Priority Based Resource Scheduling Techniques for a Multitenant Stream Processing Platform

By

Rudraneel Chakraborty

A thesis submitted to the Faculty of Graduate and Postdoctoral
Affairs in partial fulfillment of the requirements for the degree of

**Master of Applied Science in
Electrical and Computer Engineering**

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario, Canada

© 2016, Rudraneel Chakraborty

Abstract

Apache Storm is a popular distributed stream processing system which has been widely adopted by the key players in the industry including YAHOO and Twitter. An application running in Storm is called a topology that is characterized by a Directed Acyclic Graph. Isolation Scheduler, the default scheduler for a multitenant Storm platform running multiple topologies assigns resources to topologies based on static resource configuration information and does not provide any means to prioritize topologies based on their business significances. One of the problems with this scheduler is that, performance degradation even complete starvation of topologies is possible on a resource constrained cluster. Two priority based resource scheduling strategies are proposed in this thesis to overcome these problems. A performance analysis based on prototyping and measurements is conducted to demonstrate the effectiveness of the proposed techniques. A comprehensive analysis of the results leading to key insights into system behavior and performance is presented.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Shikharesh Majumdar, for his continuous guidance, support, and encouragement during the process of writing this thesis and over the duration of this program. I sincerely appreciate his invaluable feedbacks and patience during every step of this thesis.

I want to thank my parents, my sister, my aunt and uncle, and Upoma for providing me with unfailing support and continuous encouragement throughout my years of study. This would not have been possible without them.

I would like to thank my friends specially Pavel, Asif, Rabbi, Anshuman Biswas and Gurinderbeer Singh. I want to specially thank Anshuman Biswas and Gurinderbeer Singh from RADS research group for all the engaging conversations and suggestions.

Table of Contents

Abstract.....	ii
Acknowledgements	iii
Table of Contents	iv
List of Tables	vii
List of Figures.....	viii
List of Abbreviations	x
List of Symbols	xi
Chapter 1: Introduction.....	12
1.1 Background.....	12
1.2 Problem Description and Motivation	13
1.3 Proposed Solution.....	14
1.4 Scope of the Thesis.....	15
1.5 Contributions of the Thesis	16
1.6 Thesis Outline.....	16
Chapter 2: Background and Related Work.....	18
2.1 Components of a Distributed Stream Processing Platform	18
2.2 Key Rrequirements of a Distributed Stream Processing System.....	19
2.3 Stream Processing Systems Classifications.....	22
2.3.1 Centralized Stream Processing Systems	22
2.3.2 Distributed Stream Processing Systems	23
2.3.3 Massively Parallel Distributed Stream Processing Systems	24
2.4 Internals of a Distributed Stream Processing System: Apache Storm.....	24
2.4.1 Storm Architecture	25

2.4.2	Storm Stream Processing Model	26
2.5	Scheduling in Distributed Stream Processing Systems	29
2.5.1	Scheduling Techniques for Storm.....	31
2.5.1.1	Offline Schedulers for Storm.....	32
2.5.1.2	Online Schedulers for Storm	36
2.5.2	Scheduling Techniques for Other Distributed Stream Processing Systems	38
2.5.2.1	Static Operator Placement Based Scheduling	39
2.5.2.2	Dynamic Operator Redistribution Based Scheduling.....	41
2.5.3	Other Related Research.....	43
Chapter 3:	Priority Based Scheduling Techniques	44
3.1	Priority Based Scheduling	44
3.2	Static Priority Scheduler (SPS).....	45
3.3	Dynamic Priority Scheduler (DPS)	54
3.4	Implementation.....	60
Chapter 4:	Performance Evaluation of the Scheduling Techniques.....	62
4.1	Use Case-1: Systems Subjected to the Arrival of Batch Data	62
4.1.1	Data Producers	63
4.1.2	Storm Topologies	66
4.1.3	System Configuration.....	70
4.1.4	Performance Metrics	72
4.1.5	Workload Parameters	74
4.1.6	Performance Evaluation of the Priority Based Schedulers	75
4.1.7	Performance Evaluation for a Resource Constrained Storm Cluster	76
4.1.7.1	Effect of Batch Gap (B_G).....	79
4.1.7.2	Effect of Service Time (S_{high} and S_{low})	83
4.1.7.3	Effect of Batch Length (B_L)	86

4.1.7.4	Effect of Event Factor (E_P).....	89
4.1.7.5	Additional Experiments on a Smaller System.....	91
4.1.8	Performance Evaluation for a Resource Unconstrained Storm Cluster	92
4.1.8.1	Effect of Batch Gap (B_G).....	92
4.1.8.2	Effect of Service Time (S_{high} and S_{low}).....	95
4.1.8.3	Effect of Batch Length (B_L)	97
4.1.8.4	Effect of Event Factor (E_P).....	99
4.2	Use Case-2: Systems Subjected to the Arrival of Continuous Data	100
4.2.1	Data Producers	101
4.2.2	Storm Topologies	102
4.2.3	System Configuration.....	104
4.2.4	Performance Metrics	104
4.2.5	Workload Parameters	105
4.2.6	Performance Evaluation of the Priority Based Schedulers	107
4.2.7	Performance Evaluation for a Resource Constrained Storm Cluster	107
4.2.7.1	Effect of High Arrival Rate for Topic Feed1 (λ_{1high}):	111
4.2.7.2	Effect of Proportion of Tuples with High Arrival Rate for Topic Feed1 (r_1):	
	113	
Chapter 5:	Conclusions.....	116
5.1	Summary and Conclusions	116
5.1.1	Use Case 1	117
5.1.2	Use Case 2.....	120
5.2	Future Research	121
References		123

List of Tables

Table 1: Notations for the SPS and the DPS Algorithm.....	48
Table 2: The SPS Algorithm.....	49
Table 3: The Resource Allocator Algorithm	52
Table 4: The DPS Algorithm	56
Table 5: Overview of the Data Producers.....	66
Table 6: Topology Resource Provisioning	70
Table 7: System Configuration for the Clusters	71
Table 8: Topology Resource Provisioning using SPS	77
Table 9: Topology Resource Provisioning using DPS	78
Table 10: Summary of Parameters Used in the Experiments	79
Table 11: Topology Resource Provisioning	104
Table 12: Topology Resource Provisioning using SPS	108
Table 13: Topology Resource Provisioning using DPS	109
Table 14: Summary of Parameters Used in the Experiments	110

List of Figures

Figure 1: Distributed Stream Processing Platform	19
Figure 2: A Storm Cluster.....	25
Figure 3: A Storm Topology.....	29
Figure 4: SPS Architecture	45
Figure 5: DPS Architecture.....	55
Figure 6: Activity Diagram for Runtime Rescheduling of a Topology	59
Figure 7: Kafka Architecture	65
Figure 8: Topology Structure for EventTP1 and EventTP2	68
Figure 9: Topology Structure for ArchivalTP1 and ArchivalTP2	69
Figure 10: Effect of B_G on T_E	81
Figure 11: Effect of B_G on T_T (Higher Priority Topologies)	82
Figure 12: Effect of B_G on T_T (Lower Priority Topologies)	83
Figure 13: Effect of S_{high} on T_E	85
Figure 14: Effect of S_{high} on T_T (Higher Priority Topologies)	85
Figure 15: Effect of S_{low} on T_T (Lower Priority Topologies).....	86
Figure 16: Effect of B_L on T_E	87
Figure 17: Effect of B_L on T_T (Higher Priority Topologies).....	88
Figure 18: Effect of B_L on T_T (Lower Priority Topologies).....	88
Figure 19: Effect of E_P on T_E	89
Figure 20: Effect of E_P on T_T (Higher Priority Topologies)	90
Figure 21: Effect of E_P on T_T (Lower Priority Topologies).....	91
Figure 22: Effect of B_G on T_E	93

Figure 23: Effect of B_G on T_T (Higher Priority Topologies)	94
Figure 24: Effect of B_G on T_T (Lower Priority Topologies)	94
Figure 25: Effect of S_{high} on T_E	95
Figure 26: Effect of S_{high} on T_T (Higher Priority Topologies)	96
Figure 27: Effect of S_{low} on T_T (Lower Priority Topologies)	96
Figure 28: Effect of B_L on T_E	97
Figure 29: Effect of B_L on T_T (Higher Priority Topologies)	98
Figure 30: Effect of B_L on T_T (Lower Priority Topologies)	98
Figure 31: Effect of E_P on T_E	99
Figure 32: Effect of E_P on T_T (Higher Priority Topologies)	100
Figure 33: Effect of E_P on T_T (Lower Priority Topologies)	100
Figure 34: Topology Structure for TP1 and TP2	103
Figure 35: Effect of λ_1^{high} on T_T (Higher Priority Topology)	111
Figure 36: Effect of λ_1^{high} on O	112
Figure 37: Effect of λ_1^{high} on T_T (Lower Priority Topology)	113
Figure 38: Effect of r_l on T_T (Higher Priority Topology)	114
Figure 39: Effect of r_l on O	115
Figure 40: Effect of r_l on T_T (Lower Priority Topology)	115

List of Abbreviations

API	Application Program Interface
BFS	Breadth First Search
CPU	Central Processing Unit
CSV	Comma Separated Values
DAG	Directed Acyclic Graph
DPS	Dynamic Priority Scheduler
EC2	Elastic Compute Cloud
GB	Gigabyte
GHz	Gigahertz
HTTP	Hypertext Transfer Protocol
IOT	Internet of Things
JAR	Java Archive
JVM	Java Virtual Machine
PM	Priority Manager
REST	Representational State Transfer
Sec	Seconds
SOAP	Simple Object Access Protocol
SPS	Static Priority Scheduler

List of Symbols

B_G	Batch Gap (in seconds)
B_L	Batch Length (in number of tuples in a batch)
E_P	Event Factor
S_{high}	Service Time for Higher Priority Topology (in seconds)
S_{low}	Service Time for Lower Priority Topology (in seconds)
T_E	Average Complex Event Inference Latency (in seconds)
T_T	Average Tuple Processing Latency (in seconds)
O	Scheduling Overhead (in seconds)
r_1	Proportion of Tuples with High Arrival Rate for Kafka Topic Feed1
λ_1^{high}	High Arrival Rate for Kafka Topic Feed1 (in tuples/second)
λ_2	Tuple Arrival Rate for Kafka Topic Feed2 (in tuples/second)
N_T	Total Number of Tuples Generated by Each Producer

Chapter 1: Introduction

1.1 Background

With the rise of social media, IOT, wireless sensor networks and intelligent devices such as smart phones, we are experiencing a huge growth in the amount of data generated every single day. IBM reports that, we create 2.5 quintillion bytes of data every day [1]. Extracting meaningful information from this data-sets has become a key point of interest for both industry and academia. Business organizations are employing big data processing platforms to identify new business opportunities and analyze customer behavior which results in higher profit. As an example, in the report presented in [2], SAS notes that, top organizations are using various big data analytics strategies which results in immediate rewards such as reduction of operational cost for an enterprise, faster and better decision making capabilities and more customer focused product and service ideas. Another significant impact of big data can be seen in the healthcare industry. With a data-driven decision making approach, big data analytics in healthcare is being heavily used to predict epidemics, cure disease, improve quality of life and even avoid preventable deaths [3].

The term big data can be characterized by three aspects: Volume, Velocity and Variety [4] where volume refers to a large volume of data, variety refers to largely heterogeneous data and velocity refers to very low latency processing requirements for data. To extract meaningful information from big data, a big data processing platform needs to address one or more of these 3 aspects.

Existing research efforts on batch processing have successfully addressed the volume and variety aspects of big data processing. Frameworks like Hadoop [5] and Spark [6] have emerged which are widely used now a days to process high volume data with varying

structures. More recently, a new class of big data processing platform has gained significant attention. On such a platform, data generated from external sources are continuously pushed to the processing platform with a goal to extract meaningful information in near real time. This class of big data processing is known as distributed stream processing which addresses the velocity aspect of big data processing. Apache Storm [7] is a popular distributed stream processing platform which has been adopted by key players in the industry such as Yahoo, Twitter and Spotify [8].

1.2 Problem Description and Motivation

Although Storm provides a platform for streaming data analytics, effective resource management techniques are required to effectively utilize the resources supported by the underlying Storm stream processing system. Isolation Scheduler, the popular scheduling algorithm used currently with Storm cannot handle a situation when the Storm cluster is resource constrained. In such a cluster, the available number of resources are not sufficient to satisfy the desired resource requirements of the Storm applications submitted to the cluster. A more detailed discussion on this case is presented in Section 2.5.1.1. A cluster may become resource constrained after the submission of new applications or may be explicitly engineered to contain a lower number of resources to achieve high resource utilization. Moreover, Isolation Scheduler does not support application priority. In this thesis, two novel resource management techniques are proposed for Apache Storm to address these issues. The proposed techniques can effectively handle resource constrained Storm clusters. The schedulers use priority based scheduling and provide an effective sharing of the available limited number of resources by the application. A high-level description of the proposed solution is provided next.

1.3 Proposed Solution

Stream processing applications running on Storm clusters are called topologies. Developers package logics for stream processing applications as topologies and submit them to Storm clusters to run. Storm follows a master slave architecture where the master node is called Nimbus and worker nodes are called Supervisors. Submitted Storm topologies are run on the supervisor nodes. A Storm cluster can comprise a number of supervisor nodes where multiple Storm topologies are run [8]. To reduce resource contention among multiple Storm topologies, Storm currently provides Isolation Scheduler [9] which allows developers to specify dedicated number of supervisor nodes for their submitted topologies. The scheduler then schedules the topologies by dedicating the supervisor nodes to topologies per their desired requirements. Although the scheduler prevents resource contention among topologies, as indicated in the previous section, it falls short when the cluster does not have enough supervisor nodes to satisfy the desired number of supervisor node requirements for all the topologies. In such situations, Isolation Scheduler fails to schedule all the submitted Storm topologies which results in partial or complete starvation of some of the topologies that are submitted to the cluster.

This thesis proposes two schedulers for resource constrained Storm clusters where the clusters do not have enough supervisor nodes to satisfy desired number of supervisor node requirements for the submitted Storm topologies. Both the schedulers use priority indications of the topologies and based on the priority indications, proportionally allocates limited resources to the topologies and thus prevents starvation of the topologies. The first scheduler is called Static Priority Scheduler (SPS). The scheduler uses application level priority indications of the topologies and prioritizes resource assignment for topologies

with higher priority when the cluster is resource constrained. The priority indications are provided by the developers when submitting the topologies to the cluster and these priority indications are static and cannot be changed at runtime. SPS can avoid starvation of the submitted topologies provided a minimum number of supervisor nodes are present in the cluster.

The second scheduler is called Dynamic Priority Scheduler (DPS). The scheduler utilizes dynamic priority indications that are determined by the topologies at runtime based on some predefined trigger conditions. The scheduler schedules topologies based on these dynamic priority indications where a topology with higher priority receives more resources. Using this scheduler may result in temporary starvation of the lower priority topologies. The starving topologies cease to starve in time if they also regain their original priority. Both the schedulers are discussed briefly in Section 3.2.1 and Section 3.2.2 respectively.

To evaluate the performance of the proposed schedulers, prototype systems are developed to implement the techniques and associated algorithms. The design and implementation of these systems are described in details in Section 3.4, Section 4.1.1 and Section 4.1.2.

1.4 Scope of the Thesis

A distributed stream processing system is a part of a stream processing platform comprised of a number of data producers, data ingestion layer and the distributed stream processing system. This thesis exclusively focuses on devising effective resource management techniques for a resource constrained distributed stream processing system and resource management for the other components of the stream processing platform is beyond the scope of the thesis. Additionally, scheduling on distributed stream processing systems with

a fixed number of resources is considered in this thesis and auto-scaling of resources for changing the number of resources dynamically based on system load is considered to lie outside the thesis scope.

1.5 Contributions of the Thesis

The main contributions of this thesis are presented next

- *Static Priority Scheduler (SPS)*: A novel starvation free scheduler that uses static priority indications of the topologies to schedule Storm topologies in resource constrained Storm clusters running multiple topologies.
- *Dynamic Priority Scheduler (DPS)*: A novel scheduler that schedules topologies in resource constrained Storm clusters by utilizing dynamic priority indications that are provided by the topologies at runtime.
- Prototype systems are built for the two proposed schedulers and insights into the impact of the various system and workload parameters on system behavior and performance are presented.

The proposed resource management techniques are to be deployed by an entity that is responsible for providing the stream processing services to the users.

A paper [10] based on the initial research results has been published in an international conference: R. Chakraborty and S. Majumdar, "A priority based resource scheduling technique for multitenant Storm clusters," *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2016. pp. 1-6.

1.6 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 provides background information on distributed stream processing system and discusses related work. Chapter 3 describes

the priority based scheduling techniques in detail. Chapter 4 describes the design and implementation of the prototype systems including the topologies used in the various experiments. Performance evaluation of the proposed priority based schedulers based on prototyping and measurement is also included in this chapter. Lastly, Chapter 5 concludes the thesis and discusses possible directions for future research.

Chapter 2: Background and Related Work

This chapter starts by outlining the components of a distributed stream processing platform in Section 2.1. Then, characterization and classifications of distributed stream processing systems are presented in Section 2.2 and 2.3. Next, in Section 2.4 the internals of a distributed stream processing system are discussed in the context of Apache Storm. Finally, the chapter is concluded by outlining existing work focused on various resource management techniques for distributed stream processing systems in Section 2.5.

2.1 Components of a Distributed Stream Processing Platform

Figure 1 illustrates a high level architecture of a distributed stream processing platform based on [11,12,13,14]. Such a platform is composed of multiple layers of abstractions working in concert. An overview of each of the layers is presented.

- *Data Producers*: Data Producer or simply the producer layer is the entry point of data streams to the distributed stream processing platform and can support various data sources including web and social media, mobile and handheld devices, wireless sensor networks etc. The data sources in the producer layer push data to an intermediate ingestion layer where data is temporarily stored before the distributed stream processing system starts processing them.
- *Ingestion Layer*: Ingestion layer typically consists of message brokers such as Apache Kafka [15]. Data pushed by the various data sources in the producer layer is stored temporarily by the ingestion layer. Distributed stream processing system processes these data by pulling them from the ingestion layer.
- *Stream Processing Engine*: A distributed stream processing system like Apache Storm consumes data from the ingestion layer and processes these data.

Figure 1: Distributed Stream Processing Platform

Usually, a distributed stream processing platform is a part of a complete big data processing platform such as lambda architecture [11] and works in conjunction with a batch layer such as Hadoop [5] to accommodate a much broader set of queries over the ingested data.

As this research is only concerned with distributed stream processing, further details on the complete big data processing eco system are not discussed.

2.2 Key Rrequirements of a Distributed Stream Processing System

This section outlines the key requirements for a distributed stream processing system based on the study found in [16]. The distinguishing features of a distributed stream processing system in comparison to traditional data processing systems are captured in the following key attributes:

- **High data movement:** To ensure low processing latency, a distributed stream processing system must assure high data movement by processing data in memory and by reducing blocking operations (e.g. storage operation) and passive processing on data.

