CAPITAL UNIVERSITY OF SCIENCE AND
TECHNOLOGY, ISLAMABAD


# A Scheduler for Efficient Scheduling of Stream Processing Tasks on Computational Clusters

by

Asif Muhammad

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the

Faculty of Computing
Department of Computer Science

2023

# A Scheduler for Efficient Scheduling of Stream Processing Tasks on Computational Clusters

By

Asif Muhammad

(DCS151003)

**Dr. Jerry Chun-Wei Lin, Professor**

**Western Norway University of Applied Sciences, Bergen, Norway**

**(Foreign Evaluator 1)**

**Dr. Radu Prodan, Professor**

**University of Klagenfurt, Innsbruck, Austria**

**(Foreign Evaluator 2)**

**Dr. Muhammad Abdul Qadir**

**(Thesis Supervisor)**

**Dr. Abdul Basit Siddiqui**

**(Head, Department of Computer Science)**

**Dr. Muhammad Abdul Qadir**

**(Dean, Faculty of Computing)**

**DEPARTMENT OF COMPUTER SCIENCE**

**CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**ISLAMABAD**

**2023**

*Dedicated to my parents and family*

# CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
# ISLAMABAD

Expressway, Kahuta Road, Zone-V, Islamabad
Phone:+92-51-111-555-666 Fax: +92-51-4486705
Email: info@cust.edu.pk Website: https://www.cust.edu.pk


## CERTIFICATE OF APPROVAL

This is to certify that the research work presented in the thesis, entitled "**A Scheduler for Efficient Scheduling of Stream Processing Tasks on Computational Clusters**" was conducted under the supervision of **Dr. Muhammad Abdul Qadir**. No part of this thesis has been submitted anywhere else for any other degree. This thesis is submitted to the **Department of Computer Science, Capital University of Science and Technology** in partial fulfillment of the requirements for the degree of Doctor in Philosophy in the field of **Computer Science.** The open defence of the thesis was conducted on **May 05, 2023**.

**Student Name :** Asif Muhammad (DCS151003)

---

The Examination Committee unanimously agrees to award PhD degree in the mentioned field.

**Examination Committee :**

(a) External Examiner 1: Dr. Muazzam A. Khan Khattak,
Professor
QAU, Islamabad

(b) External Examiner 2: Dr. Ejaz Ahmed
Associate Professor
FAST-NUCES, Islamabad

(c) Internal Examiner : Dr. Nayyer Masood
Professor
CUST, Islamabad

**Supervisor Name :** Dr. Muhammad Abdul Qadir
Professor
CUST, Islamabad

**Name of HoD :** Dr. Abdul Basit Siddiqui
Associate Professor
CUST, Islamabad

**Name of Dean :** Dr. Muhammad Abdul Qadir
Professor
CUST, Islamabad

# AUTHOR'S DECLARATION

I, **Asif Muhammad (Registration No. DCS151003)**, hereby state that my PhD thesis titled, '**A Scheduler for Efficient Scheduling of Stream Processing Tasks on Computational Clusters'** is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/ world.

At any time, if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my PhD Degree.

05 **(Asif Muhammad)**

Dated: May, 2023 Registration No: DCS151003

# PLAGIARISM UNDERTAKING

I solemnly declare that research work presented in the thesis titled "**A Scheduler for Efficient Scheduling of Stream Processing Tasks on Computational Clusters**" is solely my research work with no significant contribution from any other person. Small contribution/ help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/ cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of PhD Degree, the University reserves the right to withdraw/ revoke my PhD degree and that HEC and the University have the right to publish my name on the HEC/ University Website on which names of students are placed who submitted plagiarized thesis.

**(Asif Muhammad)**

Dated: 05 May, 2023

Registration No: DCS151003

# *List of Publications*

It is certified that following publication(s) have been made out of the research work that has been carried out for this thesis:-

1. **Muhammad, A.** and Qadir, M.A., 2022. MF-Storm: a maximum flow-based job scheduler for stream processing engines on computational clusters to increase throughput. *PeerJ Computer Science*, 8, p.e1077.

2. **Muhammad, A.** and Aleem, M., 2021. BAN-storm: a bandwidth-aware scheduling mechanism for stream jobs. *Journal of Grid Computing*, 19(3), pp.1-16.

3. **Muhammad, A.** and Aleem, M., 2021. A3-Storm: topology-, traffic-, and resource-aware storm scheduler for heterogeneous clusters. *The Journal of Supercomputing*, 77(2), pp.1059-1093.

4. **Muhammad, A.**, Aleem, M. and Islam, M.A., 2021. TOP-Storm: A topology-based resource-aware scheduler for Stream Processing Engine. *Cluster Computing*, 24(1), pp.417-431.

**(Asif Muhammad)**
Registration No: DCS151003

# *Acknowledgement*

I want to begin by expressing my gratitude to the Almighty Allah for His divine guidance and blessings. His unwavering support, goodness, and kindness have been instrumental in helping me achieve my goals. It is through His blessings that I have been granted the motivation, passion, and diligence to plan and execute my dreams, turning them into reality.

I would like to extend my heartfelt appreciation to my supervisor, Dr. Muhammad Abdul Qadir, for his invaluable guidance and patience throughout my research work. I am also immensely grateful to all the professors at Capital University of Science and Technology (CUST) who have contributed to my academic development. In particular, I am at a loss for words to express my gratitude to Dr. Muhammad Aleem, whose constant motivation, guidance, support, and encouragement bolstered my morale during moments of uncertainty.

I would like to extend my deep appreciation and thanks to my beloved parents, wife, brothers, and sisters for their unwavering encouragement and prayers throughout my PhD studies. I am grateful to my late father, whose dream I am striving to fulfill. I thank my mother for her heartfelt prayers for my success, and I am indebted to my wife for her unending support and inspiration. I am also thankful to my wife for taking care of our children while I dedicated myself to my PhD studies. Lastly, I want to express my gratitude to my brothers and sisters for their love, care, and unwavering support as a family.

Finally, I would like to express my thanks and offer my regards to all those who have supported me in any capacity, especially Mr. Ahmad Nawaz and Ms. Raabia Asif, whose assistance was invaluable during the completion of this thesis.

**(Asif Muhammad)**

# *Abstract*

Over the past few decades, there has been a substantial surge in data in diverse domains, including medical and healthcare, transportation, finance, agriculture, and the energy sector. This exponential growth in data has become a major focus for the scientific community, as efficiently extracting relevant information from this vast volume of data has become a significant concern. During the last two decades, hardware resources have continuously improved in terms of computational power, leading to an increased demand for real-time processing capabilities. As a result, the need to process data immediately as it is generated has become increasingly crucial in various applications.

Within distributed computing, the utilization of scheduling algorithms is vital to effectively manage computational tasks across a cluster of computing nodes. The primary goal is to maximize throughput while optimizing the utilization of computational resources. To facilitate the execution and coordination of streaming applications (computational tasks) on a computational cluster, a Stream Processing Engine (SPE) is deployed. However, this field, like other emerging domains, presents several challenges that researchers must address. These challenges include resource awareness, dynamic configurations, heterogeneous clusters, load balancing, and topology awareness, all of which significantly impact the job scheduling process.

Inefficiencies in any of these aspects can hamper the achievement of maximum throughput. A notable observation is the existence of a gap in achieving the optimal mapping of computational and communication loads in streaming applications, considering the underlying computational and communication power of the cluster. It is common for frequently communicating tasks to be scheduled on different processing nodes that are connected through relatively slower communication links. This arrangement leads to increased network latency and decreased resource utilization, resulting in a significant reduction in the achieved throughput of the cluster.

To address this gap, this dissertation proposes an efficient job scheduling system that takes into account the communication demands of the jobs and the available computational resources within a heterogeneous cluster. The aim is to achieve a near-optimal schedule that minimizes communication latency while maximizing throughput. The proposed system introduces four distinct scheduling schemes, namely **TOP-Storm**, **A3-Storm**, **BAN-Storm**, and **Gr-Storm**. These schedulers are designed to enhance overall throughput by placing tasks in a resource-aware manner based on their specific requirements.

To implement the proposed schedulers, **Apache Storm** (a popular SPE) is used. The effectiveness of the proposed schedulers is evaluated through experiments conducted on a physical heterogeneous cluster consisting of eleven computing nodes. Two benchmark topologies, namely the **word count** topology and the **exclamation** topology, are employed to generate results. For comparison purposes, the performance of the proposed schedulers is compared against three state-of-the-art scheduling algorithms: the **Default Scheduler**, the **Isolation Scheduler**, and the **Resource-aware Scheduler**.

Results showed that the proposed schedulers exhibited better performance than the default scheduler, achieving an average throughput improvement of 28% while utilizing up to 40% fewer resources for the word count topology. Similarly, when compared to the isolation scheduler, the proposed schedulers showed a throughput improvement of up to 200% while using only 60% of the resources. Moreover, R-Storm achieved the same throughput with over-provisioning.

Furthermore, the improvements in throughput were observed for the exclamation topology as well. For instance, the proposed schedulers achieved enhanced throughput of up to 23% and 112% compared to the default and isolation schedulers, respectively. When equal resources were allocated, the proposed work outperformed R-Storm by 4% in terms of average throughput for the exclamation topology. Additionally, the proposed approach demonstrated significant resource savings through consolidation.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Abbreviations

| | |
|---|---|
| **A3-Storm** | Traffic-based resource-aware scheduling |
| **BAN-Storm** | Bandwidth-aware Storm scheduler |
| **BFS** | Breadth-First Search |
| **CP** | Critical Path |
| **CSL** | Customer Satisfaction Level |
| **DAG** | Directed Acyclic Graph |
| **DFS** | Depth-First Search |
| **DRL** | Deep Reinforcement Learning |
| **DS** | Design Science |
| **DSP** | Distributed Stream Processing |
| **DSR** | Design Science Research |
| **FLOPS** | Floating Point Operations Per Second |
| **Gr-Storm** | Graph-based resource-aware scheduling |
| **HPC** | High Performance Computing |
| **JVM** | Java Virtual Machine |
| **PaaS** | Platform-as-a-Service |
| **PU** | Processing Unit |
| **SLA** | Service-level Agreement |
| **SPE** | Stream Processing Engine |
| **TOP-Storm** | Topology-based resource-aware scheduling |
| **W-DAG** | Weighted Directed Acyclic Graph |

# Symbols

| | |
|---|---|
| $\omega$ | Total number of slots required for topology execution |
| $e_u$ | All unassigned executors for a topology |
| $e_a$ | All assigned executors for a topology |
| $e_t$ | Total number of executors for a topology |
| $e_{ps}$ | Maximum executor per slot |
| $e_i$ | Next unassigned executor |
| $e_s$ | Next unassigned source executor |
| $e_d$ | Next unassigned destination executor |
| $e_u.Sort$ | Sort all unassigned executors with respect to connections or traffic in descending order |
| $\alpha$ | Adjustment factor |
| s | All unassigned slots of a node |
| $s_a$ | Assigned slot of the current node |
| $s_i$ | A specific slot of the current node |
| $s_n$ | Next free slot of the current node |
| s.Sort | Sort all slots by number of executors in descending order |
| n | All unassigned nodes in the cluster |
| $n_a$ | Available nodes in the cluster |
| $n_i$ | A specific node in the cluster |
| $n_i$.GFLOPs | Calculate computation power of $i^{th}$ node in GFLOPs |
| n.Sort | Sort unassigned nodes by computation power in descending order |
| $N_e$ | The corresponding set of executors |
| $N_s$ | The set of available slots |

# Chapter 1

# Introduction

## 1.1 Introduction

A data stream is a continuous production of data from different sources. This data could be generated by many sensors in many ubiquitous services, for example, smart cities (energy supply, transportation management, garbage collection), disaster management (emergency management services), production and logistics (quality control sensors and resource-saving), entertainment (analyzing streaming data for recommendations, advertisement analysis), the financial sector (credit card transactions, fraud detection, loan application), health care (time-sensitive critical patient data) and social network (Facebook, Twitter, and LinkedIn) [1]. This type of data can be characterized as a huge volume of data generated at a very fast rate. We need to process these data streams in real-time to provide the intelligence required for decision-making, otherwise, the next data stream will be generated, and we will lose valuable intelligence. Today's business world depends upon the processing of these huge and very fast data streams in real-time by using powerful computing and communication resources.

A streaming application to process data can be modelled as a *Directed Acyclic Graph* (DAG) [2] in which each vertex corresponds to a streaming data source or a *Processing Unit* (PU), and edges indicate data flows between PUs. The data

TABLE 1.1: Few Widely Used SPEs

| **Name** | **Specified Use** |
|---|---|
| Apache Storm [4] | Realtime computation system |
| Apache Spark Streaming [5] | Unified engine for large-scale data analytics |
| Apache Flink [6] | Stateful Computations over Data Streams |
| Apache Heron [7] | Realtime, distributed, fault-tolerant stream processing engine |
| Apache Samza [8] | Near-realtime, asynchronous computational framework |
| Apex [9] | Unifies stream and batch processing in a stateful and secure way |

flow through the edges and processing happens at the vertex. Streaming data may traverse multiple PUs [3].

Processing data in real-time demands ever-increasing processing power. Multiple machines interconnected in a distributed fashion to process the data is an economical way to process this huge and very fast data in real-time. A virtual service to facilitate the processing of real-time streams by using the distributed infrastructure of computing and communication resources is termed a *stream processing engine* (SPE)[10]. Many SPEs have rapidly evolved and are being adopted by the industry. Table 1.1 illustrate a few of the widely used platforms wherein streaming data could be processed in real-time.

The organization of SPEs can be divided into different layers as shown in Figure 1.1 [11]. The top layer defines the user API where a streaming application is defined as a DAG. The DAG is then converted to an execution graph and components of this are distributed to a set of workers across a cluster of nodes. Network communications are established between the workers to complete the graph. The bottom three layers (Execution Graph, Workers, and Network Communication) can work with the resource scheduling layer to acquire the necessary computing and communication resources to run the execution graph (as shown in Figure 1.2).

This is very important to measure the performance of an SPE system to process the streaming data. One of the key components of an SPE system is the resource

scheduling layer. Different parameters can be measured, including throughput, latency, and resource utilization to assess the performance of the resource scheduling layer.

In the context of SPEs, the streaming data is called a tuple and throughput refers to the number of tuples processed in a unit time [12]. Latency is the time difference between consuming a tuple and the end of its processing [13]. Similarly, resource utilization represents the number of computing nodes used in job execution. Typically, the performance of an SPE is assessed based on system throughput, communication latency or resources used [12]. To achieve better performance, scheduling can play an important role. A scheduler assigns DAG's components to computing nodes. An efficient scheduler assigns these components in a manner that minimizes tuple processing time and increases the utilization of resources [13].

Different proposed schedulers use different techniques to enhance their overall performance like topology-aware, traffic-aware, resource-aware etc. In topology-aware scheduling, the streaming application's DAG is used for task placement. With the help of DAG, connected tasks can be identified then graph traversing algorithms like breadth-first search, depth-first search etc. are employed to perform scheduling. In this type of scheduling, the focus is to place connected tasks closer to each other. So that, inter-node communication can be reduced which ultimately minimizes latency. Once a streaming application is running, its DAG, as well as

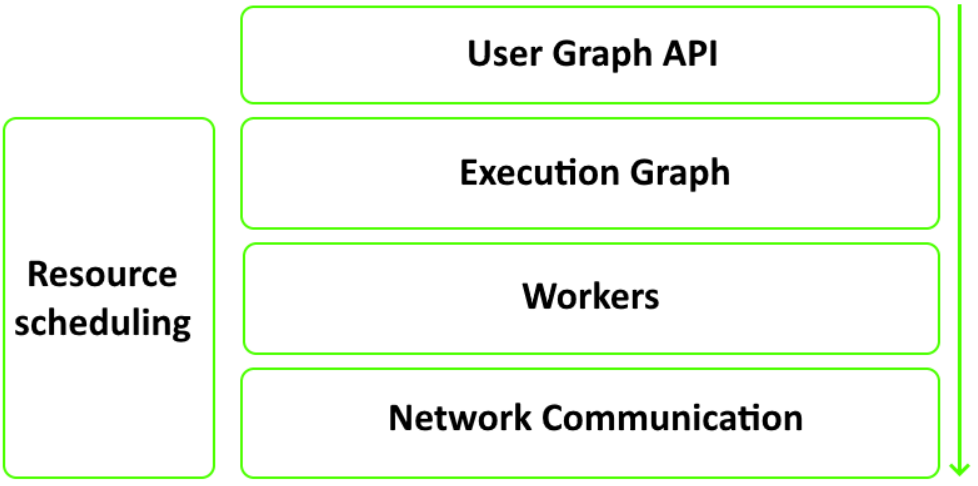



FIGURE 1.1: SPEs Layered Architecture [11].

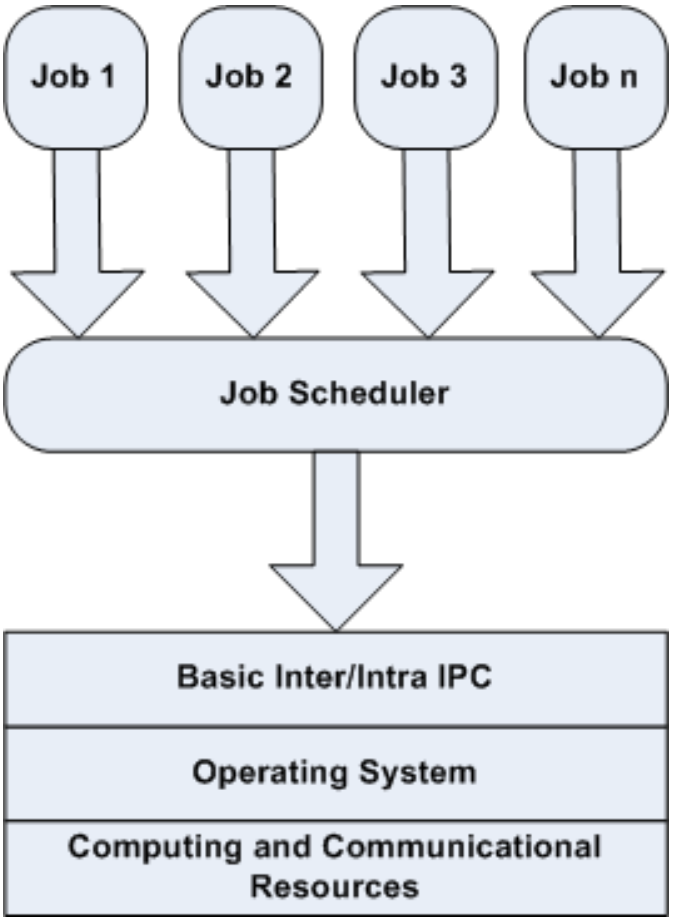



FIGURE 1.2: Scheduling of Jobs in SPEs.

communication traffic, can be used for making scheduling decisions. These schedulers are called traffic-aware scheduler that speed up data processing by employing efficient traffic-aware algorithm for job assignment, which minimizes inter-task communication. To handle the heterogeneity of the computing nodes; resource-aware schedulers are also designed to increase overall throughput by maximizing resource utilization while minimizing network latency. Resource-aware schedulers perform scheduling decisions based on available computing and communicational resources of the underlying hardware. So, the goal for the schedulers is how to schedule the tasks on nodes to process the data streams with near-optimum utilization of the available resources with maximized throughput. One of the popular SPE [14] is using the round-robin technique for scheduling; which introduces a potential performance bottleneck due to an unbalanced workload distribution for heterogeneous computational resources. As a result, Resource-aware scheduling algorithms were proposed [15–26] to address this issue. Resource-aware scheduling

considers resource availability on machines when scheduling jobs [27]. However, these are either static or nonadaptive. As a result, runtime changes are not handled by these scheduling algorithms. Some researchers [17, 19, 25, 28–33] introduced self-adaptability in their schedulers for a cost-efficient solution.

Graph partitioning algorithms are also used in research [16, 29, 31, 34–38] to place frequently communicating components of the DAG closer to each other to reduce latency which may improve overall throughput, too. But most of them are either resource-unaware or non-adaptive. As a result, it is difficult to decide on the power of cluster resources. Moreover, graph-based approaches may suffer from isomorphism problems that are not solvable in polynomial time nor to be NP-Complete.

Most of the SPEs like Apache Samza, Apex, Apache Storm, Apache Spark Streaming, Apache Flink, Apache Heron, etc., use DAG for the abstract representation of streaming applications [2]. The scheduling techniques proposed in this dissertation are also using the application's DAG for making scheduling decisions. Therefore, these proposed techniques can be implemented in any DAG-based SPE. However, we used Apache Storm to implement our proposed schedulers. Apache Storm is widely utilized because it is a reliable, fault-tolerant, and real-time SPE [39, 40]. It has been of interest in the industry as well as an area of research in academia also.

## 1.2 Apache Storm

The architecture of Storm follows a master and slave pattern [41] and coordinated by a third component called Zookeeper [42]. Figure 1.3 [43] shows the interaction between the Storm cluster and the Zookeeper cluster. The architecture of Storm allows a master node (consists of the Nimbus) and each slave node includes exactly one Supervisor. At the start-up of topology, the Nimbus distributes jobs directly to its worker nodes [40].

