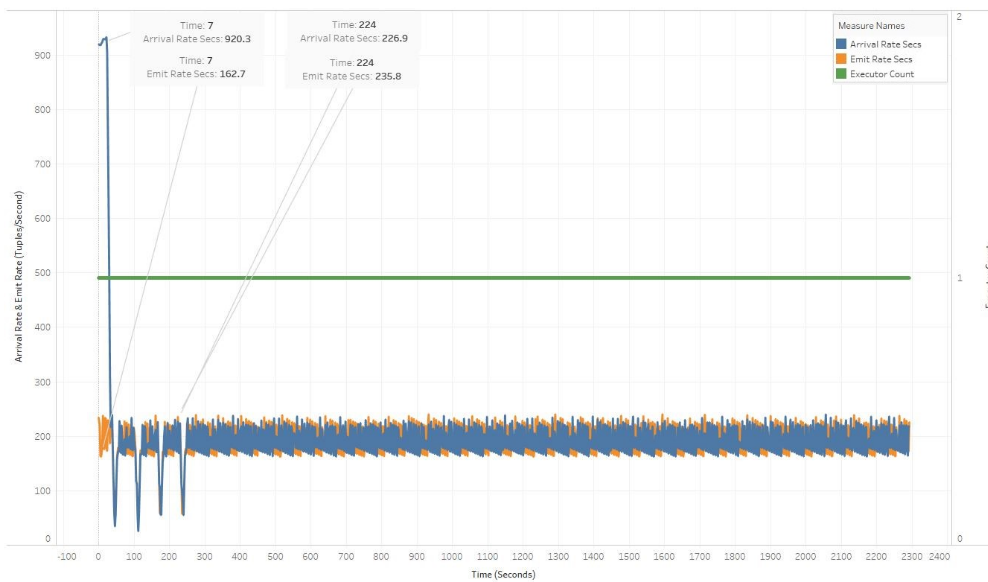
5.2 Experimenting with Proactive Scaling Mode

The other technique for re-scaling is proactive or predictive. For any application that we run on ARSTREAM, we first train and create an artificial neural network on the application's metric data, which is then deployed in the bolt to run in the background. We measure the MSE values of the model for training, testing, and real-time data in the next section. In this technique, we kept the scale-out threshold to be 40 percent of the queue size, the scale-in threshold to be 5 percent of the queue size, and we kept two different durations for persisting sleep value (i.e., 5 minutes and 1 minute), just like in reactive scaling mode. Once we have a model up and running in each bolt's task, we run the application and observe the results, which are then plotted in the form of throughput and the comparison of emit rate and arrival rate plotted on two different axes. Throughput is calculated as the average of total emitted tuples in a 30-second window. The graphs will also contain the number of tasks/executors on one axis to observe the effect of changes in the parallelism of the operator on the throughput.

Figure 5.5a and Figure 5.5b compares ARSTREAM with STORM for constant arrival rate by looking at how arrival rate changes when a new task is added in parallel. In Figure 5.5a, the arrival rate starts from 980 tuples/second (1 executor) and then decreases at each step until it comes closer to the emit rate by adding new tasks in parallel whereas in Figure 5.5b, the arrival rate here also starts from 980 tuples/second but spout throttling decreases the arrival rate. The executor count in this case always stays equal to 1. We discussed about what happens when a new task is added in parallel (i.e., arrival rate splits equally among the tasks), but to measure the overall performance, Figure 5.5c and Figure 5.5d gives us the better view. In Figure 5.5c, we can see that as new executor gets added at each step, throughput also increases whereas in STORM shown in Figure 5.5d, throughput is always constant. Also as an experiment, we ran the application for two different times in ARSTREAM for constant arrival rate to see how different the execution is with



(a) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for constant arrival rate in ARSTREAM

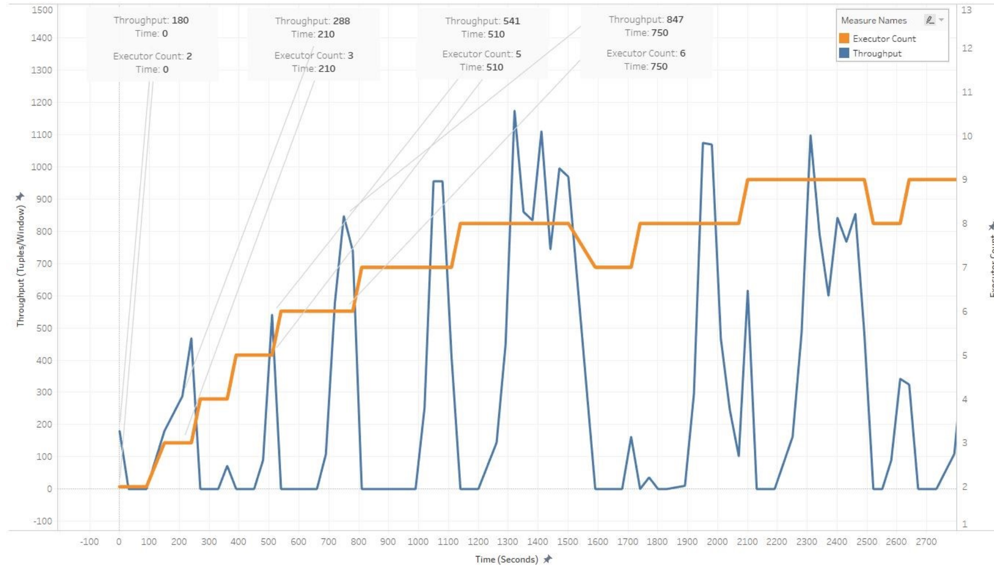


(b) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for constant arrival rate in STORM