- **Instantaneous process and response model for high volume data:** There should be a highly-optimized processing model for distributed stream processing systems which can minimize overhead and maximize the useful utilization of the resources such as CPU, memory. These systems should keep up with high data load and process data in a short period.
- **Ability to integrate stored and processed data:** Distributed stream processing systems should provide means to dispatch processed data to outside storage facilities so that application designers can combine live data and historical data to infer business critical events starting from a past time frame to the current time frame.
- **Data partitioning and scalability:** Distributed stream processing systems should anticipate a large amount of continuous data flow and provide means to process these data with low latency. To achieve this, distributed stream processing systems must support multithreading to take advantage of the modern multi core processors and facilitate the partitioning of the continuous data among multiple cores for processing. The ability to use multiple machines and processors to process a high volume of data in parallel is a key distinctive feature of the different stream processing systems. This feature ensures incremental scalability of the stream processing applications.
- **Predictable and repeatable processing:** Distributed stream processing systems should provide a deterministic and repeatable processing guarantee. That means when supplied with the same input, the processing should result in the same output every time when not influenced by other factors. This attribute is essential for fault tolerance and recovery from failure.

- **Handling of stream imperfection:** In batch style processing, data are always present before processing as the *store then process* model is followed. But in case of stream processing, data might be out of sequence or even be delayed. A distributed stream processing system should handle these types of stream imperfections by introducing strategies such as time outs, storing of data and delayed processing.
- **Stream based query language:** The traditional query language for relational database will not suffice for stream based processing. So, it is expected that a distributed stream processing system should provide means for a stream based query framework with built in stream oriented primitives and operators.
- **Processing guarantee with high availability and fault tolerance support:** Distributed stream processing systems must provide high availability and fault tolerance supports to avoid disruption and preserve integrity of the stream processing applications. There are various schemes available for distributed systems to achieve the desired fault tolerance and high availability which can be used in the distributed stream processing systems as well [17]. Distributed stream processing systems employ a combination of these strategies to provide processing guarantee. For an example: Apache Storm uses a centralized coordination system called Apache Zookeeper [18] to offer automatic failover by launching new instances of the failed stream processing applications on another machine available to the cluster [19].

2.3 Stream Processing Systems Classifications

Based on the extensive summary presented in [20], Stream Processing systems can be subdivided into three main categories: *Centralized*, *Distributed* and *Massively Parallel*. A short overview of these categories is presented in the following subsections.

2.3.1 Centralized Stream Processing Systems

Early systems proposed like AURORA [21], STREAM [22] fall into the category of centralized stream processing systems. AURORA was designed as a single site stream processing system by researchers in Brown University and MIT. Stream processing jobs in AURORA has a Directed Acyclic Graph (DAG) like structure where the nodes of the graph represent operators which represent the processing elements and the edges that represent the movement of the stream elements among the operators. A stream is viewed as an unbounded sequence of data elements called tuples. Tuples arrive at the input queues of stream operators and a scheduler selects which operators to run at a given point in time. Each operator functionality is specified in the AURORA query model. After the processing is done, the output is again moved to the input queue of the next operator in the processing graph. This process keeps on repeating until the output is presented to the sink or end users. Another stream processing system in this category is Stanford Stream Data Manager (STREAM) [22]. Both AURORA and STREAM have the same single host machine based architecture and a DAG based stream processing model. Like AURORA, STREAM also has a query model which decides the functionalities of the operators on the DAG structured stream processing jobs. While both systems have the provision to provide latency requirements for the stream processing jobs, they run on a single host machine and are not scalable to cater to growing processing demands due to added data sources and jobs.

2.3.2 Distributed Stream Processing Systems

Distributed stream processing systems like AURORA/Medusa [23] and Borealis [24] are of the second category. Unlike the centralized stream processing systems discussed earlier, these systems support a distributed processing model that can be comprised of multiple machines. The AURORA/Medusa system was built by the researchers in MIT. This system is an extension to the AURORA stream processing system which is discussed in the previous section. While the AURORA stream processing system runs on a single host machine, an AURORA/Medusa system is comprised of multiple of AURORA nodes that collaborate through an overlay network. Each AURORA node in AURORA/Medusa uses the AURORA query model where the DAG structured jobs are used to process streams. The Medusa system makes it possible for the multiple independent AURORA nodes to be organized as a loosely coupled federation under the control of one or more independent participants. Each participant is a user with financial and administrative access and holds ownership over a set of AURORA nodes, stream processing jobs and data producers feeding streams of data to the jobs.

Another system in this category, Borealis also employs the generic DAG based stream processing model and has a distributed architecture comprised of Borealis servers running on different “sites”. These sites can communicate among themselves and thus can take coordinated actions. Unlike modern distributed systems like Storm [7] or Spark Streaming [6], there is no master node or centralized control node for the Borealis sites. Each of the processing nodes in a Borealis system has an independent admin module which decides if the query should run on the local server or on a participating remote server. Each Processing node executes a single query processor, which runs the actual query fragments.

Processing nodes have a neighborhood optimizer that uses the periodically collected local and remote site statistics to balance the load among the processing nodes.

2.3.3 Massively Parallel Distributed Stream Processing Systems

In the third category, there are massively parallel distributed stream processing systems like Storm and Spark Streaming. Unlike systems in the previous two categories, these massively parallel systems are optimized to run on a cluster with a master slave architecture. These clusters can comprise hundreds or even thousands of machines and thus these frameworks have the capability to attain impressive performance goals. For example, Apache Storm is benchmarked to process a million records per second [7]. These systems provide the developers with flexibility to write fairly complex stream processing applications with the generic DAG based structure and provides high availability and fault tolerance support.

2.4 Internals of a Distributed Stream Processing System: Apache Storm

Storm provides developers with Application Program Interface (API) to write distributed stream processing applications which runs on a Storm cluster. It is to be noted that, Storm is chosen as a representative distributed stream processing system while discussing the core concepts because of two reasons: Firstly, including a plethora of systems available in the literature would make the discussion lengthy. Secondly and most importantly, this thesis is specifically focused on resource management techniques for Apache Storm which is a representative of modern day popular stream processing systems. In the following two sections, overview of the system architecture and the stream processing model of Storm are discussed.

2.4.1 Storm Architecture

Storm adopts to a master slave based cluster architecture. Figure 2 illustrates a Storm cluster with N supervisor node. There are essentially two types of nodes in a Storm cluster:

- **Nimbus Node:** A single machine running a daemon called Nimbus. Nimbus acts as the master node for a Storm cluster. Developers submit Storm job to Nimbus and Nimbus distributes the tasks of the job to the worker nodes with the help of the scheduler. Nimbus also monitors the cluster state and in case of supervisor node failure, it tries to restart the supervisor node or move the processing tasks to other supervisor nodes.
- **Supervisor Node:** There can be multiple supervisor nodes in a Storm cluster with each running a daemon called supervisor. Each supervisor node hosts a predefined number of worker processes. The actual processing of the components of Storm topologies takes place in the worker processes.

Figure 2: A Storm Cluster

Apart from the Nimbus and supervisor nodes, Storm clusters use Apache Zookeeper [18] for synchronization and cluster state management. Multiple nodes, each running the

Zookeeper daemon forms an “ensemble” and provides the Nimbus and supervisor nodes with replicated synchronization and state management service with eventual consistency.

2.4.2 Storm Stream Processing Model

A stream in Storm is modeled as an unbounded sequence of data structures called tuples. A tuple is essentially a named list of values of data of types including string, integer, float etc. Jobs processing these streams are called topologies in the Storm jargon. A topology is comprised of components of two types:

1. *Spouts*: Spouts connect to the external data sources. External data sources include various types of message brokers such as Kafka and ActiveMQ [11]. A spout captures the application logic to connect to these message brokers and pushes streams to the downstream processing components.
2. *Bolts*: Bolts are the processing components of a Storm topology. Application logic for processing of streams is captured in bolts and based on the user defined logic, bolts perform various operations such as filter, aggregate on the tuples of the stream.

To run a topology on a cluster, Storm employs three main abstractions. They are:

- A *Worker Process*, which executes a subset of a topology. Each *supervisor* has a predefined number of slots for *worker processes* and each slot is a Java Virtual Machine (JVM) [25] process where one or more executors of one or more components (*spouts/bolts*) are executed.
- An *Executor* is a thread of execution spawned by a *Worker Process* and runs within the JVM process of the *worker*. Executors run one or more tasks of a specific component. If there are more than one task of a specific component to be run inside

an *executor*, the executor runs them sequentially. Executors are the basic unit of parallelism in Storm jargon.

- A *Task* is a running instance of a Storm component (*Spout/Bolt*). The actual processing is done by the tasks and they run within their parent executors. By default, the number of tasks and number of executors for a given component (spout/ bolt) are the same. One reason for developers require more tasks than executors for a component is to take advantage of the Storm runtime scaling utility called “rebalance” [26]. Using rebalance, Storm allows the system to increase the number of executors and worker processes for a topology but the number of tasks remains the same. Thus, developers can objectively add more tasks than executors if they anticipate to scale out during runtime. However, the number of tasks can never be less than the number of executors.

Developers need to decide on the number of worker processes the application will need to run. It is also the responsibility of the developers to decide the number of concurrently running executors by setting “parallelism hint” parameter for each component. The “parallelism hint” is the number of concurrent executors for each topology component that would run inside the worker processes of the supervisor nodes.

Storm topologies form a directed acyclic graph (DAG) like structure by connecting *Spouts* and *Bolts* by various stream groupings. Developers are provided with Storm APIs to write application logic for spouts and bolts. The nodes in the graph represent components such as spouts, bolts and edges represent stream propagation among the components. Stream groupings decide the communication pattern among the components of a Storm topology by deciding how the stream is partitioned among downstream components.

Storm provides 8 stream groupings. They are:

1. *Shuffle Grouping*: Tuples are randomly distributed among the bolt executor such that each bolt executor receives an equal number of tuples.
2. *Local or Shuffle Grouping*: If the target bolt executor resides on the same worker process, then tuples will be forwarded to that bolt. Otherwise the semantics of *Shuffle Grouping* will apply. In other words, this grouping strategy tries to ensure locality to reduce network communication whenever possible.
3. *Fields Grouping*: Streams are partitioned by the pre-specified fields and tuples with the same value of the field will always go to the same bolt executor. For an example, if a stream is partitioned by a “user-name” field, tuples consisting the same user-name will always go to the same bolt executor whereas tuples having different user name will go to different bolt executors.
4. *Partial Key Grouping*: This is like the *Fields Grouping* but tries to balance the load of tuples between two downstream bolt executors to achieve better resource utilization.
5. *All Grouping*: Tuples are replicated and emitted to all the target bolts executors.
6. *Global Grouping*: This grouping strategy sends all the tuples to a single bolt executor.
7. *None Grouping*: If this grouping policy is selected, then Storm will apply the default strategy which is the *Shuffle Grouping*. Application developers select this grouping if they are oblivious of the grouping of the tuples.
8. *Direct Grouping*: The producer of the tuple decides which bolt executor receives the tuple based on some predefined logic.

Figure 3 illustrates a Storm topology with one *spout* and three *bolts* connected by *Shuffle Grouping*.

Figure 3: A Storm Topology

2.5 Scheduling in Distributed Stream Processing Systems

To process streams of data, stream processing jobs representing continuous queries are run over the data streams. Distributed stream processing systems abstract these queries with a Directed Acyclic Graph (DAG) structure. Vertices on this DAG model represents operators which captures the actual processing logic for the query fragments such as joins, filters and aggregations. The edges on the DAG model represent flow of data among operators. Different distributed stream processing systems have different semantics to design this DAG structured jobs. As an example, recall from previous section that Apache Storm allows developers to design the stream processing jobs called topologies by implementing logics for operators using two abstractions called spouts and bolts. These operator abstractions are collectively known as components. Each of the operators in the DAG model performs some operations on the incoming data and forwards the resulted transformed data to downstream operators or end systems such as a sink database or UI [27]. This DAG abstraction is then mapped to the software components of the distributed stream processing systems. Depending on the design principles of specific distributed

stream processing systems, the operators of the deployed DAG model can be replicated throughout several worker servers to ensure scalable and parallel execution of the stream processing jobs. Recall from Section 2.4.2 that, Storm provides developers to specify the number of concurrent threads for each component (spouts or bolts) of a topology by setting the “parallelism hint” parameter. These running instances are called executors. The executors are run on the worker processes of the supervisor nodes in the cluster.

Scheduling in the distributed stream processing systems refers to the process of mapping the operator instances of the DAG structured stream processing jobs to the worker servers [28]. Each of the worker servers has a limited resource capacity (CPU cores, memory etc.). Developers decide the resource requirement for their stream processing jobs based on the nature of the job (e.g. compute or memory intensive) and number of operator instances. It is the responsibility of the schedulers to make sure that, all the operator instances of all the stream processing jobs are mapped to the available worker servers while respecting their individual resource demands. Failing to do so may result in the following scenarios:

1. Starvation of the stream processing jobs where one or more stream processing jobs never get any resource and fails to process any data.
2. Partial scheduling of the stream processing jobs where some of the operator instances of a stream processing job get mapped to the worker servers whereas some operator instances do not. This situation can hamper the performance and functional correctness of the affected stream processing jobs.

In this section, state of the art scheduling techniques in the context of distributed stream processing systems are briefly discussed. In Section 2.5.1, scheduling techniques for Apache Storm are discussed. In Section 2.5.2, scheduling techniques for distributed stream

processing systems other than Apache Storm are outlined. Finally, some works which are not directly related to scheduling in distributed stream processing systems but have motivated this thesis are briefly discussed in Section 2.5.3.

2.5.1 Scheduling Techniques for Storm

Scheduling in Apache Storm refers to the process of mapping executors of a Storm topology to the worker processes hosted by supervisor nodes [29,30]. As previously discussed, executors are the concurrently running threads of the topology components and worker processes are Java Processes hosted by the supervisor nodes in a Storm cluster. Developers determine the appropriate number of executors for the topology components and the required number of worker processes to run these executors. This information is provided by the developers at design time by setting appropriate configuration parameters. Upon submitting a topology to the cluster, Nimbus invokes the scheduler. The scheduler captures the executors for the topology components and notes the desired number of worker processes for the topology. Next, the scheduler identifies the free worker processes hosted by the supervisor nodes and distributes the executors to the desired number of worker processes.

Available literature on Storm based scheduling techniques can be broadly classified in to two categories:

- *Offline Scheduling*: This type of schedulers is used to formulate an initial schedule. They are invoked once for every Storm topology submitted to the Storm cluster. This type of schedulers does not adapt to the changes in the system environment which means once a Storm topology is scheduled after the submission, a new schedule is never computed for that Storm topology.

- *Online Scheduling*: These schedulers are always live/active. This type of schedulers monitors the quality of the current schedule by periodically assessing the collected system/application level statistics and tries to formulate a new schedule if necessary. Thus, it is possible for a submitted Storm topology to be rescheduled when using an online scheduler.

2.5.1.1 Offline Schedulers for Storm

Apache Storm provides two out of the box schedulers. They are called *Isolation Scheduler* and *Default Scheduler*. Both schedulers are *offline* schedulers so they cannot adapt to the changes in the cluster environment and they are invoked once per topology submission.

Default Scheduler uses a simple round robin mechanism and produces a schedule where each worker process hosts approximately an equal number of executor of the components of a submitted topology. The scheduling is a two-stage process where in the first stage, the scheduler notes the desired number of worker processes by the topology and captures all the executors of the topology components. The scheduler then partitions the executors into multiple sets. The number of sets is equal to the desired number of worker processes and each set holds approximately an equal number of executors. In the second stage, the scheduler identifies the desired number of free worker processes from the available supervisor nodes and allocates each set of the executors to one of the worker processes. The allocation is done in a round robin fashion where each supervisor node contributes approximately an equal number of worker processes. Although this scheduler ensures fairness by making sure the number of executors mapped to each worker process remains approximately equal and each supervisor node contributes approximately equal number of worker processes, it is possible for multiple topologies to be scheduled in the worker

processes of the same supervisor node which results in resource contention among the topologies. This results in performance degradation of the topologies. Thus, this scheduler is not suitable for a multitenant Storm cluster where a Storm cluster is shared by many Storm topologies.

To solve the multitenancy problem, Storm introduced *Isolation Scheduler* which solves the multitenancy problem by providing dedicated supervisor nodes for each topology running in the cluster. Developers indicate the desired number of supervisor nodes for the topologies in the Storm configuration file. After a topology is submitted to the cluster, the scheduler reads the configuration file and allocates the desired number of supervisor nodes to the topology. The desired number of worker processes are selected from these dedicated supervisor nodes and the executor to worker process mapping follows the same round robin strategy as the *Default Scheduler*. Essentially, the supervisor nodes are “isolated” so that no two topologies are run on the same supervisor node and thus resource contention among topologies are eliminated.

Both *Isolation Scheduler* and *Default Scheduler* can result in partial scheduling or even complete starvation of the topologies in a resource constrained Storm cluster. A resource constrained Storm cluster does not have enough supervisor nodes to satisfy the requirement of all the topologies. Following are few scenarios where using *Isolation Scheduler* or *Default Scheduler* will result in partial scheduling or complete starvation of the Storm topologies:

- Topologies are submitted to the Storm cluster separately at different points in time and the cluster allocates resources to the topologies in a First in First out (FIFO) order. If topologies that are already submitted have consumed all the supervisor nodes on

the cluster and there are not enough supervisor nodes on the resource pool, it is possible for newly submitted topologies to starve or remain partially scheduled until more resources are added to cluster.

- It is possible for supervisor nodes to become unavailable at runtime due to machine failure/network failure. Even in the presence of high availability features, it takes some time to make new supervisor nodes available to the cluster. In this interim period, topologies running on those supervisor nodes can suffer from partial scheduling or even complete starvation.
- Lastly, a Storm cluster can face resource constraint due to scaling out of the topologies that require adding new resources to the respective topologies. Storm provides a feature called rebalance [26] through which it is possible for runtime scaling out of the topologies. In the research presented in [31], it is shown that it takes a significant amount of time (on the order of 6 minutes) for making new supervisor nodes available to the cluster. A Storm cluster may become resource constrained and a scaled-out topology may suffer from performance degradation during this period.

In all these resource constrained systems, the full functionality of all the submitted topologies cannot be supported at the same time. As a result, Isolation Scheduler cannot be used in such resource constrained systems.

Other than the default schedulers, a few offline schedulers are available in the literature. They are discussed next.

In the offline scheduler proposed in [28], authors modeled the executor placement problem in the context of Apache Storm as an Integer Linear Programming problem while considering heterogeneity of compute and network resources. The authors proposed a

generic framework to include various user indicated quality of service metrics to be optimized while allocating resources to the executors in a Storm application. A few observations can be made in the context of [28]. First, this solution does not provide satisfactory performance when the number of executors in a topology and the size of the cluster are large. Secondly, the scheduler is an offline scheduler, and unlike an online scheduler, it is oblivious to run time changes both at the system and the application levels. The offline scheduler presented in [30] places executors of a Storm topology on the same node based on their stream grouping. This approach ensures lower network communication. The scheduler is run when a Storm topology is submitted to the cluster. One aspect of the scheduler is that it does not consider system load as a factor which the authors themselves identify as a drawback.

A similar technique is proposed in [30] where the offline scheduler analyzes the DAG structure of the Storm topology and tries to formulate a schedule where the heavily communicating executors are placed on the same node and thus minimizes network latency.