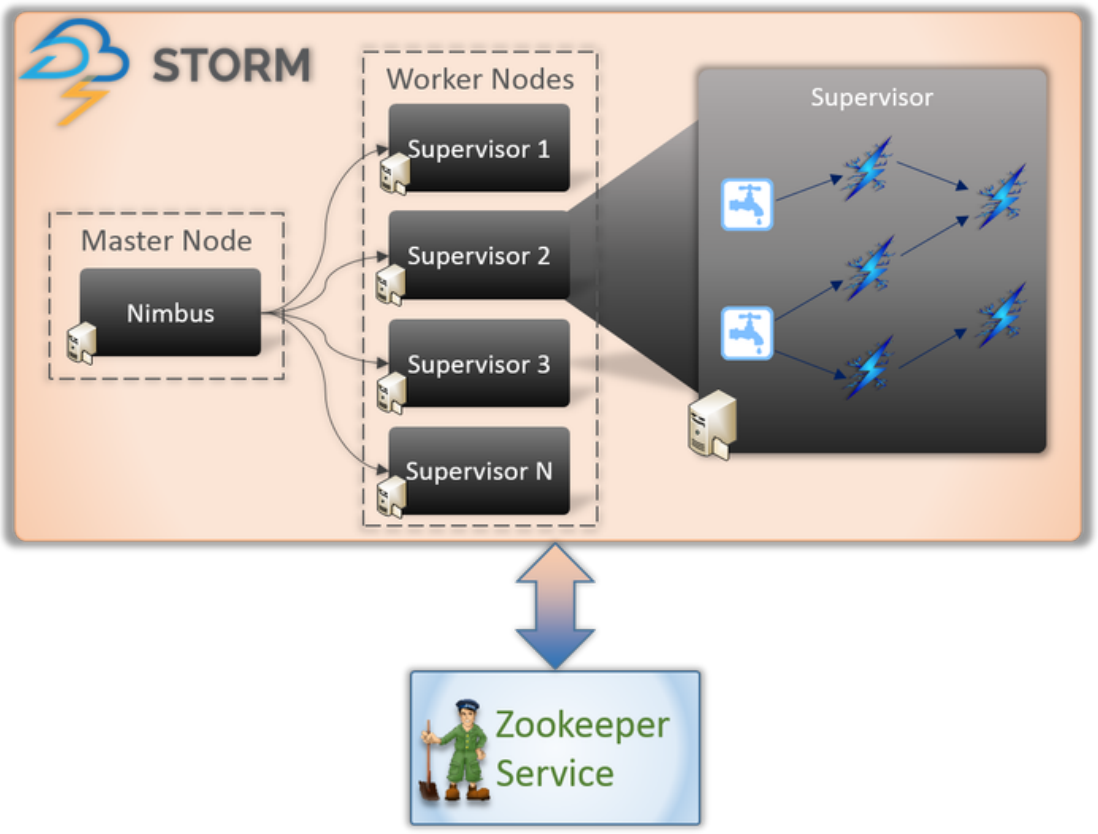



FIGURE 1.3: The Apache Storm Architecture [43].

It consists of the following major components:

- **Nimbus** represents the master component in the architecture of Storm and distributes the work among multiple workers [40]. It assigns a task to a slave node, monitors the task progress, and reschedules the task if a slave node fails. The Nimbus component is stateless because it stores all its data in the Zookeeper cluster, which is the third major component of Storm. This pattern does not cause the problem of a single point of failure. If a Nimbus daemon fails, it can be restarted without interrupting the running task. The architecture of Storm allows only a single Nimbus component.

- **Zookeeper** [44] coordinates and shares the information between Nimbus and Supervisor components. All states of the running tasks and the Nimbus and Supervisor daemon are stored in the Zookeeper cluster. Therefore, the state-less components, Nimbus and Supervisor, are fail-fast. These can be killed and restarted without interrupting the running topology.

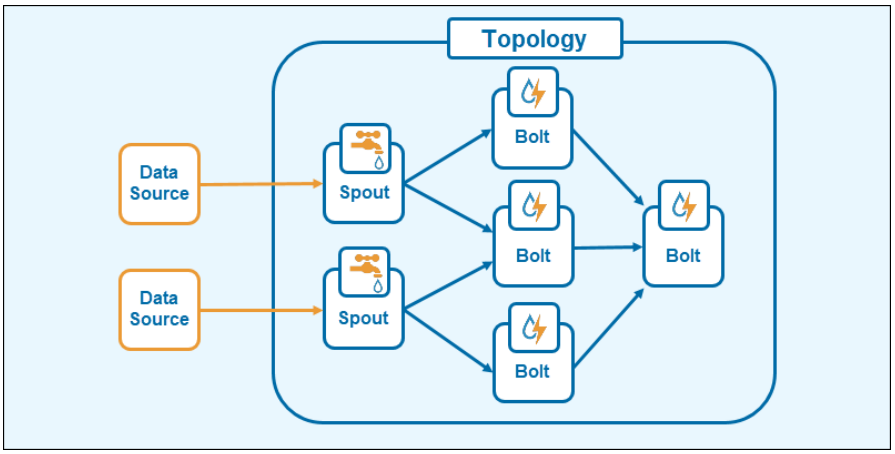



FIGURE 1.4: An Example Topology [45].

- **Supervisor** is the slave or worker node, which executes the actual tasks. The architecture of Storm may contain one or more Supervisor nodes. It initiates a new Java Virtual Machine (JVM) instance for each worker process. Like the Nimbus component, the Supervisor stores its data in the Zookeeper cluster, which makes it a fail-fast process. A Supervisor daemon in the Storm architecture normally runs an instance of multiple workers wherein each worker can spawn new threads.

- **Topology** represents logic / structure of a real-time Apache Storm application which is defined as DAG. It consists of two components for example, Bolts and Spouts (as shown in Figure 1.4 [45]). Spouts and Bolts are connected using stream which represents an infinite sequence of tuples [40].

- **Spout** specifies the source of tuples in Apache Storm. It reads data from source and supplies to the Storm topology [40].

- **Bolt**, the real handling of a tuple is done by Bolts. It reads, processes and releases them to other Bolt [40].

### 1.2.1 Levels of Parallelism

Storm provides four different levels of parallelism in its architecture. The following levels are used to run a topology in Storm:

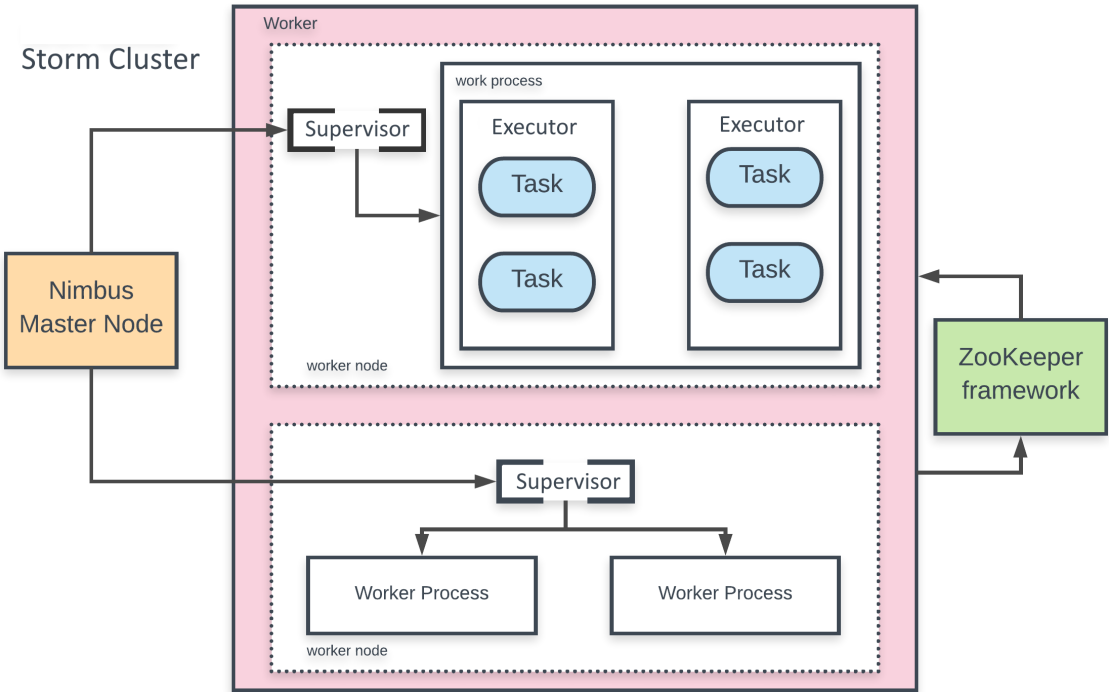



FIGURE 1.5: Level of Parallelism in Storm.

- Supervisor (Slave)
- Worker (JVM)
- Executor (Thread)
- Task

Apache Storm distributes tasks among supervisors where multiple processes can be running. Similarly, each process can use multiple Executor threads which can execute one or more tasks. These different levels of parallelism in Storm are shown in Figure 1.5, which shows two supervisor daemons, including two Worker processes. Normally, Apache Storm executes one Task per each Executor thread, but an Executor may execute multiple Tasks. Apache Storm's behaviour varies with workload velocity. It efficiently handles low workloads with low latency. It scales dynamically across multiple nodes with medium workloads. High workloads may introduce increased latency, but Storm's scaling mechanisms maintain stability but extremely high workloads requires careful tuning of the execution plan to optimize performance.

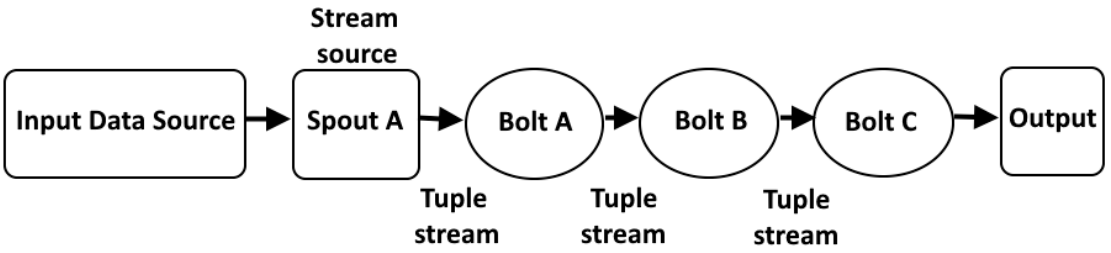



FIGURE 1.6: Example Linear Topology.

The default Storm scheduler employs a round-robin strategy for task scheduling. In the first step, it iterates through the DAG's components and allocates them to the worker processes. In the next step, the worker processes are evenly assigned to the computing nodes. As a simple case study, the working of default Storm is shown here using a linear topology (in Figure 1.6). A cluster based on two nodes is assumed (as shown in Figure 1.7). When Storm schedules the linear topology on the cluster, it assigns Spout A to supervisor 1, Bolt A to supervisor 2, Bolt B to supervisor 1, and Bolt C to supervisor 2. As a result, Spout A and Bolt B are assigned to supervisor 1, and Bolt A and Bolt C get placed on supervisor 2 node. Now Spout A and Bolt A are in different nodes, therefore inter-node communication is required. Similarly, inter-node communication will be required for other mapped nodes ((between Bolt A and Blot B) and (Blot B to Blot C)) of the example topology. This mapping increases internode communication causes execution delays and decreases throughput.

## 1.3 Problem Formulation with Significance

During the last decade, data has grown in almost every field of life. Studies have shown that this data will double every two years [46, 47] . Efficient processing of this huge data is important to take maximum benefits from it. Real-time processing is also possible with the rapid improvement in hardware resources, but this domain is still under development as new tools are still emerging. Different frameworks have been developed for this paradigm. In these frameworks, high latency is one of the major factors affecting performance. The scheduler of these frameworks

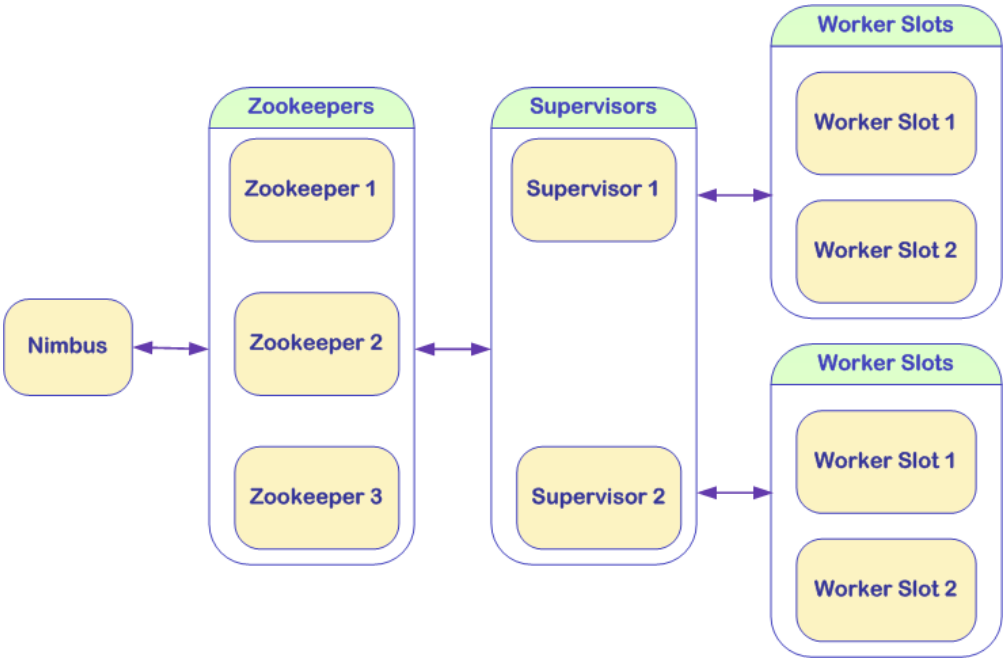



FIGURE 1.7: Cluster with Two Nodes.

considers different aspects while scheduling for example, available resources in the cluster, communication pattern between computing devices, dynamic changes in the network configurations, runtime changes in submitted jobs, heterogeneity in physical resources, etc.

Therefore, A problem statement is formulated as:

"***Existing scheduling schemes are unable to optimally schedule jobs on a computational cluster. An optimum schedule achieves maximum throughput with minimum resources.***"

## 1.4 Research Questions

To solve the above research problem, the following research questions are required to be answered:

1. How to design a resource-aware scheduler to distribute jobs among heterogeneous cluster to improve resource utilization? (TOP-Storm Scheduler, BAN-Storm Scheduler)

2. How to assign a workload based on data communication (traffic) to achieve maximal throughput and reduce data communication between computing nodes (machines)? (A3-Storm Scheduler, Gr-Storm Scheduler)

## 1.5 Research Objectives and Scope

After an extensive literature survey, it is revealed that the achieved throughput by existing SPEs can be enhanced using efficient scheduling. Therefore, this research proposes resource provisioning and scheduling techniques for SPEs that will ensure efficient resource utilization with enhanced throughput. The primary objective of this dissertation is to propose a scheduling scheme for SPEs that achieves near-maximum throughput in a way that a minimum number of resources are employed.

The focus of this dissertation is to design an efficient scheduling scheme for SPEs. Apache Storm (a popular SPE) will be used to implement the proposed scheme. Average throughput per node along with resource utilization will be used to quantitatively access the performance of the proposed work.

## 1.6 Research Methodology

A methodology is "a system of principles, practices, and procedures applied to a specific branch of knowledge"[48] to solve problems or add new knowledge. Such a methodology helps researchers to produce quality research. Similarly, "*Design Science* (DS) seeks to consolidate knowledge about the design and development of solutions, to solve problems and create new artifacts"[49]. Thus, *Design Science Research* (DSR) is a research method that is focused on problem-solving [50].

In this work, we have adopted the DS [48] methodology as it creates and evaluates information technology artifacts intended to solve problem, and communicate the results to the audiences [51]. Our research will be incremental, that is to quantitatively study the schedulers to maximize throughput with the minimum number

of resources and then propose a scheme to achieve these objectives. DS process includes six steps [52]. 1) problem identification and motivation, 2) the definition of the objectives for a solution, 3) design and development, 4) demonstration, 5) evaluation, and 6) communication.

1. To identify the research problem, a literature review will be carried out. The review will converge on the problem formulation and research questions. This may involve some experimentation with different scenarios for empirical validation of the problem identified. This will help in inferring the solution to the problem.

2. Then the objectives for a solution inferred from the problem specification will be finalized in the light of the available options and resources to carry out the research.

3. The next step is mainly related to the design and development of the artifact. Such artifacts can be models, methods, or frameworks [51]. Conceptually, a design research artifact can be any designed object in which a research contribution is embedded in the design. In our case, the proposed system is the outcome of this step.

4. Then the implementation of the artifact to solve the problem. This could be experimentation, simulation, or any other appropriate activity.

5. Then results and observations will be measured to evaluate how good the artifact helps a resolution to the problem. This step includes assessing the goals of a solution to actual examined results from the usage of the artifact in the experiment.

6. Finally, the problem, solutions, results, evaluation, utility, and novelty will be communicated to the researcher community in the form of research publications and reports. Design science research steps along their outputs for this dissertation is shown in Figure 1.8 [48].

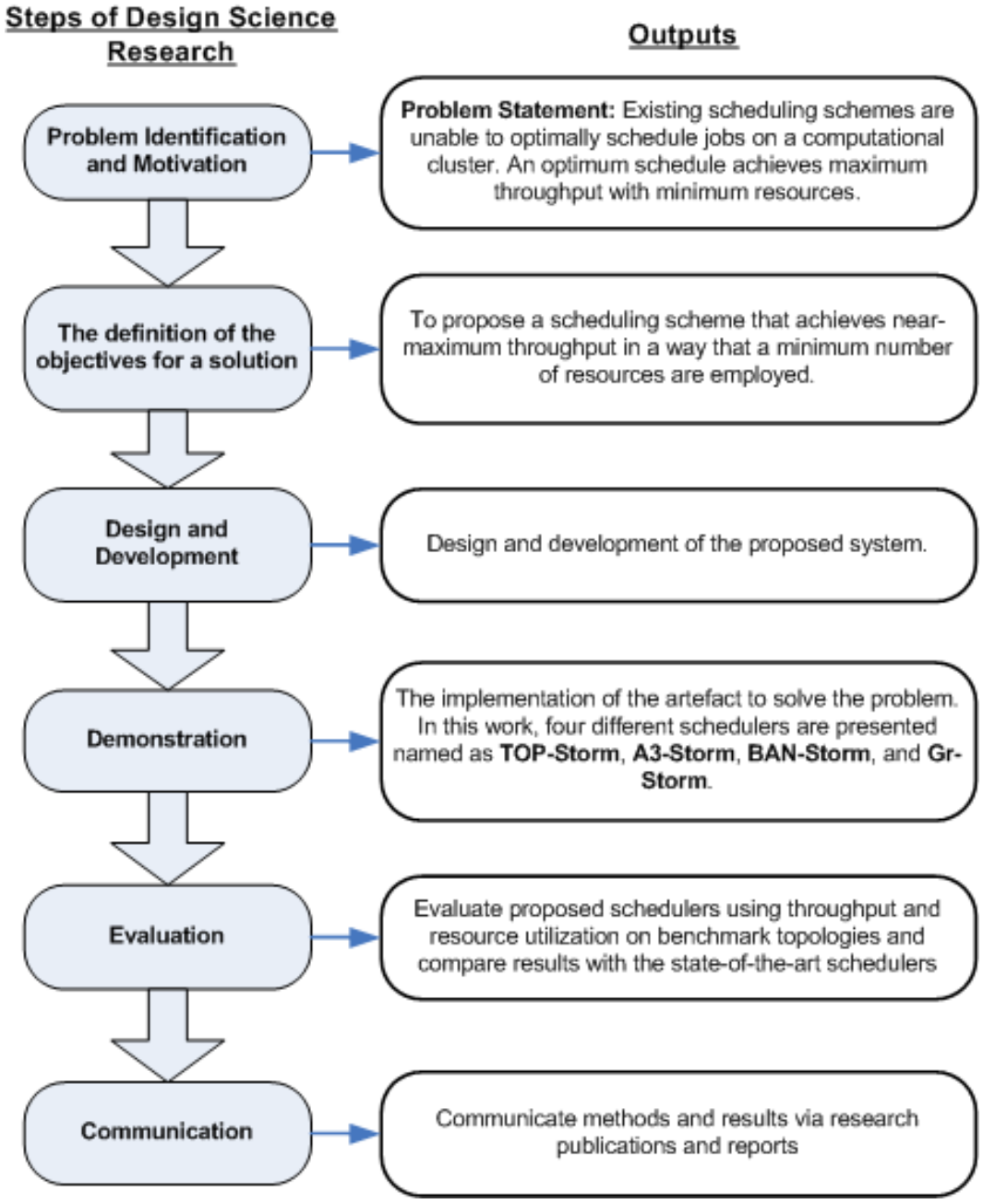



FIGURE 1.8: Design Science Research Steps and Their Outputs.

## 1.7 Evaluation Metrics

Usually, an SPE performance is assessed based upon throughput, latency or resources used. The same throughput can be achieved with different employed resources. It means throughput does not truly represent the efficiency of a scheduler. To depict the impact of both measures (achieved throughput and utilized resources), an evaluation matrix named average throughput per node (Equation

1.2) is used in this dissertation.

$$AverageThroughput = TotalTuplesProcessed \div TotalTimeTaken \quad (1.1)$$

$$AverageThroughputPerNode = AverageThroughput \div TotalNodesUsed \quad (1.2)$$

## 1.8 Research Findings

In this dissertation, topology, traffic, and resource-aware schedulers are proposed that can find frequently communicating tasks and assign them closer to each other in a heterogeneous cluster such that minimum nodes are utilized for topology execution. It has **four** different schemes:

- Topology-based resource-aware scheduling (named as **TOP-Storm**) finds executors that will be mapped in a single worker process. After that, it reads the executor's connectivity from the topology's DAG. Based on this connectivity, the longest path is calculated and assigned to the most powerful computing machine.
- Traffic-based resource-aware scheduling (named as **A3-Storm**): In this scheme, traffic is also used with topology structure. A3-Storm reads inter-task communication and sorts them in descending order. It starts assigning tasks to the computing nodes according to their traffic. For the next executor assignment, traffic communication of unassigned executors is compared with assigned executors and the executor with the highest communication with the already assigned executor is selected. After all executor's assignments to slots, the physical assignment is made starting from the most powerful machine and so on.
- Bandwidth-aware Storm scheduler (**BAN-Storm**) that finds regularly communicating computing tasks and maps them closer to each other. The employed mechanism results in a closely mapped and a smaller number of the

provisioned nodes of a heterogeneous cluster. Next, the physical mapping is performed based on the node's computation power which includes FLOPs, memory, and bandwidth as well.