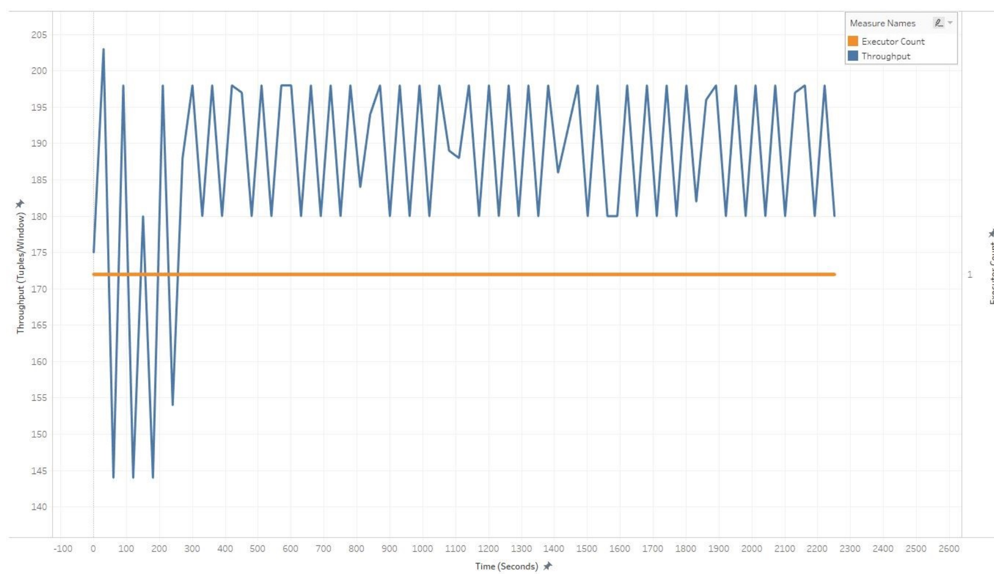
Figure 5.5: A comparison between ARSTREAM and STORM executed with constant arrival rate, compared by observing the relation between arrival rate, emit rate and executor count, using Proactive Scaling Mode.

respect to re-scaling. Overall shape of the graph looks same except for the dips at executor count 8 and 9 in Figure 5.5c.

The next set of graphs tries to explain what happens when we have different arrival rate persist durations with varying arrival rate and what effect does quickly changing arrival rate has on the



(c) Graph showing the relation between Throughput, Executor Count and Time for constant arrival rate in ARSTREAM



(d) Graph showing the relation between Throughput, Executor Count and Time for constant arrival rate in STORM

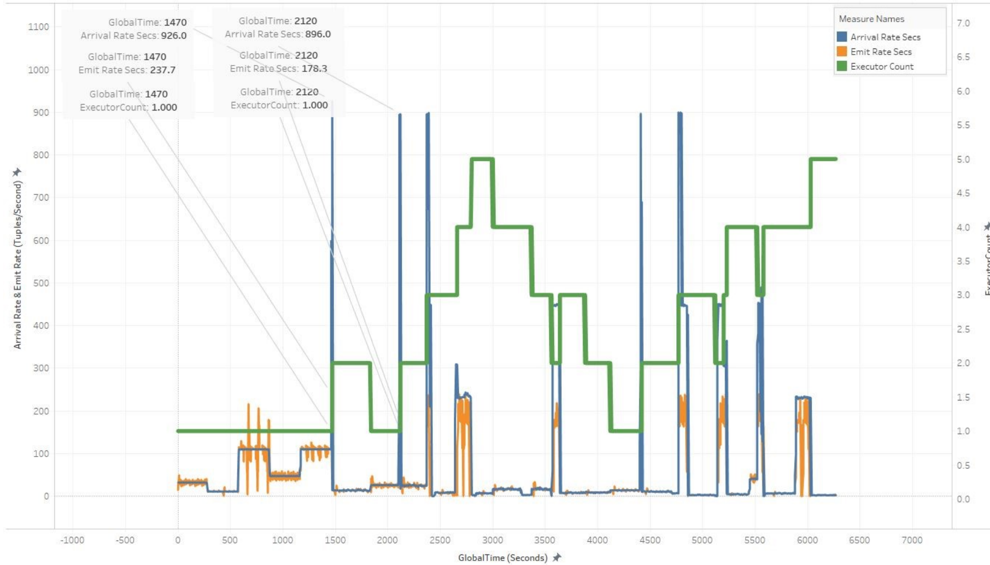
Figure 5.5: A comparison between ARSTREAM and STORM executed with constant arrival rate, compared by observing the relation between executor count and throughput, using Proactive Scaling Mode.

predictions and hence, re-scaling. We start with 5 minutes of sleep persist duration, and then later, we will plot for 1 minute. Sleep persist duration of 1 minute will not give enough time to the model to make correct predictions given the fact that it is predicting 1 minute in the future, which is equal to the sleep persist duration.

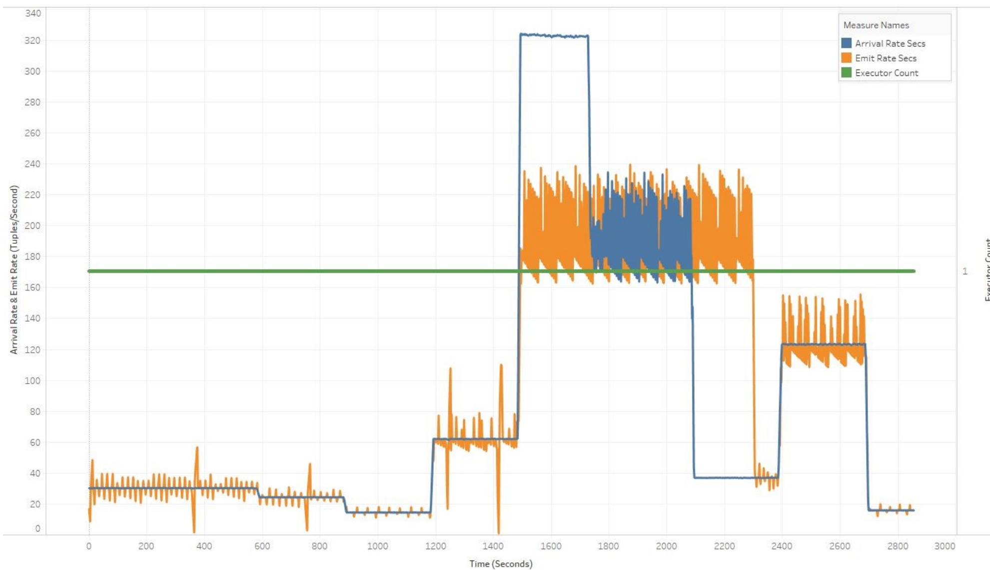
Figure 5.6a shows that ARSTREAM has been prompt in re-scaling on seeing the arrival rate going up and with speed changing every 5 minutes. There has been a scale-out operation executed by ARSTREAM for every peak, which has a value greater than the emit rate. STORM, which is not auto-scalable, will not do anything to increase the throughput if it gets the data arrived at the speed, which is more than the emit rate, as shown in Figure 5.6b. Figure 5.6c proves that as new executors are added in ARSTREAM, throughput also increases while in STORM, throughput stays the same which is equal to the emit rate multiplied by the number of tasks as shown in Figure 5.6d. When STORM does not have enough tasks in parallel to process all the data which arrives at a relatively high speed, queues will get filled up quickly resulting in spout activating its throttling mechanism, after which it stops sending messages to the bolt task or decreases it to a speed which is equal to the emit rate.