A resource aware offline scheduler for Apache Storm is proposed in [32]. This scheduler considers both demands and availability of resources in terms of CPU, memory and network bandwidth in a Storm cluster. To achieve this, the users are provided with APIs to indicate CPU, memory and network bandwidth requirements of a Storm topology when submitting the topology to the cluster. Similarly, each worker node in the cluster has its resource capacity statically configured in a configuration file. The statistics about resource availability and resource consumption are collected and stored periodically in the *GlobalState* module and the resource aware scheduler utilizes these statistics while

allocating resources to a submitted Storm topology. The scheduling process has two sequential steps. The first step obtains a partial order of the executors to be scheduled using the popular Breadth First Search (BFS) algorithm. Use of *BFS* ensures that adjacent executors are placed in close sequence so that heavily communicating executors can be mapped to the worker processes of the same supervisor node and thus network communication latency is reduced. The next step focuses on selecting a supervisor node to host executors such that the selected node has the most available resources to fulfill resource requirement of the executor. While the scheduler investigates the feasibility of running multiple Storm topologies, it does not consider a resource constrained Storm cluster. Thus, partial scheduling and starvation of topologies remain a possibility when the cluster is resource constrained.

2.5.1.2 Online Schedulers for Storm

In the research presented in [33], the authors argue that in the case in which the system is experiencing lighter workload, significant performance improvement in terms of average tuple processing latency can be achieved by using fewer supervisor nodes as the inter supervisor node communication latency is more expensive than the inter worker process communication latency for lighter workloads. This technique falls short, however, when the system is experiencing higher traffic and the nodes become heavily overloaded. Based on these observations, a periodically running online scheduler is proposed. The proposed scheduler tries to formulate a schedule where the executors are mapped to the worker processes fewer supervisor nodes at the same time makes sure, no supervisor node has exceeded its CPU capacity. The capacity is predefined for every supervisor node. CPU load due to tuple arrival or traffic across executors is collected periodically by daemons which

are deployed throughout the supervisor nodes in the cluster. Using this information, the scheduler can determine if a participating worker node is approaching its CPU capacity.

A prediction based scheduling framework for Apache Storm is proposed in [34]. There are essentially two aspects of the proposed framework: a prediction component which models and predicts average tuple processing latency, and a scheduler which assigns executors to worker nodes based on the prediction model. Authors model average tuple processing latency as the summation of average tuple processing latency at each executor and average tuple transfer latency between the executors which includes both queuing delay and network communication latency. A popular supervised learning algorithm called Support Vector Regression is used for the prediction of average tuple processing latency. The prediction is used as a guideline to take intelligent decision while mapping executors to the worker processes of the supervisor nodes. To facilitate the learning, statistics on average tuple processing latency and tuple transfer latency are collected by daemons implanted across supervisor nodes which in turn send these statistics to a centralized location. Based on this collected data, a prediction model is built. Finally, a greedy scheduling algorithm tries to find a feasible schedule using the prediction model with a goal of reducing average tuple processing latency.

A graph partition based scheduler for Apache Storm is presented in [35]. The goal of this research is to minimize network load due to data movement among the nodes in a Storm cluster while making sure there is no load imbalance in the nodes in terms of compute capacity. A weighted graph called communication graph of an application DAG is constructed where the weights on the nodes represent the compute capacity requirements of the tasks and weights on the edges represent the accumulated total number of all the

tuples that are sent from one task to the other. Based on the periodically collected application performance statistics from the cluster, this communication graph is optimized with an objective to minimize network communication cost due to inter node communication while keeping the load imbalance in terms of computational load under a certain given limit.

While the above proposed offline and online schedulers aim at maximizing a given performance metric such as average tuple processing latency by formulating intelligent executor to worker process mapping, none of them considers a resource constrained environment where the available number of resources are not adequate to satisfy the resource requirements of the all the submitted topologies. Moreover, resource contention due to multiple running topologies is also not considered. The proposed priority based schedulers in this thesis aim at formulating a schedule addressing all these issues by utilizing application level priority in a static and dynamic manner.

2.5.2 Scheduling Techniques for Other Distributed Stream Processing Systems

Scheduling techniques for other distributed stream processing systems available in the literature can be broadly categorized in two ways

- **Static Operator Placement based scheduling:** Upon submission of a DAG structured stream processing job, an initial one time operator to worker mapping is formulated by this type of schedulers. This type of schedulers does not adapt to run time changes and thus stream processing jobs are never rescheduled.
- **Dynamic Operator Redistribution based Scheduling:** This type of schedulers utilizes periodically collected performance and system statistics to assess the quality of the current schedule and if required, can formulate a new schedule by

voiding the current one. It is to be noted that, there is an overhead associated with the dynamic redistribution of the operators.

The available researches provide varied opinion while evaluating both the techniques. Some are in favor of using dynamic operator redistribution based techniques citing long term benefits while the others find the overhead to be a significant factor impacting the performance and advocate static operator placement based scheduling.

2.5.2.1 Static Operator Placement Based Scheduling

In the research presented in [36], worker nodes hosting the operators of a stream processing job in a Borealis stream processing system experience loads which can be formulated as linear functions of system input stream rates. Authors argue that, dynamic operator redistribution in the presence of short term load variation is associated with significant overhead making it completely infeasible for and for some distributed stream processing systems. To overcome this problem, authors propose a static operator distribution algorithm, which aims to formulate a schedule such that operator to worker node mappings are "*resilient*" to unpredictable load variations. The algorithm is a coherent combination of two heuristics where the first heuristic aims at balancing the load of each input stream for all the worker nodes while the second heuristic tries to avoid bottlenecks by looking at the effects of different input streams on each of the nodes and thus identifying overloaded nodes which can be a bottleneck. The algorithm presented also consider communication costs and knowledge of specific workload characteristics such as a predefined lower bound on the input stream rate.

In [37], authors proposed a static operator placement based scheduler for System S developed by IBM. This system uses admission control which means a submitted job can

be rejected if the scheduler deems necessary which is rarely seen in any other distributed stream processing system. The periodically running scheduling process is decomposed into four sequential steps where each step is a mathematical optimization problem. The first step decides the jobs to be admitted, parallelism of the operators in the job and the CPU capacity constraints for each job. In the next step, the worker nodes for the admitted jobs are selected. The third step updates the processing capacity constraints with respect to the selected worker nodes in the second step. The fourth and final step decides the allocation of the job operators to the worker nodes.

Finally, the static priority based scheduler proposed in [27] for the Quasit stream processing system [38] is probably the closest research effort to the static priority scheduler proposed in this thesis. Like the static priority scheduler, the framework utilizes user determined static priority indications for scheduling jobs submitted to the system. Although, it is important to note that, while the static priority scheduler proposed in this thesis allows priority indication for a job, [27] uses priority indications for operators for a specific job. This priority indications are expressed using "priority schemas" of two types: 1) *Absolute priority* where an operator is assigned a priority indication in such a way that, a higher priority operator is sure to get resources even at the expense of starvation of a lower priority operator. 2) *Proportional priority* where operators are assigned with weights depending on their significance. Allocated resources are proportionally shared among these operators using a proportional share algorithm. This approach is like our proposed scheduler. The key differences between the schedulers proposed in this thesis and the scheduler described in [27] are summarized. Firstly, the scheduler in [27] uses static priority indications on operator level while our proposed static priority scheduler uses

priority indications on job level. Secondly, unlike our proposed schedulers, there is no provision of specifying minimum resource requirement for a job in [27]. The significance of minimum resource requirement will be detailed in Chapter 3. Thirdly, in addition to the static priority scheduler, a dynamic priority scheduler is investigated in this thesis whereas research presented in [27] only uses static priority indications for scheduling.

2.5.2.2 Dynamic Operator Redistribution Based Scheduling

In the research presented in [39], authors propose a scheduler based on load distribution for the Borealis Stream Processing system. The objective of the greedy algorithm presented in this paper is to reduce load variation across worker nodes of a Borealis cluster. Authors considered a non-linear load model where the input stream rates are unpredictable and have transient variations. Authors show that for such a system, average variance and average correlation of the measured load across the worker nodes are inversely related and minimizing the later helps maximizing the former and vice versa. The proposed scheduler tries to formulate a schedule following this objective which translates to better load distribution across worker nodes and improved performance of the stream processing jobs. System load is measured as CPU utilization and load across worker nodes are collected periodically as time series data. Correlation of average load between any two worker nodes or operators can be calculated by calculating a correlation coefficient (a real number between -1 and +1) of their respective time series data. This concept of correlation facilitates the scheduling in two ways: Firstly, if the load correlation between two operators is small, then they are placed on the same worker node which results in less variance of load and secondly, to reduce frequent operator migration and thus overhead, a *pair-wise load redistribution algorithm* is periodically run with an objective to maximize average

node load correlation between two adjacent worker node pairs.

A simulation based researching the context of Borealis [24] is proposed in [40]. Authors proposed a scheduling technique which focuses on minimizing network communication cost in a simulated distributed stream processing environment by utilizing the knowledge of the underlying network. The scheduler decides placement of operator based on two heuristic approaches. In the first approach, the pair wise inter-worker node communication latency is considered and a cost model is constructed. Using the second heuristic, the operator placement utilizes the cost model developed using the first heuristic. It is to be noted that this approach does not consider operator processing cost as well as queuing delay. Also unlike researches in [39] [36] [41] load due to continuous flow of streams in the system and required load balancing is not taken into account.

On the other hand, in [41], authors presented another simulation based research which is not tied to any specific distributed stream processing Platform. Authors strongly emphasize on the importance of run time dynamic operator re-distribution even if it incurs a short-term overhead. Authors used 3 heuristic based approaches where the first heuristic is used to balance the load across worker nodes. The second heuristic aims at limiting the number of worker nodes for the distribution of the operators of a query and finally, the third heuristic tries to minimize the total communication cost by employing a data-flow aware query migration strategy which tries to ensure locality by keeping frequently communicating operators on the same node whenever possible. It is to be noted that, contrary to the research presented earlier in this section, the authors did not introduce any new load balancing strategy and used the existing well known receiver initiated load balancing strategy described in [42]. Load distribution is done at the query fragment level

(which contains the subset of operators) whereas the research presented in [39] and [36] are concerned with operator level load migration.

2.5.3 Other Related Research

Although not directly related to distribute stream processing, the following works motivated this thesis and require discussion.

Proportional share based resource scheduling is briefly discussed in operating system literature [43]. The idea of using priority based proportional share scheduling for distributed data processing platforms is used frequently outside the stream processing architecture. For an example, a priority based proportional share scheduler is proposed in [44] for Hadoop. Users who are willing to pay a higher price gains higher priority and a higher capacity in terms of higher number of map and reduce slots. The pricing policy is determined by the overall resource demand in the Hadoop cluster. Resources are proportionally shared among the users based on the capacity they are allocated.

As pointed out, while all the schedulers discussed in this chapter try to maximize/minimize a certain performance metric (e.g. average tuple processing latency) none of them addresses the problem of scheduling multiple Storm topologies in a resource constrained Storm cluster. To the best of our knowledge, no other work has addressed this issue so far. In the next chapter (Chapter 3) the priority based schedulers proposed in this thesis are discussed where it is shown that by introducing priority based scheduling, it is possible to overcome the partial scheduling and starvation problem of the topologies for a resource constrained Storm cluster running multiple Storm topologies.

Chapter 3: Priority Based Scheduling Techniques

This chapter discusses the two-proposed priority based schedulers for Apache Storm. Section 3.1 provides a short overview of the two priority based schedulers. Section 3.2 describes the algorithm and other internal details for the Static Priority Scheduler (SPS). The second scheduler called Dynamic Priority Scheduler (DPS) is discussed in Section 3.3. Finally, Section 3.4 discusses the implementation details for the proposed schedulers.

3.1 Priority Based Scheduling

Two priority based scheduling techniques are proposed in this thesis for multitenant resource constrained Storm clusters. The first scheduler is called Static Priority Scheduler (SPS) and it uses static priority of the topologies based on their business significance as indicated by the respective topology developers. Apart from priority indications, developers also need to indicate a minimum number of supervisor node requirements for the topologies which SPS uses to prevent starvation of the topologies. SPS is agnostic to the run time priority changes.

The second scheduler is called Dynamic Priority Scheduler (DPS). Rather than using user indicated static priority, DPS uses topology indicated runtime priority indications and formulates the schedule in an *online* fashion. To use DPS, developers need to write application logic for the topologies to send priority indications based on some predetermined trigger conditions.

Storm allows implementation of custom schedulers through its *IScheduler* [51] Java Interface where a single method called *schedule* needs to be implemented. The scheduler is packaged as a *.jar* file and placed in the library folder of the Nimbus node. The Storm configuration file is also updated to instruct the Nimbus to use the custom scheduler. It is

also to be noted that Nimbus invokes scheduler periodically with a default period of 10 seconds. Further discussion for each of the schedulers is provided next.

3.2 Static Priority Scheduler (SPS)

SPS utilizes static priority indication to schedule topologies in a resource constrained Storm cluster. This priority indications are provided by application developers before submitting the topology to the Storm cluster and cannot be altered at runtime. The SPS architecture has two components. The first one is the SPS scheduler itself and the second one is the Priority Manager (PM). Before submitting topologies, application developers need to provide PM with the priority indications and minimum supervisor node requirement of the topologies which PM stores in a database. SPS is placed on the Nimbus node and Nimbus is instructed to use SPS through appropriately setting the Storm configuration parameters. Figure 4 illustrates the SPS architecture.

Figure 4: SPS Architecture

SPS schedules the topologies in a resource constrained Storm cluster by proportionally sharing available supervisor nodes among topologies based on their priority indications. A lower numeric priority value indicates higher priority. It is possible for multiple topologies to be assigned the same priority level. Topologies in a higher level get more resources than the topologies in a lower priority level.

In order to schedule topologies in a resource constrained Storm cluster, SPS needs to know four important parameters for every submitted topology. They are:

1. *Desired Number of Supervisor Nodes*: As discussed in Chapter 2, the default Isolation Scheduler requires the developers to provide the number of desired supervisor nodes for every submitted topology in the Storm configuration file. SPS provides the developers with the same provision and uses this information while scheduling topologies.
2. *Desired Number of Worker Processes*: In line with the Isolation Scheduler, this information is also provided by the developers at the topology submission time and SPS uses this information when scheduling topologies.
3. *Minimum Number of Supervisor Nodes*: This parameter is introduced for SPS and captures the minimum number of supervisor nodes needed for a topology to run. Developers provide this information in the Storm configuration file and SPS uses it while scheduling topologies.
4. *Priority*: As noted earlier, developers need to specify the priority of each of the submitted topologies. It is up to the developers to decide the priority of a topology which depends on the business significance of the topology. As an example, a

Storm topology used for climate monitoring in a data center can have a higher priority than a topology used for data archival.

SPS can prevent starvation of the topologies if the cluster can satisfy the minimum supervisor node requirement of all the topologies. It is to be noted that the notion of minimum number of supervisor nodes is different from the desired number of supervisor nodes. Developers decide on the desired number of supervisor nodes needed by analyzing the nature of the topology, the number of executors for each component and the hardware specification of the participating supervisor nodes in the cluster. As an example, consider a topology with 4 components (spout/bolt) with each component having 4 executors. Additionally, consider each of the supervisor nodes in the cluster to have dual core CPU and two worker processes. To ensure the 4 executors of the topology components can run in parallel, it is natural to select 4 supervisor nodes as the desired number of supervisor nodes for the topology. Doing so ensures that one worker process in each supervisor node hosts one executor of each component and executors are not competing for resources (e.g. CPU). On the other hand, for the same topology, a minimum of 2 supervisor nodes will be needed for the executors of the topologies to run. This is because, for this example, each supervisor node has 2 CPU cores and thus 2 worker processes each. The topology has 4 components and 4 executors for each component. In order to make sure that 4 executors can run at least 4 CPU cores need to be allocated to the topology. Thus 2 supervisor nodes are selected as the minimum number of supervisor nodes for the topology.

The complete SPS algorithm is presented in Table 2. Notations used to formulate the algorithm are described in Table 1.

Table 1: Notations for the SPS and the DPS Algorithm

Symbol	Meaning
$PR = \{1, 2, 3 \dots p\}$	Set of distinct priority levels; p is the total number of priority levels available. A lower numeric value indicates higher priority
$l_j (j=1, 2 \dots p)$	Total number of topologies in j^{th} priority level.
$TP_j = \{tp_{ij}\} (j=1, 2 \dots p; i=1, 2 \dots l_j)$	Set of all the topologies in the j^{th} priority level; tp_{ij} is the i^{th} topology of the j^{th} priority level. There are $\sum_{j=1}^p l_j$ topologies submitted to the cluster
$SD = \{sd_{ij}\} (j=1, 2 \dots p; i=1, 2 \dots l_j)$	Set of desired number of supervisor nodes for the i^{th} topology in the j^{th} priority level
C	Total number of supervisor nodes available to the cluster
$W_{ij} (j=1, 2 \dots p; i=1, 2 \dots l_j)$	Total number of worker slot desired by the i^{th} topology of the j^{th} priority level
$CA = \{ca_{ij}\} (j=1, 2 \dots p; i=1, 2 \dots l_j)$	Set of already allocated number of supervisor nodes for each topology; ca_{ij} is the number of supervisor nodes already allocated to the i^{th} topology of the j^{th} priority level. A newly submitted topology has a default value 0.
$CN = \{cn_{ij}\} (j=1, 2 \dots p; i=1, 2 \dots l_j)$	Set of newly computed number of supervisor nodes for each topology; cn_{ij} is the number of supervisor nodes computed to be allocated to the i^{th} topology of the j^{th} priority level
$SM = \{sm_{ij}\} (j=1, 2 \dots p; i=1, 2 \dots l_j)$	Set of minimum number of supervisor node requirement of the topologies; sm_{ij} is the minimum number of supervisor node required by the i^{th} topology of the j^{th} priority level
$E_{ij} (j=1, 2 \dots p; i=1, 2 \dots l_j)$	Set of executors for the i^{th} topology of the j^{th} priority level
ST	Total number of supervisor node desired by all the topologies submitted
$SP_j (j=1, 2 \dots p)$	Total number of supervisor node required by all the topologies in the j^{th} priority level
PF_j	A number representing the maximum of the supervisor nodes that can be allotted to the topologies of the j^{th} priority level

Table 2: The SPS Algorithm

```

1      Get C from the cluster
2      
$$\text{if } C == \sum_{j=1}^p \sum_{i=1}^{l_j} sm_{ij}$$

3          for each  $j \in PR$ 
4              for each  $tp_{ij} \in TP_j$ 
5                   $cn_{ij} = sm_{ij}$ 
6                  add  $cn_{ij}$  to set CN
7              end for
8          end for
9      else
10          $ST = \sum_{j=1}^p \sum_{i=1}^{l_j} sd_{ij}$ 
11          $X = \min(C, ST)$ 
12         for each  $j \in PR$ 
13              $SP_j = \sum_{i=1}^{l_j} sd_{ij}$ 
14              $PF_j = \min(X, \text{round}(C * (\frac{SP_j}{ST})))$ 
15              $Y = PF_j$ 
16             for each  $tp_{ij} \in TP_j$ 
17                  $cn_{ij} = \min(Y, \text{round}(PF_j * (\frac{sd_{ij}}{SP_j})))$ 
18                 add  $cn_{ij}$  to set CN
19                  $Y = (Y - cn_{ij})$ 
20             end for
21              $X = (X - PF_j)$ 
22         end for
23     end else
24     call ResourceAllocator()

```

The SPS algorithm presented in Table 2 starts by querying the cluster for the total number of available supervisor nodes C by invoking Storm library functions [line 1]. If the cluster has only the minimum number of supervisor nodes to satisfy the cumulative minimum supervisor node requirements of the topologies [line 2], SPS allocates each topology with their minimum number of supervisor node requirements and adds the number of supervisor nodes to be allocated to the set of newly computed number of supervisor nodes CN [Lines 5,6].