- Graph-based resource-aware scheduling (named as **Gr-Storm**): This scheme maps computational requirements of the topology on the available computation power of workers (JVM) in an optimal way to maximize throughput and resource utilization. The first phase formulates the topology's graph (DAG) with intra-executor computation requirements and inter-executor communication requirements to generate a modelled DAG, which is then given to the max-flow min-cut algorithm [53] to produce a partitioned graph of executors. The second phase intelligently maps to a pool of workers by assigning all the group of executors to a worker based on the worker's computation power (starting from most powerful to the least) with less communication need as per the modelled DAG.

## 1.9 Thesis Organization

In this chapter, the research area has been introduced, the problem statement has been presented, and the objectives of the study have been outlined. The remaining chapters of the thesis are organized as follows:

### Chapter 2

Chapter 2 describes the literature review by presenting the state-of-the-art work done by the researchers. Different aspects have been used by the researcher in this context for example, topology-aware, traffic-aware, resource-aware, dynamic, task migration, SLA-Based scheduling algorithms etc. Therefore, category-wise schemes have been discussed and issues are highlighted followed by a critical analysis of the existing work.

### Chapter 3

In chapter 3, a topology-aware, traffic-aware, and resource-aware system for stream processing engines has been proposed. Based on the proposed system, four different scheduling schemes (TOP-Storm, A3-Storm, BAN-Storm and Gr-Storm) are presented to address the research questions. The working of each scheme is discussed in detail.

### Chapter 4

In chapter 4, first of all, the experimental setup is discussed then the benchmarked schedulers are listed with which the performance analysis of the proposed system is performed in terms of throughput, and resource utilization. The topologies used in experiments are also mentioned and the result is discussed.

### Chapter 5

Finally, the last chapter concludes the thesis document. The key findings of this research, open challenges and importantly the future directions are discussed in detail.

# Chapter 2

# Literature Review

## 2.1 Introduction

Like other domains, SPE is also not a new field of study for researchers. In the last decade, a broad range of research has been performed to improve the throughput, enhance resource utilization and reduce latency of the scheduling algorithm of SPEs. Different researchers targeted different aspects in their scheduling scheme. In this dissertation, we have used the following most frequently used aspects to investigate the existing literature:

- **Topology-aware**: As we know that topology can be represented as a DAG. A scheduler which is making scheduling decisions based on topology's DAG is called a topology-aware scheduler
- **Traffic-aware**: Once a topology is in execution then we can have communication traffic between executors for example some bolt A send processed data to bolt B etc. So, if scheduling is being done based on communication traffic is known as traffic-aware scheduling
- **Resource-aware**: hardware specifications like CPU, RAM, Bandwidth, Number of cores, etc. are also considered while assigning tasks among the computing node in a cluster

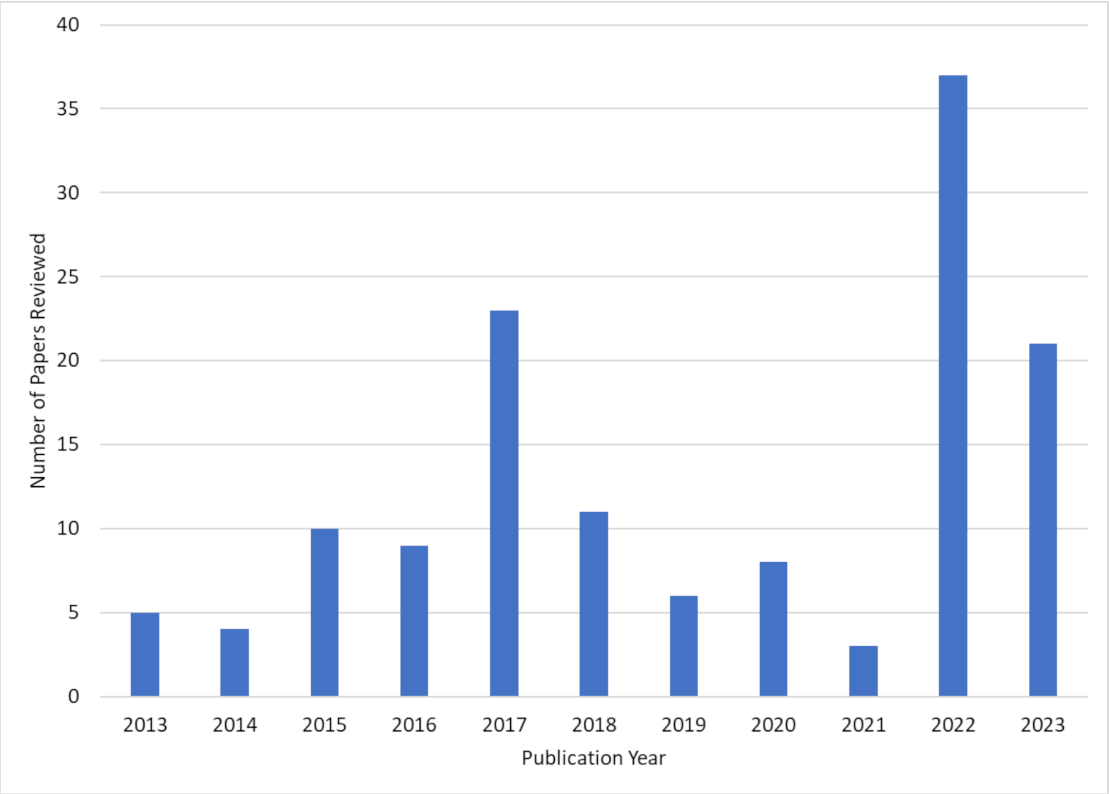



FIGURE 2.1: Year-wise Distribution of the Reviewed Papers.

- **Heterogeneous**: computing environment that can contain processors and devices with different bandwidth and computational capabilities
- **Dynamic**: Typically, a scheduler makes scheduling decisions at compile time but if a scheduler makes changes in the execution plan during runtime is called a dynamic scheduling scheme
- **Self-adaptive**: includes a task monitor to collect runtime statistics and continuously adapts the scheduling plan accordingly to enhance the performance
- **Network-aware**: scheduling based on the physical distance between components that communicate with each other

Year-wise distribution of the reviewed papers (see Fig. 2.1) with respect to the scheduling aspects (as shown in Fig. 2.2) covered in each paper is presented in

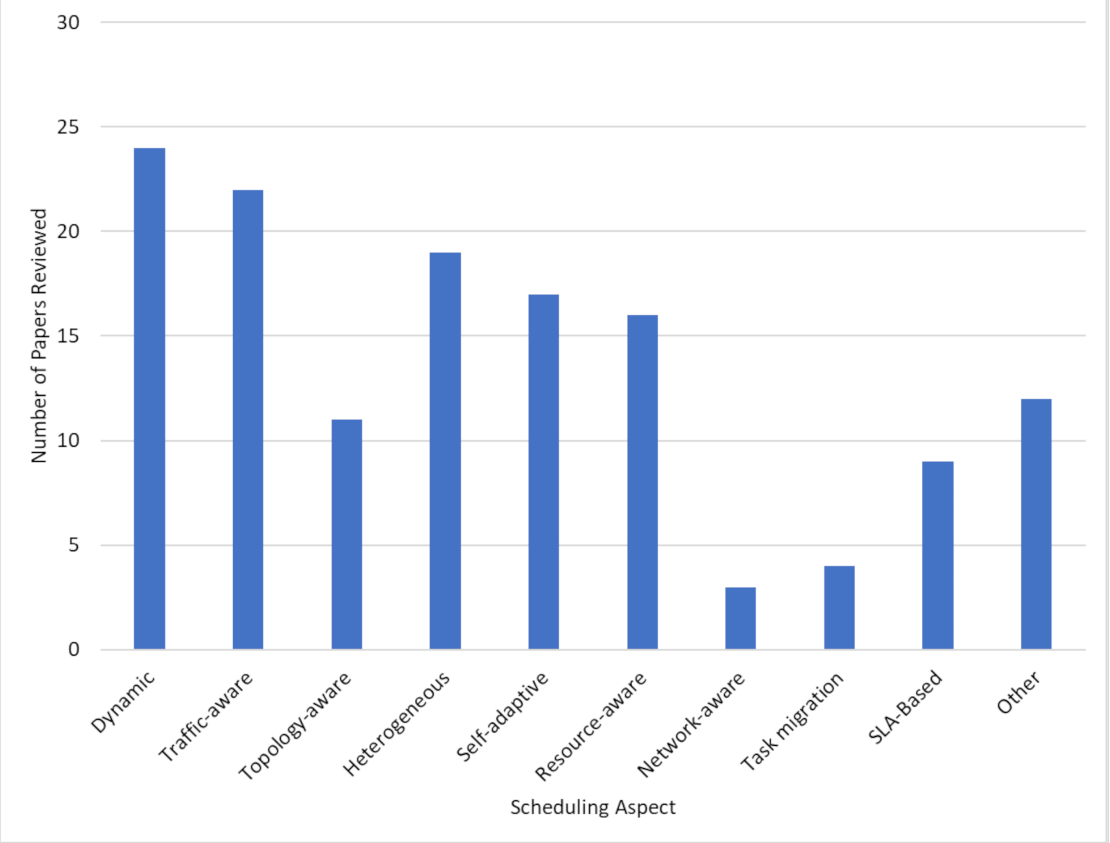



FIGURE 2.2: Aspect-wise Distribution of the Reviewed Papers.

Table 2.1. Similarly, Year-wise & Aspect-wise distribution of the reviewed papers are also shown in Fig. 2.3. If a paper targeted multiple aspects then that paper is counted against each category.

## 2.2 Related Work

The simplest scheduling scheme used by the Storm scheduler is round-robin, which involves iterating through the topology executors and allocating them to the configured number of worker processes. In the next step, according to the slot availability, the worker processes are evenly assigned to the computing nodes. This scheduling produces worker processes with an equal number of executors and assigns worker processes among the computing nodes such that each node runs an equal number of worker processes.

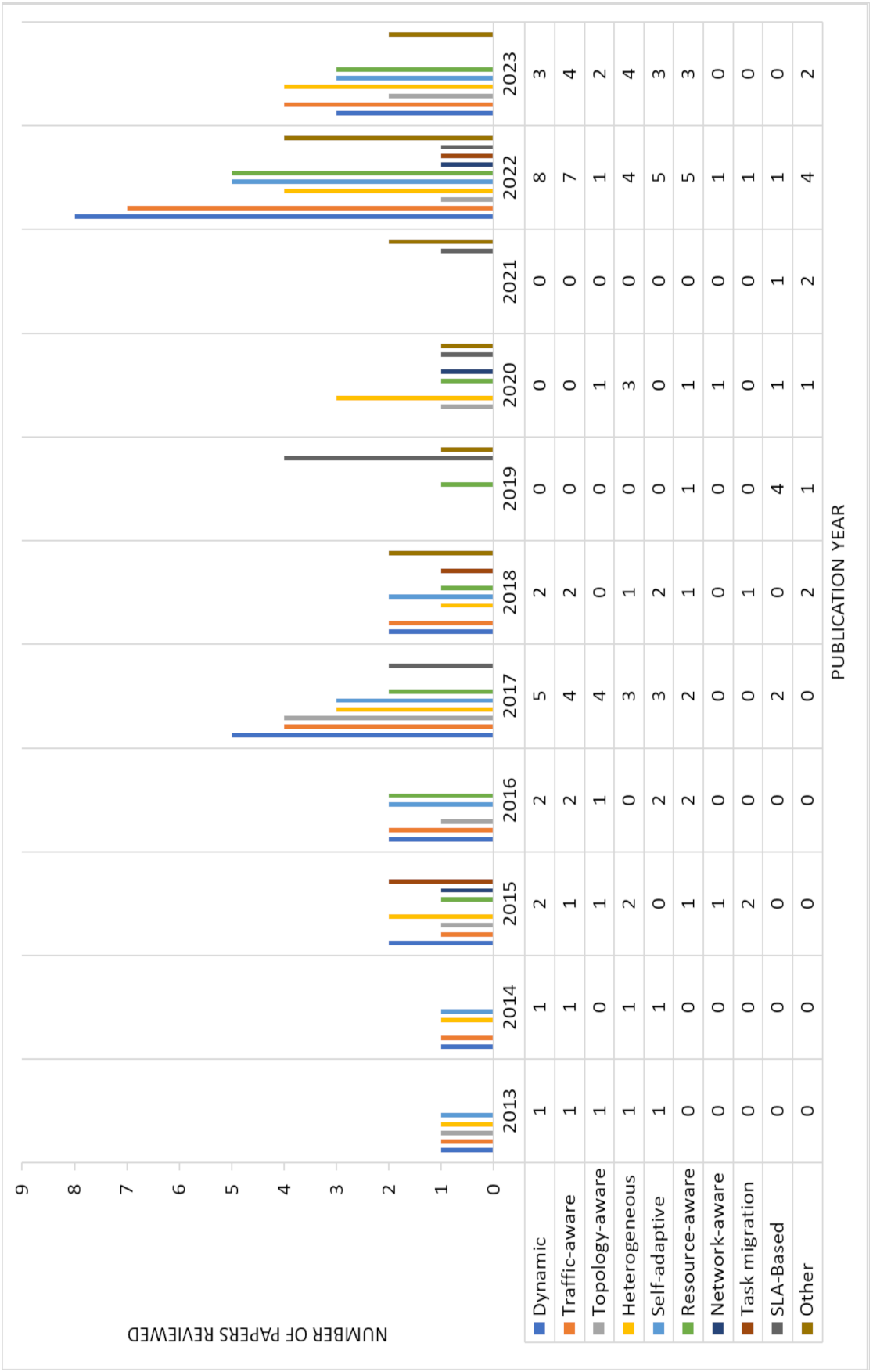



FIGURE 2.3: Year-wise & Aspect-wise Distribution of the Reviewed Papers.

TABLE 2.1: Year-wise Distribution of Reviewed Papers.

| | Scheduling Aspects | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year** | **Dynamic** | **Traffic-aware** | **Topology-aware** | **Heterogeneous** | **Adaptive** | **Resource-aware** | **Network-aware** | **Task migration** | **SLA-Based** | **Other** | **Total (Year-wise)** |
| **2013** | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 5 |
| **2014** | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 4 |
| **2015** | 2 | 1 | 1 | 2 | 0 | 1 | 1 | 2 | 0 | 0 | 10 |
| **2016** | 2 | 2 | 1 | 0 | 2 | 2 | 0 | 0 | 0 | 0 | 9 |
| **2017** | 5 | 4 | 4 | 3 | 3 | 2 | 0 | 0 | 2 | 0 | 23 |
| **2018** | 2 | 2 | 0 | 1 | 2 | 1 | 0 | 1 | 0 | 2 | 11 |
| **2019** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 4 | 1 | 7 |
| **2020** | 0 | 0 | 1 | 3 | 0 | 1 | 1 | 0 | 1 | 1 | 8 |
| **2021** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 |
| **2022** | 8 | 7 | 1 | 4 | 5 | 5 | 1 | 1 | 1 | 4 | 37 |
| **2023** | 3 | 4 | 2 | 4 | 3 | 3 | 0 | 0 | 0 | 2 | 21 |
| **Total (Aspect-wise)** | 24 | 22 | 11 | 19 | 17 | 16 | 3 | 5 | 9 | 12 | |

The default Storm scheduler is a fair scheduler that considers each node for scheduling tasks; however, it has several drawbacks. It evenly assigns workload to worker processes across the cluster without considering inter-task communication, which may adversely effect the performance [28]. This even mapping increases inter-node communication, causes execution delays and decreases throughput. Several researchers have proposed scheduling algorithms to improve throughput and resource utilization of the Apache Storm cluster.

### 2.2.1 Traffic-aware Scheduling Algorithms

Aniello et al. [28] introduced the traffic-aware scheduling algorithm based on two sub-schedulers, offline and online schedulers. The offline scheduler simply analyses the topology graph and identifies inter-connected bolts to be scheduled on the same node. This approach is simple and only considers topology for mapping. Offline scheduling is executed before the topology is started, so neither the workload nor the traffic is considered for mapping decisions. The application computing requirements and traffic aspects are ignored, which results in higher inter-node and inter-slot traffic at runtime. To overcome the limitations of the offline scheduler, the authors proposed a dynamic scheduler named an online scheduler. It is a traffic-aware scheduler and considers the application load at runtime. The dynamic scheduler monitors runtime traffic and re-adapts the application schedule to improve its performance and reduce communication delays. The major strength of the online scheduler is that it reduces inter-node communication. Experiments are performed using 9 computing nodes and a 30% reduction in latency is achieved w.r.t. default scheduler. Reference topology and Grand Challenge topology are used for evaluation purposes. The online scheduler reduces inter-node communication; however, it ignores inter-slot communication, which may become a performance bottleneck. The online scheduler performs better for variable input traffic velocity.

Another problem with the default scheduler is that it occupy all machines available in a cluster despite of workload. In case of a light workload, the operational cost

TABLE 2.2: Summary of Traffic-aware Scheduling Algorithm

| **Authors / Algo.** | **Scheduling Aspects** | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Dynamic** | **Traffic-aware** | **Topology-aware** | **Heterogeneous** | **Adaptive** | **Resource-aware** | **Network-aware** |
| Aniello et al. [28] / Offline & Online scheduler | √ | √ | √ | √ | √ | × | × |
| Xu et al. [33] / T-Storm | √ | √ | × | √ | √ | × | × |
| Eskandari et al. [29] / P-Scheduler | √ | √ | √ | × | √ | × | × |
| Li and Zhang [31] / TS-Storm | √ | √ | √ | √ | √ | × | × |
| Selim et al. [54] / BSS | √ | √ | × | √ | × | √ | × |

can be reduced by turning off idle ones. In 2014, Xu et al. [33] proposed a traffic-aware scheduler (called T-Storm) which uses runtime states to accelerate data processing for dynamic task allocation. This minimizes inter-process traffic in a load-balancing fashion. T-Storm was implemented using Storm 0.8.2 with the overall time complexity of $O(N_e log N_e + N_e N_s)$ where $N_e$ representing the corresponding set of executors, and $N_s$ is the set of available slots. Experiments show up to 84% enhancement in average throughput using 30% fewer computing nodes as compared to the default scheduler. However, the proposed work faces the cold start problem which means that initially some data is required for this type of scheduler to work.

The Storm scheduler equally assigns the tasks across the cluster. It has two main disadvantages. First, it ignores the communication pattern between executors, which results in low execution performance and high communication latencies. Secondly, it harnessed all the computing nodes regardless of the load, which may cause high inter-node traffic. To address these issues, Eskandari et al. [29] presented an adaptive scheduling algorithm called P-Scheduler. To reduce the communication traffic, P-Scheduler calculates the required resources for topology execution. Afterwards, it consolidates the cluster after computing the estimated

load. P-Scheduler maximizes communication efficiency by placing frequently communicating pairs at a minimum distance. Experiments were performed using 10 computing nodes resulting in 50% less latency to default Storm 0.9.5. In-house, test throughput and top trending topics topologies were employed in these experiments. Despite these features, P-Scheduler has a major drawback in that it is designed for homogeneous clusters only. It also assumes an equal number of slots for all the nodes. That's why P-Scheduler will not perform better in a heterogeneous environment with a variable number of slots on each computing node.

Default Storm Scheduler is also not optimum in terms of traffic performance and resource utilization. Moreover, while scheduling the topologies, it ignores the computing node's load and the topology structure. Zhang et al. [31, 55] proposed a traffic-aware real-time scheduling algorithm based on worker nodes load to solve the problems. This algorithm is divided into two steps. The first step assigns slots to the executors, according to inter-component communication and topology structure. The second step chooses the lowest loaded worker node and assigns slots that require maximum resources. During the allocation process, CPU and memory utilization is also considered. This scheduler is compared with default scheduler 0.8.2, R-Storm, and topology-based schedulers using linear, star, and diamond topologies. The evaluation carried out with 8 nodes showed a 91% improvement in throughput and a 50% reduction in latency.

Recently, Balance Scheduling on Storm (BSS) is presented [54] to improve the scheduling on the Storm cluster. BSS collects runtime details to create an execution plan using the graph partitioning technique. BSS collects the communication data at runtime, partitions the graph, and balances the workload among the computing nodes. Then nodes are assigned the partitions, and the tasks are mapped to the slots w.r.t. their communication, which reduces the network traffic and resulted in enhanced throughput. BSS is compared with the Fischer and Bernstein and A3-Storm schedulers using four different topologies for example, SOL topology, Rolling count topology, word count, spike detection topology etc. The

TABLE 2.3: Strength / Weakness of Traffic-aware Scheduling Algorithm

| **Authors** | **Strength** | **Weakness** |
|---|---|---|
| Aniello et al. [28] | Reduced inter-node communication | Inter-slot communication is ignored |
| Xu et al. [33] | Minimizes inter-process traffic using load-balancing | Cold start problem |
| Eskandari et al. [29] | Graph-based adaptive scheduling | Homogeneous solution |
| Li and Zhang [31] | CPU utilization-based load balancing algorithm | Resource unawareness problem |
| Selim et al. [54] | A hybrid approach is employed | Available nodes equal to the no. of partitions |

experimental study is done on a heterogenous cluster of 4 physical machines in which results show a 70 percent improvement in throughput.

### 2.2.2 Dynamic Scheduling Algorithms

In Storm, the job configuration remains static during job execution. This static configuration may not be aware of data stream properties (i.e., data transfer rate). As a result, significant resources can be wasted on fluctuating data flow. Similarly, the static configuration might also limit the throughput if the arrival rate of data flow surpasses the system capability. To dynamically change the instances, Apache Storm must pause the whole topology, recompile, and redeploy the complete topology. This will incur latency during runtime. Madsen et al. [56] provided the solution to this problem by over-provisioning resources (one redundant operator per node) with polynomial time complexity of $O(|\mathrm{N}| \times \#\text{checkpoints})$. Experiments were carried out on Amazon EC2 (25 instances) using Airline On-Time dataset which reduced latency by 20% w.r.t. default scheduler. The major weakness of this work is the redundant operators on each node which results in high resource costs and most of the operators are not utilized during execution.