In Figure 5.7a, Figure 5.7b, Figure 5.7c, Figure 5.7d, we compare STORM and ARSTREAM using 1 minute sleep persist duration (i.e., arrival rate changing after every 60 seconds). Figure 5.7a shows that ARSTREAM scales-out whenever it sees an arrival rate higher than the capacity of the task but then arrival rate changes again to which the scale-in operation's request will have to wait for cool-down period. Mostly, ARSTREAM has been able to take actions whenever the arrival rate was higher. Throughput is also observed to increase with executor count in Figure 5.7c whereas STORM worked at its capacity and with low throughput.

Overall, the proactive mode also effectively manages the arrival rate. But as we moved closer to the entirely random case (i.e., 1 minute persist duration), the performance must have been degraded, which we will measure in the next section. But proactive mode by predicting in advance minimizes the data loss in contrast to reactive mode, which does not minimize.



(a) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for varying arrival rate with 5 minutes sleep persist duration in ARSTREAM

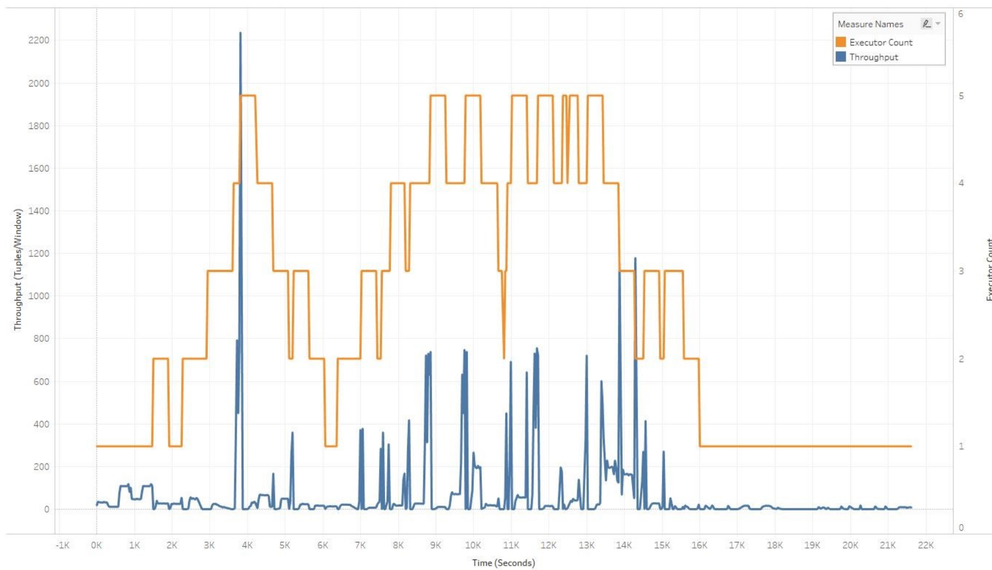


(b) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for varying arrival rate with 5 minutes sleep persist duration in STORM

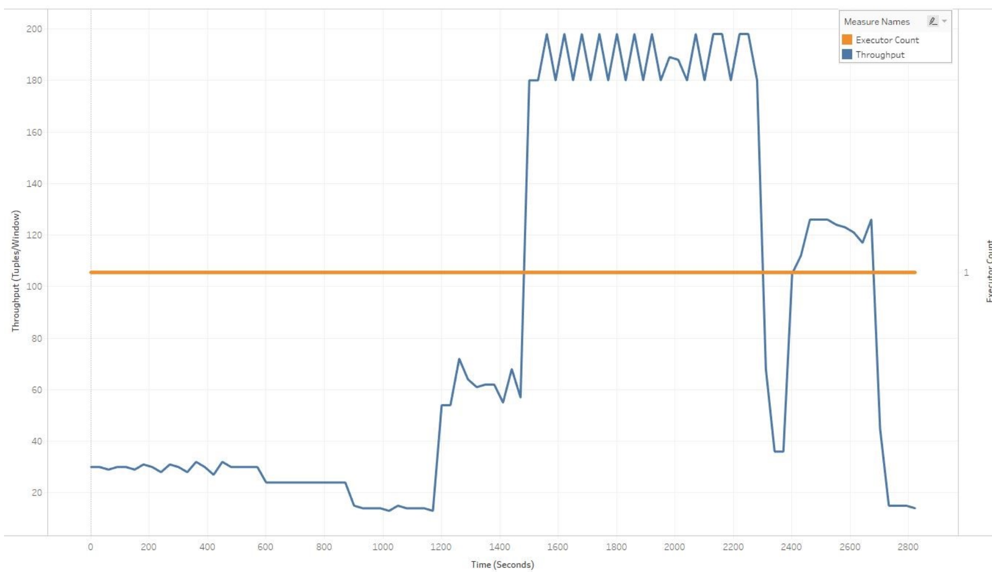
Figure 5.6: A comparison between ARSTREAM and STORM executed with varying arrival rate and sleep persist duration of 5 minutes, compared by observing the relation between arrival rate, emit rate and executor count, using Proactive Scaling Mode.

5.3 Proactive Model Accuracy

We trained the proactive model on the metrics data generated by MFM, which was sent to nimbus and written to a file at the end of the process. We generated input data for different arrival