If the total number of supervisor nodes in the cluster is more than the minimum number of supervisor node required by all the topologies, SPS schedules topologies by proportionally allocating the available supervisor nodes. To accomplish that, the total number of desired supervisor nodes required by all the topologies ST is calculated by adding the individual supervisor node requirements of the topologies [line 10]. It is possible for the cluster to have more supervisor nodes than ST . In such a situation, the extra nodes should not be considered. Thus, the minimum of the C and ST values is chosen [Line 11] to discard extra supervisor nodes.

As discussed earlier, multiple topologies can have the same priority level. Priority levels are captured in set PR and are sorted in ascending order where a lower numeric value indicates a higher priority. For the j^{th} priority level in PR , the total number of supervisor nodes required by the topologies with the given priority level is calculated as SP_j [Line 13]. Next using the equation in line 14, the maximum number of supervisor nodes that can be allocated to the topologies in this priority level is calculated as PF_j . Finally, using the equation in line 17, the number of supervisor nodes to be allocated to each of the topologies in the priority level is computed as cn_{ij} and added to the set of newly computed number of

supervisor nodes CN . The computed cn_{ij} value is subtracted from the number of supervisor nodes allocated to the topologies of the priority level so that the allocated supervisor nodes can be discarded from the subsequent calculations [Line 19]. The steps contained by lines 13 to 21 are repeated for every priority level in PR and when the number of supervisor nodes to be allocated for each of the submitted topologies is calculated and added to CN , the *ResourceAllocator* function is called with CN as an input parameter.

It is to be noted that, SPS computes proportional share of the supervisor nodes in two steps. First, the proportional share of supervisor nodes is calculated for the individual priority levels considering the collective desired number of supervisor nodes of the topologies belonging to each of these priority levels (Line 14). This is necessary to prevent starvation of the topologies with the lower priorities as it ensures that topologies in a higher priority level is not allocated with all the available supervisor nodes by starving the lower priority topologies. As SPS does not allow dynamic priority change, starvation of these topologies can continue indefinitely. In the second step, the computed number of supervisor nodes for a priority level are proportionally distributed among the topologies with the given priority level based on their individual desired number of supervisor nodes (Line 17). As it is possible for topologies with the same priority level to have varying number of desired supervisor nodes, this strategy ensures that individual topologies get proportional share of supervisor nodes with respect to other topologies belonging to the same priority level. Both *SPS* and *DPS* use the *ResourceAllocator* function to allocate the computed supervisor nodes to the topologies. The *ResourceAllocator* function is presented in Table 3 and is discussed next.

Table 3: The Resource Allocator Algorithm

1	input: CN, CA
2	for each $j \in PR$
3	for each $tp_{ij} \in TP_j$
4	if $cn_{ij} \neq ca_{ij}$
5	de allocate tp_{ij} by flushing currently occupying worker processes
6	end if
7	else
8	remove cn_{ij} from CN and tp_{ij} from TP_j
9	end else
10	end for
11	end for
12	for each $j \in PR$
13	for each $cn_{ij} \in CN$
14	Using the round robin strategy (similar to the Isolation Scheduler), select W_{ij} worker processes from cn_{ij} supervisor nodes
15	Using the round robin strategy (similar to the Isolation Scheduler), assign executors from E_{ij} to the selected W_{ij} worker processes
16	end for
17	end for

The *ResourceAllocator* algorithm starts by deallocating resources from the topologies that are consuming numbers of supervisor nodes that are not equal to the respective computed values of supervisor nodes. There can be two scenarios where deallocation of resources is required while using SPS:

1. *SPS* is invoked every time a new topology is submitted to the cluster. Based on the priority indication of the newly submitted topology, already scheduled topologies may need to give up resources as the previously computed schedules become invalid with the addition of a new topology. As a result, topologies consuming more supervisor nodes need to free up the extra supervisor nodes and need to be rescheduled according to the newly computed value of supervisor nodes to be allocated.
2. As noted earlier, Nimbus periodically invokes the scheduler with a default period of 10 seconds. With passing time, it is possible for some topologies to be killed by the user or new supervisor nodes to be added to the cluster pool. In situations like these, the previously computed schedule also becomes invalid as the free supervisor nodes are left unused. The existing topologies can utilize these free supervisor nodes for additional performance gain. So, a new schedule will be computed by SPS and topologies consuming less resources than the computed values will be rescheduled. In line with the above discussion, *ResourceAllocator* compares the number of supervisor nodes already allocated with the computed number of supervisor nodes to be allocated for each of the topologies [Line 4 of Table 3]. If the numbers are not equal, a new supervisor node allocation will be required for the topology and its current allocation is removed [Line 5 of Table 3]. On the other hand, if both the

numbers are equal for a topology, then a new allocation is not required and the topology with its computed number of supervisor nodes to be allocated value are not considered in the further scheduling steps [Line 8 of Table 3]. This step is required for preventing unnecessary deallocation of topology resources. The steps are repeated for every priority level in PR [Lines 3 to 8 of Table 3].

Finally, topologies are allocated the computed number of supervisor nodes and executors of the topologies are mapped to their desired number of worker processes. For the i^{th} topology in the j^{th} priority level, the desired number of worker processes W_{ij} are selected from the computed cn_{ij} supervisor nodes [Line 14 of Table 3]. This is done in a round robin fashion similar to the default scheduler (presented in Section 2.5.1). Similarly, the executors of the topology are also mapped to the selected worker processes following the round robin strategy [Line 15 of Table 3].

3.3 Dynamic Priority Scheduler (DPS)

Dynamic Priority Scheduler (DPS) uses dynamic priority indication from topologies at runtime to formulate schedules in an online fashion. Unlike SPS, application developers do not determine priority indications of the topologies, instead all of the submitted topologies are assigned with the same initial priority values. Topologies may assume a higher or lower priority at runtime based on some trigger conditions such as an increase in tuple arrival rate. The DPS architecture is illustrated in Figure 5.

Similar to SPS, the DPS architecture has two components: the DPS scheduler and the Priority Manager (PM). As noted earlier, while using SPS, developers specify the priority levels of the submitted topologies in PM whereas in DPS, all the topologies are assigned a default priority value by the PM when submitted to the cluster. At runtime, a topology may

indicate a change of priority based on some predefined trigger conditions. When PM receives the indication, it changes the priority of that topology. Developers need to define the required logics for the trigger conditions and implement the logics that will be included in the Storm topologies. As the logic for the trigger conditions are highly specific to business use cases and the system environments, the responsibility is left to the application developers to provide the appropriate trigger logics. When a trigger condition occurs, a topology sends the *Trigger On* indication and PM updates the priority of the topology. DPS and then generates a new schedule. Apart from this, PM also assumes two important responsibilities which are activation and deactivation of topologies that are explained after the presentation of the DPS algorithm.

Figure 5: DPS Architecture

The DPS algorithm is described in Table 4. The notations used are already discussed in Table 1.

Table 4: The DPS Algorithm

1	<i>Get C from the cluster</i>
2	$ST = \sum_{j=1}^p \sum_{i=1}^{l_j} sd_{ij}$
3	$X = \min(C, ST)$
4	for each $j \in PR$
5	$SP_j = \sum_{i=1}^{l_j} sd_{ij}$
6	$P = \min(X, SP_j)$
7	for each $tp_{ij} \in TP_j$
8	$cn_{ij} = \min(X, \text{round}(P * (\frac{sd_{ij}}{SP_j})))$
9	add cn_{ij} to set CN
10	$X = (X - cn_{ij})$
11	end for
12	end for
13	end else
14	call <i>ResourceAllocator()</i>

DPS starts by querying the cluster for total number of available supervisor nodes [Line 1]. Next, the total number of supervisor nodes required by all the topologies ST is calculated in [Line 2]. Next, the minimum of the C and ST values is chosen [Line 3] and stored in variable X to discard the extra supervisor nodes that may be present in the cluster. Next, DPS Iterates through the priority levels of the topologies. For topologies in the j^{th} priority level of PR , total number of supervisor nodes required by the topologies is calculated as SP_j [Line 5]. Next, minimum of X and SP_j is selected and stored in variable P to make sure the available supervisor nodes to be allocated to the topologies in the priority levels are not more than the desired SP_j value [Line 6]. Using the equation in Line 8, proportional allocation of P supervisor nodes is computed for each of the topologies with the priority level and added to set of newly computed number of supervisor nodes CN . The steps are repeated for all the priority levels and finally ResourceAllocator function is called with CN

as input parameter. The ResourceAllocator algorithm is presented in Table 3 and already discussed in Section 3.1.1 while discussing SPS.

It is to be noted that, unlike SPS, DPS does not compute the proportional share of supervisor nodes for the priority levels. Instead all the available supervisor nodes are directly allocated to the topologies in a priority level [Line 6] and only topologies in the same priority level shares the supervisor nodes proportionally [Line 8]. Thus, it is possible for lower priority topologies to starve in the presence of higher priority topologies. The starving topologies may cease to starve in time if they also assume a higher priority based on some trigger conditions or topologies running currently at a higher priority assume lower priority due to the occurrence of trigger off signals.

It is established from the above discussion that using the DPS involves runtime rescheduling of topologies. Whenever topology priorities are changed, some topologies may need to give up supervisor nodes and some topologies may get awarded with more supervisor nodes. This nullifies the previously computed schedules for the topologies and deallocation of supervisor nodes from some topologies are required. Run time rescheduling of topologies can give rise to a number of problems including:

- **Tuple Loss:** If a running topology is immediately rescheduled without deactivating and waiting for existing tuples to be processed first, then the existing tuples already present in the system will be lost which may hamper the functional correctness of the Storm application [33] .
- **Unexpected Termination and Duplication of Executors:** A supervisor daemon checks Zookeeper for new executor to worker process assignments every 10 seconds. As a result, if a topology is rescheduled, it is possible that the some of the

previous worker processes the topology was occupying are still active while the some of the new worker processes from the new schedule have not started. In such a situation, if a newly rescheduled topology is immediately activated without waiting for some time for the worker processes to properly shut down/start, then there can be unpredictable outcome for the topologies including tuple loss, duplicate executors, unexpected worker process termination or even complete supervisor node termination [45,33]

Because of the issues mentioned above, ensuring a smooth run time rescheduling thus incurs rescheduling overhead [33,30] . These problems are well known and reported by other researches such as [33] where the authors introduced two strategies to prevent tuple loss and unexpected worker process terminations. Firstly, authors introduced a static delay of 30 seconds while rescheduling a topology which gives the worker processes ample time to properly shut down/start. Secondly, authors introduced intermediate message dispatcher daemons for the worker processes where tuples from one worker process to another is transported through the dispatcher daemon. While the approaches introduced in [33] ensure a smooth run time rescheduling for the single topology used in the research, they do not address rescheduling multiple Storm topologies. Additionally, introducing daemon message dispatchers for every single worker processes may also limit the performance and scalability of the Storm system as a typical Storm cluster can be comprised of many supervisor nodes where each supervisor node can host a number of worker processes. In this thesis, a novel approach is introduced for runtime rescheduling of topologies in a multitenant Storm cluster. Using an UML Activity Diagram [46] presented in Figure 6, the rescheduling approach is described.

Figure 6: Activity Diagram for Runtime Rescheduling of a Topology

From Figure 6, it can be seen that after the submission of a topology to the cluster, DPS reads the initial priority indication from the PM and schedules the topology accordingly (see Section 3.2.2 for the DPS algorithm). At runtime, topologies communicate with PM with an indication of the onset of trigger condition. Upon receiving the trigger indication from a topology, PM deactivates the topology using the Storm Representational State Transfer (REST) API [47]. Deactivating a topology ensures that the spout executors of the topology remain inactive and do not pull any new tuples from Kafka. Next, PM waits for the topologies to finish processing the existing tuples in the system. Topologies inform PM after they have finished processing existing tuples. To accomplish this, topologies use two Storm utilities called “tick tuples” [48] and the “Nimbus client” [49]. Tick tuples are signals

that are generated from within the topologies with a preconfigured frequency. And the Nimbus client makes it is possible for topologies to know if they are in a deactivated state. Combining these two utilities, whenever a topology identifies itself in an inactive state and there are no tuples left to process, it waits for a predefined number of tick tuples and decides that it has finished processing the existing tuples and signals PM. PM receives the signal and understands that it is safe to change the priority of the topology. PM then changes the priority of the topology and waits for the scheduler to perform the rescheduling. When DPS finishes rescheduling the topology, PM activates the topology, and the topology can start processing tuples with the new resource allocation.

3.4 Implementation

Prototypes for the two schedulers were implemented for Storm version 0.9.6 [50]. Both the SPS and DPS scheduling algorithms discussed in Section 3.2 and Section 3.3 are implemented in JAVA 1.6 [25]. Storm allows the incorporation of the custom schedulers through its *IScheduler* [51] plugin. The plugin is a Java interface where a single method called *schedule* needs to be implemented by the developers. The Nimbus daemon invokes this *schedule* method when it needs to run the scheduler. The custom scheduler is packaged as a standard *jar* package and placed on the library folder of the Nimbus node. Nimbus then needs to be instructed to use the custom scheduler through appropriate configuration parameters in the Nimbus configuration file. Some Storm APIs that are heavily used to implement the scheduling steps are listed in [51]. Using these APIs, various operations such as allocating executors to worker processes, computing total number of supervisor nodes present in the cluster, freeing up worker slots are implemented.

The PM module is also developed using JAVA. A MySQL [52] database is used in the

backend to store priorities of the topologies. The database is accessed using the Java Database Connectivity (JDBC) [53] API. The communication between various topologies and the PM module is done through message passing using the Kafka [15] message broker. Chapter 4 discusses implementation details specific to Kafka in more details.

Chapter 4: Performance Evaluation of the Scheduling Techniques

The performance of the proposed schedulers is evaluated by conducting a number of experiments based on two popular use cases of distributed stream processing systems. This chapter describes the use cases and experiments that have been conducted to study the performance of the proposed priority based schedulers. Implementation details are provided for the Storm topologies and data producers used in the two use cases. Workload and system parameters are outlined and performance metrics that are used to evaluate the proposed schedulers are identified. The results of the experiments are then presented and insights into system behavior and performance resulting from the performance analyses are summarized.

4.1 Use Case-1: Systems Subjected to the Arrival of Batch Data

The first use case considers a stream processing platform where several producers are sending data in small batches to the data ingestion layer. Each batch has a fixed number of tuples and after sending a batch of tuples, producers wait for a predefined fixed amount of time before sending the next batch. Multiple Storm topologies are then used to process the tuples from the ingestion layer.

Some popular use cases similar to this scenario is the use of sensors / Internet of Things (IOT) devices for business critical data analytics in the field of healthcare [54], urban development [55], automotive industry [56], aviation [57] and many more. As an example, consider a sensor based real time climate monitoring application for a data center where various types of sensors are used to harvest real time climate information including: temperature, air-flow, relative humidity etc. for better energy management and protection of valuable devices, specifically the servers. A use case of such an application is presented

in [58] where Intel partnered with Baidu to deploy a real-time sensor based data center monitoring application which resulted in up to 30% lower power consumption and better operational management in Baidu data centers. A similar application can be seen in [59] where HP uses “smart on demand cooling” in their data centers which reduces their operational costs by up to 40%.

The Sensor/IoT devices are usually power and resource constrained devices and use short range low rate network communication protocols like ZigBee, MQTT, and Bluetooth etc. for data transfer [60]. Devices send measurement data to centralized "IoT Gateways" which can communicate over the internet to the outside world. Referring to the data center climate monitoring example, servers in a data center are mounted on racks where each rack accommodates a number of servers and a data center can have thousands of such servers mounted on several racks [61,62]. A number of sensors are placed on each rack on different measuring points [63]. These sensors send their measurement data to "Gateways" [64,65]. A typical IoT gateway has the responsibility to upload these collected device data to the data analytics platform.

Use Case-1 used in the experimental analysis considers a similar application involving sensors where measurement data are collected for two phenomena. Multiple producer applications modelling gateways that push batches of synthetically generated data on the phenomenon being modelled (e.g. sensor readings) to the data ingestion layer are considered. Four Storm topologies are used to process these ingested data. The details of the data producers and Storm topologies are discussed in the next sections.

4.1.1 Data Producers

For the experiments that are run for Use Case 1, 8 producer applications push batches of

synthetically generated readings for the respective phenomena to the ingestion layer of the distributed stream processing platform. The internals of the stream processing platform are discussed in Section 2.1 of this thesis. Apache Kafka [15] used in the ingestion layer is a popular distributed publish/subscribe based messaging system. Kafka can be run as a cluster on one or more machines where each machine hosts an independent Kafka message broker. Kafka has four important key concepts. They are:

- Topic: Streams of incoming data are categorized into topics. A message is defined as a payload of bytes and a topic is a category or feed name to which messages are published.
- Partitions: A topic is divided into multiple parts. Each of these parts is called a partition and the partitions are distributed throughout the broker nodes in the Kafka cluster. Partitions are the unit of parallelism of a Kafka topic.
- Producers: Producers publish data to topics. If a topic has multiple partitions, then multiple brokers can publish data to that topic in parallel, each sending data to a separate partition (see Figure 7).
- Consumers: Consumers subscribe to topics and read the published messages. Depending on the number of partitions of a topic, multiple consumers can read data from the same topic in parallel. Figure 7 illustrates a Kafka topic T with N partitions. N producers are publishing data on the topic and M consumers are reading data from the topic in parallel.

Figure 7: Kafka Architecture

It is to be noted that, Kafka is not a permanent storage solution for the ingested data. A configurable retention period is used for each topic and after the retention period is over, data is flushed out from the system.

To send data into partitions of a topic, Kafka provides developers with the producer API [66]. Producers publish streams of messages to topics and Storm topologies are used as consumers to process the ingested messages. For each type of stream, a separate Kafka topic is created with multiple partitions. For each topic, a producer application is created using Java programming language [25] that pushes messages to the respective Kafka topics using the producer API. The producer applications are called *Producer1* and *Producer2*. For sending messages to multiple partitions of a topic in parallel, Java threads are used by each producer application.

Based on the type of the data readings, producers send batches of data to two different Kafka topics where one of the Kafka topics: *Topic1* is for synthetic data readings for the first phenomena and the other Kafka topic *Topic2* is for synthetic data readings for the second phenomenon (see Table 5). Each of the topic has a partition count of 8 which means in each topic, 8 producers can push data in parallel. Based on the format and structure of

real world IOT device data collected and examined from [67], a generic message format is considered where each message a producer sends to the ingestion layer is structured as a Comma Separated Value (csv) line with 4 fields. The format is

➤ **<date-time:(String)>,<source:(String)>,<readingValue:(Number)>,<other:(String)>**

The first field <date-time> contains the time stamp of the reading, the second field contains the originating source (gateway ID), the third field contains the reading value specific to the respective phenomena and the fourth field contains other information which serves as meta-data such as: device id, serial number.

Various workload parameters that are used to control the content of the messages, the number of messages in each batch and the time difference between two subsequent batches sent by the producer applications are discussed in Section 4.1.3.