The question of how to assign tasks among the available resources has important consequences for the performance of SPEs. Fischer et al. [35] presented a workload

TABLE 2.4: Summary of Dynamic Scheduling Algorithm

| **Authors / Algo.** | **Scheduling Aspects** | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Dynamic** | **Traffic-aware** | **Topology-aware** | **Heterogeneous** | **Adaptive** | **Resource-aware** | **Network-aware** |
| Fischer and Bernstein [35] | √ | √ | √ | × | × | × | × |
| Liu et al. [57] / E-Storm | √ | √ | √ | × | × | × | × |
| Qiann et al. [58] / S-Storm | √ | × | √ | × | × | × | × |
| Weng et al. [25] / AdaStorm | √ | √ | × | × | √ | √ | × |
| Li et al. [30] / DRL-Based framework | √ | √ | × | × | √ | × | × |
| Sun al. [59] / Lc-Stream | √ | √ | × | × | √ | √ | √ |

scheduling strategy that is based on a graph partitioning algorithm. The proposed scheduler collects the communication behaviour of running applications and creates the schedules by partitioning the communication graph using the METIS. The proposed scheduler showed better results with decreases in network bandwidth of up to 88% and increased throughput values of up to 56%, respectively. Experimental evaluation was performed on a cluster of 80 machines and a comparison was made with default Storm 0.9.0.1 and Aniello et al. [28] schedulers using OpenGov, Parallel, Payload, and Reference topologies. The major limitation of the proposed approach is that it ignores the computational resources such as RAM or disk space while scheduling.

As an essential part of the fault-tolerance mechanism, Apache Storm's state management is accomplished by a checkpointing system, which commits states frequently and recovers lost states from the latest checkpoint. However, this method requires a remote location for state storage and retrieval, resulting in substantial overheads to the performance of error-free execution. To fix this overhead, a

replication-based state management system (E-Storm) is proposed [57] that actively maintains multiple state backups on different worker nodes. E-Storm guarantees application integrity when failover occurs. The replication of the state is autonomous and high-performance, which allows multiple transfers to take place simultaneously. E-Storm outperforms the existing checkpointing method in terms of the resulting application performance, obtaining 9 times better throughput while reducing the application latency down to 9.8% when a comparison was made with Apache Storm version 1.0.2. These results are achieved using 12 nodes cluster at the expense of storage space for multiple backups and processing required to restore the lost state from a checkpoint. Moreover, the computational complexity of the provided solution is *NP-Hard* which is also a drawback of E-Storm.

The even scheduler ignores the allocation and dependency relationship among slots. This would bring the load-unbalancing problem when the topology run failed and is killed by its user, or new machines are extended in the Storm cluster. To address this issue, a slot-aware scheduler (S-Storm) is proposed [58] which uses a sorting queue and merger factor. First, it evenly allocates slots for multiple topologies in the load balancing cluster. Second, when load-unbalancing happens, it distributes workers to slots among light-load worker nodes. Experimental results show that when compared to Storm 0.10.0, S-Storm can achieve over 18.7% speedup on the average processing time and 1.25 times improvement on throughput within a ten-minute interval for Word count and Throughput test topologies. The major limitation of the S-Storm is its equal slots allocation for each topology. In practice, different topologies require a different number of slots for execution.

In Storm, the number of slots for topology execution can't be increased or decreased at runtime. This static configuration might limit the processing throughput if the arrival rate of data flow exceeds the system capability and more slots are required for processing. Weng et al. [25] presented AdaStorm with time complexity of $O(n^2)$ to solve the above-mentioned issue of the default Storm scheduler. AdaStorm dynamically adjusts the topology configuration according to the data communication rate using machine learning algorithms. Firstly, different resource

TABLE 2.5: Strength / Weakness of Dynamic Scheduling Algorithm

| **Authors** | **Strength** | **Weakness** |
|---|---|---|
| Fischer and Bernstein [35] | Workload based graph partitioning | Ignores the computational resources |
| Liu et al. [57] | Replication-based state management | Computational complexity is NP-Hard |
| Qiann et al. [58] | Slot-aware scheduler | Equal slots allocation for each topology |
| Weng et al. [25] | Resource-efficient scheduling using ML | Topology unawareness problem |
| Li et al. [30] | DRL employed to minimize processing time | Suffers from a cold start problem |
| Sun al. [59] | Network location-aware scheduling | Experiments lacked required environment |

usage models are trained with historical data. Secondly, these models are used to derive the resource-efficient topology configuration for deployment. AdaStorm was compared with R-Storm using a cluster of 10 nodes. GeoLife GPS Trajectories dataset was used in these experiments. Results showed that AdaStorm reduces CPU and memory usage by about 15% and 60% respectively. AdaStorm has solved the static configuration to some extent, however, it does not consider network topology while scheduling. It handles local and remote computing nodes in the same manner, which may affect the overall throughput.

To minimize average end-to-end tuple processing time, T. Li et al. [30] developed a model-free approach that can learn to control an SPE from its experience, just as a human learns a skill (such as cooking, driving, swimming, etc). Deep Reinforcement Learning (DRL) is introduced to minimize processing time by learning the system environment via collecting runtime statistics and making decisions using Deep Neural Networks. Extensive experiments show that the proposed framework reduces average tuple processing by 33.5%. This work was compared with the Actor-critic-based method, default scheduler, model-based method, and the DQN-based DRL method. Three topologies are used (continuous queries, log

stream processing, and word count) on a cluster of 11 computing nodes. The proposed system is based on runtime statistics; therefore, it suffers from a cold start problem.

Processing real-time streaming data efficiently is always a challenging task, especially for a Geo-Distributed cluster. The default Storm uses a pooling method while allocating tasks, which often causes load imbalance. Uneven resource allocation results in low system efficiency. Similarly, the waiting time caused by the communication between nodes is also ignored. To address these problems, an elastic scheduling strategy with latency constrained (Lc-Stream) is proposed [59] Lc-Stream mainly focuses on 3 aspects. First, an optimized redirection method based on queuing network algorithm, a computing resource model, a latency-constrained scheduling model and an energy consumption model. Second, a node selection method based on the inter-task relation, to reduce the communication latency between groups at the operator level. Finally, a cluster distribution for a Geo-Distributed computing environment to ensure energy saving under low transmission latency. Lc-Stream is implemented using Storm (version 2.1.0) and experiments are conducted in the Alibaba Cloud Computing cluster consisting of 28 machines using Word Count topology. Results demonstrated that Lc-Stream reduces latency by 19% and increases the throughput by up to 37%.

### 2.2.3 Resource-aware Scheduling Algorithms

While scheduling, the Default Storm Scheduler does not consider resource availability in the underlying cluster, and resource requirement of Storm topologies. This can lead to resource over-utilization or under-utilization causing failure or execution inefficiency. To overcome this issue, Peng et al. [22] presented resource-aware scheduling (R-Storm) in Storm at compilation time. When scheduling

the tasks, R-Storm fulfils resource restrictions (CPU, bandwidth, and memory) and can also minimize network distance between communicating components. PageLoad and Processing topologies are used for experimentation on a cluster of 13 machines. Results showed a 50% higher throughput and 350% better CPU utilization when compared to the Storm scheduler. However, due to compile-time scheduling, R-Storm is unaware of the actual runtime workload and remains unchanged during the whole lifecycle of the application. Secondly, the scheduling is performed during compile-time only. Therefore, it is impossible to adjust the execution plan according to runtime changes.

In general, inter-node and inter-slot communication are important factors that affect the performance of a topology. However, they are inversely proportional to each other. While scheduling on Storm, handling the trade-off between intern-ode, and inter-slot is a crucial process. Fan et al. [17] presented an adaptive task scheduler, which uses runtime statistics of the topologies and cluster load while scheduling. The overall throughput is improved by 67% with 3% extra resources consumed as compared to the default scheduler (version 0.9.3) using the SOL benchmark on a cluster of 5 blade servers. Throughput has been increased; however, the authors have not presented the details of runtime statistics used in this scheduler.

To reduce the network latency, there is a need to avoid the data movement over WAN as much as possible. Placing applications closer to the data source can decrease network cost and latency. Even though, this approach is not well explored in data stream processing applications. To address this issue, SpanEdge, a novel approach is proposed [23] that reduces latency incurred by WAN links by distributing stream processing applications to near-the-edge data centres. It provides a programming environment from which programmers specify parts of their applications that need to be close to the data source. Results show that SpanEdge reduces the bandwidth consumption and the response latency by optimally deploying applications in a geo-distributed infrastructure. SpanEdge is evaluated in an emulated environment using the CORE network emulator.

TABLE 2.6: Summary of Resource-aware Scheduling Algorithm

| **Authors / Algo.** | **Scheduling Aspects** | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Dynamic** | **Traffic-aware** | **Topology-aware** | **Heterogeneous** | **Adaptive** | **Resource-aware** | **Network-aware** |
| Peng et al. [22] / R-Storm | × | × | × | √ | × | √ | √ |
| Fan et al. [17] / Adaptive-Scheduler | √ | √ | × | × | √ | √ | × |
| Liu and Buyya [19] / D-Storm | √ | √ | × | × | √ | √ | × |
| Al-Sinayyid and Zhu [15] / MT-Scheduler | × | × | √ | √ | × | √ | √ |
| Farrokh et al. [60] / SP-Ant | √ | √ | × | √ | √ | √ | × |

The stream processing model works on continuous data flow. Due to this, a large amount of data arrives dynamically that cannot be evaluated in advance. Smirnov et al. [24] presented a performance-aware scheduling algorithm for real-time data streaming systems. The common scheduling problem is to assign tasks to the best-fit slots, minimize inter-node communication, and maximize CPU utilization for all nodes. With the help of a genetic algorithm, these problems are addressed by the authors [24]. The proposed methodology is compared with the default and R-Storm schedulers showed a 40% improvement in throughput. However, this study only considers a linear topology consisting of 1 spout and 3 bolts. Moreover, the authors have not presented the implementation details of the proposed methodology.

SPEs lack an intelligent system, which can adjust the scheduling plan based on resource usage. In 2017, Liu [19] presented D-Storm to address this problem with the time complexity of $O(n^2)$. D-Storm monitors topologies at runtime to obtain resource usage and communication patterns to avoid resource contention and reduce inter-node communication. D-Storm makes runtime decisions to schedule tasks closely to reduce inter-node communication. It automatically re-schedules the topology whenever resource contention is detected. Performance evaluation

of D-Storm was performed on Nectar Cloud (using 19 nodes) with the default scheduler (1.0.2) and R-Storm where D-Storm achieved a 16.25% improvement on throughput for Tweet sentiment topology. Despite all these results, D-Storm has the following limitations. It is based on runtime statistics; therefore, it suffers from a cold start issue means in the beginning some statistics are required to generate schedule and It is implemented for homogeneous clusters only. Similarly, D-Storm is slower (approximately 20 times) to generate a scheduling plan as compared to the default scheduler. D-Storm is suitable in a situation where input data flow is fluctuating.

Another scheduling mechanism to map and process the streaming data collected from different geo-distributed heterogeneous IoT sensors is presented in [18]. The authors emphasized that IoT-based applications generating stream data should be scheduled and processed considering both the network topology and heterogeneity of the processing resources. To address these research challenges, the authors proposed a scheduling mechanism that considers latencies, bandwidth, and computational resources in a heterogeneous topology environment. The results presented by the authors highlight the fact that the scheduling mechanism when employed considering the service quality metric produces better placements of the dataflow jobs as compared to considering only topology-related aspects. The proposed work relies only on the simulation to evaluate the potential performance benefits of the scheme in contrast to our proposed work that is implemented using a real steam processing framework (i.e., Apache Storm) to highlight the factual performance results including all machines and runtime system overheads. Also, their proposed scheduling scheme only employs an abstract level resource-aware mechanism (such as resource requirement classes of low, medium, and high) in contrast to the fine-grained resource-aware mapping employed by our proposed schemes.

While scheduling topology, Apache Storm ignores cluster heterogeneity. As a result, scheduling in a heterogeneous environment is a problem. To address this issue, AL-SINAYYID & ZHU proposed MT-Scheduler [15] to maximize throughput for a heterogeneous cluster based on computational as well as communicational

TABLE 2.7: Strength / Weakness of Resource-aware Scheduling Algorithm

| **Authors** | **Strength** | **Weakness** |
|---|---|---|
| Peng et al. [22] | Compact executor assignment | In-adaptive scheduling |
| Fan et al. [17] | Adaptive task scheduler | Runtime statistics are missing |
| Liu and Buyya [19] | Dynamic scheduling to reduce inter-node traffic | Slowest to calculate a scheduling plan |
| Al-Sinayyid and Zhu [15] | Allows the users to configure the data locality | Polynomial-time heuristic solution |
| Farrokh et al. [60] | Hybrid heuristic-metaheuristic optimization | Cold start problem |

requirements, while considering the capability of the cluster. First, the topology's DAG is linearised by applying a topological sort. Then the critical path (CP) is found by employing the longest path algorithm. The mapper function determines the mapping schema for the topological components in the CP. The remaining components (not on the CP) are assigned with the help of a layer-oriented greedy method that may fail to achieve an optimal solution. The MT-Scheduler lowers latencies by 28–46% and enhances the throughput by 17–54% when compared to the default scheduler. It also permits the clients to organize the data locality to execute tasks as close to the data, which decreases the communication cost. MT-Scheduler was compared with the default scheduler and an adaptive online scheduler [28].

A key feature of SPEs is the task scheduling on the cluster's nodes. While scheduling, the tasks to node mapping, the task's requirements, and the computing power of the cluster's resources must be considered with the goal of finding a trade-off to achieve maximum efficiency. This is a challenging goal, especially in a heterogeneous environment, because at the runtime the computing node's load is not known. To address this problem, an ant colony algorithm (SP-Ant) [60] is proposed as a joint venture of the bin-packing algorithm and the ACO algorithm.

Its discoveries the best operator mapping considering the inter-operator communication by assigning highly communicating operators on the same nodes using the bin-packing algorithm and schedules the remaining operators using the ant colony optimization (ACO) algorithm. It continues to improve based on real-time feedback from the already assigned operators. The time complexity of SP-Ant is $O(T_c \times 2 \times Max_{it} \times N_{Ant} \times |E|)$, and It is compared with Storm (2.1.0) and R-Storm using Word Count topology on a cluster of 5 virtual machines. Results show a 50% reduction in response time.

### 2.2.4 Scheduling Algorithms for Heterogeneous Cluster

Due to the heterogeneity of computing clusters, blindly increasing the number of workers for a job could degrade the overall performance. So the question of how to distribute the computational task over the heterogeneous cluster to maximize performance is addressed by Xue et al. [26]. Based on the practical constraints of current systems, the problem is addressed in two scenarios. For systems using a hash-based partition method (for avoiding the overhead of indexing and searching vertex), a coarse-grained mechanism is proposed to greedily select suitable workers set to execute the job. For systems allowing arbitrary graph partition, a heterogeneity-aware streaming graph partitioning model that can assign workload at a fine-grained level is presented.

The proposed framework [26] reduces the execution time by 55.9% for the lab cluster and 44.7% for the EC2 cluster, respectively. However, the computing complexity of the coarse-grained scheduler is O($n^3$) and NP-hard for graph partitioning problems. The graph datasets employed in the experiments include social network graphs with different scales of vertices (i.e., 25 million and 114 million). Comparison is performed with PageRank, random walk, and single-source shortest path. The experiments were conducted on a cluster of 46 workers and an Amazon EC2 cluster with 100 instances.

The default Storm scheduler distributes tasks equally among computing nodes in a round-robin fashion regardless of their heterogeneity. This may introduce a potential performance bottleneck due to unbalanced workload distribution with respect to computation resources. Yangyang Liu [61] proposed TOSS – the topology-based scheduling algorithm to address this issue. TOSS improves performance by reducing the communication overhead with the time complexity of *O(V+E)*. TOSS works in two phases to achieve efficiency. In the first phase, TOSS analyses the topology structure and partition executors in such a way that it minimizes the communication overhead between executors. Then, it utilizes run-time workload information to estimate the current workload. TOSS was compared with default Storm scheduler version 0.8.2 which resulted in a 24% boost for throughput and a 20% reduction in latency for SOL and Rolling WordCount topology. A small cluster of 5 machines was used for experiments. TOSS has addressed performance bottleneck issues; however, it also suffers from a cold start problem. Similarly, while generating a schedule for a topology the network structure is not considered by TOSS.

In 2018, Eskandari et al. [16] proposed T3-Scheduler in Apache Storm 1.1.1, a Topology, and Traffic-aware two-level scheduler which finds highly communicating executors and assigns them to the same node in a heterogeneous cluster such that each node remains fully utilized. In the first level, it divides topology into multiple parts based on the executor's communication frequency to reduce traffic by placing highly communicating executors closer. In the second level, it finds the suitable allocation by arranging highly communicating parts in the same worker process to minimize the communication between the worker processes within a node. T3-Scheduler was compared with Online Scheduler and R-Storm and results show that T3-Scheduler increases throughput by up to 32% for the two real-world applications on a 10-nodes cluster. T3-Scheduler has addressed the heterogeneity issue but for linear and star topologies there is a minor improvement in average throughput.

TABLE 2.8: Summary of Scheduling Algorithm for Heterogeneous Cluster

| **Authors / Algo.** | **Scheduling Aspects** | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Dynamic** | **Traffic-aware** | **Topology-aware** | **Heterogeneous** | **Adaptive** | **Resource-aware** | **Network-aware** |
| Xue et al. [26] | × | √ | √ | √ | √ | × | × |
| Yangyang Liu [61] / TOSS | × | √ | √ | √ | × | × | × |
| Eskandari et al. [16] / T3-Scheduler | √ | √ | √ | √ | × | √ | × |
| Nasiri et al. [32] | √ | × | √ | √ | √ | × | × |
| Hadian et al. [62] / ER-Storm | √ | √ | × | √ | √ | √ | × |

The aspect that controls the overall performance in Apache Storm is the scheduling strategy of the topology's components. Gulzar Ahmad et al. [37] proposed their strategy which comprises of two phases. First, the topology's DAG is segregated in a way that least amount of data transfer is required between partitions. In the second phase, the allotment of each partition on a single node decreases the data movement within the task of a partition which improves the performance of job execution. Moreover, task replication is also integrated to further decreases the execution time. Results have shown 21% improvement in execution time for different data sets. Performance comparison of this work was made with default scheduler, R-Storm, and GA using EURExpressII workflow on a 5 nodes cluster. The proposed solution focuses on the reduction of inter-node traffic while ignoring inter-slot traffic which may result in performance degradation. Moreover, each partition is mapped to a single machine. If available machines are less than the required number, then this scheduler will not work. Similarly, partial task duplication is overprovisioning which is also an overhead.

Elasticity has become an essential attribute of SPEs to handle variations in the input data rate as well as encountering heavy workloads. Elasticity can limit

the performance degradation in this situation negatively impacting the overall throughput. Hamid et al. [62] proposed ER-Storm to address the issue of elasticity in a heterogeneous environment. The working principles of ER-Storm are to find out suitable operators for replication. Followed by the selection of worker nodes for hosting while avoiding frequent scaling operations. Operators with the least communications that consume computational resources equal to the bottleneck operator are selected and replicated to other nodes using Markov Decision Process. Reinforcement learning (a subfield of machine learning (ML)) is used to determine the suitable node. A heuristic algorithm based on the bin-packing (FFD) algorithm is employed for the generation of an initial execution plan. In the learning phase, the number of scaling decisions is controlled by using the Naïve Bayes classifier (a probabilistic ML algorithm used for classification tasks) to guess whether scaling would enhance throughput or not. ER-Storm is implemented using Storm 2.1.0 with the overall time complexity of $O(|W|log|W|)$ and a comparison was made with R-Storm and Online-Scheduler. Word Count topology and SentiStorm topology are used for experimentation. The underlying cluster comprises 10 virtual machines which were divided into three computational capacity categories, namely 3 high, 4 medium, and 3 low VMs. In addition, a physical server with 12 cores and 16 GB of memory was dedicated to the Nimbus and Zookeeper. Results showed up to 60% reduction in the response time. Despite the results obtained, ER-Storm exhibits some limitations when compared to our proposed schedulers. The major weakness of ER-Storm is that it operates on the principle of elasticity to achieve scalability, which leads to increased end-to-end delays. Similarly, operators' parallelism (duplication) or relocation to other nodes incurs additional computational and memory costs. These factors have an added negative impact due to the operators' statelessness in ER-Storm. Moreover, frequent scaling decisions can potentially result in performance degradation. To identify overutilized worker nodes, a threshold of 70 percent CPU utilization is utilized. Once the CPU utilization exceeds this threshold, the corresponding node is identified as overutilized. Consequently, approximately 30 percent of the node's resources remain

TABLE 2.9: Strength / Weakness of Scheduling Algorithm for Heterogeneous Cluster

| **Authors** | **Strength** | **Weakness** |
|---|---|---|
| Xue et al. [26] | Heterogeneity-aware graph partitioning model | Solution's Computing complexity is NP-hard |
| Yangyang Liu [61] | Graph algorithm is used for traversing DAG | Network structure is ignored |
| Eskandari et al. [16] | Optimizes assignment by clustering communicating tasks | Minor improvement in average throughput |
| Nasiri et al. [32] | Optimal solution | Utilizes a time-consuming brute-force algorithm |
| Hadian et al. [62] | Utilizes ML and heuristics for scheduling | Performs replication to alleviate bottlenecks |

unutilized.