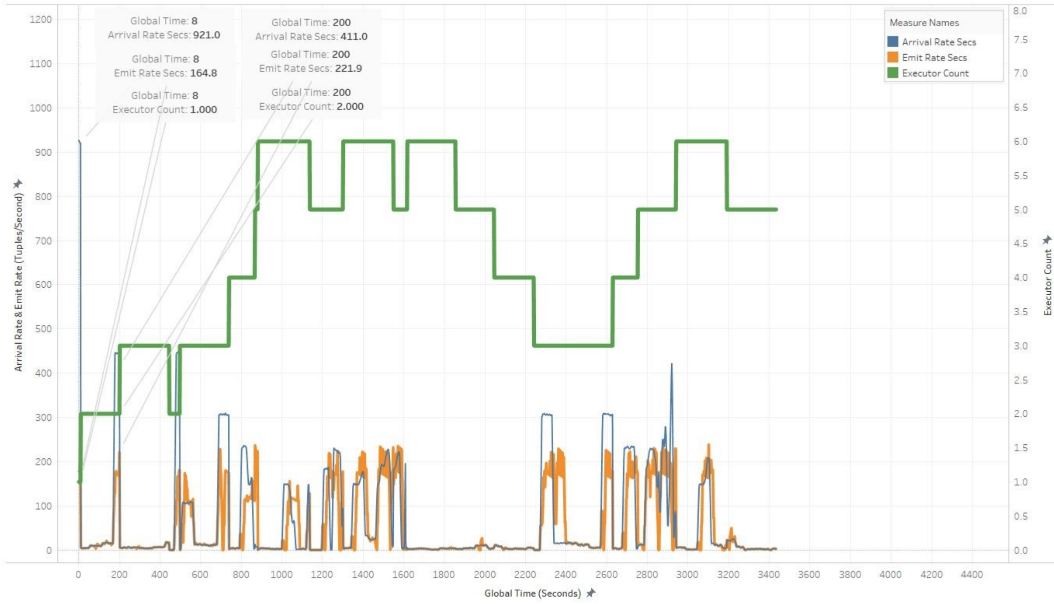


(c) Graph showing the relation between Throughput, Executor Count and Time for varying arrival rate with 5 minutes sleep persist duration in ARSTREAM

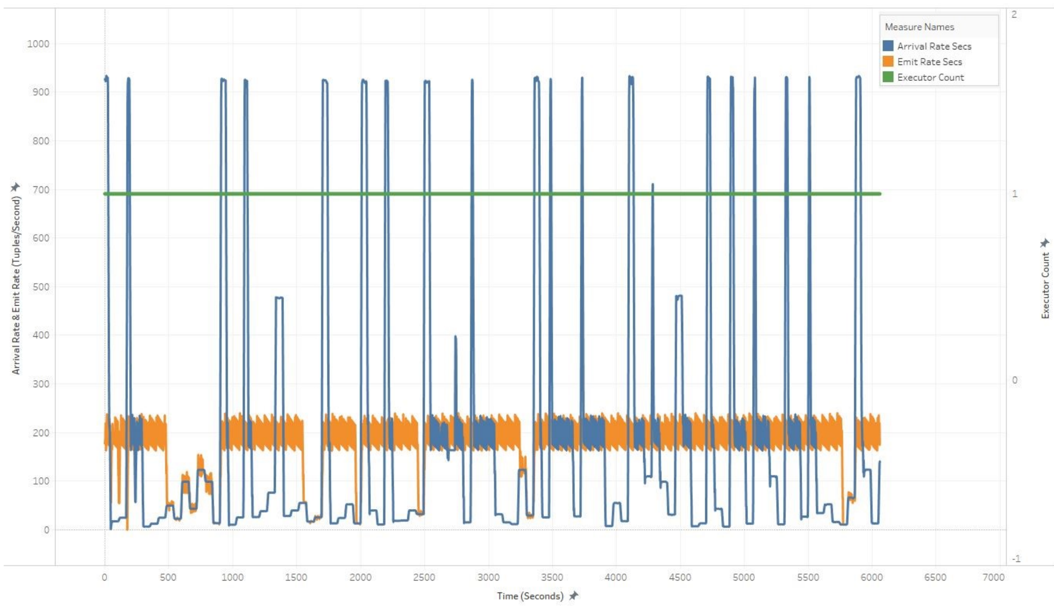


(d) Graph showing the relation between Throughput, Executor Count and Time for varying arrival rate with 5 minutes sleep persist duration in STORM

Figure 5.6: A comparison between ARSTREAM and STORM executed with varying arrival rate and sleep persist duration of 5 minutes, compared by observing the relation between executor count and throughput, using Proactive Scaling Mode.



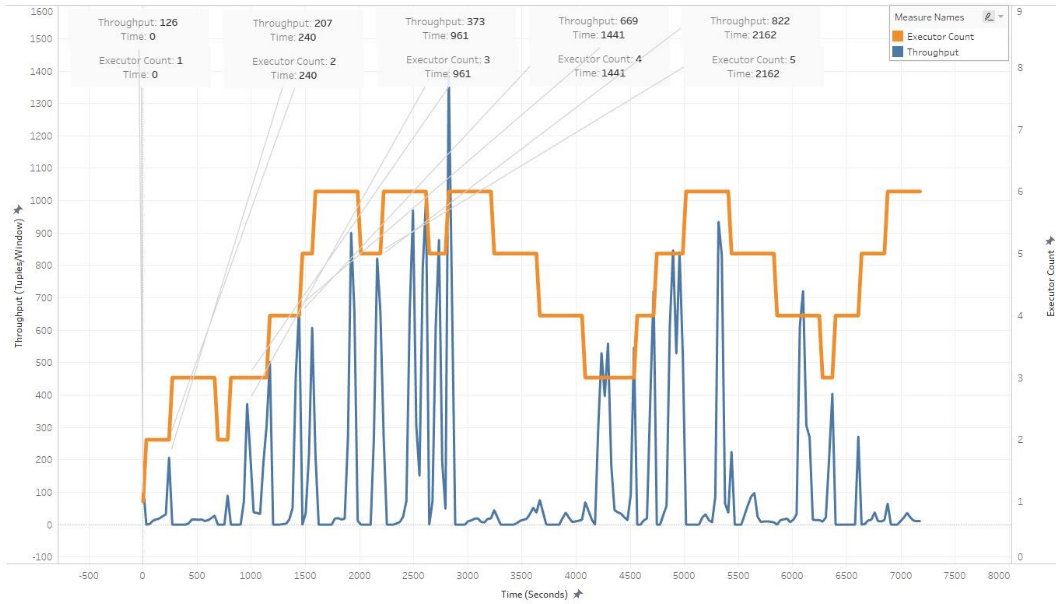
(a) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for varying arrival rate with 1 minutes sleep persist duration in ARSTREAM