Table 5: Overview of the Data Producers

Parameter	Value	Description
Types of data readings	2	Two types of readings are considered, each for one specific phenomena
Number of Producers for each type of readings	8	8 data producers, modeled as gateways that push synthetically generated messages to Kafka in small batches are considered
Number of Kafka topics	2	One of the Kafka topics: <i>Topic1</i> is for data readings for the first phenomena and the other one: <i>Topic2</i> is for the data readings for the second phenomena.
Number of partitions for each topic	8	8 producers can transmit batches of data to a topic in parallel

4.1.2 Storm Topologies

4 Storm topologies are used to process the ingested batches of tuples from Kafka. The purpose of using the 4 Storm topologies is twofold. Two of the Storm topologies are used to infer "Complex Events". A complex event is a simultaneous occurrence of two or more independent raw events where each raw event signifies an onset of certain situations that

are analyzed over streams of data [68] and has been the subject of attention for research on stream processing systems. In this use case, the two raw events are described as following:

- Raw Event 1: A certain proportion of the tuples representing measurement readings of the first phenomenon in a batch has crossed a predefined threshold.
- Raw Event 2: A certain proportion of the tuples representing measurement readings of the second phenomenon in a batch has crossed a predefined threshold.

The first topology named *EventTP1* is used to detect Raw Event 1 which processes tuples from the Kafka topic *Topic1*. The second topology named *EventTP2* is used to detect Raw Event 2 by processing tuples from Kafka topic *Topic2*. Processing time for each tuple is characterized by a parameter called service time which is discussed in Section 4.1.5.

Simultaneous occurrence of Raw Event 1 and Raw Event 2 signifies the occurrence of the complex event. Timely detection of a complex event is of interest to the users of the stream processing application. Consider the data center climate control application example for instance. A complex event in such an application can mean that both temperature and humidity levels are out of an operational range and alarms can be generated for external automated climate control systems which can take corrective actions such as increasing the airflow or decreasing the air cooler temperature to compensate for the degrading humidity and temperature.

The other two topologies are used to persist the ingested data to a storage device (e.g. hard drive/ block storage such as "Amazon S3" [69]). As discussed, Kafka is not a permanent storage solution and it holds the ingested data for a configurable amount of limited time. Thus, data needs to be shipped from Kafka to a permanent storage facility such that further

analysis can be performed on these data (e.g. historical analysis by the batch data analytics system). Storm topologies are heavily used for data archival and transportation in a big data analytics framework and a very detailed example can be found in the Twitter technology blog [70].

Storm topologies are developed using the Storm library. Both topologies, *EventTP1* and *EventTP2*, have the same structure as illustrated in Figure 8.

Figure 8: Topology Structure for EventTP1 and EventTP2

As shown in Figure 8, both *EventTP1* and *EventTP2* have four components. The first component in each topology is a spout which reads data from a Kafka topic (*Topic1* for *EventTP1* and *Topic2* for *EventTP2*) and pushes tuples to the downstream bolts for further processing. Storm provides support for reading data from Kafka by a spout library called "KafkaSpout" [71]. The next component is a bolt called "Parse Bolt" which parses the comma separated value (csv) structured tuples in multiple fields and forwards the <source> and <reading-value> field of each tuple to downstream "Event Detector Bolt". Next, the "Event Detector Bolt" in each topology identifies the occurrence of the raw events (Raw Event 1 for *EventTP1* and Raw Event 2 for *EventTP2*) on a batch by processing the contents of <reading-value> field for each tuple. Note that, for both the topologies, a raw event is

detected if 80% tuples are a batch has crossed a predefined threshold. The event detection result is then passed to the next downstream bolt called the "Event Dispatcher Bolt" which writes the event detection decisions to a Kafka topic. End systems such as a web based User Interface (UI)/Alarm system reads from the Kafka topic to signal the detection of the raw events. Simultaneous occurrence of both the raw events signifies onset of a complex event.

The remaining two topologies that are used for data archival are called *ArchivalTP1* and *ArchivalTP2*. *ArchivalTP1* is used for archiving ingested data from *Topic1* and *ArchivalTP2* is used for archiving ingested data from *Topic2*. The topologies have a similar structure with 3 components (see Figure 9). The first component is the Kafka spout like the ones used in *EventTP1* and *EventTP2*. The next component is a bolt called "Archival Bolt" which appends a new time stamp to each ingested tuple, adds an archival tag (e.g. a storage device identifier) and forwards the tuple to next bolt. Finally, the last bolt on the processing chain "Batch Dispatcher Bolt" which batches up the tuple for predefined batch length and dispatches them to disk for archival.

Figure 9: Topology Structure for ArchivalTP1 and ArchivalTP2

Table 6 lists the resources that need to be provisioned for the 4 topologies. Both *EventTP1* and *EventTP2* are allotted 8 supervisor nodes. Recall that, 8 producers send batches of tuples in parallel to the Kafka. Thus, each component (spout/bolt) of *EventTP1* and *EventTP2* are assigned 8 concurrent executors (see Table 6 Column 5) where one batch is processed on the worker process of one supervisor node. The rest of the topologies (*ArchivalTP1* and *ArchivalTP2*) are assigned 4 supervisor nodes each because they have 4 concurrent threads per topology component. The number of supervisor nodes and the worker processes for a given topology are kept equal following the resource provisioning best practices provided by Storm [72].

Table 6: Topology Resource Provisioning

Topology	Desired Number of Supervisor Nodes	Desired Number of Worker Processes	Number of Topology Components	Number of Executors/ concurrent threads per topology component
<i>EventTP1</i>	8	8	4	8
<i>Event TP2</i>	8	8	4	8
<i>ArchivalTP1</i>	4	4	3	4
<i>ArchivalTP2</i>	4	4	3	4

4.1.3 System Configuration

To run the experiments, Storm, Zookeeper and Kafka clusters are set up on an Amazon EC2 cloud infrastructure [73]. Storm and Kafka both use Zookeeper for state management and cluster synchronization. In total 32 nodes, running on the Amazon EC2 cloud infrastructure are used. Amazon provides different types of EC2 nodes depending on the

CPU and memory capacity. Among the different types of EC2 nodes available, 26 c4.large type EC2 nodes are used for the Storm cluster. 3 m4.large type EC2 nodes are used for the Zookeeper cluster and the 3 m4.large type EC2 nodes are used for the Kafka cluster. Table 7 captures the detailed system configuration for the individual clusters.

Table 7: System Configuration for the Clusters

Cluster Type	Configuration Parameter	Value
Storm cluster	Number of nodes in the cluster	26
	Type of EC2 instance	c4.large
	Number of CPU cores/instance	2
	CPU clock frequency	2.8 GHz
	RAM	3.75 GB
Zookeeper cluster	Number of nodes in the cluster	3
	Type of EC2 instance	m4.large
	Number of CPU cores/instance	2
	CPU clock frequency	2.8 GHz
	RAM	8 GB
Kafka cluster	Number of nodes in the cluster	3
	Type of EC2 instance	m4.large
	Number of CPU cores/instance	2
	CPU clock frequency	2.8 GHz
	RAM	8 GB

Among the 26 nodes dedicated for Storm clusters, 1 node is used for running the Nimbus, 1 node is used for running the User Interface (UI), and 24 nodes are used for running the supervisor nodes where each supervisor node hosts 2 worker processes. The producer applications are run on a local machine equipped with 16 GB of RAM and an Intel Core-i7 processor with a clock speed of 2.4 GHz running on Windows-7 operating system. The local machine communicates with the clusters set up on the cloud through internet provided

by Carleton University with a download speed of 97 Mbps and upload speed of 98.21 Mbps.

Please note that, the average tuple transfer time between the local machine and the Kafka cluster that is set up on the Amazon EC2 cloud, is in the order of few milliseconds. This is significantly smaller in comparison to the average tuple processing latency, that is in the order of hundreds of milliseconds.

4.1.4 Performance Metrics

Performance of the schedulers is evaluated using 2 performance metrics:

- (1) Average complex event inference latency (T_E): A complex event is the simultaneous occurrence of Raw Event 1 (Detected by *EventTP1*) and Raw Event 2 (Detected by *EventTP2*). Detection of both Raw Event 1 and Raw Event 2 contributes to the inference of the occurrence of a complex event. The complex event inference latency is the estimate from the raw event detection latencies that are measured first. Detection latency for Raw Event 1 is measured by taking two timestamps using `System.currentTimeMillis()` from Java Library, one after *Producer1* has sent a batch of tuples to Kafka and another after the topology *EventTP1* finishes detecting the raw event by processing this batch of tuples from Kafka. These two timestamps are taken by a separate application called “Perf” that is run on the local machine after it receives the respective messages from *Producer 1* and *EventTP1*. The detection latency computed by “Perf” is the difference between these two timestamps (in seconds). For Raw Event 2, detection latency is measured in a similar way after receiving similar messages from *Producer2* and *EventTP2*. A complex event inference latency is estimated as the maximum of these two raw event

detection latencies. Average complex event inference latency is thus computed by taking average of all the computed complex event inference latencies.

- (2) Average tuple processing latency (T_T): Average tuple processing latency is the average time in seconds a tuple takes to be processed by the Storm topologies. To compute T_T , batch processing latencies are computed for topologies. To compute batch processing latency of *EventTP1*, two timestamps are taken using `System.currentTimeMillis()` from Java Library, one after the *Producer1* finishes sending a batch and one after *EventTP1* finishes processing the batch. These two timestamps are taken by the “Perf” application after it receives the respective messages from the producer and the topology. The “Perf” application then takes the difference between the two timestamps to compute the batch processing latency. By dividing the batch processing latency by the number of tuples in that batch, tuple processing latency for that batch is computed. Average tuple processing latency for *EventTP1* is then computed by computing tuple processing latency for every batch in the experiment and taking the average of all the computed tuple processing latencies. Similarly, average tuple processing latency for *EventTP2*, *ArchivalTP1* and *ArchivalTP2* are computed by “Perf” in a similar way after receiving similar messages from corresponding producers and the topologies.

The average tuple processing latency for the higher priority topologies is computed as the average of the mean tuple processing latency for *EventTP1* and *EventTP2*. Similarly, the average tuple processing latency for the lower priority topologies is computed as the average of the mean tuple processing latency for *ArchivalTP1* and *ArchivalTP2*.

4.1.5 Workload Parameters

This section describes the various workload parameters used in the experiments.

- Batch Gap (B_G): The Batch gap, is the time difference in seconds between successive batches of tuples sent by the producers. After sending a batch of tuples, a producer waits for the time specified by the batch gap before sending the next batch of tuples. If an event is to be inferred from a batch, it must be done within the time frame of the B_G before the next batch of tuples arrives.
- Batch Length (B_L): Batch length is the number of tuples present in each batch sent by the producers.
- Total Number of Batches (B_T): Total number of batches is the total number of batches that are sent by the producers in each experiment.
- Event Factor (E_P): Event factor is the ratio of the number of batches that will generate raw events for the two topologies (*EventTP1* and *EventTP2*) to the total number of batches in the experiment (B_T). It is a real number between 0 and 1. This parameter essentially indicates the proportion of batches that will cause the trigger to be in the ON state. As an example, if B_T has a value of 80 and E_P has the value of 0.2 then for 16 consecutive batches, raw events will be detected and the trigger will be in the on state for the two topologies detecting raw events.
- Priority Levels (PR): Two distinct priority levels are used for the experiments, they are:
 - Priority Level, PR = 1: Denotes topologies with higher priorities
 - Priority Level, PR = 2: Denotes topologies with lower priorities

While using SPS, two of the topologies: *EventTP1* and *EventTP2* are assigned

higher priority (PR=1) and topologies *ArchivalTP1* and *ArchivalTP2* are assigned lower priority (PR=2).

- Service Time (S_{high} and S_{low}): Service time is the amount of CPU time in seconds each tuple takes to complete its processing on a Storm topology. In line with the researches in [39,41,31], fixed values of service times are used in the experiments. Specific values of service time used in the synthetic workload that is used in the different experiments are generated with the help of a method that computes the factorial of a large number. The method is called iteratively with the number of iterations chosen in such a way that a desired value of service time is achieved. Two different service times are used for the topologies depending on their priority levels. Tuples processed by a topology with higher priority (Priority Level, PR = 1) has a service time S_{high} whereas tuples processed by a topology with lower priority (Priority Level, PR = 2) has a service time of S_{low} . Note that the values of S_{low} used in an experiment is set to half the value of S_{high} .

Each experiment is repeated 3 times and average results are computed. Note that, fixed values of workload parameters (e.g. service time, batch length, batch gap) are used in each experiment, thus close values for a given metric are achieved in each of the three repetitions.

4.1.6 Performance Evaluation of the Priority Based Schedulers

The experiments to evaluate the performance of the proposed priority based schedulers are run for two scenarios. In the first scenario, a resource constrained Storm cluster is considered. The proposed priority based schedulers are intended to be used in a resource constrained Storm cluster where the available cluster resources are not sufficient to fulfill

the desired resource requirements of the submitted topologies but are sufficient to meet their minimum resource requirements. Recall from Section 2.5.1.1 that, in such situations, the default Isolation Scheduler cannot be used because of the partial scheduling or even complete starvation of the topologies that are submitted to the cluster. Thus, the Isolation Scheduler is not considered during evaluation of system performance in this scenario and only the performances of SPS and DPS are evaluated. In the second scenario, the cluster has adequate number of supervisor nodes to fulfill the desired supervisor node requirements of all the topologies and performance of the proposed priority based schedulers are evaluated against the default Isolation Scheduler.

4.1.7 Performance Evaluation for a Resource Constrained Storm Cluster

The proposed priority based schedulers are intended to be used in a resource constrained Storm cluster where the cluster does not have adequate number of supervisor nodes to satisfy desired number of supervisor node requirements for all the topologies. As noted earlier, in such a cluster, Isolation Scheduler fails to schedule all the topologies and thus, is not considered any further. The two proposed schedulers use priority indication from the topologies to proportionally allocate the available limited resources to the topologies. SPS uses static priority indication of the topologies which are provided by the developers in the topology submission time while DPS uses runtime dynamic priority indication of the topologies which are generated by the topologies at runtime based on some predefined trigger conditions.

Recall from Table 6 that a total number of 24 supervisor nodes are desired by the 4 topologies used in the experiment. For the resource constrained case, the available number of supervisor nodes in the cluster is reduced to 16. For such a cluster, resource provisioning

of the topologies using SPS is captured in Table 8.

Table 8: Topology Resource Provisioning using SPS

Topology Name	Priority Level (PR)	Desired Number of Supervisor Nodes	Minimum Number of Supervisor Nodes	Number of Supervisor Nodes Allocated
<i>EventTP1</i>	1	8	4	6
<i>EventTP2</i>	1	8	4	6
<i>ArchivalTP1</i>	2	4	2	2
<i>ArchivalTP2</i>	2	4	2	2

SPS allocates topologies with available supervisor nodes in the resource constrained cluster using the algorithm described in Section 3.2. Developers provide the priority indication and minimum number of supervisor node requirements for each topology at topology submission time (see Column 2 and 4 of Table 8). The priority indications of the topologies are determined by the developers based on the business significances of the topologies. For the minimum number of supervisor node selection, the system configuration of the cluster needs to be taken into consideration. As described in Table 7, each supervisor node in the cluster has 2 CPU cores and thus each supervisor node hosts 2 worker processes each. Recall from Table 6 that both *EventTP1* and *EventTP2* have 8 executors per component and 8 worker processes. Thus, at least 4 supervisor nodes are needed for these topologies to run. Similarly, 2 supervisor nodes are needed at minimum for the low priority *ArchivalTP1* and *ArchivalTP2* (Column 2 and 4 of Table 8). The resource provisioning for the topologies remains unchanged throughout the experiment while using SPS as it does not allow run time priority change of the topologies.

On the other hand, DPS reacts to the reported trigger indications from the topologies and

reschedules the topologies accordingly. Recall from Section 3.3 that topologies report the trigger indications based on predefined logic. For the experiments in this use case, *EventTP1* and *EventTP2* report the trigger condition when a predefined percentage of tuples in a batch crosses threshold. Table 9 captures the supervisor nodes allocated to the topologies when using DPS. Initial priority levels of the topologies can be seen from Column 2 and it can be seen that all topologies are assigned the same initial priority level. In such a situation, using the DPS algorithm described in Section 3.3, the available 16 supervisor nodes are proportionally shared among the 4 topologies (see Column 4 of Table 9). When a trigger ON indication is reported by *EventTP1* and *EventTP2* at runtime, they are assigned a higher priority level (Column 3) and all the available supervisor nodes are allocated to them (see column 5). The lower priority topologies e.g. *ArchivalTP1* and *ArchivalTP2* are temporarily starved (0 supervisor nodes are allocated) until the higher priority topologies finish inferring complex events and send trigger OFF indication.

Table 9: Topology Resource Provisioning using DPS

Topology	Initial Priority Level (PR)	Priority Level (PR) when Trigger ON	Number of Supervisor Allocated (Event Trigger OFF)	Number of Supervisor Allocated (Event Trigger ON)
<i>EventTP1</i>	2	1	5	8
<i>EventTP2</i>	2	1	5	8
<i>ArchivalTP1</i>	2	2	3	0
<i>ArchivalTP2</i>	2	2	3	0

A summary of the parameters used in the experiments is presented in Table 10. The parameters are divided into 2 categories: workload and system. The experiments are run following a factor at a time method where one of the parameters is changed while others are held at their default values (indicated in bold at Table 10).

Table 10: Summary of Parameters Used in the Experiments

Parameter Type	Parameter	Value
Workload	Batch Gap (B_G)	{40, 60 ,80} seconds
	Batch Length (B_L)	{60, 80 ,100} tuples
	Total Number of Batches (B_T)	80
	Event Factor (E_P)	{0.2, 0.5 ,0.8}
	Service Time for Higher Priority Topologies (S_{high})	{0.1, 0.2 ,0.3} seconds
	Service Time for Lower Priority Topologies (S_{low})	($S_{high}/2$)
System	Total Number of Topologies	4
	Total Number of Supervisor Nodes Present in the Cluster	16
	Total Number of Supervisor Nodes Desired by the Topologies	24
	Number of Kafka Topics	2
	Number of Parallel Producers in Each Topic	8

4.1.7.1 Effect of Batch Gap (B_G)

As discussed before, Batch Gap (B_G) indicates the time difference in seconds between two successive batches that are sent by a producer. Complex events need to be inferred within this time frame so that a decision on the occurrence of events in the current batch is available before the next batch of tuples arrive on the system. Following three outcomes are possible from processing a batch of tuples by the topologies inferring events:

1. Trigger OFF: Trigger condition is not satisfied for the topologies that are detecting raw events (*EventTP1* and *EventTP2*). Thus, no trigger ON indication is sent by the topologies and their priority is also not changed by the PM (see Table 9 Column 4).
2. Trigger ON: Trigger condition is satisfied by *EventTP1* and *EventTP2* and a trigger on indication is reported to PM. PM changes the priority of the topologies to a higher priority level (from priority level 2 to priority level 1) and *EventTP1* and *EventTP2*, the topologies with higher priority get all the available supervisor nodes (see table 9, Column 5).
3. Event Occurred: Raw event is inferred. The raw events and the composition of complex event are discussed on Section 4.1.2

Figure 10 captures the effect of B_G on complex event inference latency T_E . For any value of B_G , SPS results in a higher T_E than that of DPS. Being an offline scheduler, SPS cannot react to run time trigger indications sent by the topologies. Thus, both event inferring high priority topologies are allocated 6 resources statically whereas DPS can react to the priority indications of the topologies and allocates 8 supervisor nodes to each of the high priority topologies. Thus, high priority topologies are allocated more supervisor nodes when DPS is used and that results in a 38% lower T_E than that of SPS. It can be also seen that, increasing the value of B_G has no effect on the T_E values obtained by any of the schedulers. This is because, while the number of tuples in a batch and the service time for each tuple remains the same, the event inference latency is not impacted by a higher value of B_G which only provides a longer time window for topologies to infer complex events.