First suitable executors' selection and then finding an appropriate mapping between these executors and supervisor nodes have a decisive effect on throughput and resource utilization on large-scale clusters. To overcome this effect, a heterogeneity-aware scheduler is proposed [32] that finds the proper number of executors of a DAG and maps them to the most suitable node. The algorithm starts to scale up the topology's DAG over a given cluster by increasing the topology input rate and generating new tasks from bottlenecked executors until an optimal solution is reached. When compared to the default scheduler of Storm (version 0.9.5), the proposed scheduler provides up to 44% throughput enhancement. PageLoad topology, Processing topology, and Network Monitoring topology, are used to evaluate this work on a small network of 4 computing nodes. A brute-force algorithm is used for scheduling which takes 18 hours for producing results which is a major drawback of this work.

As the volume of data increases over time, it poses a challenge for SPEs to predict the resource and application requirements for processing and may cause problem in achieving maximum throughput. Rizwan et al. proposed WG-Storm [63] to address this issue. WG-Storm is based on DAG that enhances the resource usage and overall throughput using efficient tasks assignment. WG-Storm scheduler is divided into four phases. First, monitors the connectivity between tasks which

makes WG-Storm topology-aware. In the second step, a weighted graph based on the communication between tasks is constructed. In the third step, the task is logically grouped using topology's DAG which reduces the inter-group communication according to inter-tasks traffic. Finally, these groups are mapped to the physical machines. WG-Storm scheduler is compared with 5 state-of-art schedulers with 30% improved throughput has been reported.

### 2.2.5 Task Migration Scheduling Algorithms

Real-time scaling of distributed data processing system causes the migration. Storm does not handle task states during task migration. Storm shuts down all worker processes of the topology and starts new ones with the new configuration. During this period, storm waits for in-progress worker processes to complete their tasks. Yang et al. [64] proposed methods for worker/executor level migration in Storm. Instead of killing all the worker processes, the storm only kills the migrating executors and starts them on a newly assigned node. This system improves migration performance by reducing migration time.

Van der Veen et al. [65] have also proposed a scheduler for Apache Storm with a different approach. First, resource over-utilization or under-utilization is identified. Addition or removal of nodes is performed for under-provisioned or over-provisioning of the resources. The proposed methodology is compared with Storm version 0.9.2 and experiments are carried out on a cluster of 1 master node, 1 zookeeper node, and several processing nodes. The major drawback of this work is that it ignores network structure while assigning tasks to a cluster. Due to this, equal tasks assignment is performed despite their physical distance which may introduce latency. With the help of this scheduler, the operational cost can be reduced by provisioning only the required resources for topology execution.

Distributed stream processing (DSP) applications are typically deployed on large-scale Cloud data centres that are often distant from data sources. As data increase in size, pushing them toward the Internet can introduce unnecessary latency. A

solution to reduce this latency lies in moving the computation to the edges of the network close to data sources. Within a set of available distributed nodes, the nodes that should host and execute each operator of a DSP application, to optimize the Quality-of-Service attributes of the application are known as the operator placement problem. Cardellini et al. studied the operator placement problem for distributed DSP applications in their work [66]. They provided a general formulation of the optimal DSP placement as an Integer Linear Programming problem for a heterogeneous environment. An Optimal DSP Placement (ODP-based) scheduler with NP-Hard complexity is presented using Apache Storm (named S-ODP). Experiments have been performed using Apache Storm 0.9.3 on a cluster of 8 worker nodes, each with 2 worker slots, and 2 further nodes for Nimbus and Zookeeper. In the ODP formulation, different assumptions have been made for example, the data stream traffic between operators does not saturate the logical link bandwidth. Therefore, S-ODP is suitable for a situation where available bandwidth is always greater than the required bandwidth.

The existing task migration schemes in Apache Storm are inefficient because they use a process-level migration scheme, which requires the entire process to be stopped and restarted during a task migration. This can lead to significant performance degradation, especially for long-running tasks. To address this issue, a new scheduler for Storm, called N-Storm [67] is presented. N-Storm schedules tasks on a per-thread basis, which allows it to better utilize the resources available on each worker. It uses several techniques including thread-level migration, incremental migration and Load balancing etc. Results showed increased throughput and saved significant migration time.

Task scheduling in SPEs is an NP-complete problem. Due to the instability and variability of input data streams, current schemes have a slow start process, which affects the performance e.g. throughput, latency etc. Moreover, idle nodes at runtime may result in large energy consumption. To address these issues, an energy efficient and runtime-aware framework (Er-Stream) has been presented [68] in this work. It handles changes in data streams at runtime to adjust task migration adaptively. Karush-Kuhn-Tucker mathematical constraints are used to

model node resources and energy consumption to increase the energy utilization of the cluster. Task pairs with high communication are mapped onto the same node to minimize the network communication. At runtime, task migration is performed based on node communication and nodes' CPU frequencies are adjusted dynamically based on resource usage information to lower the energy consumption.

### 2.2.6 SLA-Based Scheduling Algorithms

In 2017, X. Cai et al. presented [69], an SLA-aware energy-efficient scheduler that allocates resources to MapReduce applications using YARN (Yet Another Resource Negotiator) architecture. During task scheduling, the data locality information is carefully used to reduce network traffic. A dynamic voltage and frequency scaling scheme is designed to dynamically change the CPU frequency for upcoming tasks. The proposed scheme outperforms the existing scheduler for resource utilization and energy efficiency. The weakness of this work is that the proposed scheduler orders jobs by the Earliest Deadline First algorithm, as a result, the least priority jobs will have to wait long for their turn.

Similarly in another work [70], an expandable resource provisioning method is proposed which ensures SLA-compliant scheduling of jobs while maximizing resource usage and minimizing the operational cost. As a result, a cost reduction of around 12% was achieved as compared to the existing methods. The major limitation of this work is its time complexity which is NP-hard for worst case scenarios.

In 2019, An adaptive watermarking mechanism for streaming frameworks was proposed [71] to enhance system throughput while maintaining the SLA-based end-to-end system latency. It focuses on tuning the system parameters on the expectation of incoming jobs and evaluates whether a given job will violate an SLA based on throughput, and latency. Different modes are used to maintain the trade-off between latency and throughput.

As more and more consumers assign their tasks to the cloud, SLAs between consumers and providers are emerging as an important aspect. To meet this requirement, [72] proposed an SLA-aware load balancing algorithm for cloud computing which migrates tasks from overloaded nodes to the underloaded nodes having the highest capability. The experimental results prove that the SLA-aware load-balancing algorithm is cost-effective and has a minimum makespan as compared to RR, max-min, and ABC algorithms. It also reduces SLA violations using prediction models with well-timed scaling.

Altaf et al. [73] focus on the low-resource utilization problem in Cloud scheduling to ease the load imbalance issue. It schedules the SLA-based jobs in a load-balanced manner to improve resource utilization along with execution cost. Performance evaluation reveals that the proposed scheduler attains increased resource utilization as compared to existing state-of-the-art schedulers using the Google Cloud Jobs dataset. Similarly, an SLA-aware optimal resource allocation algorithm [74] was presented for multi-tier computing architecture. Two different models were designed. (i) a realistic mixed-integer nonlinear programming model which adopts the requirements of the services, and (ii) a mixed-integer linear programming (MILP) model that employs linear approximation to convert it into a MILP model for solving the time and space complexity problems. Furthermore, a nearest-fit algorithm is also presented where the optimization models are unable to obtain a viable solution within an appropriate time limit.

In this paper [75], a CSL-driven scheduler to enhance energy efficiency in the cloud data center is proposed. Both CSL requirements and energy saving are studied in this approach. Three scheduling approaches (aiming to minimize energy-saving, maximize CSL and maximize CSL per energy, respectively) are employed to consolidate resources according to different CSL states.

A novel SLA-based cloud coalition approach is presented in this work [76] in which, each request includes the required resources with SLA requirements. Then it finds the best coalition of cloud providers to fulfil the request while satisfying SLA

requirements. The main objectives of this approach are 1) to maximize coalition profit while ensuring a fair profit-sharing between members; 2) to minimize the number of coalition members, and 3) to reduce the penalty who fail to offer the promised resources. The proposed solution leverages Irving's roommate matching algorithm to propose an SLA-based cloud federation formation methodology. The drawback of this technique is its time complexity which is $O(n^3)$ in the worst case. When the coalition size becomes large, the execution time of this approach becomes huge which results in a big execution delay.

Cloud computing is a flexible, low-cost, on-demand service platform that exists over the internet. Like other fields, this domain also demands proper management of resources for efficient jobs execution. As this is a shared infrastructure and different customers using it in a platform-as-a-Service (PaaS) fashion. Due to a shared environment, service-level agreement (SLA) between customers and service providers is becoming an important ingredient to measure customer satisfaction level (CSL). SLA helps in imposing different constraints like security, quality of service, pricing, and period of service. At this time, SPEs missing an SLA-based scheduling mechanism that can improve CSL as well as resource utilization on Cloud. To address this need, SLA-A3-Storm scheduler to map submitted jobs on a heterogeneous cluster to enhance resource usage according to end-users budget in an SLA-aware fashion is presented. End-user selects SLA plan at the time of job submission which consists of performance-sensitive, standard, and cost-sensitive. All tasks are positioned according to their communication traffic requiring high computation power then mapped to nodes with high computing power listed under the selected SLA plan. This resource-aware assignment ensures better throughput and optimized resource utilization according to end-user needs. As compared to the benchmark schedulers, SLA-A3-Storm saves up to 50% in terms of operational cost with better throughput.

The proliferation of data generated by the Internet in recent years has presented significant challenges in terms of energy consumption for data processing in computational clusters. These clusters consume energy while executing tasks for data processing, making energy-efficient scheduling a crucial concern. Efficient task

scheduling can help minimize energy wastage, and it is important for the scheduler to minimize the energy consumption of the cluster while satisfying SLAs, including deadline constraints. In this paper [77], two energy-efficient schedulers are proposed to reduce energy consumption in SPEs. The first scheduler focuses on minimizing the energy consumption of tasks by assigning them to low-energy consumption nodes. The second scheduler optimizes both load balance and energy consumption by sorting the slot utilization of low-energy consumption nodes in the cluster and assigning task priorities to nodes with low slot utilization. While the proposed schedulers may increase the execution time for different workloads, it still satisfies SLA conditions and reduces the energy consumption of the cluster. The results demonstrated that the proposed schedulers reduced energy consumption by up to 32% compared to the benchmark scheduler, while still satisfying the deadline constraints.

### 2.2.7 Other Scheduling Algorithms

A scheduling scheme for SDN-based cluster systems is proposed in [78]. SDN provides a centralized view of the entire network and allows controlled operations related to packet switching with ease to manipulate large-scale cloud systems. According to the authors, most of the existing solutions do not consider fine-grained bandwidth-related information and are unable to utilize the full network information for scheduling decisions. To address these issues, the authors proposed an SDN-enabled scheduling scheme called BAHS. The principal idea for the BAHS is to utilize the information on network interference by incorporating only those paths resulting in the lowest network interference. To analyze the path interference or conflicts, the proposed scheme uses the Dynamic Online Routing Algorithm mechanism. The experiments are performed in a simulated environment called Mininet with an improved completion time of up to 14.49%. In contrast to this work, we propose a resource-aware mechanism that considers both computing and network-related aspects for scheduling decisions. Moreover, we use a real

stream processing system (i.e., Apache Storm) to implement and demonstrate the performance as compared to the other state-of-the-art schemes.

In [79], the authors presented a heuristic approach to Cloud resource allocation and scheduling. The authors combined several approaches such as the Modified Analytic Hierarchy Process (MAHP), BAR optimization applied to a Bandwidth-Aware divisible scheduling (BATS), and to address the load imbalance issue longest expected process time (LEPT) preemption is used along with the divide-and-conquer methods. For resource allocation and mapping, first, the tasks are ranked using the MAHP approach. After that, the resources are assigned to the ranked tasks using Bandwidth-Aware divisible scheduling with BAR optimization. Using LEPT methods, Virtual machines' load is checked and an overloaded VM is managed using a divide-and-conquer scheme. The proposed scheduling framework is evaluated using the CloudSim simulator. In contrast to this work, our proposed solution is capable of schedule and process real-time stream jobs in a Big-Data Cloud. Moreover, we use a real stream processing system (i.e., Apache Storm) to implement and demonstrate the performance in contrast to using a simulated Cloud environment.

Mortazavi-Dehkordi & Zamanifar [80] proposed a framework, which examines the topology to determine the output size of its operators for analysis, and partitions. Different scheduling approaches are applied for each. The operators assigned to thread computing units and threads that are identified are assigned to processes. The authors also proposed a scheduler for dynamic environments (offline schedulers). The offline scheduler assigns processes to the available nodes. The purpose of a scheduler is to reduce the latency and balance the operator workload. However, the authors did not consider memory in resources.

Stavros Souravlas et al. [81] proposed a method based on pipeline modular arithmetic (PMOD scheduler), which is based on each node using tuples at the same time to process only from another node. The scheduler organizes all necessary operations in a pipelined manner, such as tuple packing, tuple transmission, and tuple processing. As a result, the overall execution time is decreased compared

to other approaches. The scheduler maximizes load balancing, but also increases throughput and minimizes buffer requirements.

The increasing amount of data generated by the Internet of Things (IoT) devices is driving the need for efficient stream processing solutions. SPEs are not well-suited for edge computing due to the heterogeneity of the computational and network resources available at the edge. Amnis [82] proposed a new stream processing framework designed for edge computing. It first employs a unique data locality-aware approach to optimize resource allocation and considers the resource requirements for each operation. It dynamically changing load while scheduling tasks to further improve performance. Results showed 200 times improvement on latency and 10 times on the throughput compared to the RA-Storm.

Tianyu Qi et al. [83] introduced a dynamic scheduling strategy based on graph partitioning and real-time traffic data monitoring. The scheduler monitors the exchange rate of a tuple within different communication tasks. Assign tasks according to the resource requirements and the traffic pattern to achieve high throughput and low latency. The proposed approach is a two-level technique. First, the topology graph is assigned a weight based on the type of communication. At the second level, the schedulers find multiple partitions and also find unfeasible partitions, by these partitions the authors achieve load balancing of the tasks and minimize tuple communication. After identifying the best partition, the scheduler assigns the tasks in the cluster.

Sun et al. [84] proposed a model to determine the cumulative processing delay (latency) and computing power of each operator and also the similarity of bolts. The authors also considered bandwidth and the communication latency between inter-worker communications. The author's approach also reduces the overall communication latency by putting the highly communicating tasks into the same worker and also increases the overall throughput. However, the authors ignore the I/0 bounds and memory bounds jobs.

This work [85] discusses a learned cost estimation model for Stream Processing Framework to provide accurate cost predictions for executing queries, enabling

optimal operator placement. Once trained, the proposed model can predict performance measures such as latency and throughput, even in the presence of runtime changes. This model can be applied to achieve optimal operator placement, thereby minimizing latency in unstable environments. Additionally, a zero-shot cost model is presented, which can handle unseen scenarios and predict different hardware configurations using a transferable feature representation. The model is capable of predicting Quality of Service metrics for queries across a vast collection of configurations. The results demonstrate that the proposed model accurately predicts the performance of queries for unknown tasks, even without explicit training for them.

SPEs are widely used for real-time processing of large data volumes due to their low latency. Stream joins serve as the basis for SPEs as they analyze data from multiple sources. However, stream joins face load imbalance issues, where a few nodes become bottlenecks, resulting in increased latency. To address this issue, an adaptive non-migrating load-balancing method is proposed [86] based on stream window join problems. This method monitors load distribution at runtime, controls tuple replication and forwarding through routing tables, and adjusts tuple partitioning to achieve load balancing with minimal overhead. A distributed stream window join system called NM-Join was built on Flink, based on the proposed method. Experimental results demonstrated reduced latency with increased throughput compared to static load-balancing methods with modest input skew. It also adapted to input fluctuations, similar to migration-based methods, while significantly reducing the overhead of load balancing and processing latency.

SPEs are used to run mission-critical applications for real-time monitoring that require high throughput to process incoming data in real time. Changes in the distribution of incoming data can result in partition skew, causing an uneven distribution of data partitions and sub-optimal processing. To address this issue, Josip et al. [87] propose a solution that balances the load and reduces network traffic by ensuring the ideal locality of the data. The proposed solution offers three modes: first, determining where the join is performed; second, detecting imbalances; and lastly, adapting to the detected imbalances. It employs modern

principles to monitor the load, detect imbalances, and dynamically redistribute partitions to achieve optimal load balance. The proposed solution leverages the collocation of streaming data while considering the processing load of the join. The evaluation demonstrated that the dynamic mode effectively manages load imbalances through dynamic partitioning and load balancing among the workers.

Quick processing with low latency is a critical requirement for big data applications. However, interferences between tasks can decrease resource utilization and increase latency. This work [88] focuses on studying an optimal scheduling method for processing big data applications with reduced interferences. The scheduling problem was modeled considering multiple factors, such as data items, processing tasks, computing resources, and internal cores. To address task interferences, a two-stage scheduling problem is formulated, which examines the causes of interferences and emphasizes the relationship between the task's type and execution using linear regression. The completion time of each node can be estimated. The proposed scheduler was compared with static, reactive, proactive, and a coarse-grained scheduler in terms of load balance, resource utilization, and throughput. The results demonstrated that the proposed scheduler achieved low interference with high resource utilization.

## 2.3 Critical Analysis of Related Work

Tables 2.2, 2.4, 2.6, and 2.8 present a summary of the scrutinized scheduling algorithms. The tables contain the scheduling aspects that have been addressed by those papers. Although different aspects have been investigated in the literature but here the most widely used aspects are selected. The $\surd$ symbol means the presence of an aspect in the scheduler. Conversely, $\times$ shows that a particular scheduler does not contain this aspect. Similarly, Tables 2.3, 2.5, 2.7, and 2.9 contain strength and weakness of these algorithms. Moreover, Table 2.10 shows performance metrics used in the literature by different researchers. Here + sign is

showing improvement and - sign indicates the other way. For example, +30% for throughput means that 30% improvement in throughput has been achieved. However, -20% under the latency heading shows that 20% less latency has occurred.

The Default scheduler allocates tasks without considering inter-task traffic, which makes a substantial effect on the performance [28]. Additionally, the application's computing requirements and communication pattern are ignored by the default Storm scheduler which causes delays and lower throughput. This issue is addressed in [28] by proposing a traffic-aware scheduler. The major drawback of this scheduler is that it reduces inter-node communication; moreover, it also ignores inter-process communication, which may become a performance bottleneck. To address the same problem, P-Scheduler [29] is proposed. P-Scheduler has a major drawback in that it is designed for homogeneous clusters only.

Another problem of the Default Storm Scheduler is that it occupies all the available resources, regardless of the load which resulted in low resource utilization and higher computing costs. In [33] above mentioned issue has been addressed; however, the proposed methodology suffers from a cold start problem. The same issue is also targeted by [65] but the proposed model does not consider the physical distance between nodes which may cause a delay due to bandwidth differences.

Another limitation of Default Storm Scheduler is that its topology is static. [56] provided a solution to this problem by over-provisioning resources. The same problem is addressed in [25] by presenting AdaStorm but AdaStorm neither considers network topology nor handles any change in the cluster. While scheduling, the default scheduler neither considers resource availability in the underlying cluster nor the resource requirement of Storm topologies. Peng et al. [22] overcome this issue but their scheduler does not handle run-time topological changes. In [19] D-Storm is presented to address a similar problem, but it suffers a cold start issue. It is implemented for homogeneous clusters and does not consider the network topology while scheduling. The Storm scheduler is intended for a homogeneous environment which may establish performance bottleneck problems. [89] addressed performance bottleneck issue, but it suffers from a cold start problem. Similarly,

the network structure is not considered while generating a schedule for a topology [24, 55].

The detailed critical analysis of the existing research work done in this domain reveals that the existing scheduling schemes are unable to optimally schedule jobs on a computational cluster. Therefore, it is desirable to have a dynamic resource-aware scheduler that can distribute jobs among heterogeneous clusters to achieve maximum throughput with minimum resources.

TABLE 2.10: Performance Metric Used in the Literature.

| Reference / Algorithm Name | Performance Metric | | | |
|---|---|---|---|---|
| | **Resource Utilization** | **Throughput** | **Latency** | **Time Complexity** |
| Aniello et al. [28] / Offline & Online Scheduler | - | +30% | - | Greedy heuristic |
| Xu et al. [33] / T-Storm | - | +84% | - | $O(N_e logN_e + N_e N_s)$ |
| Fischer and Bernstein [35] | - | +56% | - | - |
| Madsen and Zhou [56] / MPSPE | - | - | -20% | $O(\|N\| \times$ #checkpoints) |
| Xue et al. [26] | - | - | -55.90% | $O(n^3)$<br>NP-Hard |
| Peng et al. [22] / R-Storm | +350% CPU utilization | +50% | - | - |
| Eskandari et al. [29] / P-Scheduler | - | - | -50% | NP-Complete |
| Fan et al. [17] / Adaptive-Scheduler | -3% extra resources used | +67% | - | - |
| Yangyang Liu [61] / TOSS | - | +24% | -20% | O (V + E) |
| Smirnov et al. [24] | - | +40% | - | - |
| Li and Zhang [31] / TS-Storm | - | +91% | -50% | - |
| Qiann et al. [58] / S-Storm | - | +1.25 times | -18.70% | - |
| Liu et al. [57] / E-Storm | - | +9 times | -9.80% | NP-Hard |

**Table 2.10 continued from previous page**

| Reference / Algorithm Name | Performance Metric | | | |
|---|---|---|---|---|
| | Resource Utilization | Throughput | Latency | Time Complexity |
| Weng et al. [25] / AdaStorm | +15% CPU<br>+60% Memory | - | - | $O(n^2)$ |
| Liu and Buyya [19] / D-Storm | - | +16.25% | - | $O(n^2)$ |
| Eskandari et al. [16] / T3-Scheduler | - | +32% | - | - |
| Li et al. [30] / DRL-Based framework | - | +33.50% | - | - |
| Gulzar Ahmad et al. [37] / PDWA | - | +21% | - | - |
| Nasiri et al. [32] | - | +44% | - | - |
| Al-Sinayyid and Zhu [15] / MT-Scheduler | - | +54% | -46% | Polynomial-time |

# Chapter 3

# The Proposed System and Schedulers

With extensive literature review, it is revealed that the existing schedulers are unable to optimally schedule jobs for SPEs. An optimum schedule achieves maximum throughput with minimum resources. Therefore, a dynamic resource-aware scheduler is desirable to distribute jobs among heterogeneous clusters to achieve maximum throughput with minimum resources. To address this need, a topology-aware, traffic-aware, and resource-aware system for heterogeneous cluster has been proposed in this chapter. Moreover, based on this system, four different scheduling schemes (**TOP-Storm**, **A3-Storm**, **BAN-Storm** and **Gr-Storm**)[1] are also presented to answer the research questions raised in section 1.4.