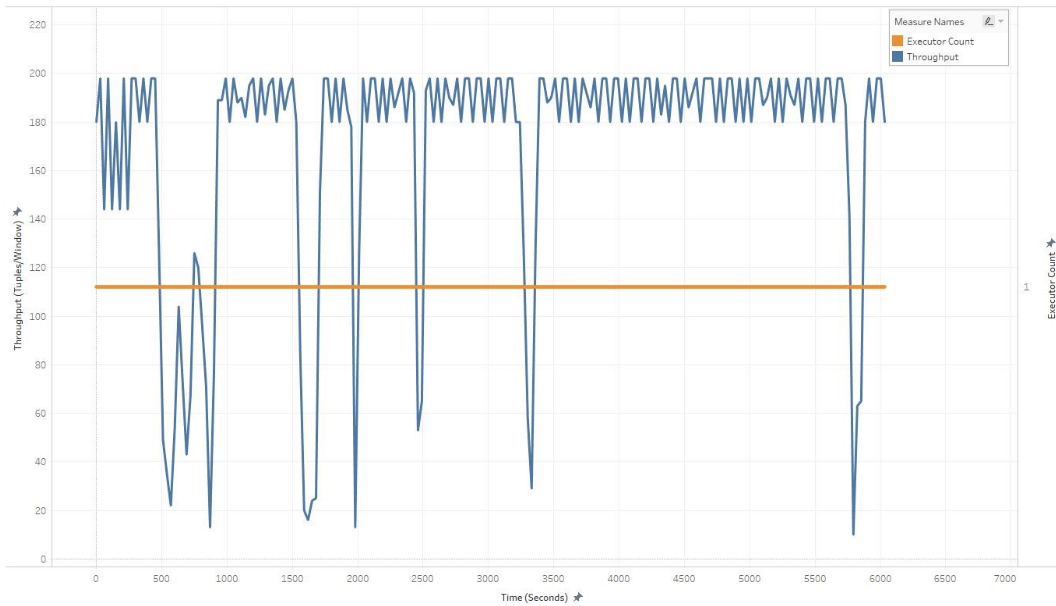
(b) Graph showing the relation between Arrival Rate, Emit Rate, Executor Count and Time for varying arrival rate with 1 minutes sleep persist duration in STORM

Figure 5.7: A comparison between ARSTREAM and STORM executed with varying arrival rate and sleep persist duration of 1 minute, compared by observing the relation between arrival rate, emit rate and executor count, using Proactive Scaling Mode.

rates, and then using that, we trained an artificial neural network. The model, which consisted of 6



(c) Graph showing the relation between Throughput, Executor Count and Time for varying arrival rate with 1 minutes sleep persist duration in ARSTREAM



(d) Graph showing the relation between Throughput, Executor Count and Time for varying arrival rate with 1 minutes sleep persist duration in STORM

Figure 5.7: A comparison between ARSTREAM and STORM executed with varying arrival rate and sleep persist duration of 1 minute, compared by observing the relation between executor count and throughput, using Proactive Scaling Mode.

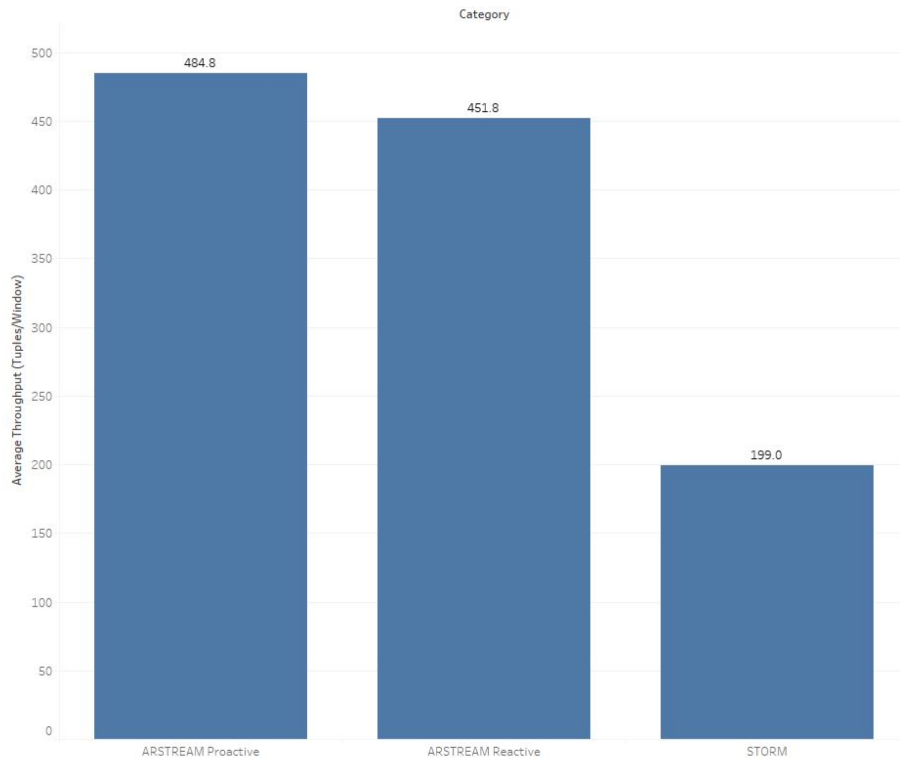


Figure 5.8: Throughput comparison for original STORM (0 re-scaling operations) versus auto-scalable ARSTREAM (average after 7 re-scaling operations) with constant arrival rate.

total layers, performed exceptionally well on the training and testing set; we plotted MSE and R2 for the training and testing results.

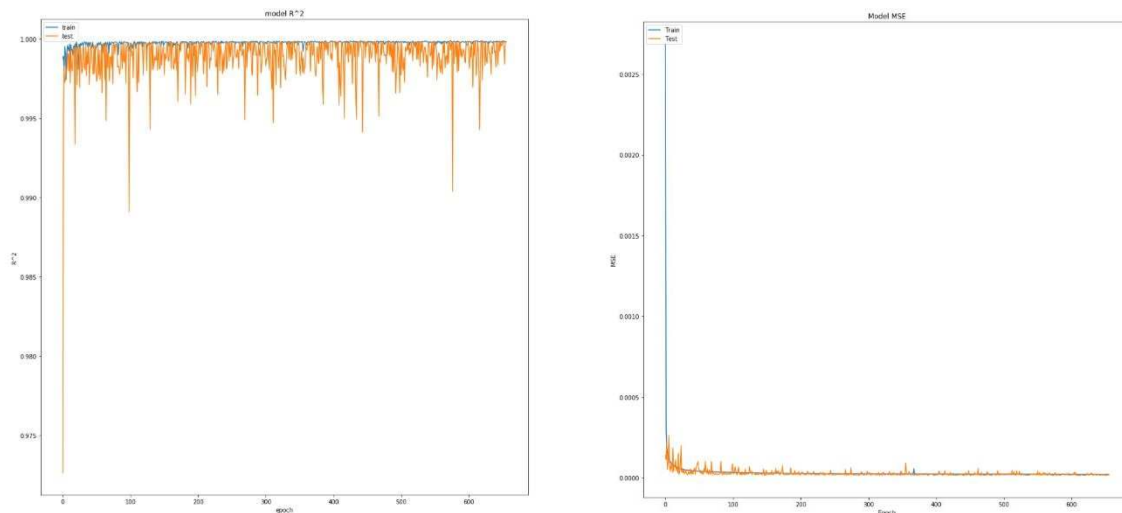
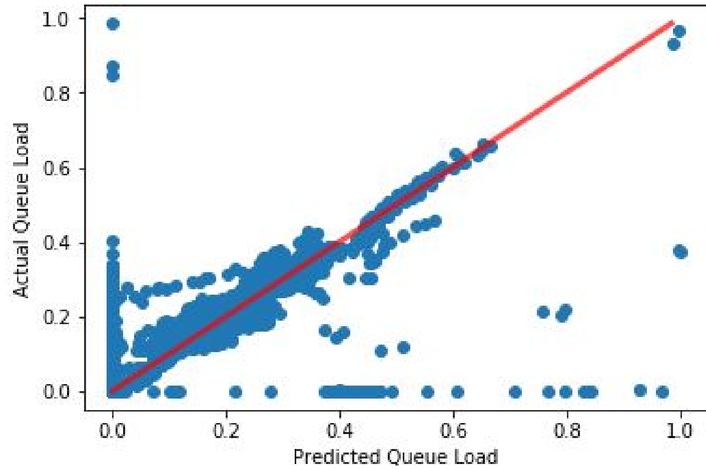