Figure 10: Effect of B_G on T_E

For every value of B_G , average tuple processing latency (T_T) for higher priority topologies is higher for SPS compared to that of DPS (see Figure 11). Again, DPS allocates more supervisor nodes to topologies while a trigger condition is satisfied (see Column 5 of Table 9). Thus, for the default E_P value of 0.5 and total number of batches (B_T) of 80, trigger condition will remain satisfied for the topologies detecting raw events (*EventTP1* and *EventTP2*) for 40 batches and these topologies will get 8 supervisor nodes each. For the remaining 40 batches, DPS will allocate 5 supervisor nodes to the 2 topologies. On the other hand, SPS will keep the initial allocation of 6 supervisor nodes to the higher priority topologies throughout the experiment. Thus, for any given B_G value, DPS results in a lower T_T value for topologies with higher priority compared to that of SPS (see Figure 11).

Figure 11: Effect of B_G on T_T (Higher Priority Topologies)

Conversely, Figure 12 shows that for every value of B_G , T_T for lower priority topologies is significantly higher for DPS than that for SPS. With DPS, when higher priority topologies are inferring complex events, lower priority topologies are temporarily starved as all the available supervisor nodes are allocated to the higher priority topologies. Thus, for E_p value of 0.5 and total number of batches (B_T) of 80, lower priority topologies will starve for 40 batches as all of the available supervisor nodes in the cluster are allocated to the higher priority topologies (*EventTP1* and *EventTP2*) while lower priority topologies (*ArchivalTP1* and *ArchivalTP2*) starve (see Column 5 of Table 9). Note that this starvation is temporary and lower priority topologies are allocated supervisor nodes again when the higher priority topologies report a trigger off indication after successfully inferring all the complex events in those 40 batches. Also, increasing the B_G values increases the starvation time of the lower priority topologies while using DPS. For an example, when B_G is 40 seconds, topologies starve for $40 \times 40 = 1600$ seconds whereas when $B_G = 60$ seconds, topologies starve for $40 \times 60 = 2400$ seconds before getting back the supervisor nodes to start processing the tuples that are waiting in Kafka to be processed. As discussed in Section 4.1.1, each of the ingested tuples in Kafka has a sequential id called *offset* and Kafka stores these tuples

for a preconfigured retention period. Topologies keep track of the offsets of the tuples that are pulled from Kafka and periodically stores the information on Zookeeper [71]. Thus, when the starving lower priority topologies get back their supervisor nodes, they can start processing the leftover tuples by looking at the last offset information stored in Zookeeper. The default retention period in Kafka is 168 hours or 7 days [74] which is more than enough for the lower priority topologies to resume operation and complete processing of the leftover tuples. On the other hand, as SPS is oblivious to run time priority indication and topologies are never rescheduled thus lower priority topologies are never starved. Thus, increasing B_G values has no effect on T_T when using SPS.

Figure 12: Effect of B_G on T_T (Lower Priority Topologies)

4.1.7.2 Effect of Service Time (S_{high} and S_{low})

From Figure 13, it can be seen that T_E increases with the increase in S_{high} . Service time signifies the amount of CPU time in seconds each tuple takes to be processed. Thus, a higher value of S_{high} signifies a higher CPU time for each tuple. To infer complex events, all the tuples in the batch needs to be processed. For any given value of S_{high} , T_E for SPS is

higher than that of DPS. This is because of the allocation of more supervisor nodes to topologies inferring complex events by DPS than that of SPS (see Table 8 and Table 9). Another observation that can be made from Figure 13 is that for higher values of S_{high} , the difference in T_E values between SPS and DPS is higher. For an example, from Figure 13, when $S_{high}=0.1$, T_E resulted using SPS is 21% higher than that of DPS whereas when $S_{high}=0.3$, T_E resulted using SPS is 39% higher than that of DPS. To explain this, recall that Storm is a pull based system. The number of tuples that Storm pull into the system from Kafka per topology is dictated by two parameters: (1) *fetchSizeBytes* and (2) *maxSpoutPending* [71,72]. The first parameter defines how many tuples are pulled at a time from Kafka and the second parameter defines the maximum number of tuples that can be present in the system per topology at a given time. If the system already has tuples equal to the number set by *maxSpoutPending* parameter, no tuples are pulled from Kafka until the existing tuples are processed. Thus, in Storm, at any given time, the maximum number of tuples that can be present in the system per topology is fixed. Thus, while the number of supervisor nodes allocated to the topologies and the number of tuples that are pulled in the system per topology remain same, an increase in S_{high} signifies more contention for CPU resources. Thus, in lower values of S_{high} , difference in T_E values achieved using SPS and DPS is lower and vice versa (see Figure 13).

Same discussion applies for the effect of S_{high} on average tuple processing latency T_T for higher priority topologies presented in Figure 14.

Figure 13: Effect of S_{high} on T_E

Figure 14: Effect of S_{high} on T_T (Higher Priority Topologies)

From Figure 15, for lower priority topologies, T_T values increase with the increase in S_{low} for both SPS and DPS for the reasons discussed earlier. Recall from Section 4.1.5 that that $S_{low} = (S_{high}/2)$. Values of T_T resulted using DPS are much higher than that of SPS for any values of S_{low} because of the starvation of lower priority topologies that occurs using DPS. The reasoning behind the starvation of the lower priority topologies and thus a significantly higher T_T values while using DPS is already discussed in Section 4.1.7.1.

Figure 15: Effect of S_{low} on T_T (Lower Priority Topologies)

4.1.7.3 Effect of Batch Length (B_L)

From Figure 16, one can see that as value of B_L increases, T_E value also increases for both SPS and DPS. A higher value of B_L signifies more number of tuples in a batch. As discussed on Section 4.1.2, occurrence of the complex events is determined by processing all the tuples in a batch. Thus, a higher value of B_L translates to a higher T_E value for both SPS and DPS. Also, it can be observed from Figure 16 that for any value of B_L , T_E resulted using SPS is approximately 50 % higher than that of DPS. This is because, DPS allocates a higher number of supervisor nodes to the topologies while they are inferring complex events than that of SPS (see Table 8 and Table 9).

Figure 16: Effect of B_L on T_E

From Figure 17, for any value of B_L , SPS results in a higher Average tuple processing latency (T_T) than that of DPS for higher priority topologies. Recall from Table 8 and 9 that DPS allocates more supervisor nodes to the topologies detecting events (*EventTP1* and *EventTP2*) when a trigger condition is in ON state. Thus, for E_P value of 0.5 and total number of batches (B_T) of 80, lower priority topologies will starve for 40 batches as all of the available supervisor nodes in the cluster are allocated to the higher priority topologies (*EventTP1* and *EventTP2*). For the remaining 40 batches, DPS will allocate 5 supervisor nodes to the 2 topologies inferring complex events. On the other hand, SPS will keep the initial allocation of 6 supervisor nodes to the higher priority topologies throughout the experiment (see Table 8). Thus, DPS results in lower T_T ($\sim 13\%$) values for topologies with higher priority. Also, recall from Section 4.1.7.2 that, at any given time, the maximum number of tuples that can be present in the system per topology is fixed. Thus, T_T does not change with a change in B_L for any of the schedulers.

Figure 17: Effect of B_L on T_T (Higher Priority Topologies)

Conversely, as seen on Figure 18, for lower priority topologies, DPS results in much higher values of T_T for any B_L . The reasoning behind this is already discussed on Section 4.1.7.1.

Figure 18: Effect of B_L on T_T (Lower Priority Topologies)

4.1.7.4 Effect of Event Factor (E_P)

From Figure 19, it can be seen that T_E values obtained using DPS are approximately 35% higher than that of SPS for any E_P . Increasing E_P increases the number of complex events to be inferred by the topologies. Recall that, DPS allocates the event detecting topologies with 8 supervisor nodes when their trigger indication is in ON state and does not change the schedule until all the complex events are successfully inferred (see Table 9). When E_P is 0.2, topologies are allocated 8 supervisor nodes for 16 consecutive batches, when E_P is 0.5, topologies are allocated 8 supervisor nodes for 40 consecutive batches and when E_P is 0.8, topologies are allocated 8 supervisor nodes for consecutive 64 batches out of $B_T=80$ batches. Whereas, SPS allocates the event inferring topologies with 6 resources and never changes the schedule (see Table 8).

Figure 19: Effect of E_P on T_E

Similar observations can be made from Figure 20 on the effect of E_P on T_T obtained by higher priority topologies.

Figure 20: Effect of E_P on T_T (Higher Priority Topologies)

From Figure 21, one can see that a higher value of E_P results in higher T_T for lower priority topologies while using DPS. A higher E_P value translates to a longer starvation time while using DPS. Given the default batch gap B_G is 60 seconds and total number of batches B_T is 80, when E_P is 0.2, lower priority topologies starve for $16 \times 60 = 960$ seconds before they regain 3 supervisor nodes each and resumes processing tuples (Table 9). Similarly, when E_P is 0.5, lower priority topologies starve for $40 \times 60 = 2400$ seconds. This is reflected on Figure 21. T_T values for SPS remains uninfluenced by E_P as SPS keeps the initial supervisor node allocation throughout the experiment and lower priority topologies never starve (Table 8).

Figure 21: Effect of E_P on T_T (Lower Priority Topologies)

4.1.7.5 Additional Experiments on a Smaller System

In order to investigate the effect of the number of resources on performance, the experiments for the resource constrained case presented in Section 4.1.7 are repeated on a smaller system comprising 8 supervisor nodes. The smaller cluster was set up using machines available in Carleton University's Real Time and Distributed Systems' lab. The 4 topologies used in the experiments are already described in Section 4.1.2. The performance metrics and the workload parameters for the experiments are the same as those described in Section 4.1.4 and Section 4.1.5 respectively. The VMware [86] virtualization hypervisor is used to set up the nodes (Virtual Machines). The nodes are connected through a private local area network using a 100 Mbps Ethernet connection. Once again, the experiments are run following a factor at a time method where one of the parameters is changed while others are held at their default values.

All the experiments investigating the impacts of B_G , B_L , S_{high} , S_{low} and E_P show that the relative performances of the scheduling policies do not change with a change in the number of nodes. The performance trends for SPS and DPS are similar to the ones observed in the

corresponding experiments performed on the Amazon EC2 cluster comprising 16 supervisor nodes described Section 4.1.7.

4.1.8 Performance Evaluation for a Resource Unconstrained Storm Cluster

A resource unconstrained Storm cluster has the adequate number of supervisor nodes to satisfy requirements for all the topologies. For such a cluster, SPS, DPS and Isolation Scheduler are expected to perform comparably with one another as all 3 of the schedulers provision the topologies with their desired number of supervisor nodes. To validate this, performance of SPS and DPS are compared with that of the Isolation Scheduler in a resource unconstrained Storm cluster comprising 24 supervisor nodes. 4 Storm topologies that were discussed in Section 4.1.2 are used. Total number of supervisor node desired by the topologies can be calculated from Table 6 which is 24. The cluster also has 24 supervisor nodes available. Other than the total number of supervisor nodes present in the cluster, all other parameters and their values are kept same as Table 10. Similar to the experiments in Section 4.1.7, a factor at a time method is used to run the experiments where one of the parameters is changed while others are held at their default values (indicated in bold at Table 10).

4.1.8.1 Effect of Batch Gap (B_G)

Batch Gap refers to the amount of time producers wait between sending of subsequent batches of tuples to Kafka. Topologies inferring complex events need to identify individual raw events within the time duration B_G so that before the next batch of tuples arrive, a decision for the presence of complex event on the current batch is made. A resource unconstrained Storm cluster has adequate number of supervisor nodes to satisfy the desired number of supervisor node requirements of all the topologies, so SPS, DPS and Isolation

Schedulers produce the same resource allocation for the topologies where each topology gets their desired number of supervisor nodes (see Table 6). Thus, average complex event inference latency (T_E) values also remain similar as seen on Figure 22. For any value of B_G , T_E achieved by all 3 schedulers remain same (~18 seconds). It is to be noted that, while SPS and Isolation Scheduler results in very similar values of T_E , values of T_E are slightly higher for DPS. This is because of the temporary deactivation overhead of topologies while using DPS that is discussed briefly Section 3.2.2. It can be also seen that, increasing the value of B_G has no effect on the T_E values obtained by any of the 3 schedulers. This is because, while the number of tuples in a batch and the service time for each tuple remains the same, the event inference latency is not impacted by a higher value of B_G which only provides a longer time window for topologies to infer complex events.

Figure 22: Effect of B_G on T_E

Similar observations can be made for the average tuple processing latency (T_T) for higher priority topologies (as seen on Figure 23) as well as for the lower priority topologies (see Figure 24). Note that, T_T is the average tuple processing latency while T_E refers to average complex event inference latency. A complex event is inferred by processing all the tuples in a batch. Thus, as expected T_E is significantly higher than T_T . SPS and Isolation Scheduler results in very similar T_T values whereas because of the additional overhead resulting from the temporary deactivation of topologies, DPS results in slightly higher values of T_T . Once again, increasing B_G values also have no effect on the T_T values as a higher value of B_G .

Figure 23: Effect of B_G on T_T (Higher Priority Topologies)

Figure 24: Effect of B_G on T_T (Lower Priority Topologies)

4.1.8.2 Effect of Service Time (S_{high} and S_{low})

Service time indicates the amount of CPU time each tuple takes to be processed on a Storm topology. As shown in Figure 25, as S_{high} increases, T_E also increases as each tuple in a batch takes longer time to be processed. Also, in line with the previous experiments, for any given value of S_{high} , T_E values for SPS and Isolation Scheduler result in a very small difference ($\sim 5\%$) whereas T_E values obtained using DPS are slightly higher ($\sim 8\%$) than that of the Isolation Scheduler. This is due to the overhead associated with the deactivation of topologies discussed earlier.

Figure 25: Effect of S_{high} on T_E

As discussed in Section 4.1.4, higher priority topologies have service time S_{high} while the lower priority topologies have a service time $S_{low} = (S_{high}/2)$. Figure 26 depicts the effect of S_{high} on average tuple processing latency (T_T) of higher priority topologies. As S_{high} increases, tuples spend longer time on the system and tuple processing latency also increases. In line with the previous experiments, for any value of S_{high} , SPS and Isolation Scheduler results in similar T_T values while DPS results in a slightly higher T_T than that of Isolation Scheduler because of the topology deactivation overhead discussed earlier. A

similar set of observations are obtained for the variation in T_T with S_{low} and the relative performance of the three schedulers for lower priority topologies (see Figure 27).

Figure 26: Effect of S_{high} on T_T (Higher Priority Topologies)

Figure 27: Effect of S_{low} on T_T (Lower Priority Topologies)

4.1.8.3 Effect of Batch Length (B_L)

Figure 28 shows that T_E increases as B_L increases. A higher value of B_L signifies more number of tuples in a batch. As discussed on Section 4.1.2, occurrence of the complex events is determined by processing all the tuples in a batch. Thus, a higher value of B_L translates to a higher T_E value. As discussed in the previous experiments, for a resource unconstrained Storm cluster, Isolation Scheduler, SPS and DPS all result in a schedule where each topology gets their desired number of supervisor nodes. As a result, T_E values obtained using all 3 schedulers are similar. While for a given B_L , SPS and Isolation Scheduler results in T_E values with a small difference ($\sim 5\%$), T_E for DPS is slightly higher ($\sim 8\%$) than that of Isolation Scheduler. This is because of the added overhead due to deactivation of topologies discussed earlier.

Figure 28: Effect of B_L on T_E

For any of the three schedulers, an increase in B_L has no effect on the average tuple processing latency T_T for higher and lower priority topologies as seen on Figure 29 and Figure 30 respectively. The reasoning behind this is already discussed on Section 4.1.7.3.

Figure 29: Effect of B_L on T_T (Higher Priority Topologies)

Figure 30: Effect of B_L on T_T (Lower Priority Topologies)

4.1.8.4 Effect of Event Factor (E_P)

Event Factor determines the proportion of batches that contains events and give rise to a complex event. Thus, increasing E_P increases the number of complex events to be inferred by the topologies. Figure 31 shows, changes in E_P values has no effect on the average complex event inference latency T_E . This is because, in a resource unconstrained Storm cluster, topologies are always provisioned with their desired number of supervisor nodes which does not change over time. In line with the previous experimental results, T_E values obtained using all 3 schedulers are similar with a smaller difference ($\sim 5\%$). For a given E_P , a slightly higher ($\sim 8\%$) T_E is achieved by DPS in comparison to the other 2 schedulers because of the deactivation overhead of topologies discussed earlier (see Figure 31).

Figure 31: Effect of E_P on T_E

Average tuple processing latency T_T also remains unaffected by E_P for both higher priority topologies (Figure 32) and lower priority topologies (Figure 33). Once again, for a given E_P , T_T obtained using DPS is slightly higher than that of Isolation Scheduler whereas T_E obtained with SPS is closer to Isolation Scheduler than achieved with DPS.

Figure 32: Effect of E_P on T_T (Higher Priority Topologies)

Figure 33: Effect of E_P on T_T (Lower Priority Topologies)

4.2 Use Case-2: Systems Subjected to the Arrival of Continuous Data

For the second use case, a stream processing platform where data producers are continuously sending data at variable rates is considered. An example of such applications is continuous mining of social network data such as, analysis of twitter streams for trend analysis and sentiment analysis [75,76]. Usually, social networking platforms expose APIs to tap on their stream data and continuously push the streams to a data analytics platform.

For example: Twitter allows developers to access the continuous twitter streams of different topics of interest (e.g. music, politics etc.) through a Twitter Firehose API [77]. Using this API, developers collect live twitter feeds and continuously push the data to a data analytics platform for analysis. A good example of such a system has been described in [78].

For Use Case 2 that corresponds to stream processing platforms subjected to the arrival of continuous streams of data, several producers are continuously pushing synthetic data to the ingestion layer and are then processed by two Storm topologies. Specific implementation details of the producers and the developed Storm topologies are discussed in the next sections.

4.2.1 Data Producers

Data producers considered in Use Case 2 are sending data streams of two types to two different Kafka topics. The two types of streams are characterized by two different subject of interests. As an example, one subject of interest can be twitter feeds about politics and the other can be twitter feeds related to music.

Like the producers that are used in Use Case 1, the producer applications for Use Case 2 are also developed using the Java programming language [25]. The producer applications continuously generate synthetic data related to 2 subject of interests that pushed to 2 Kafka topics at a given rate using the Kafka producer API [66]. The first topic is called “*Feed1*” which has 16 partitions. The other topic is called “*Feed2*” which has 8 partitions. For each topic, a producer application is created using Java programming language that pushes messages to the respective Kafka topics using the producer API. The producer applications are called *Producer1* and *Producer2*. For sending messages to multiple partitions of a topic

in parallel, Java threads are used by each producer application.

Based on the dataset available from Stanford Social Network Datasets site [79], a generic structure is developed with 4 fields. The format is

- <date-time:(String)>,<source:(String)>,<Content:(String)>,<other:(String)>

The first field contains the timestamp for, the second field contains the name of the user, the third field contains the content of the message (e.g. tweets) and the fourth field contains the platform specific meta-data information, for example the language of the message, time-zone offset etc. Such a format of data with four fields can be used in the context of various streaming data scenarios. The workload parameters used to control various aspects of the producer applications that push data to Kafka are discussed on Section 4.2.3.