The proposed system assigns topologies to a heterogeneous cluster to improve resource utilization and increase throughput. It maps topologies by contemplating computational requirements of topologies and computation power of the node in the cluster. All unassigned executors are arranged either according to their connectivity (represented by DAG) or communication traffic (historical communication of topology) requiring high computation power. These (frequently communicating) executors are mapped to a node with high computing power. This resource-aware assignment ensures higher throughput with better resource utilization.

[1] https://github.com/asif-muhammad-malik

## 3.1 System Architecture

The system architecture is depicted in Fig. 3.1. The proposed system consists of 5 major modules:

1. **Resource-aware Adaptive Storm Scheduler**
2. **Network Agent**
3. **Resource Manager**
4. **Tune Scheduling**
5. **Provenance Data**

As discussed earlier, the Apache Storm cluster consists of a nimbus and supervisor nodes. The main job of the nimbus is to run the Storm topology. Nimbus analyzes the topology and collects the tasks to be executed. Next, it distributes the tasks among the available supervisors. In the proposed system, the Resource-aware Adaptive Storm Scheduler module is running on the nimbus node. It is a generic module and in the subsequent sections this module is replaced with four different schedulers (TOP-Storm, A3-Storm, BAN-Storm, and Gr-Storm). All four schedulers are using the same architecture as shown in Fig. 3.1.

The **Resource-aware Adaptive Storm Scheduler** running on the nimbus node receives input from other modules and makes critical decisions related to a topology mapping. After reading required inputs (such as unassigned executors, inter-executor traffic and executor's connectivity, etc.), a logical mapping of executors to slots is prepared which is then forwarded to supervisor nodes for physical mapping. Once this mapping is applied, the supervisor nodes start executing the scheduled topologies.

**Network Agent** is an important module which is running on each supervisor node. It provides two different types of information related to each supervisor node to the Resource Manager module. Mainly, it retrieves and shares hardware

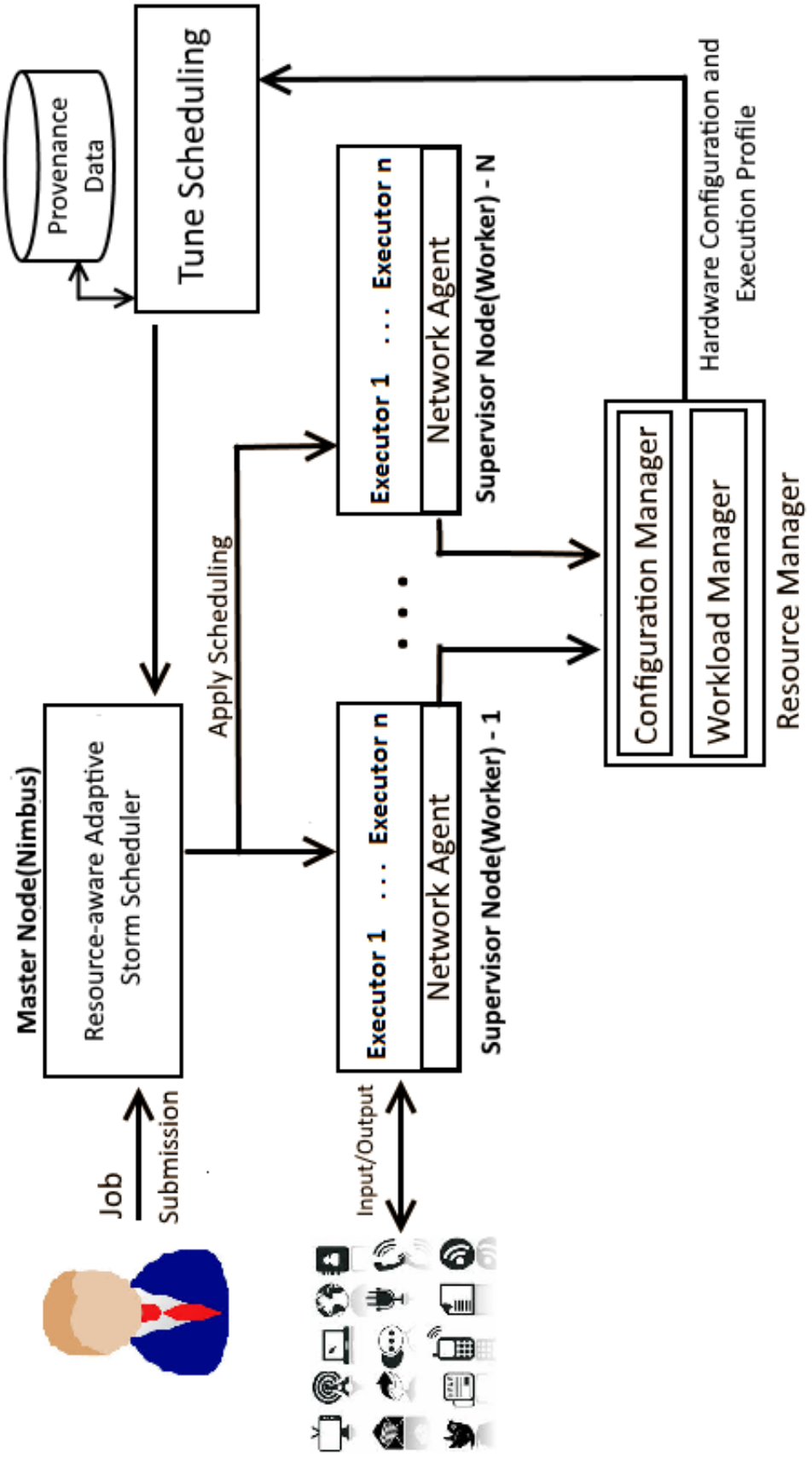



FIGURE 3.1: The Proposed System.

configuration (number of sockets, number of cores, frequency, number of FLOPs, installed RAM and bandwidth) with the Resource Manager which is used to calculate the computation power of a supervisor node later on. Secondly, the network agent provides inter-executor traffic information to the Resource Manager. This information is employed for traffic-based scheduling decisions and is used by the Tune Scheduling module for making critical decisions related to the execution plan. This information is stored in a data repository called **Provenance Data**.

**Resource Manager** consists of two sub-modules.

a) **Configuration Manager**, which keeps track of the available supervisor nodes of the cluster. If a supervisor node joins or leaves the cluster then this module will update the list of available supervisors. Secondly, this module also calculates the computation power of each supervisor node in sorted order for future use.

b) **Traffic Manager**, logs communication traffic occurring among executors at the time of topology execution. A list of frequently communicating executors is generated with the help of this communication log which is used to group executors to reduce inter-node communication.

**Tune Scheduling** is a real-time monitoring module which measures how the streaming application performs under a real workload. With the help of this module, adaptiveness in the proposed system is achieved. First using the **Provenance Data**, it acquires the computation power of each node. Next, the overall traffic is calculated based on inter-node, inter-slot, and inter-executor traffic patterns. Considering the traffic pattern and the resources utilized, it is determined whether any amendment in the execution plan is required or not. Based on the amendments, the execution plan (task mapping) is re-adjusted and fine-tuned accordingly.

Another important responsibility performed by the **Resource Manager** is the calculation of the computation power of each supervisor node. The computation power of each node is measured in terms of Floating Point Operations Per Second

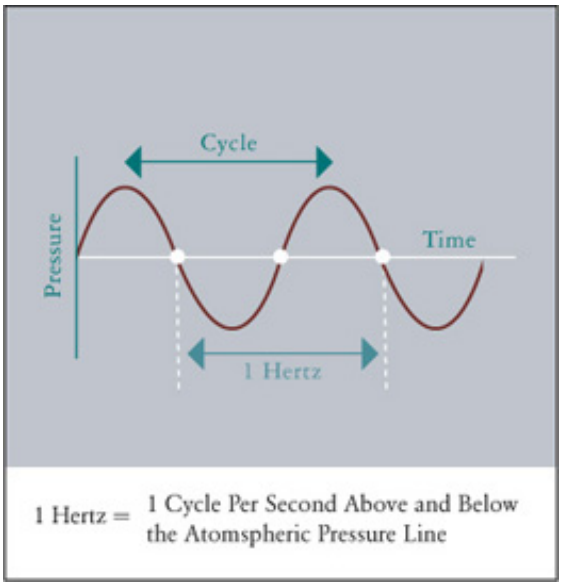



FIGURE 3.2: Exploring the Speed of Processors: Cycles per Second (Hz) [90].

(FLOPS) which is calculated using Equation 3.1 [91, 92]. To deal with the heterogeneity of the computing node, the FLOPS of that node is employed. FLOPS are used to measure the performance of a processor while clock speed indicates the speed of a processor that can not define the total number of calculations a processor can perform in a second. Therefore, FLOPS is a better way of knowing the throughput of a processor [93].

$$Processing\_Speed = number\_of\_cores \times cycles\_per\_second \times flops\_per\_cycle \tag{3.1}$$

[91, 92]

where *number of cores* represents the total cores of a processor, *cycles per second*, also known as Hertz (Hz), is a unit of frequency that represents the number of clock cycles a processor can execute in one second (as shown in fig. 3.2). Similarly, *flops per cycle* is a measure of how many floating-point operations a processor can perform in a single clock cycle. To accommodate the computer's memory resource as it also plays an important role in processing, Equation 3.2 is devised to highlight the processing and memory resources in a weighted form.

TABLE 3.1: Node Indexing for $\alpha$ Value 0.8

| **Name** | **CPU (GHz)** | **RAM (GB)** | **FLOPs /Cycle** | **# Cores** | **CPU Index ($\alpha$=0.8)** |
|---|---|---|---|---|---|
| Node-D | 3.4 | 12 | 16 | 4 | 1 |
| Node-E | 3.2 | 16 | 8 | 8 | 2 |
| Node-C | 3.2 | 10 | 16 | 2 | 3 |
| Node-B | 2.8 | 8 | 4 | 4 | 4 |
| Node-A | 2.4 | 4 | 4 | 4 | 5 |
| Node-F | 2.8 | 12 | 4 | 2 | 6 |

$$ComputationPower = \alpha \times Processing\_Speed + (1 - \alpha) \times RAM \tag{3.2}$$

### 3.1.1 Analysis of Adjustment Factor $\alpha$

In a computing node, installed memory plays an important role. To accommodate the effect of memory resources, an adjustment factor $\alpha$ with a value of 0.8 for the proposed scheduling heuristic is employed in Equation 3.2. Different $\alpha$ values from 0 to 1.0 are tried and finally, 0.8 is selected. With $\alpha$ equal to 0.8, the machines with high computing power are indexed on the top. Similarly, if $\alpha$ is assigned to 0.2 then the machines are indexed with respect to the RAM size.

According to the employed experimental cluster, Node-D represents the most powerful machine in the cluster followed by Node-E. Similarly, Node-E has a maximum of 16 GB of memory whereas Node-D and Node-F contain 12 GB of memory. If $\alpha$ is assigned to 0.8 then the Node-D is indexed on top followed by Node-E (resulting in the desired ordering) as shown in table 3.1.

Similarly, if $\alpha$ is assigned 0.2 then the Node-E is indexed on top of the cluster followed by Node-D. Considering the memory installed in the cluster nodes, the node's indexing seems fair (see Table 3.2). This adjustment factor can plan an

TABLE 3.2: Node Indexing for $\alpha$ Value 0.2

| Name | CPU (GHz) | RAM (GB) | FLOPs /Cycle | # Cores | Memory Index ($\alpha$=0.2) |
|---|---|---|---|---|---|
| Node-E | 3.2 | 16 | 8 | 8 | 1 |
| Node-D | 3.4 | 12 | 16 | 4 | 2 |
| Node-C | 3.2 | 10 | 16 | 2 | 3 |
| Node-B | 2.8 | 8 | 4 | 4 | 4 |
| Node-F | 2.8 | 12 | 4 | 2 | 5 |
| Node-A | 2.4 | 4 | 4 | 4 | 6 |

important role when we have different nature of topologies for example CPU-bound, Memory-bound etc. For CPU-bound topology, we will assign $\alpha$ to 0.8 and indexed nodes according to their computing power. The first node from this list will be consumed and so on. Similarly, $\alpha$ will be assigned to 0.2 for memory-bound topology.

In the remaining part of this chapter, we will discuss 4 different scheduling algorithms which will be deployed one by one in the Resource-aware Adaptive Storm Scheduler module.

## 3.2 TOP-Storm: A Topology-based Resource aware Scheduler for SPE

To address the first research question given in section 1.4, a topology-based resource-aware scheduler named TOP-Storm is proposed. The TOP-Storm assigns topologies to a heterogeneous cluster to improve resource utilization and increase throughput. The TOP-Storm maps topologies by contemplating the communicational requirements of topologies and the computation power of the node. All unassigned executors are arranged according to their connectivity (represented by DAG). Computationally fast machines are used first ensures higher throughput.

TOP-Storm consists of **two** phases:

1. **Logical grouping** involves the grouping of executors which is done using DAG. With DAG-based grouping, connected executors are placed together closer to each other which reduces latency and improves throughput. This is topology-based scheduling.

2. **Physical mapping** assigns frequently communicating executors group to a node based on the node's computation power (starting from most powerful to least). In this way, Resource-aware scheduling is achieved. This is an iterative process until all groups are mapped to nodes.

### 3.2.1 TOP-Storm Algorithm

For efficient job execution in a topology-based system, a scheduler needs to find directly communicating executors (longest path), which can be assigned to the same supervisor node, reducing the inter-node communication. To achieve this, TOP-Storm uses a heuristic to find such groups to minimize inter-node communication. These groups are assigned to a supervisor node with relative capacity. TOP-Storm consists of **two** main steps:

1. **Executor Grouping**: Topology's DAG is used for executor grouping. Connected executors are placed as close as possible. This is done in Algorithm 2, which is explained below.

2. **Slot Assignment**: Above created groups are assigned to slots. Slots are assigned starting from the most computationally powerful node. This is achieved using Algorithm 3, which is discussed below. Table 3.3 listed the notations used in the TOP-Storm algorithms.

Algorithm 1 is responsible for generating an execution plan based on available resources and connectivity between topology's executors. It takes unassigned topologies as input and node-wise executor assignment is the output of this algorithm.

TABLE 3.3: List of Notations Used in the TOP-Storm Algorithms

| **Notations** | **Description** |
|---|---|
| $\omega$ | Total number of slots required for topology execution |
| $e_u$ | All unassigned executors for a topology |
| $e_a$ | All assigned executors for a topology |
| $e_t$ | Total number of executors for a topology |
| $e_{ps}$ | Maximum executor per slot |
| $e_i$ | Next unassigned executor |
| $e_s$ | Next unassigned source executor |
| $e_d$ | Next unassigned destination executor |
| $e_u.Sort$ | Sort all unassigned executors w.r.t. connections in descending order |
| $\alpha$ | Adjustment factor |
| s | All unassigned slots of a node |
| $s_a$ | Assigned slot of the current node |
| $s_i$ | A specific slot of the current node |
| $s_n$ | Next free slot of the current node |
| s.Sort | Sort all slots by number of executors in descending order |
| n | All unassigned nodes in the cluster |
| $n_a$ | Available node in the cluster |
| $n_i$ | A specific node in the cluster |
| $n_i$.GFLOPs | Calculate computation power of $i^{th}$ node in GFLOPs |
| n.Sort | Sort unassigned nodes by computation power in descending order |

**Algorithm 1:** TOP-Storm : Main

1 function TOP-Storm ($T$)

**Input :** Unassigned Topologies $T$

```
// This function will receive all unassigned topologies as input, which need
   to be scheduled
```

**Output:** Node-wise executor mapping to cluster's node

```
// The output of the function will be the mapping of all executors of
   unassigned topologies to the cluster's nodes
```

2 **foreach** *topology* $t_i \in T$ **do**

```
// Iterate through all unassigned topologies one by one
```

3 HashMap<source, destination> $e_u = t_i$.getUnassignedExecutors();

```
// Get the inter-executor connectivity of all unassigned executors in
   the form of source-destination pairs
```

4 $\omega = t_i$.NumWorker;

```
// Retrieve the required number of worker processes for topology
   execution
```

5 HashMap<slot,executor> slotMap = MapExecutorToSlot($e_u$, $\omega$);

```
// Invoke the 'MapExecutorToSlot' function, provide it with the
   inter-executor connectivity and the required number of worker
   processes as input
```

6 HashMap<node,slot> nodeMap = MapSlotToNode(slotMap);

```
// Invoke the 'MapSlotToNode' function, provide it with the executor to
   slot mapping as input
```

7 AllocateSlotsToNodes(nodeMap);

```
// Finally, physical mapping is handed over to Apache Storm for topology
   execution
```

8 **end**

As a first step, inter-executor connectivity for all unassigned executors for each topology is retrieved (line 3). Next, it reads the number of slots (worker process) to be used for topology execution (Algorithm 1, line 4). For the logical grouping of executors to a slot, Executor Grouping (Algorithm 2) is used (line 5). Similarly, for physical mapping of a slot to a node, Slot Assignment (Algorithm 3) is invokes (line 6). Finally, physical mapping is handed over to Apache Storm for execution (line 7).

**Algorithm 2:** TOP-Storm : Executor Grouping

```
1  function MapExecutorToSlot (HashMap<source, destination> e_u, ω)
   Input  : HashMap<source, destination> e_u, ω
   // Inter-executor connectivity details
   Output: HashMap<slot,executor> slotMap
   // Executor(s) to slot(s) mapping
2  s_a = 0;
3  e_a = 0;
4  e_t = e_u.Count();
5  e_ps = Ceiling (e_t / ω);
6  while e_u ≠ null do
7      e_s = e_u.getSourceExecutor();
8      e_d = e_u.getDestinationExecutor();
9      if e_s ≠ assigned && e_d ≠ assigned then
10         slotMap.add(s_a,e_s);
11         e_a++;
12         slotMap.add(s_a,e_d);
13         e_a++;
14         counter = 2;
15         while counter < e_ps do
16             e_k = e_u.getFrequentlyInteractingExecutor(e_s, e_d);
               // Frequently interacting executor w.r.t. source or destination
17             slotMap.add(s_a,e_k);
18             e_a++;
19             counter++;
20         end
21         s_a++;
22     end
23 end
24 if e_a < e_t then
25     counter = 1;
26     while e_a < e_t do
27         e_k = e_u.getNextExecutor();
           // Get next unassigned executor
28         slotMap.add(counter % ω,e_k);
           // Round-robin assignment
29         e_a++;
30         counter++;
31     end
32 end
33 return slotMap
```

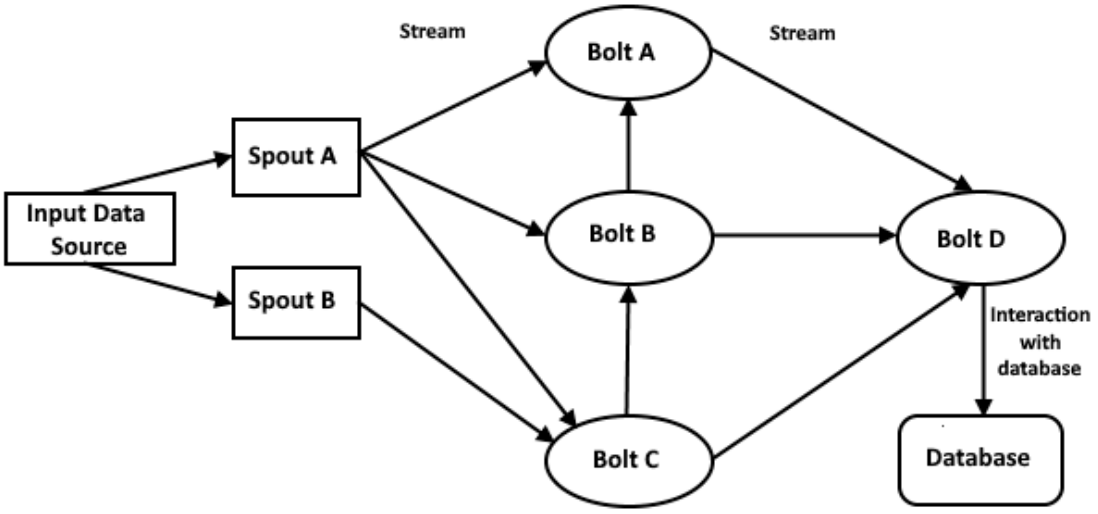



FIGURE 3.3: An Example Topology with Inter-executors Connectivity.

Algorithm 2 receives inter-executor connectivity of unassigned executors and the required number of slots ($\omega$) as input. After processing, algorithm 2 returns the executor to slot mapping as an output. First, It calculates the maximum executors to be mapped in a slot by dividing the total number of executors by the required number of slots (line 5).

Initially, a pair of unassigned executors which are directly connected are selected from the top of the list and assigned to the current slot (lines 7–14). Then, a heuristic is used to select the next executor for the assignment. From the unassigned list, the executor with the highest interacting frequency with the already assigned executors is selected and assigned (lines 15–20). This process continues until the maximum limit per slot is reached. Once this limit reaches then the next slot will be selected for another iteration (line 21). At the end of this process, if there are still unassigned executors, then they will be assigned in a round-robin fashion (lines 24–32).