Figure 5.9: Graph showing the result of training the neural network in the form of R2 and MSE graphs

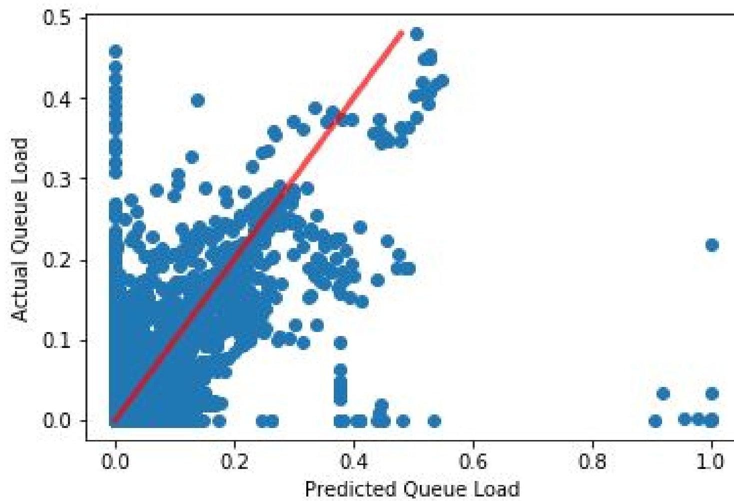
The model performed well on the training and testing data, but the real test would be on the data generated from the real run. We generated the results for all the above cases of proactive mode for variable speed, and we see that, as we reduce the sleep persist duration (i.e., moving closer to the completely random arrival rates which change quickly), the model error increases.
We compare the errors for 5 minutes, and 1-minute sleep persists runs with MAE, MSE, and RMSE values plotted with the graphs between predicted and actual values for each data point and talk about the results.

In Figure 5.10a, the points lie more closer to the line than in Figure 5.10b. The error for 5 minutes sleep persisted run is also lesser than the 1-minute sleep persisted run, which proves our hypotheses that as we move closer to the more random case wherein the arrival rate changes quickly, the proactive model performs poorly. Therefore, for this technique to function correctly, the same arrival rate should persist for some duration, which is also our assumption.




```
Mean absolute error (MAE):      0.007914
Mean squared error (MSE):       0.001845
Root mean squared error (RMSE): 0.042958
```

(a) Graph showing the points plotted for 5 minute sleep persisted run




```
Mean absolute error (MAE):      0.025536
Mean squared error (MSE):       0.004325
Root mean squared error (RMSE): 0.065762
```

(b) Graph showing the points plotted for 1 minute sleep persisted run

Figure 5.10: Comparison of the accuracy measures between 5 minutes sleep persisted run, and 1-minute sleep persisted run by plotting the actual versus predicted points and showing the errors related to each run

Chapter 6

Conclusions and Future Work

This study describes ARSTREAM, which alleviates the bottlenecks from the running application by adapting to the changing workloads. The bottlenecks are predicted in advance using a proactive model that runs in the background of each operator's task to prepare for the future bottleneck by scaling the application [RQ1]. ARSTREAM incrementally provides more resources to an operator in which bottleneck is detected and, therefore, may take time to reach a balanced state. Since ARSTREAM uses STORM's re-balance API under the hood to scale an application, we understand that it introduces data loss in the system, which is also a limitation. But we try to minimize the data loss as much as possible by predicting the data processing queue utilization in advance and even by including a call to an on-demand scaling to deal with any unobserved event and abrupt traffic fluctuations [RQ2]. Moreover, our system is short-sighted in terms of its application. ARSTREAM is designed to work with a federated learning approach to deep learning in which workers migrate the learned weights to the parameter server and, therefore, prevents the loss of states that occurs when all the task instances are killed during a re-scaling operation. These types of applications are computationally expensive and resource-intensive, and thus, we account for both the computational workload and the traffic for making accurate predictions [RQ3]. We have evaluated ARSTREAM and compared its performance with the original STORM engine using the deep learning application mentioned in the text. The results showed that ARSTREAM is effective in increasing throughput by adapting to the arrival rate of the stream in contrast to the STORM, in which throughput stays the same because of spout's throttling. But this increase in throughput is achieved at the cost of introducing data loss in the system.

6.1 Future Works

The approach we used for building ARSTREAM uses STORM's re-balance API internally to scale an application. However, one thing we tend to ignore while using the re-balance API of

STORM is that it induces faults in the system in the form of data loss and loss of the operator states. When we re-scale an application using re-balance API in ARSTREAM, we lose all the data that is present in the queues because all the workers, executors, and tasks are killed and queues are dropped along with the data in it which creates unnecessary faults in the system. Since all the components are killed, we also lose the state that gets attached to the task instance during the execution, which may affect the correctness of some application use-cases. In the future, we would like to update the STORM's re-balance API, which we use in ARSTREAM to make it fault-tolerant such that we don't lose the data that is present in the data processing queues at the time of re-scaling and also to preserve the states. An essential requirement for any DSPE is to be able to orchestrate multiples jobs simultaneously without affecting each other's execution. Since we use STORM as the underlying system, ARSTREAM faces issues while running multiple topologies at the same time, and for the same reason, HERON was developed at Twitter as the replacement of STORM. Since this research focused on just one application use-case and without sensing the need for running multiple applications simultaneously, STORM suited best. But this is a serious issue and must be dealt with to enable the execution of various jobs in parallel and isolation. We can implement or port ARSTREAM's auto-scaling mechanism to a system like HERON, which will allow us to run multiple applications simultaneously without any side effects and enabling proactive scaling to prevent future bottlenecks. We can also think of enabling shared cluster mode in STORM to allow for the reliable and correct execution of multiple jobs simultaneously.