4.2.2 Storm Topologies

While continuous processing of tuples can correspond to a number operations such as monitoring, analytics, ads targeting, data synchronization [80], sentiment and trend analysis of twitter feeds [75,76], the topologies in Use Case 2 are not tied to any specific computation. Instead processing time for each tuple is characterized by a parameter called service time which is discussed in Section 4.2.5.

Two Storm topologies are used to process the continuously ingested tuples from Kafka. The first Storm topology called *TP1* is used to process streams of tuples from the Kafka topic *Feed1*. The second topology, called *TP2* is used to process streams of tuples from Kafka topic *Feed2*. Figure 34 illustrates the structure of the topologies.

Figure 34: Topology Structure for TP1 and TP2

The topologies have similar structure. As seen on Figure 34, the first component in these topologies is a Kafka Spout which subscribes to a Kafka topic and emits the tuples to downstream bolts. The next component is the "Parse Bolt" which parses the csv structured tuples and forwards the contents of the <Content> field to downstream "Process Bolt". The Process bolt processes each tuple for a time characterized by the service time parameter (discussed in Section 4.2.5) and forwards the tuple to the "Dispatcher Bolt" (see Figure 34) which writes the result to a Kafka topic for the end systems (e.g. external UI) to realize.

Table 11 lists the resource provisioned for the 2 topologies. *TP1* is allocated 16 supervisor nodes as it has 16 executors per component (see Column 5) to process tuples from 16 producers sending tuples in parallel. Similarly, *TP2* is assigned 8 supervisor nodes because it has 8 concurrent threads per topology component to process tuples from 8 producers. The number of supervisor nodes and the worker processes for a given topology are kept at equal following the resource provisioning best practices provided by Storm [72].

Table 11: Topology Resource Provisioning

Topology	Desired Number of Supervisor Nodes	Desired Number of Worker Processes	Number of Topology Components	Number of Executors/ concurrent threads per topology component
<i>TP1</i>	16	16	4	16
<i>TP2</i>	8	8	4	8

4.2.3 System Configuration

The system configuration for Use Case 2 is same as used in Use Case 1 (see Section 4.1.3) where 32 nodes running on an Amazon EC2 cloud are used for setting up Storm, Zookeeper and Kafka clusters. Refer to Table 7 for an overview of the system configuration for the clusters that are used in the experiments for Use Case 2.

4.2.4 Performance Metrics

Performance of the schedulers is evaluated using two performance metrics that are described.

- Average tuple processing latency (T_T): Average tuple processing latency is the average time in seconds a tuple takes to be processed by the Storm topologies. To compute T_T of *TP1*, two timestamps are taken using `System.currentTimeMillis()` from Java Library, one after the *Producer1* starts sending a tuples at the beginning of the experiment and one after the *TP1* finishes processing all the tuples in that experiment. These two timestamps are taken by a separate application called “Perf” that is run on the local machine after it receives the respective messages from the producer and the topology. The “Perf” application then takes the difference between the two timestamps and divides the result by the total number of tuples

used in the experiment to get the average tuple processing latency for *TP1*. Average tuple processing latency for *TP2* is computed following the similar approach.

- **Scheduling Overhead (O):** Scheduling overhead is the amount of time in seconds it takes for a topology to be rescheduled at runtime when DPS is used. To measure the scheduling overhead, one time stamp is generated when a topology sends a trigger indication and another time stamp is generated when the topology is successfully rescheduled by DPS and has started processing with the new resource allocation. The difference between these two timestamps is the scheduling overhead. Recall from Section 3.3 that run time rescheduling of a topology by DPS is a three -stage process where in the first stage, a topology sends a trigger indication and PM immediately deactivates the topology and waits till the topology finishes existing tuples. In the second stage, the topology signals PM that it has finished processing the existing tuples and PM changes the priority of the topology. Finally, in the third stage, when DPS successfully reschedules the topology using the algorithm described in Section 3.3, and the PM activates the topology and the topology starts processing tuples with the new resource allocation. All the three stages contribute to the scheduling overhead.

Note that, resource management is performed by SPS in an offline fashion and SPS is invoked only during system initialization. Since SPS does not incur any scheduling overhead at runtime, *O* for SPS is zero.

4.2.5 Workload Parameters

For the experiments that are run in Use Case 2, it is assumed that producers are sending data to Kafka following the “On/Off” model [81,82]. In an On/Off model, a period of data

arrival with a low rate is followed by a period of data arrivals with a high rate. It is assumed that, among the two Kafka topics, 16 producers that are sending data to the Kafka topic *Feed1* are characterized by this On/Off model. The mean arrival rate for the tuples sent to the Kafka topic *Feed1* can be expressed by the following equation:

$$\lambda_1^M = (\lambda_1^{high} * r_1 + \lambda_1^{low} * r_2) / (r_1 + r_2) \quad (1)$$

Where

- λ_1^{high} is the high rate for the tuple arrival to topic *Feed1* using the previously described On/Off model.
- λ_1^{low} is the low rate for the tuple arrival for topic *Feed1*.
- r_1 is the proportion of tuples with rate λ_1^{high} and r_2 is the proportion of tuples with rate λ_1^{low}

Note that: $r_1 + r_2 = 1$.

In all the experiments λ_1^{low} is considered to be half of λ_1^{high} . Thus, $\lambda_1^{low} = (\frac{\lambda_1^{high}}{2})$. Fixed rates are used for λ_1^{high} and λ_1^{low} such that inter arrival times for the tuples remain fixed.

In addition to the workload parameters used in Equation 1, the following parameters are used in the experiments:

- Service Time (S_{high} and S_{low}): The parameter service time is already discussed in Section 4.1.5.
- Tuple Arrival Rate for Topic *Feed2* (λ_2): Unlike *Feed1*, tuple arrival for topic *Feed2* does not follow the On/Off model. Instead 8 producers send tuples at a fixed rate λ_2 to the topic *Feed2*.
- Total Number of Tuples Generated by Each Producer (N_T): is the total number of

tuples sent by a producer in an experiment.

Each experiment is repeated 3 times and average results are computed. Note that fixed values of workload parameters (e.g. service time, tuple arrival rate) are used in each experiment, thus close values for a given performance metric are achieved in each of the three repetitions.

4.2.6 Performance Evaluation of the Priority Based Schedulers

The experiments to evaluate the performance of the proposed priority based schedulers are run for a resource constrained Storm cluster where the available cluster resources are not sufficient to fulfill the desired resource requirements of the submitted topologies but are sufficient to meet their minimum resource requirements. Note that, using the default Isolation Scheduler results in partial scheduling or even complete starvation of the topologies depending on the resource deficit in such situations. Thus, the Isolation Scheduler is not considered during evaluation of system performance in this scenario and only the performances of SPS and DPS are evaluated.

4.2.7 Performance Evaluation for a Resource Constrained Storm Cluster

Recall from Table 11 that a total number of 24 supervisor nodes are desired by the 2 topologies used in the experiment. For the resource constrained case, the available number of supervisor nodes in the cluster is reduced to 20. For this resource constrained cluster, SPS uses static priority indication of the topologies which are provided by the developers in the topology submission time while DPS uses runtime dynamic priority indication of the topologies which are generated by the topologies at runtime based on some predefined trigger conditions.

Table 12 captures the resource provisioning of the two topologies using the SPS algorithm.

TP1 that processes tuples from Kafka topic *Feed1* is assigned a higher priority while *TP2* that processes tuples from Kafka topic *Feed2* is assigned a lower priority (see Column 2). Column 5 of Table 12 shows that *TP1* has a higher number of concurrent executors per topology components (e.g. 16) compared to that of *TP2* (e.g. 8). Thus, a higher number of supervisor nodes is assigned to *TP1* compared to *TP2* (see Column 5 of Table 12). The supervisor nodes are allocated to the topologies using the SPS algorithm discussed in Section 3.2. The rationale behind selecting the minimum and desired number of supervisor nodes are similar to that presented in Section 4.1.7.

Table 12: Topology Resource Provisioning using SPS

Topology Name	Priority Level (PR)	Desired Number of Supervisor Nodes	Minimum Number of Supervisor Nodes	Number of Supervisor Nodes Allocated
<i>TP1</i>	1	16	8	13
<i>TP2</i>	2	8	4	7

For DPS, tuple arrival rates at the Kafka topics are used to determine the runtime trigger indications. Kafka allows to query tuple arrival rate at a certain topic using its jmxtrans API [83]. Using this API, topologies can determine the tuple arrival rate at a given topic from which, they are consuming tuples and can notify the PM if the rate remains above a certain predefined threshold for a predefined amount of time (e.g. 3 seconds for the experiments in this use case). The logic for this is implemented by in Counter Bolt component (see Section 4.2.2) of *TP1* and *TP2*.

As discussed in Section 4.2.4, producers that send tuples to *Feed1* follows the On/Off model where a portion of tuples are sent with a high rate and portion of tuples are sent with a low rate. Topology *TP1* processes tuples from the Kafka topic *Feed1* and upon

experiencing a higher rate of tuple arrival, it sends a trigger indication to PM so that it can assume a higher priority value. *TP2* processes tuples from topic *Feed2* which experiences a fixed rate of tuple arrivals pushed by the producers.

Table 13 presents the resource provisioning data for the topologies when DPS is used. Topologies are assigned the same priority level by PM when submitted to the cluster (see Column 2 of Table 13). At runtime topologies send trigger indications based on the arrival rate of tuples on the Kafka topics. Based on the trigger indications, topology priorities are changed by the PM (Column 3). The supervisor nodes that are allocated to the topologies are calculated using the DPS algorithm described in Section 3.3.

Table 13: Topology Resource Provisioning using DPS

Topology	Initial Priority Level (PR)	Priority Level (PR) when Trigger ON	Number of Supervisor Nodes Allocated (Event Trigger OFF)	Number of Supervisor Nodes Allocated (Event Trigger ON)
<i>TP1</i>	2	1	13	16
<i>TP2</i>	2	1	7	4

A summary of the parameters used in the experiments is captured on Table 14. The parameters are divided into two categories: workload and system parameters. The experiments are run following a factor at a time method where one of the parameters is changed while others are held at their default values (indicated in bold at Table 14).

Table 14: Summary of Parameters Used in the Experiments

Parameter Type	Parameter	Value
Workload	High Arrival Rate for topic Feed1 (λ_1^{high})	(40, 53.33 , 80) tuples/sec
	Proportion of tuples with high rate for topic Feed1 (r_1)	{0.2, 0.5 ,0.8}
	Tuple Arrival rate for Kafka topic Feed2 (λ_2)	10 tuples/sec
	Service Time for Higher Priority Topology (S_{high})	0.4 seconds
	Service Time for Lower Priority Topology (S_{low})	($S_{high}/2$)
	Total Number of tuples generated by each producer (N_T)	5000
System	Total Number of Topologies	2
	Total Number of Supervisor Nodes in the Cluster	20
	Total Number of Supervisor Nodes Desired by the Topologies	24
	Number of Kafka Topics	2
	Number of Producers for topic Feed1	16
	Number of Producers for topic Feed2	8

4.2.7.1 Effect of High Arrival Rate for Topic Feed1 (λ_1^{high}):

From Figure 35, it can be seen that as λ_1^{high} increases, average tuple processing latency (T_T) for higher priority topology also increases for both SPS and DPS. An increase in λ_1^{high} signifies more tuples per second being pulled by the topology. While the service time S_{high} remaining fixed, an increase in the number of tuples increases resource contention among the executors of the high priority topology *TP1*. For any value of λ_1^{high} , T_T achieved by DPS is lower than that of SPS. This is because, DPS allocates the higher priority topology with more resources than SPS (see Table 12 and Table 13) when topologies are experiencing high rate of tuple arrival. With the proportion of tuples with high rate for topic Feed1 (r_1) and Proportion of tuples with low rate for topic Feed2 (r_2) remaining equal (e.g. 0.5), *TP1* will process half of the total number tuples with a higher number of supervisor nodes while using DPS in comparison to the case in which SPS is used.

Figure 35: Effect of λ_1^{high} on T_T (Higher Priority Topology)

From Figure 36, it can be seen that SPS does not incur any scheduling overhead as expected while scheduling overhead for DPS increases as λ_1^{high} increases. Recall from Section 4.2.5 that the scheduling overhead includes the existing tuple processing time by the topology after deactivation and the new resource allocation time by DPS. Increasing λ_1^{high} increases number of tuples in the system and thus the existing tuple processing time after topology deactivation for *TP1* also increases and so does the scheduling overhead *O*. The highest *O* is observed when λ_1^{high} is 80 (tuples/sec) and the lowest value for *O* is observed when λ_1^{high} is 40 (tuples/sec).

Figure 36: Effect of λ_1^{high} on *O*

Figure 37 captures the effect of λ_1^{high} on the performance of the lower priority topology *TP2* that processes tuples from Kafka topic Feed2. Note that, λ_2 , the arrival rate for *TP2* is fixed and does not change with time. It is observed that, T_T for lower priority topology achieved using SPS is much lower than that achieved using DPS. This can be understood by referring to Table 12 and 13 where it can be seen that while DPS awards a topology in

a trigger ON state with more supervisor nodes (16 compared to 13 when using SPS), lower priority topologies are assigned a lower number of supervisor nodes (4 compared to 7 when using SPS). The lower number of supervisor nodes leads to a higher value of T_T for SPS. Another important observation from Figure 37 is that T_T using DPS increases as λ_1^{high} increases. This is because of increase in the scheduling overhead reported in Figure 36. It is important to understand that, when DPS allocates higher priority topology with a higher number of supervisor nodes, it also reschedules lower priority topologies and assigns them a lower number of supervisor nodes. Thus, the rescheduling overhead affects both the higher and the lower priority topologies.

Figure 37: Effect of λ_1^{high} on T_T (Lower Priority Topology)

4.2.7.2 Effect of Proportion of Tuples with High Arrival Rate for Topic Feed1 (r_1):

Figure 38 illustrates the effect of r_1 on average tuple processing latency T_T for higher priority topology. From Figure 38, it can be seen T_T for both SPS and DPS increase as r_1 increases. This is because, a higher value of r_1 signifies a higher proportion of tuples with a high arrival rate and that increases resource contention. From Figure 38, it is also seen

that DPS performs better than SPS and the performance difference increases with r_1 . This is because, using DPS for a higher value of r_1 , *TPI* processes more tuples with higher number of supervisor nodes leading to a lower T_T in comparison to SPS for which the number of supervisor nodes allocated to *TPI* remains fixed.

Figure 38: Effect of r_1 on T_T (Higher Priority Topology)

The overhead incurred using DPS seems insensitive to r_1 (see Figure 39). The rationale behind such a behavior is presented. Increasing r_1 signifies the change in the proportion of tuples with rate λ_1^{high} . Recall that a trigger condition is reported by the topology *TPI* if it experiences tuple arrival rates has crossed a threshold for 3 consecutive seconds. Thus, the number of tuples that can be accumulated in the system in these 3 seconds does not change with r_1 and the existing tuple processing time after topology deactivation for *TPI* does not change as well. Thus, scheduling overhead O remains insensitive to r_1 . Also, it can be seen from Figure 39 that SPS does not incur any scheduling overhead which expected as SPS does not perform run time rescheduling of topologies.

Figure 39: Effect of r_1 on O

Increasing r_1 does not have any effect on T_T achieved by the lower priority topology using DPS as seen on Figure 40. This is because, scheduling overhead incurred using DPS is unaffected by r_1 which is already discussed for Figure 40. It can be also seen that T_T achieved using DPS is higher than that of SPS. This is expected as SPS does not have any rescheduling overhead.

Figure 40: Effect of r_1 on T_T (Lower Priority Topology)

Chapter 5: Conclusions

This chapter provides a summary of the thesis and presents concluding remarks.

Directions for future research are also included.

5.1 Summary and Conclusions

This thesis addresses resource management techniques for resource constrained Storm clusters where the available cluster resources are not sufficient to satisfy desired resource requirements for the submitted Storm topologies. The default scheduler provided by Storm, called Isolation Scheduler results in partial scheduling or even complete starvation of topologies when used for such resource constrained Storm clusters. To alleviate this problem, two schedulers are proposed to schedule Storm topologies for such resource constrained clusters. The first scheduler called Static Priority Scheduler (SPS) uses static priority for the topologies. SPS schedules topologies by proportionally allocating available limited resources where topologies with higher priority receives a higher proportion of resources. The priority indications are provided by the developers when submitting topologies to the cluster and cannot be changed at runtime. Given a minimum number of resources are present in the cluster, SPS can prevent starvation of the topologies. The second scheduler is called Dynamic Priority Scheduler (DPS) and it uses dynamic priority indications from the topologies. The priority indications are determined by the topologies at runtime based on some predefined trigger conditions. DPS schedules topologies based on these dynamic priority indications where topologies with higher priority receives a higher proportion of resources. Although DPS gives rise to a higher performance for the higher priority topologies, it can lead to a deterioration of performance for the lower

priority topologies that may face a temporary starvation during the time when the trigger condition remains active.

To evaluate the performance of the proposed schedulers, prototype systems were developed using Java programming language. Experimental results demonstrate the effectiveness of each of these scheduling algorithms. The two Storm schedulers were developed using the Storm *IScheduler* [51] Plugin. Topologies and producer applications were developed using the Storm and Kafka APIs respectively. The experiments were run considering two use cases: Use Case 1 and Use Case 2. A summary of key observations and concluding remarks are presented for the two use cases.

5.1.1 Use Case 1

The first use case considers batch arrival of data where data producers are sending data in small batches in fixed time intervals. Four Storm topologies are used in this use case. Clusters are configured on Amazon EC2 cloud infrastructure (see Section 4.1.2). Experiments are run considering two scenarios. In the first scenario, a resource constrained cluster is considered. In such a cluster, Isolation Scheduler fails to schedule all the topologies thus, only the performance of SPS and DPS are evaluated and compared. Various system and workload parameters are used for the experiments. For all the experiments that are conducted in this scenario, performance of higher priority topologies is better using DPS than using SPS. This is because, DPS allocates higher priority topologies more resources than SPS. Conversely, performance of the lower priority topologies is much worse using DPS than SPS. This is because of the temporary starvation of the lower priority topologies that occurs when DPS is used. The starving lower priority topologies cease to starve when topologies currently running at a higher priority level lower

their priority based on trigger conditions. Lower priority topologies can then process the backlog of tuples that were accumulated during the starvation period. The impact of various workload and system parameters on the performance of the schedulers are summarized next.

- Increasing B_G does not have any effect on T_E and T_T achieved by higher priority topologies for any of the schedulers. This is because, increasing B_G provides a longer time window for the high priority topologies to detect complex events that does not affect T_T . Also, it is observed that, T_E and T_T for higher priority topologies using DPS is lower compared to that of SPS because DPS allocates higher priority topologies with more resources. T_T achieved by lower priority topologies increase when B_G is increased using DPS. This is because an increase in B_G signifies higher temporary starvation period for lower priority topologies.
- An increase in service time increases CPU time for each tuple. Thus, T_E and T_T increases with an increase in service time. T_E and T_T for higher priority topologies using DPS are lower compared to that of SPS. It is observed that for higher values of service time, the difference in T_E and T_T values between SPS and DPS is higher. This is because while other parameters remain fixed, an increase in service time signifies more contention for CPU resources. Also, using both SPS and DPS, T_T achieved by lower priority topologies increases when service time is increased because of the increase in tuple processing time for the lower priority topologies.
- Increasing batch length (B_L) increases T_E for higher priority topologies for both SPS and DPS. A higher value of B_L signifies more number of tuples in a batch that needs to be processed to detect events. The observation that T_E and T_T for higher

priority topologies using DPS is lower compared to that of SPS is consistent with other experiments. For lower priority topologies, T_T achieved using DPS is higher than that of SPS because of the temporary starvation of the lower priority topologies.