For a simple case study, consider a topology given in Fig. 3.3 as an example. Here, Spout A, Spout B, Bolt A, Bolt B, Bolt C, and Bolt D represent the executors of the topology and arrows represent the connectivity between the executors. Initially, all these executors are unassigned. With the help of topology's DAG, we can make the following pairs (as shown in Table 3.4):

TABLE 3.4: Inter-executor Connectivity for Example Topology Given in Figure 3.3

| S. # | Source | Destination |
|---|---|---|
| **1** | Spout A | Bolt A |
| **2** | Spout A | Bolt B |
| **3** | Spout A | Bolt C |
| **4** | Spout B | Bolt C |
| **5** | Bolt A | Bolt B |
| **6** | Bolt A | Bolt D |
| **7** | Bolt B | Bolt C |
| **8** | Bolt B | Bolt D |
| **9** | Bolt C | Bolt D |

As a starting point, Algorithm 2 will select Spout A to Bolt A pair for assignment. Then with the help of getFrequentlyInteractingExecutor($e_s, e_d$), Bolt B will be selected because of having maximum interaction with already selected executors (Spout A and Bolt A). Other executors are either connected to Spout A or Bolt A but Bolt B is the only one which is communicating with both executors. Therefore, Bolt B will be selected. Now, we have Spout B, Bolt C and Bolt D in the unassigned list and Spout A, Bolt A and Bolt B are in the assigned list. For the next selection, Spout B does not have any interaction with already selected executors. That's why, Spout B can not be selected. However, Bolt C is connected with Spout A and Bolt B. Similarly, Bolt D is connected with Bolt A and Bolt B. Therefore, Bolt C and Bolt D are equal candidates for the next selection. In this way, executors with different interaction frequencies will be grouped. This process continues until the

maximum limit per slot is reached.

Once executors are assigned to slots by algorithm 2, the next step is to map these slots to computing nodes which is done by algorithm 3. Algorithm 3 receives executors to slots mapping as input and produces slots to nodes mapping. First, algorithm 3 retrieves all supervisor nodes of the cluster and calculates the computation power (in terms of GFLOPs) of each node with the help of equation 3.2 (Algorithm 3, lines 2–14). Heterogenous resource awareness is also handled by this equation. All nodes in the cluster are ranked in terms of GFLOPs irrespective of their hardware specifications.

Next, these nodes are sorted in descending order according to their computation power (lines 15–17). Similarly, slots are sorted in descending order according to the available number of executors (lines 18–20). Lastly, the slot having the highest executors is allocated to the most powerful node and so on. Slots are assigned to a node until all slots are consumed then the next node is used (lines 21–27). In this way, minimum nodes are utilized for topology execution, and inter-node communication is also reduced.

Algorithm 3 is responsible for the calculation of the computation power of each supervisor node. The computation power of each node is measured in Floating Point Operations Per Second (FLOPS) [94] which is calculated using equation 3.2. To index each node, the FLOPS of that node is used. FLOPS are used to measure the performance of a processor while clock speed indicates the speed of a processor. The clock speed can not define the total number of calculations a processor can perform in a second. Therefore, FLOPS is a better way of knowing the throughput of a processor [93].

GFLOPs are calculated using equation 3.2 where noOfSockets represents the total number of sockets on the motherboard, noOfCores represents the total cores of a processor, and frequency represents the clock frequency of a core (in Hz), and noOfFlop represents the total number of floating-point operations. To represent the installed memory, $\alpha$ is used to give weightage to the processing speed as well as the installed RAM (see details in section 3.1.1).

**Algorithm 3:** TOP-Storm : Slot Assignment

```
1  function MapSlotToNode (HashMap<slot,executor> slotMap)
   Input  : HashMap<slot,executor> slotMap
   // Executor(s) to slot(s) mapping
   Output: HashMap<node,slot> nodeMap
   // Slot(s) to Node(s) mapping
2  ClusterNode Nodes = ClusterInfo.getNodes();
   // Retrieve all nodes of the cluster
3  Set α = 0.8;
   // Set α to 0.8 to select compute-intensive nodes
4  foreach Node n_i ← Nodes do
       // Iterate through all available nodes
5      if n_i.FreeSlots > 0 then
           // if a node has free slot(s) then calculate its computation power
6          noOfSockets = n_i.getSockets();
7          noOfCores = n_i.getCores();
8          frequency = n_i.getFrequency();
9          noOfFlop = n_i.getFlop();
10         RAM = n_i.getRAM();
11         n_i.GFLOPs = α (noOfSockets x noOfCores x frequency x noOfFlop)
             + (1 - α)RAM;
           // α is used to normalize the score
           // Equation 3.2
12         n_a++;
13     end
14 end
15 if n_a ≥ 2 then
16     n = n.Sort('Desc'); // Sort available nodes by computation power,
         available slots in descending order
17 end
18 if slotMap.length() ≥ 2 then
19     slotMap = slotMap.Sort('Desc'); // Sort mapping by number of executors
         in descending order
20 end
21 foreach Slot s_i ϵ slotMap do
22     if n_i.FreeSlots == 0 then
23         n_i = n.getNextNode();
24     end
25     nodeMap.add(s_i, n_i);
26     n_i.FreeSlots–;
27 end
28 return nodeMap
```

## 3.3 A3-Storm: Topology, Traffic, and Resource-aware Storm Scheduler for Heterogeneous Cluster

To answer the second research question given in section 1.4 which is mainly related to communication traffic of topology, a traffic-aware scheduler named **A3-Storm** is presented. This scheduler is an extension of the previously presented TOP-Storm Scheduler. In this scheme, traffic, as well as the topology structure, is employed while preparing the execution plan. A3-Storm reads inter-operator traffic and sorts them in descending order. It starts assigning operators to slot according to their traffic. For the next executor assignment, traffic communication of unassigned executors is compared with assigned executors and the executor with the highest communication with the already assigned executor is selected. After all executors assignment to slots, the physical assignment is made starting from the most powerful machine and so on.

### 3.3.1 A3-Storm Algorithm

For efficient topology (in the form of a DAG) mapping, the scheduler needs to find groups of highly communicating executors, which can be assigned to the same or nearby nodes to reduce inter-node communication costs. Therefore, A3-Storm uses a heuristic to find groups of different sizes such as inter-group communication is reduced. Each group can be assigned to a node with relative capacity. A3-Storm consists of two main steps:

1. **Executor Assignment**: For executor assignment, topology communication is used. Frequently communicating executors are placed as close as possible. This is done in Algorithm 5, which is explained below.

2. **Slot Assignment**: Above created groups are assigned to slots. Slots are assigned starting from the most computationally powerful node. This is achieved using Algorithm 6, which is discussed below.

**Algorithm 4:** A3-Storm : Main

1 function A3-Storm ($T$)

**Input :** Unassigned Topologies $T$

```
// This function will receive all unassigned topologies as input, which need
   to be scheduled
```

**Output:** Node-wise executor assignment to cluster's node

```
// The output of the function will be the mapping of all executors of
   unassigned topologies to the cluster's nodes
```

2 **foreach** *topology* $t_i \in T$ **do**

```
// Iterate through all unassigned topologies one by one
```

3 $\omega = t_i$.NumWorker;

```
// Retrieve the required number of worker processes for topology
   execution
```

4 HashMap<source, destination> $e_u = t_i$.getUnassignedExecutors();

```
// Get the inter-executor traffic of all unassigned executors in the
   form of source-destination pairs
```

5 HashMap<slot,executor> slotMap = MapExecutorToSlot($e_u$, $\omega$);

```
// Invoke the 'MapExecutorToSlot' function, provide it with the
   inter-executor connectivity and the required number of worker
   processes as input
```

6 HashMap<node,slot> nodeMap = MapSlotToNode(slotMap);

```
// Invoke the 'MapSlotToNode' function, provide it with the executor to
   slot mapping as input
```

7 AllocateSlotsToNodes(nodeMap);

```
// Finally, physical mapping is handed over to Apache Storm for topology
   execution
```

8 **end**

Algorithm 4 presents the detailed steps for the proposed A3-Storm scheduler. The scheduler is responsible for generating the execution plan based on available resources and communication traffic (among the executors). The proposed scheduling algorithm takes as input unassigned topologies and produces the node to executor mapping.

**Algorithm 5:** A3-Storm : Executor Assignment

1 function MapExecutorToSlot (HashMap<source, destination> $e_u$, $\omega$)
**Input :** HashMap<source, destination> $e_u$, $\omega$
// Inter-executor communication details
**Output:** HashMap<slot,executor> slotMap
// Executor(s) to slot(s) mapping

1: $s_a = 0$;
2: $e_a = 0$;
3: $e_t = e_u$.Count();
4: $e_{ps}$ = Ceiling ($e_t$ / $\omega$);
5: **if** $\omega == 1$ **then**
6: slotMap.add($s_a$,$e_u$);
7: $s_a$++;
8: $e_a = e_t$;
9: **else if** $\omega \geq e_t$ **then**
10: **for all** $e \in e_t$ **do**
11: slotMap.add($s_a$,$e$);
12: $s_a$++;
13: $e_a$++;
14: **end for**
15: **else**
16: $e_u = e_u$.Sort('desc');
// Sort executors w.r.t. traffic in descending order
17: **while** $e_u \neq$ null **do**
18: $e_s = e_u$.getSourceExecutor();
19: $e_d = e_u$.getDestinationExecutor();
20: **if** $e_s \neq$ assigned && $e_d \neq$ assigned **then**
21: slotMap.add($s_a$,$e_s$);
22: $e_a$++;
23: slotMap.add($s_a$,$e_d$);
24: $e_a$++;
25: counter = 2;
26: **while** counter < $e_{ps}$ **do**
27: $e_k = e_u$.getFrequentlyCommunicatingExecutor($e_s$, $e_d$); // Frequently communicating executor w.r.t. source or destination
28: slotMap.add($s_a$,$e_k$);
29: $e_a$++;
30: counter++;
31: **end while**
32: $s_a$++;
33: **end if**
34: **end while**
35: **end if**

**Algorithm 5:** A3-Storm : Executor Assignment (Part 2)

```
1:  if e_a < e_t then
2:    counter = 1;
3:    while e_a < e_t do
4:      e_k = e_u.getNextExecutor();
        // Get next unassigned executor
5:      slotMap.add(counter % ω,e_k);
        // Round-robin assignment
6:      e_a++;
7:      counter++;
8:    end while
9:  end if
10: return slotMap
```

Initially, it reads the number of worker processes to be used for topology execution (Algorithm 4, line 3). Inter-executor traffic for all the unassigned executors is retrieved from the traffic log (line 4). For logical mapping of the executors to a slot, Executor Assignment (listed as Algorithm 5) is invoked (Algorithm 4, line 5). Similarly, for physical mapping of a slot to a node, Slot Assignment (listed as Algorithm 6) is invoked (Algorithm 4, line 6). Finally, physical mapping (i.e., slot—node) is handed over to Apache Storm for execution (line 8).

Algorithm 5 inputs inter-executor traffic (communication) ($e_u$) and the number of worker processes ($\omega$) to determine executor(s) to slot(s) mapping. ($e_u$) represents the inter-executor traffic for all the unassigned executors which is retrieved from the traffic log (Provenance Data module). ($e_u$) plays an important role in traffic-aware executor(s) to slot(s) mapping.

As a first step, Algorithm 5 calculates the maximum executors to be mapped in a worker process (line 4). It checks the desired number of worker processes for job scheduling (by default = 1). In this case, all executors will be assigned to a single slot (Algorithm 5, lines 5-8). Alternatively, if the user specifies that the number of worker processes is equal to the number of executors, then each worker process will contain a single executor (lines 9–14).

In the end, if the number of worker processes is less than the total executors, then executors will be assigned in the following manner. Executors are sorted in descending order with respect to their communication-related traffic (line 16). A pair of unassigned executors are selected from the top of the list and assigned to the current slot (lines 18–25). For the selection of the next executor, getFrequentlyCommunicatingExecutor($e_s$,$e_d$) will be invoked. In this method, a heuristic is used to select the next executor for the assignment. From the unassigned list of executors, an executor with the highest communication granularity (with already mapped executors) is selected and scheduled (lines 26–32). During this selection, all in-bound and out-bound traffic is considered.

For a case study, consider topology t in Fig. 3.4 as an example. Here, A, B, C, D, E, and F represent the executors of the topology and the numbers on the edge of the topology DAG represent the communication traffic between the executors. Inter-executor traffic for all the unassigned executors is retrieved from the traffic log (Algorithm 4, line 4) as shown in Table 3.5.

As described in Algorithm 5 at line 16, the unassigned executors are sorted in descending order with respect to their communication traffic (as shown in Table 3.6). Then a pair with the highest communication traffic is selected from the list. In this case, B and E will be selected first because of having maximum traffic. Next, with the help of getFrequentlyCommunicatingExecutor($e_s$,$e_d$), D will be

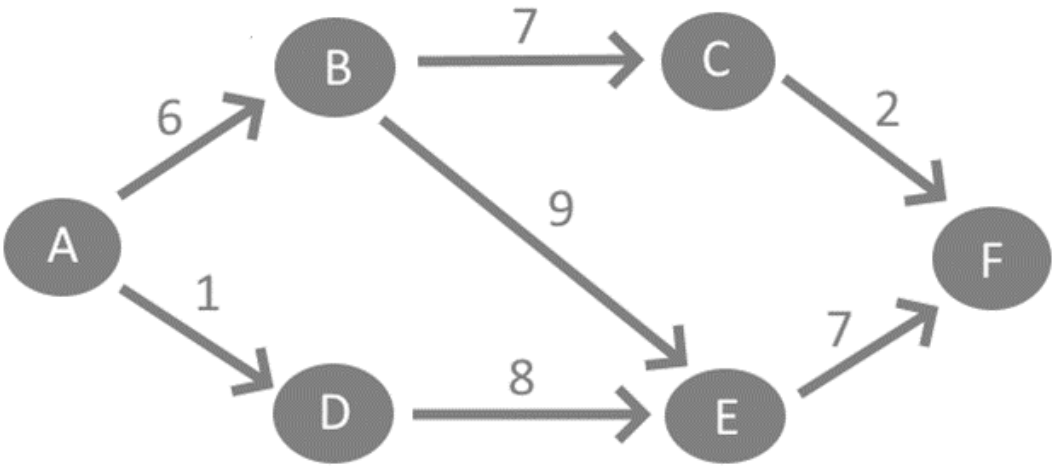



FIGURE 3.4: An Example Topology with Inter-executor Communication Traffic.

TABLE 3.5: Inter-executor Communication Traffic for Topology Given in the Figure 3.4

| S. # | Source | Destination | Traffic |
|---|---|---|---|
| **1** | A | B | 6 |
| **2** | A | D | 1 |
| **3** | B | C | 7 |
| **4** | B | E | 9 |
| **5** | C | F | 2 |
| **6** | D | E | 8 |
| **7** | E | F | 7 |

selected because of having maximum communication traffic with already selected executors (B or E). Now, B, D, and E are selected.

For the next selection from unassigned executors, C and F both have traffic communication of value 7 with already selected executors. Therefore, both are equal candidates for selection. This process continues until the maximum limit per slot is reached (lines 27–31). In this way, executors with high communication traffic are grouped and placed in the same slot which increases intra-slot traffic and reduces inter-slot traffic. Finally, if there are still unassigned executors, then they will be assigned in a round-robin fashion among the unassigned slots (Part 2, lines 1-9).

Once executors are assigned to slots, the next step is to map slots to the computing nodes. First, Algorithm 6 fetches all nodes of the cluster and calculates the computation power (in terms of GFLOPs) for each node (if it has free slots) using Equation 3.2 (Algorithm 6, lines 4–12). Equation 3.2 handles the heterogeneity of resources too. All nodes in the cluster are ranked in terms of GFLOPs irrespective of their hardware specifications. The nodes are sorted in descending order according to their computation power (line 16). Similarly, slots are sorted in descending

TABLE 3.6: Sorted Inter-executor Traffic for Example Topology Given in Figure 3.4

| Source | Destination | Traffic | Index |
|---|---|---|---|
| B | E | 9 | 1 |
| D | E | 8 | 2 |
| B | C | 7 | 3 |
| E | F | 7 | 3 |
| A | B | 6 | 4 |
| C | F | 2 | 5 |
| A | D | 1 | 6 |

order according to their inter-executor traffic (line 19).

Finally, the slot with the highest communication is mapped to the most powerful node, and so on. Instead of a round-robin fashion, a node-wise assignment is performed. Slots are assigned to node until all slots are consumed then the next node is used (lines 21–27). In this way, minimum nodes are utilized for topology execution, and inter-node communication is reduced. The complete flowchart for the A3-Storm scheduler is illustrated in Fig. 3.5.

## 3.4 BAN-Storm: A Bandwidth-aware Scheduling Mechanism for Stream Jobs

The essential component of a stream processing system is the processing framework responsible for crunching the data. To cope with the growing computing demands of real-time streaming applications, researchers have proposed several computing frameworks. The core of the computing frameworks i.e., the scheduling mechanisms for real-time stream processing need to accommodate several important

**Algorithm 6:** A3-Storm : Slot Assignment

```
1  function MapSlotToNode (HashMap<slot,executor> slotMap)
   Input  : HashMap<slot,executor> slotMap
   // Executor(s) to slot(s) mapping
   Output: HashMap<node,slot> nodeMap
   // Slot(s) to Node(s) mapping
2  ClusterNode Nodes = ClusterInfo.getNodes();
3  Set α = 0.8;
4  foreach Node n_i ← Nodes do
5      if n_i.FreeSlots > 0 then
           // if a node has a free slot then calculate its computation power
6          noOfSockets = n_i.getSockets();
7          noOfCores = n_i.getCores();
8          frequency = n_i.getFrequency();
9          noOfFlop = n_i.getFlop();
10         RAM = n_i.getRAM();
11         n_i.GFLOPs = α (noOfSockets x noOfCores x frequency x noOfFlop)
               + (1 - α)RAM; // α is used to normalize the score
           // Equation 3.2
12         n_a++;
13     end
14 end
15 if n_a ≥ 2 then
16     n = n.Sort('Desc');
       // Sort available nodes by computation power, available slots in
          descending order
17 end
18 if slotMap.length() ≥ 2 then
19     slotMap = slotMap.Sort('Desc');
       // Sort mapping by total inter-executor traffic in descending order
20 end
21 foreach Slot s_i ϵ slotMap do
22     if n_i.FreeSlots == 0 then
23         n_i = n.getNextNode();
24     end
25     nodeMap.add(s_i, n_i);
26     n_i.FreeSlots–;
27 end
28 return nodeMap
```

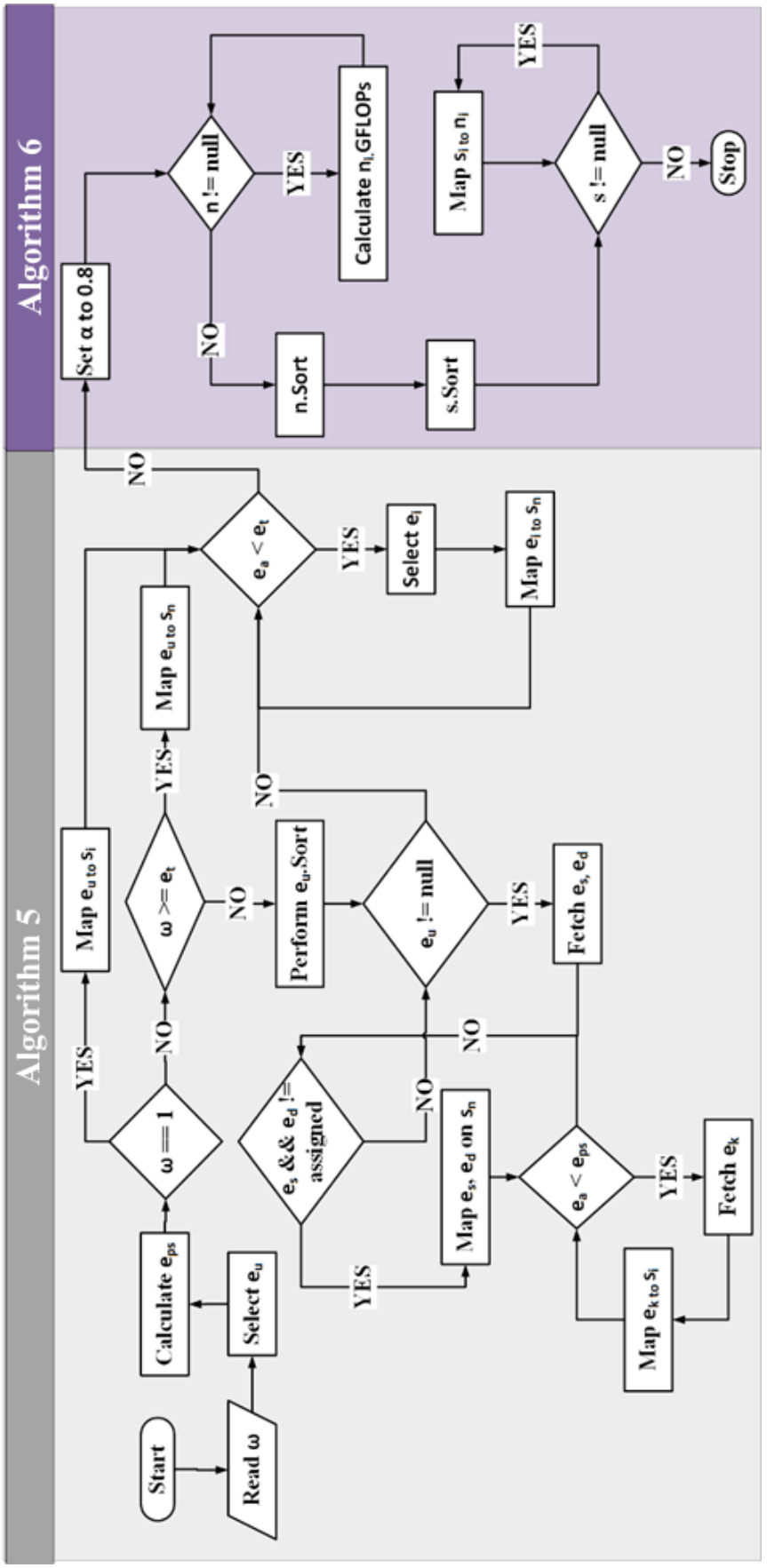



FIGURE 3.5: The A3-Storm Scheduler's Flowchart.

aspects such as incorporating resource awareness, heterogeneity of the computing resources, load balancing, etc. These aspects contribute significantly to the attained performance of the stream processing frameworks. Therefore, ignoring any one of these aspects may lead to degraded performance.