The current version of ARSTREAM is restricted in terms of features and the use-case. The monitoring and scaling are enabled only on one bolt in the topology. But if there are other bolts in the topology which tend to become a bottleneck, we need to allow scaling on them too. In the future, ARSTREAM should be able to scale multiple bolts at the same time in one scaling operation if it observes bottleneck in multiple bolts. We also need to be unrestrictive to allow for any topology structure, which is limited only to the linear topology in the current version. The application of the cool-down period between two re-scaling operations is essential to give topology enough time to get stable and to provide time to the system for spawning all the workers before

a re-scaling operation is called again. We assume that the time which re-scaling action takes is always constant, but it is not correct; therefore, the cool-down period must take into account the time each re-scaling action takes. Currently, there is no way to find out the time we actually need for cooling down. In the future, we can make this process adaptive, which will learn from the behavior and use appropriate wait time between two re-scaling operations in ARSTREAM.

The last and an essential thing in which we can improve is in the proactive scaling technique. The proactive mode uses an artificial neural network model for making predictions, and it requires prior training, which is done by the developer and requires changing model-specific parameters in the ARSTREAM's code for each new training. Due to the above reasons, the system is application-specific dependent on the developer to train the model and modify the source code. We can use an online machine learning technique that will train and predict in real-time, eliminating the need for prior training and developer dependency giving control to the users.

Bibliography

[1] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[2] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on apache spark. *International Journal of Data Science and Analytics*, 1(3-4):145–164, 2016.

[3] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[4] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Albert Bifet. Large-scale learning from data streams with apache samoa. In *Learning from Data Streams in Evolving Environments*, pages 177–207. Springer, 2019.

[5] Apache Software foundation. Apache SAMOA. https://samoa.incubator.apache.org/, December 2019.

[6] Apache Software foundation. Apache STORM. http://storm.apache.org, December 2019.

[7] Wissem Inoubli, Sabeur Aridhi, Haithem Mezni, Mondher Maddouri, and Engelbert Nguifo. A comparative study on streaming frameworks for big data. 2018.

[8] Tarek M Ahmed, Farhana H Zulkernine, and James R Cordy. Proactive auto-scaling of resources for stream processing engines in the cloud. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pages 226–231. IBM Corp., 2016.

[9] Apache Software Foundation. Understanding the parallelism of storm topology. https://storm.apache.org/releases/1.1.2/Understanding-the-parallelism-of-a-Storm-topology.html, December 2019.

[10] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[11] LinkedIn Engineering Contributors. Samza 1.0: Stream Processing at Massive Scale. https://engineering.linkedin.com/blog/2018/11/samza-1-0--stream-processing-at-massive-scale. December 2019.

[12] Apache Software foundation. Samza core concepts. http://samza.apache.org/learn/documentation/latest/core-concepts/core-concepts.html, December 2019.

[13] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[14] thenewstack.io Contributors. Apache samza, linkedin's framework for stream processing. https://thenewstack.io/apache-samza-linkedins-framework-for-stream-processing/, December 2019.

[15] ScottLogic Blog Contributors. Comparing Apache Spark, Storm, Flink and Samza stream processing engines - Part 1. https://blog.scottlogic.com/2018/07/06/comparing-streaming-frameworks-pt1.html. December 2019.

[16] Apache Software Foundation. Samza Containers. https://samza.apache.org/learn/documentation/0.7.0/container/samza-container. December 2019.

[17] Xiaowei Jiang. A Year of Blink at Alibaba: Apache Flink in Large Scale Production. https://www.dataversity.net/year-blink-alibaba, May 2017 (accessed December 2019).

[18] Apache Software Foundation. Re-scale a job in FLINK using RestAPI. https://ci.apache.org/projects/flink/flink-docs-stable/monitoring/rest_api.html#jobs-jobid-rescaling, December 2019.

[19] Nico Kruber. Specifying max parallelism in FLINK for an operator. https://www.ververica.com/blog/4-steps-flink-application-production-ready, December 2019.

[20] Here Contributors. Flink's Best Practices. https://developer.here.com/olp/documentation/pipeline/topics/stream-processing.html, December 2019.

[21] Apache Software Foundation. Parallel Execution in FLINK. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/parallel.html, December 2019.

[22] Apache Software Foundation. FlinkML - Machine Learning for Flink. https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/ml/index.html, December 2019.

[23] Apache Software foundation. Micro second latency barrier in storm. https://storm.apache.org/2019/05/30/storm200-released.html, May 2019 (accessed December 2019).

[24] Medium Website Contributors. Spark Streaming: Dynamic Scaling And Backpressure in Action. https://medium.com/@pmatpadi/spark-streaming-dynamic-scaling-and-backpressure-in-action-6ebdbc782a69, October 2018 (accessed December 2019).

[25] Jan Sipke van der Veen, Bram van der Waaij, Elena Lazovik, Wilco Wijbrandi, and Robert J Meijer. Dynamically scaling apache storm for the analysis of streaming data. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pages 154–161. IEEE, 2015.

[26] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12:2012, 2012.

[27] Ali Yadavar Nikravesh, Samuel A Ajila, and Chung-Horng Lung. Cloud resource auto-scaling system based on hidden markov model (hmm). In *2014 IEEE International Conference on Semantic Computing*, pages 124–127. IEEE, 2014.

[28] Thomas Cooper. Proactive scaling of distributed stream processing work flows using workload modelling: doctoral symposium. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 410–413. ACM, 2016.

[29] Jeff Barr. New predictive-scaling for ec2 powered by machine-learning. https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/, 2018 (accessed December 2019).

[30] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE, 2011.

[31] Demetris Trihinas, George Pallis, and Marios D Dikaiakos. Jcatascopia: Monitoring elastically adaptive applications in the cloud. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 226–235. IEEE, 2014.