- T_E achieved by higher priority topologies does not change when Event Factor (E_P) is increased. In line with previous experiments, for a given E_P , T_E for higher priority topologies using DPS is lower compared to that of SPS. Increasing E_P increases the number of complex events that are to be inferred by the higher priority topologies. Thus, for higher values of E_P , DPS allocates high priority topologies with more resources for a higher number of batches and the difference in T_T values between SPS and DPS becomes higher. Conversely, T_T achieved by lower priority topologies using DPS is much higher than that of SPS and an increase in E_P increases T_T using DPS. This is because lower priority topologies starve temporarily for a higher duration of time when E_P values are higher.

Next, a resource unconstrained cluster is considered for the second scenario of Use Case 1 to compare the performance of SPS and DPS with the default Isolation Scheduler. The same workload and system parameters as well as the performance metrics from the resource constrained case were used while the cluster was provided with enough supervisor nodes to satisfy the desired resource requirement for all the topologies. The same set of experiments from the resource constrained case were repeated for this new resource allocation. For all the experiments, SPS, DPS and Isolation Scheduler perform provide a comparable performance while DPS results in a

slightly higher T_E and T_T values than Isolation Scheduler because of rescheduling overhead.

5.1.2 Use Case 2

The second use case considers data producers continuously sending tuples with one set of data producers sending tuples with variable arrival rates following the On/Off model (see Section 4.2.4). For Use Case 2, a resource constrained cluster is considered and performance of SPS and DPS are evaluated. The impact of various workload and system parameters on the performance of the schedulers are summarized next.

- With an increase in λ_1^{high} , T_T for the higher priority topology is increased for both SPS and DPS. This is because, while other parameters remain fixed, an increase in λ_1^{high} signifies more tuples in the system which increases resource contention among the executors of the high priority topology. It is also observed that, for the high priority topology, T_T achieved using DPS is lower than that of SPS. This is because DPS allocates the high priority topology with higher number of resources compared to SPS. Also, SPS does not perform run time rescheduling of the topologies and thus, does not incur any scheduling overhead. Scheduling overhead (O) for DPS is increased however with an increase in λ_1^{high} . When a topology sends a trigger indication to PM, PM deactivates the topology and before changing the priority it waits for the topology to finish processing existing tuples that are accumulated in the system. A higher λ_1^{high} increases the number of tuples that are accumulated in the system and thus the topology takes a longer time to finish processing these tuples. This increases the scheduling overhead. Because of this,

T_T achieved by lower priority topologies also increases with an increase in λ_l^{high} when using DPS.

- Changing r_1 signifies a change in the proportion of tuples with high arrival rate processed by the higher priority topology. While using DPS, at higher values of r_1 , the high priority topology processes a higher proportion of tuples with a higher number of resources in comparison to that of SPS, for which the number of supervisor nodes allocated to the high priority topology remains fixed. Thus, DPS performs better than SPS and the performance difference increases with r_1 . For the lower priority topology, DPS results in a higher T_T than SPS because the number of resources that gets allocated to the lower priority topology by DPS is lower than that of SPS.

5.2 Future Research

This section presents a number of issues that are worthy of further investigation.

- It has been observed that DPS is associated with scheduling overhead that impacts system performance of the topologies, especially on systems that are subject to continuous arrival of tuples. Investigation of techniques to minimize scheduling overhead of DPS can form a direction for future research.
- Storm clusters that are considered in this thesis are comprised of a homogenous set of machines. The limited resources in the resource constrained Storm clusters are shared among the topologies with a proportional share based scheduling algorithm. Proportionally sharing resources in a non-homogenous storm cluster can result in over utilization and underutilization of some machines. Extending the scheduling

techniques presented in this thesis to such non-homogeneous systems warrants investigation.

- Investigation of resource management techniques for the other layers in a stream processing platform (e.g. data ingestion layer and data producer layer) can form an interesting future research direction.
- Effects of additional parameters, such as tuple arrival rate for topic Feed2 (λ_2) warrants further investigation. Additionally, this thesis used two priority levels for the experiments. Investigating systems with a higher number of priority levels is worthy of research.

References

- [1] IBM, "Bringing Big Data to the Enterprise", [Online]. Available at <https://www-01.ibm.com/software/data/bigdata/what-is-big-data.html>. [Accessed June 2016].
- [2] SAS, "Big Data in Big Companies", [Online]. Available at <http://www.sas.com/resources/asset/Big-Data-in-Big-Companies-Executive-Summary.pdf>. [Accessed June 2016].
- [3] Forbes, "How Big Data Is Changing Healthcare", [Online]. Available at <http://www.forbes.com/sites/bernardmarr/2015/04/21/how-big-data-is-changing-healthcare/#431e59c132d9>. [Accessed June 2016].
- [4] Gartner, "3D Data Management: Controlling Data Volume, Velocity and Variety" [Online]. Available at <http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf>. [Accessed June 2016].
- [5] Apache Hadoop, "Apache Hadoop", [Online]. Available at <http://hadoop.apache.org/>. [Accessed June 2016].
- [6] Apache Spark, "Apache Spark", [Online]. Available at <http://spark.apache.org/>. [Accessed June 2016].
- [7] Apache Storm, "Apache Storm", [Online]. Available at <http://storm.apache.org/>. [Accessed June 2016].
- [8] Apache Storm, "Apache Storm Powered By", [Online]. Available at <http://storm.apache.org/Powered-By.html>. [Accessed June 2016].
- [9] Apache Storm, "Storm-Scheduler", [Online]. Available at <http://storm.apache.org/releases/2.0.0-SNAPSHOT/Storm-Scheduler.html>. [Accessed June 2016].
- [10] R. Chakraborty and S. Majumdar, "A priority based resource scheduling technique for multitenant storm clusters," *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2016, pp. 1-6.
- [11] M. Kira, P. Murphy, I. Monga and J. Dugan, "Lambda architecture for cost-effective batch and speed big data processing," *IEEE International Conference on Big Data (Big Data)*, 2015, pp. 2785-2792.

- [12] J. Park and S. Chi, "An implementation of a high throughput data ingestion system for machine-logs in manufacturing industry," *Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2016, pp. 117-120.
- [13] R. Ranjan, "Streaming big data processing in datacenter clouds," *IEEE Cloud Computing*, vol. 1, no. 1, pp. 78-83, 2014.
- [14] N. Marz, *Big Data*. NY: Manning Publications Co, 2015. [Online].
<http://www.semantikoze.com/blog/lambda-architecture-velocity-volume-big-data-hadoop-storm/>
- [15] Apache Kafka, "Apache Kafka", [Online]. Available at <https://kafka.apache.org/>. [Accessed June 2016].
- [16] M. Stonebraker, U. Çetintemel and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec*, vol. 34, no. 4, pp. 42-47, 2005.
- [17] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker and S. Zdonik. "High-availability algorithms for distributed stream processing", *21st International Conference on Data Engineering (ICDE)*, 2005, pp. 779-790.
- [18] Apache Zookeeper. "Apache Zookeeper", [Online]. Available at <https://zookeeper.apache.org/>. [Accessed June 2016].
- [19] Apache Storm, "Fault Tolerance", [Online]. Available at <http://storm.apache.org/releases/current/Fault-tolerance.html>. [Accessed June 2016].
- [20] B. Lohrmann, D. Warneke and O. Kao, "Nephele streaming: stream processing under QoS constraints at scale", *Cluster Computing*, vol. 17, no. 1, pp. 1-18, 2013.
- [21] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal* , vol. 12, no. 2, pp. 120-139, 2003.
- [22] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein and J. Widom, "STREAM: the stanford stream data manager," *2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)* , 2003, pp. 665-676.
- [23] M. Balazinska, H. Balakrishnan and M. Stonebraker, "Contract-based load management in federated distributed systems," *1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, 2004, pp. 15-15.
- [24] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing and S. Zdonik, "The

- design of the borealis stream processing engine," *2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, 2005, pp. 872-884.
- [25] Oracle, "Java", [Online]. Available at <https://www.oracle.com/java/index.html>. [Accessed June 2016].
- [26] Apache Storm, "Command Line Client", [Online]. Available at <http://storm.apache.org/releases/1.0.1/Command-line-client.html>. [Accessed June 2016].
- [27] P. Bellavista, A. Corradi, A. Reale and N. Ticca, "Priority-based resource scheduling in distributed stream processing systems for big data applications," *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC '14)*, 2014, pp. 147-156.
- [28] V. Cardellini, V. Grassi, F. L. Presti and M. Nardelli, "Optimal operator placement for distributed stream processing applications," *10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*, 2016, pp. 69-80.
- [29] M. Hosseini, Z. Hong, R. Farivar, R. Campbell and B. Peng, "R-Storm: Resource-aware scheduling in storm," *16th Annual Middleware Conference (Middleware '15)*, 2015, pp. 76-87.
- [30] L. Aniello, R. Baldoni and L. Querzoni, "Adaptive online scheduling in Storm," *7th ACM international conference on conference on Distributed event-based systems*, 2013, pp. 207-218.
- [31] J. S. v. d. Veen, B. v. d. Waaij, E. Lazovik, W. Wijbrandi and R. J. Meijer, "Dynamically scaling apache storm for the analysis of streaming data," *IEEE First International Conference on Big Data Computing Service and Applications (BigDataService)*, 2015, pp. 154-161.
- [32] B. Peng, M. Hosseini, Z. Hong, R. Farivar and R. Campbell, "R-Storm: Resource-aware scheduling in storm," in *16th Annual Middleware Conference (Middleware '15)*, 2015, pp. 149-161.
- [33] J. Xu, Z. Chen, J. Tang and S. Su, "T-Storm: Traffic-aware online scheduling in storm," *IEEE 34th International Conference on Distributed Computing Systems (ICDCS 13)*, 2014, pp. 535-544.
- [34] T. Jian and J. Xu, "A predictive scheduling framework for fast and distributed stream data processing," *IEEE International Conference on Big Data (Big Data)*, 2015, pp. 333-338.

- [35] L. Fischer and A. Bernstein, "Workload scheduling in distributed stream processors using graph partitioning," *IEEE International Conference on Big Data (Big Data)*, 2015, pp. 124-133.
- [36] Y. Xing, J. Hwang, U. Çetintemel and S. Zdonik, "Providing resiliency to load variations in distributed stream processing," *32nd International Conference on Very Large Data Bases (VLDB '06)*, 2006, pp. 775-786.
- [37] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K. Wu and L. Fleischer, "SODA: An optimizing scheduler for large-scale stream-based distributed computer systems," *9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08)*, 2008, pp. 306-325.
- [38] P. Bellavista, A. Corradi and A. Reale, "Design and implementation of a scalable and QoS-aware stream processing framework," *IEEE International Conference on Green Computing and Communications (GreenCom)*, 2012, pp. 458-467.
- [39] Y. Xing, S. Zdonik and J. Hwang, "Dynamic load distribution in the borealis stream processor," *21st International Conference on Data Engineering (ICDE '05)*, April 2005, pp. 791-802.
- [40] Y. Ahmad and U. Çetintemel, "Network-aware query processing for stream-based applications," *13th International Conference on Very Large Data Bases (VLDB '04)*, 2004, pp. 456-467.
- [41] Y. Zhou, B. Ooi, K. Tan and J. Wu, "Efficient dynamic operator placement in a locally distributed continuous query system," *Confederated International Conference on The Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I (ODBASE'06/OTM'06)*, 2006, pp. 54-71.
- [42] D. Eager, E. D Lazowska and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '85)*, 1985, pp. 1-3.
- [43] J Kay and P Lauder, "A fair share scheduler," *Communications of the ACM*, vol. 31, no. 1, pp. 44-55, 1988.
- [44] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," *15th International Conference on Job scheduling Strategies for Parallel Processing (JSSPP'10)*, 2010, pp. 110-131.

- [45] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal and D. Ryaboy, "Storm@twitter," *ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, 2014, pp. 147-156.
- [46] Wikipedia, "Activity Diagram", [Online]. Available at https://en.wikipedia.org/wiki/Activity_diagram. [Accessed June 2016].
- [47] Apache Storm, "REST API", [Online]. Available at <http://storm.apache.org/releases/0.9.6/STORM-UI-REST-API.html>. [Accessed June 2016].
- [48] Apache Storm, "Acking", [Online]. Available at <http://storm.apache.org/releases/current/Acking-framework-implementation.html>. [Accessed June 2016].
- [49] Apache Storm, "Nimbus Client", [Online]. Available at <http://storm.apache.org/releases/current/javadocs/org/apache/storm/generated/NimbusClient.html>. [Accessed June 2016].
- [50] Apache Storm, "Download Storm", [Online]. Available at <http://www.apache.org/dyn/closer.lua/storm/apache-storm-0.9.6/apache-storm-0.9.6.tar.gz>. [Accessed June 2016].
- [51] Apache Storm, "javadocs", [Online]. Available at <https://storm.apache.org/releases/0.9.7/javadocs/backtype/storm/scheduler/IScheduler.html>. [Accessed June 2016].
- [52] MySQL. [Online]. Available at <https://www.mysql.com/>. [Accessed June 2016].
- [53] JDBC. [Online]. Available at <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>. [Accessed June 2016].
- [54] T. Sapna, A. Amit and M. Piyush, "A conceptual framework for IoT-based healthcare system using cloud computing," *6th International Conference Cloud System and Big Data Engineering (Confluence)*, 2016, pp. 503-507.
- [55] A. Abrar, S. Mohamed, N. Haja and M. B. Seyed, "A cross-layer framework for sensor data aggregation for IoT applications in smart cities," in *IEEE International Smart Cities Conference (ISC2)*, 2016, pp. 1-6.
- [56] L. Charissis and C. Fehling Tobias Haberle, "The connected car in the cloud: A Platform for Prototyping Telematics Services," *IEEE Software*, vol. 32, no. 6, pp. 11-17, 2015.

- [57] P. Comitz and A. Kersch, "Aviation analytics and the Internet of Things," *Integrated Communications Navigation and Surveillance (ICNS)*, 2016, pp. 72-80.
- [58] Intel, "Using real-time data to improve efficiency and reduce total cost of ownership", [Online]. Available at:
<http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/real-time-data-improves-efficiency-reduces-ownership-cost-paper.pdf>. [Accessed August 2016].
- [59] HP Labs, "Innovating for the environment", [Online]. Available at
<http://www.hpl.hp.com/environment/datacenters.html>. [Accessed June 2016].
- [60] T. P. Parthasarathy, S. Chatterjee and N. Krishna, "Integrated wireless sensor network for large scale intelligent systems," *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2013, pp. 1849-1854.
- [61] datacenterknowledge, "The Facebook Data Center FAQ", [Online]. Available at
<http://www.datacenterknowledge.com/the-facebook-data-center-faq-page-2/>. [Accessed June 2016].
- [62] Bloomberg, "5 Numbers That Illustrate the Mind-Bending Size of Amazon's Cloud", [Online]. Available at <http://www.bloomberg.com/news/2014-11-14/5-numbers-that-illustrate-the-mind-bending-size-of-amazon-s-cloud.html>. [Accessed June 2016].
- [63] ashraetcs, "IT Equipment Thermal Management and Controls", [Online]. Available at
https://tc0909.ashraetcs.org/documents/ASHRAE%202012%20IT%20Equipment%20Thermal%20Management%20and%20Controls_V1.0.pdf. [Accessed June 2016].
- [64] Google, "Overview of Internet of Things", [Online]. Available at
<https://cloud.google.com/solutions/iot-overview>. [Accessed June 2016].
- [65] Amazon Web Services "How the AWS IoT Platform Works", [Online]. Available at
<https://aws.amazon.com/iot/how-it-works/>. [Accessed June 2016].
- [66] Kafka, Apache, "Kafka Producer API", [Online]. Available at
<https://kafka.apache.org/090/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html>. [Accessed June 2016].
- [67] UCI Machine Learning Repository. [Online]. <https://archive.ics.uci.edu/ml/datasets.html>
- [68] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.
- [69] Amazon Web Services. (2016, May) S3. [Online]. <https://aws.amazon.com/s3/>

- [70] Twitter, "Handling five billion sessions a day – in real time". [Online]. Available at <https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time>. [Accessed June 2016].
- [71] Apache Storm, "Storm kafka integration", [Online]. Available at <http://storm.apache.org/releases/2.0.0-SNAPSHOT/storm-kafka.html>. [Accessed June 2016].
- [72] Apache Storm, "FAQ", [Online]. Available at <http://storm.apache.org/releases/1.0.0/FAQ.html>. [Accessed June 2016].
- [73] Amazon Web Serrvices. [Online]. <https://aws.amazon.com/ec2/>
- [74] Apache Kafka, "Documentation", [Online]. Available at <https://kafka.apache.org/documentation>. [Accessed June 2016].
- [75] A. Rahnema and A. Hossein, "Distributed real-time sentiment analysis for big data social streams," *International Conference on Control Decision and Information Technologies (CoDIT)*, 2014, pp. 206-214.
- [76] T. Chardonens, P. Cudre-Mauroux, M. Grund and B. Perroud, "Big data analytics on high velocity streams: a case study," *IEEE International Conference on Big Data*, 2013, pp 172-180.
- [77] Twitter, "Twitter Firehose", [Online]. Available at <https://dev.twitter.com/streaming/firehose>. [Accessed June 2016].
- [78] serrate, "Analysis of twitter streams with Kafka and Storm", [Online]. Available at <http://www.serrate.net/2016/01/05/analysis-of-twitter-streams-with-kafka-and-storm/>. [Accessed June 2016].
- [79] SNAP, "476 million Twitter tweets", [Online]. Available at <https://snap.stanford.edu/data/twitter7.html>. [Accessed June 2016].
- [80] Apache Storm, "Use Storm - Apache Storm - The Apache Software Foundation", [Online]. Available at <http://storm.apache.org/releases/current/Powered-By.html>. [Accessed June 2016].
- [81] T. Repantis, X. Gu and V. Kalogeraki, "QoS-aware shared component composition for distributed stream processing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 7, pp. 968-982, 2009.

- [82] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid, "Scheduling for shared window joins over data streams," *29th International Conference on Very Large Data Bases (VLDB '03)*, 2003, pp. 297-308.
- [83] jmxterm. [Online]. Available at <https://cwiki.apache.org/confluence/display/KAFKA/jmxterm+quickstart>. [Accessed June 2016].
- [84] Wikipedia, "Java", [Online]. Available at [https://en.wikipedia.org/wiki/Java_\(programming_language\)](https://en.wikipedia.org/wiki/Java_(programming_language)). [Accessed June 2016].
- [85] R. Berta, L. Scalise, G. Revel, A. Gloria, G. Orlandi and L. Pescosolido, "An IoT-inspired cloud-based web service architecture for e-Health applications," *IEEE International Smart Cities Conference (ISC2)*, 2016, pp 15-29.
- [86] VMWARE, [Online]. Available at <https://www.vmware.com>. [Accessed June 2016].