[32] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 22–31. IEEE, 2016.

[33] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[34] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[35] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, 2013.

[36] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 783–798, 2018.

[37] Databricks Contributors. Introducing Databricks Optimized Autoscaling on Apache Spark. https://databricks.com/blog/2018/05/02/introducing-databricks-optimized-auto-scaling.html. December 2019.

[38] Apache Software foundation. Dynamic resource allocation in spark. https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation, December 2019.

[39] PhuDuc Nguyen. Auto-Scaling systems with Elastic Spark Streaming. https://www.slideshare.net/SparkSummit/auto-scaling-systems-with-elastic-spark-streaming-spark-summit-east-talk-by-phuduc-nguyen, December 2019.

[40] Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Fox. An overview of the granules runtime for cloud computing. In *2008 IEEE Fourth International Conference on eScience*, pages 412–413. IEEE, 2008.

[41] Shrideep Pallickara, Jaliya Ekanayake, and Geoffrey Fox. Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[42] Thilina Buddhika and Shrideep Pallickara. Neptune: Real time stream processing for internet of things and sensing environments. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1143–1152. IEEE, 2016.

[43] Thilina Buddhika, Ryan Stern, Kira Lindburg, Kathleen Ericson, and Shrideep Pallickara. Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3553–3569, 2017.

[44] Geoffrey Fox, Shrideep Pallickara, and Xi Rao. Towards enabling peer-to-peer grids. *Concurrency and Computation: Practice and Experience*, 17(7-8):1109–1131, 2005.

[45] Ahmet Uyar, Shrideep Pallickara, and Geoffrey C Fox. Towards an architecture for audio/video conferencing in distributed brokering systems. In *Communications in Computing*, pages 17–23, 2003.

[46] Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Hermes: Federating fog and cloud domains to support query evaluations in continuous sensing environments. *IEEE Cloud Computing*, 4(2):54–62, 2017.

[47] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Minerva: proactive disk scheduling for qos in multitier, multitenant cloud environments. *IEEE Internet Computing*, 20(3):19–27, 2016.

[48] Sangmi Lee Pallickara, Shrideep Pallickara, Milija Zupanski, and Stephen Sullivan. Efficient metadata generation to enable interactive data discovery over large-scale scientific data collections. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 573–580. IEEE, 2010.

[49] Matthew Malensek, Walid Budgaga, Ryan Stern, Shrideep Pallickara, and Sangmi Lee Pallickara. Trident: Distributed storage, analysis, and exploration of multidimensional phenomena. *IEEE Transactions on Big Data*, 5(2):252–265, 2018.

[50] Sangmi Lee Pallickara, Shrideep Pallickara, and Milija Zupanski. Towards efficient data search and subsetting of large-scale atmospheric datasets. *Future Generation Computer Systems*, 28(1):112–118, 2012.

[51] Yogesh L Simmhan, Sangmi Lee Pallickara, Nithya N Vijayakumar, and Beth Plale. Data management in dynamic environment-driven computational science. In *Grid-based problem solving environments*, pages 317–333. Springer, 2007.

[52] Thilina Buddhika, Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. Synopsis: A distributed sketch over voluminous spatiotemporal observational streams. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2552–2566, 2017.

[53] Daniel Rammer, Thilina Buddhika, Matthew Malensek, Shrideep Pallickara, and Sangmi Pallickara. Enabling fast exploratory analyses over voluminous spatiotemporal data using analytical engines. *IEEE Transactions on Big Data*, 2019.

[54] Daniel Rammer, Walid Budgaga, Thilina Buddhika, Shrideep Pallickara, and Sangmi Lee Pallickara. Alleviating i/o inefficiencies to enable effective model training over voluminous, high-dimensional datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 468–477. IEEE, 2018.

[55] Walid Budgaga, Matthew Malensek, Sangmi Lee Pallickara, and Shrideep Pallickara. A framework for scalable real-time anomaly detection over voluminous, geospatial data streams. *Concurrency and Computation: Practice and Experience*, 29(12):e4106, 2017.

[56] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Fast, ad hoc query evaluations over multidimensional geospatial datasets. *IEEE Transactions on Cloud Computing*, 5(1):28–42, 2015.

[57] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Analytic queries over geospatial time-series data using distributed hash tables. *IEEE Transactions on Knowledge and Data Engineering*, 28(6):1408–1422, 2016.

[58] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Evaluating geospatial geometry and proximity queries using distributed hash tables. *Computing in Science & Engineering*, 16(4):53–61, 2014.

[59] Saptashwa Mitra, Paahuni Khandelwal, Shrideep Pallickara, and Sangmi Lee Pallickara. Stash: Fast hierarchical aggregation queries for effective visual spatiotemporal explorations. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE, 2019.

[60] Kevin Bruhwiler and Shrideep Pallickara. Aperture: Fast visualizations over spatiotemporal datasets. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 31–40, 2019.

[61] Apache Software Foundation. Spark MLlib for Machine Learning. https://spark.apache.org/docs/latest/ml-guide.html, December 2019.

[62] Nicolas Kourtellis, Gianmarco De Francisci Morales, Albert Bifet, and Arinto Murdopo. Vht: Vertical hoeffding tree. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 915–922. IEEE, 2016.

[63] S Hido. Jubatus: Distributed online machine learning framework for big data. *Proc. of the 1st Extremely Large Databases (XLDB) Asia*, 2012.

[64] Github Project Contributors. StormMOA. http://github.com/vpa1977/stormmoa, May 2013 (accessed December 2019).

[65] Right Scale Contributors. What is Auto Scaling? https://docs.rightscale.com/faq/What_is_auto-scaling.html, December 2019.

[66] Wikipedia Contributors. Autoscaling. https://en.wikipedia.org/wiki/Autoscaling, December 2019.

[67] Apache STORM Release Notes. Automatic Backpressure in Apache STORM. https://storm.apache.org/2016/04/12/storm100-released.html. December 2019.

[68] Navin Kumar Manaswi. Understanding and working with keras. In *Deep Learning with Applications Using Python*, pages 31–43. Springer, 2018.

[69] Md Rezaul Karim. *Java Deep Learning Projects: Implement 10 real-world deep learning applications using Deeplearning4j and open source APIs*. Packt Publishing Ltd, 2018.

[70] Apache Software Foundation. Why is task level parallelism not supported in storm? https://issues.apache.org/jira/browse/STORM-2286, December 2019.

[71] Yann Lecun. The MNIST Database of handwritten digits. http://yann.lecun.com/exdb/mnist/, December 2019.