The mechanism employed by stream processing frameworks is normally based on round-robin scheduling. The round-robin approach considers all the computing nodes for mapping tasks. As it is a fair scheduler (equally assigning the workload across the supervisor nodes in a cluster) therefore inter/-node/-process communication is not considered which adversely impacts the performance (especially in heterogeneous computing architectures). Using fair-scheduling to map stream jobs equally to all cluster nodes produces degraded performance and induces communication latencies [17, 31, 89].

As most of the large parallel machines (such as compute clusters and Cloud) are heterogeneous; therefore, the round-robin scheduling mechanism may cause load imbalance as well. As a result, researchers proposed resource-aware scheduling mechanisms [16, 19, 25, 34, 95] to address the issue. But most of these algorithms consider only CPU and memory as resources ignoring the important aspect of bandwidth. To address this need, a scheduling technique called **BAN-Storm** is proposed that builds upon A3-Storm [20] considering bandwidth as an additional aspect of scheduling. In BAN-Storm, the physical mapping is performed based on the node's computation power which includes FLOPs, memory as well as bandwidth.

### 3.4.1 BAN-Storm Scheduler

The BAN-Storm scheduler generates an execution plan for a stream processing job to increase throughput and improve resource utilization. This is achieved with the help of historical communication of topology and the available computation

power of the cluster. Frequently communicating tasks are mapped closely. The BAN-Storm scheduling mechanism consists of **two** phases:

a) **Logical grouping:** For the task grouping, the topology's traffic is monitored and used for mapping-related decision-making. Frequently communicating tasks are scheduled closely as shown in Algorithm 7;

b) **Physical grouping:** The groups identified in Step 1 are then assigned to the cluster, starting from computationally powerful nodes as listed in Algorithm 8.

Algorithm 7 generates the execution plan considering the cluster's computing resources and historical communication data between the tasks. The algorithm receives unscheduled topology its type along the required worker processes for topology execution and produces the execution plan as an output. Firstly, all unassigned executors with their communication are retrieved from the Provenance data module (i.e., Algorithm 7, line 1). The maximum tasks which can be scheduled for a worker process are calculated (Algorithm 7, line 6) then tasks are sorted in decreasing order considering the amount of communication traffic (Algorithm 7, line 7). An unassigned task pair is retrieved from the sorted list and added to the current group (lines 10–14). To select the next unassigned task for assignment, getMaxTrafficExecutor() is invoked which implements a mechanism for this. From the remaining list, the tasks which are involved in a high number of communication events are selected and grouped (lines 16–22). All inbound and outbound communication is measured during this process. This is an iterative process and is continued until the maximum limit as shown in Algorithm 7 (lines 16–22). Finally, the unassigned tasks are assigned using a round-robin fashion (lines 26–37).

For the physical assignment of selected groups to a cluster's node, Physical Grouping (see Algorithm 8) is invoked (see Algorithm 7, line 38). Lastly, the physical mapping is passed to the Apache Storm framework for execution (Algorithm 7, line 40).

**Algorithm 7:** BAN-Storm : Logical Grouping

**Input:** topology, topologyType, workerProcess
**Output:** physicalMap

```
1  unassignedExecutors ← getExecutors(topology);
2
3  assignedSlots = 0;
4  assignedExecutors = 0;
5  totalExecutors = unassignedExecutors.Count();
6  executorPerSlot = Ceiling (totalExecutors / workerProcess);
7  unassignedExecutors = unassignedExecutors.Sort('desc');
8
9  while unassignedExecutors ≠ null do
10     sourceExecutor = unassignedExecutors.getSourceExecutor();
        destinationExecutor = unassignedExecutors.getDestinationExecutor();
11
12     if sourceExecutor ≠ assigned && destinationExecutor ≠ assigned then
13        executorMap.add(assignedSlots, sourceExecutor);
           executorMap.add(assignedSlots, destinationExecutor);
           assignedExecutors = assignedExecutors + 2;
14         counter = 2;
15
16        while counter < executorPerSlot do
17           e = getMaxTrafficExecutor(sourceExecutor, destinationExecutor);
18            executorMap.add(assignedSlots, e);
19
20           assignedExecutors++;
21            counter++;
22        end
23        assignedSlots++;
24    end
25 end
26 if assignedExecutors < totalExecutors then
27    counter = 1;
28
29    while assignedExecutors < totalExecutors do
30       e = unassignedExecutors.getNextExecutor();
31
32       executorMap.add(counter % workerProcess, e);
33
34       assignedExecutors++;
35        counter++;
36    end
37 end
38 physicalMap = Physical_Mapping(executorMap, topologyType);
39
40 return physicalMap;
```

**Algorithm 8:** BAN-Storm : Physical Mapping

**Input:** executorMap, topologyType

```
// This function will receive executors to slots mapping and topology type
   as input
```

**Output:** physicalMap

```
// Physical mapping will be the output of this function
```

```
ClusterNode Nodes = ClusterInfo.getNodes();
 // Fetch details of all available nodes of the cluster
α = Get_Value('Alpha', topologyType);
// Read α value based on topology type
β = Get_Value('Beta', topologyType);
// Read β value based on topology type
γ = Get_Value('Gamma', topologyType);
// Read γ value based on topology type

foreach node ∈ Nodes do
    node.FLOPS = node.getCores() × node.getFrequency() ×
     node.getFLOPs();
     // Equation 3.1
    node.power = α × node.FLOPS + β × node.getRAM() + γ ×
     node.getBandwidth();
     // Equation 3.3
end

Nodes = Nodes.Sort('Desc');

executorMap = executorMap.Sort('Desc');

foreach e ∈ executorMap do
    physicalMap.add(e, n);
    n.FreeSlots = n.FreeSlots - 1;

    if n.FreeSlots = 0 then
        n = Nodes.getNextNode();
    end

end

return physicalMap
```

Once the tasks are logically grouped then the next step is to map these groups to the computing nodes. As a first step, all computing nodes of the cluster are retrieved (line 1). Based on the topology type, values for $\alpha$, $\beta$, and $\gamma$ are determined with the help of Algorithm 9 (lines 3-5) to be used in Equation 3.3. To index each node, the Floating Point Operations Per Second (FLOPS) [94] of that node is employed [91] which is a better way of expressing the throughput of a processor [93] (Algorithm 8, line 8). To accommodate other resources, Equation 3.3 is devised to highlight the system memory and bandwidth resources in a weighted form (line 10).

$$NodePower = (\alpha \times Processing\_Speed) + (\beta \times DRAM) + (\gamma \times bandwidth) \quad (3.3)$$

In the next step, the cluster resources are ranked and sorted in descending order according to their computation power irrespective of hardware specifications (see Algorithm 8, line 14). Similarly, the selected groups are sorted in descending order with respect to their communication pattern (Algorithm 8, line 16). Now, the last step is to map the selected groups to the computing nodes. The group having maximum traffic is assigned to the most powerful machine (line 19). The process continues until all slots of a given node are mapped then the next node is used (Algorithm 8, lines 22–24). This ensures the minimum number of nodes utilization for topology execution.

### 3.4.2 Value Selection for $\alpha$, $\beta$, $\gamma$

In a heterogeneous cluster, different computational resources (for example CPU, RAM, Bandwidth, etc.) have a different effect on the overall performance of the system. To accommodate this effect, $\alpha$, $\beta$, and $\gamma$ are employed for the proposed scheduling heuristic (Algorithm 8, line 10). We have different topology types in Apache Storm for example CPU-bound, Memory-bound, IO-bound, and Network-bound. Nodes participating in the execution of any topology should be selected

**Algorithm 9:** BAN-Storm : Get Alpha, Beta and Gamma Values

**Input:** valueType, topologyType

```
/* Value type and topology type as input                                    */
```

**Output:** value

```
1  if valueType = 'Alpha' AND topologyType = 'CPU-Bound' then
2      return 0.5;
3  else
4      if valueType = 'Beta' AND topologyType = 'Memory-Bound' then
5          return 0.5;
6      else
7          if valueType = 'Gamma' AND topologyType = 'Network-Bound' then
8              return 0.5;
9          else
10             return 0.25;
11         end
12     end
13 end
```

concerning the running topology type. To achieve this, $\alpha$, $\beta$, and $\gamma$ are used in equation 3.3 to index the computing nodes according to the need of the executing topology (Algorithm 9). Like, in a heterogeneous cluster different nodes may be suitable for different topologies i.e., compute-intensive, memory-intensive and bandwidth-intensive etc. So, by employing $\alpha$, $\beta$, and $\gamma$ values, we want to list down the cluster's computing nodes in sorted order with respect to the type of topology. After that, a one-by-one node from the top of the list will be consumed for topology execution.

Some suitable values for $\alpha$, $\beta$, and $\gamma$ are required for equation 3.3 to work. In this regard, different values from 0.1 to 1.0 are tried. For example, consider a cluster of five computing nodes presented in table 3.7. The number of cores, cycles per second, floating-point operations per cycle, GFLOPS, RAM and bandwidth are given in this table. Different values are used to depict a heterogeneous environment.

Any value between 0 to 1 can be assigned to $\alpha$, $\beta$, $\gamma$. After some trial and error,

TABLE 3.7: Available Computing Nodes for Indexing

| Name | Cores | Cycles /second | FLOPs /Cycle | GFLOPS | RAM (GBs) | Bandwidth (Mbps) |
|---|---|---|---|---|---|---|
| **Node A** | 4 | 2.8 | 2 | 22 | 12 | 100 |
| **Node B** | 4 | 2.4 | 2 | 19 | 12 | 100 |
| **Node C** | 2 | 2.2 | 2 | 9 | 16 | 100 |
| **Node D** | 2 | 2 | 1 | 4 | 20 | 100 |
| **Node E** | 2 | 1.8 | 1 | 4 | 24 | 100 |

we analyzed that with $\alpha$ equal to 0.5, the nodes with high computing power are indexed on the top. So, if we have a compute-intensive topology for scheduling, we require computing nodes with high computational power. In such a scenario, we will assign $\alpha$ to 0.5, and as a result, nodes will be ranked according to their processing capability as shown in table 3.8.

Similarly, if we consider a memory-intensive topology and intend to index computing nodes based on their installed memory, we can assign a value of 0.5 to $\beta$. Consequently, the machines will be indexed according to their installed memory,

TABLE 3.8: Computing Node Indexing for $\alpha = 0.5$.

| Name | Cores | Cycles /second | FLOPs /Cycle | GFLOPS | RAM (GBs) | Bandwidth (Mbps) | using equation 3.3 ($\alpha = 0.5$) |
|---|---|---|---|---|---|---|---|
| **Node A** | 4 | 2.8 | 2 | 22 | 12 | 100 | 39 |
| **Node B** | 4 | 2.4 | 2 | 19 | 12 | 100 | 38 |
| **Node C** | 2 | 2.2 | 2 | 9 | 16 | 100 | 33 |
| **Node E** | 2 | 1.8 | 1 | 4 | 24 | 100 | 33 |
| **Node D** | 2 | 2 | 1 | 4 | 20 | 100 | 32 |

TABLE 3.9: Computing Node Indexing for $\beta = 0.5$.

| Name | Cores | Cycles /second | FLOPs /Cycle | GFLOPS | RAM (GBs) | Bandwidth (Mbps) | using equation 3.3 ($\beta = 0.5$) |
|---|---|---|---|---|---|---|---|
| **Node E** | 2 | 1.8 | 1 | 4 | 24 | 100 | 38 |
| **Node A** | 4 | 2.8 | 2 | 22 | 12 | 100 | 37 |
| **Node B** | 4 | 2.4 | 2 | 19 | 12 | 100 | 36 |
| **Node D** | 2 | 2 | 1 | 4 | 20 | 100 | 36 |
| **Node C** | 2 | 2.2 | 2 | 9 | 16 | 100 | 35 |

as shown in table 3.9.

Moreover, for a bandwidth-intensive job, if $\gamma$ is set to 0.5, machines will be indexed based on available bandwidth connectivity. For more details, refer to table 3.10, which provides an example of computing node indexing based on bandwidth-intensive topology type. When we assign 0.5 to either $\alpha$, $\beta$ or $\gamma$, the remaining two are assigned a weightage of 0.25 each. Therefore, we assign different weightages to GFLOPS, RAM, and bandwidth to index nodes accordingly.

TABLE 3.10: Computing Node Indexing for $\gamma = 0.5$.

| Name | Cores | Cycles /second | FLOPs /Cycle | GFLOPS | RAM (GBs) | Bandwidth (Mbps) | using equation 3.3 ($\gamma = 0.5$) |
|---|---|---|---|---|---|---|---|
| **Node A** | 4 | 2.8 | 2 | 22 | 12 | 100 | 59 |
| **Node B** | 4 | 2.4 | 2 | 19 | 12 | 100 | 58 |
| **Node E** | 2 | 1.8 | 1 | 4 | 24 | 100 | 57 |
| **Node C** | 2 | 2.2 | 2 | 9 | 16 | 100 | 56 |
| **Node D** | 2 | 2 | 1 | 4 | 20 | 100 | 56 |

## 3.5 Gr-Storm: A Graph Algorithm-based Job Scheduler for Stream Processing Engines

A data stream is a continuous flow of data generated by a variety of applications, for example, e-commerce applications, utility services, location updates, log files, stock exchange data, blockchains, etc. Processing of real-time streaming data applications demands ever-increasing processing (computational and communication) power. Multiple machines combined in a distributed fashion with appropriate tools to process the continuous data generated by many sources, is an economical way. The execution of continuous data streams on a cluster of distributed computing elements can be handled by SPEs [10]. SPEs have rapidly evolved in the last two decades with several SPE prototypes being adopted by the industry, for example, Apache Storm [4], Apache Spark Streaming [96], Apache Flink [97], Apache Heron [7], etc. One of the important responsibilities of SPEs is how to schedule the processing tasks on computing nodes to process the streams with near-optimum utilization of the available resources with maximized throughput and minimized latency.

Different SPEs modelled a data stream application as DAG which represents the processing tasks and communication between these tasks required to complete the DSP application [11]. This abstraction layer helps a scheduler to understand how to place these tasks on multiple processing elements [66]. To schedule these tasks optimally, graph partitioning algorithms are being used in research [16, 29, 31, 34–38] to place frequently communicating components of the DAG closer to each other to reduce latency which may improve overall throughput, too. But most of them [35, 37, 57, 58, 61] are either resource-unaware or non-adaptive. As a result, it is difficult to decide on the power of cluster resources. Similarly, some techniques [24, 35, 37, 58] are proposed for homogenous clusters only and are unable to assign workload according to the node's computing power (heterogeneity).

In recent years, resource-aware schedulers [15, 19–22, 25, 26] are also proposed which are either non-adaptive or static in nature and can not accommodate real-time (dynamic) changes. Moreover, CPU, Memory, and bandwidth are considered a resource by existing resource-aware schedulers, however, they do not truly represent the computation power of a node [93]. To address these issues, we present **Gr-Storm**: A Graph algorithm-based job scheduler for Stream Processing Engines (Apache Storm) which assigns frequently communicating DAG components closer to each other such that minimum nodes are employed for job execution.

### 3.5.1 Max-flow Min-cut Algorithm

Since we implemented Gr-Storm using the Max-flow min-cut algorithm [53], we briefly introduce it here. The Max-flow min-cut theorem states that in a flow network, the largest amount of flow passing from the source to the sink is equal to the total weight of the edges in the minimum cut. A cut is a division of the vertices of the network into two parts, with the source in one part and the sink in the other. For example, a network having a value of the flow of 7 is shown in Fig. 3.6. The mark on each arrow, in the form x/y, shows the flow (x) and the capacity (y). The flows coming from the source total seven (4+3=7), as do the flows into the sink (3+4=7).

The vertex in white and grey form the subsets S and T of an s-t cut, whose cut-set holds the dashed edges. The capacity of the s-t cut is 7, which equals the value of flow. The theorem says that the value of flow and the capacity of the s-t cut are both optimal in this network. The flow through each of the dashed edges is at full capacity. By contrast, there is spare capacity in the right-hand part of the network. If there were no flow between nodes 1 and 2, then the inputs to the sink would change to 4/4 and 3/5 with the total flow remaining 7. But, if the flow from node 1 to node 2 were doubled to 2, then the inputs to the sink would change to 2/4 and 5/5 which is still seven [53]. With the help of this algorithm, Gr-Storm partitions the application's DAG when needed. If an application is needed to be scheduled on 2 nodes, then DAG is divided into 2 sub-graphs.

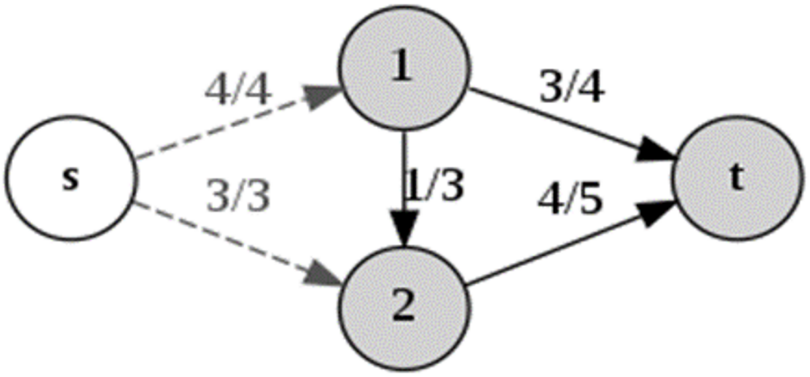



FIGURE 3.6: Example [53].

### 3.5.2 Gr-Storm Scheduler

The Gr-Storm scheduler maps Storm topology (weighted acyclic graph of computation) on a heterogeneous cluster with improved resource utilization and increased throughput. Gr-Storm maps computational requirements (processing and communication requirements) of a topology on the available network of workers (with a given processing and communication power) in an optimal way to maximize the average throughput per node of the cluster against the given topology.

Gr-Storm schedule the jobs in **two** phases:

1. The first phase, termed **Graph Partitioning**, formulates a weighted directed acyclic graph (W-DAG) from the topology with intra-executors (spouts and bolts) communication requirements. The graph is then processed by the **max-flow min-cut** algorithm [53] which gives us limiting communication links in the network and maximum flow which can be achieved with the limiting links. The limiting links can then be used to distribute the group of executors between different workers of the cluster. Groups of frequently communicating executors can be assigned to the same or nearby workers to reduce inter-worker communication costs.

2. The second phase, **Physical Mapping**, intelligently maps to a pool of workers by assigning all the groups of executors to a worker based on the worker's computation power (starting from most powerful to the least) and reduced

communication needs as per the modelled W-DAG. A group with maximum processing requirements can be assigned to a worker with higher computational power.

Algorithm 10 is a driver program for the proposed **Gr-Storm** scheduler. The algorithm gets unassigned jobs as input and produces the physical map for scheduling. Initially, with the help of the topology's code, the topology's DAG is retrieved (line 2). Inter-executor traffic is retrieved from the Provenance Data to generate modelled DAG (line 3) as W_DAG.

For Graph partitioning of the executors, W_DAG is passed to Max-flow_Min-cut [53] (line 4). Finally, Map_Partitions_To_Workers is invoked for the physical mapping of partitions to workers (line 5).

**Algorithm 10:** Gr-Storm Scheduler

**Input:** Unassigned Topologies (T[ ])

**Output:** Topologies to Cluster Nodes Physical Mapping

1 **while** *all t in T[ ] are not assigned* **do**
2 $\quad$ *DAG* ← get_Topology_DAG(*t*)
3 $\quad$ *W_DAG* ← get_Modeled_DAG(*DAG*)
4 $\quad$ *Graph_Partitions* ← max-Flow_Min-Cut(*W_DAG*)
5 $\quad$ *Physical_Mapping* ← map_Partitions_To_Workers(*Graph_Partitions*)
6 **end**
7 **return** Physical_Mapping

After the executor's partitioning into groups, the next step is to map these partitions to workers which is performed by Map_Partitions_To_Workers (as shown in algorithm 11). Map_Partitions_To_Workers after retrieving the cluster's node, calculates the computation power in Floating Point Operations Per Second (FLOPS) for each node using Equation (3.1)(lines 2–6). As a result, all computing nodes are ranked in terms of FLOPS irrespective of their hardware specifications to handle

heterogeneity [91]. FLOPS define the total number of calculations a processor can perform in a second [93].

**Algorithm 11:** Gr-Storm : Map Partitions to Workers

**Input:** Graph Partitions

**Output:** Physical Mapping

```
1  ClusterNode Nodes ← ClusterInfo.getNodes()
2  foreach n ∈ Nodes do
3      cores ← n.getCores()
4      frequency ← n.getFrequency()
5      flops ← n.getFLOPs()
6      n.GFLOPS ← cores × frequency × flops
       // Equation 3.1
7  end
8  Nodes ← Nodes.Sort('Desc')
9  Graph_Partitions ← Graph_Partitions.Sort('Desc')
10 foreach p ∈ Graph_Partitions do
11     if n.FreeSlots equals to 0 then
12         n = Nodes.getNextNode()
13     end
14     Physical_Mapping.add(p, n)
15     decrement n.FreeSlots to 1
16 end
17 return Physical_Mapping
```

After FLOPS calculation, the nodes are sorted in descending order with respect to their computing capacity (line 8). As a result, nodes with the highest FLOPS value will come on the top. Also, graph partitions are sorted in descending order according to their inter-executor communication (line 9). Similarly, the graph partition having the highest communication traffic will be on the top of the list.

In the end, the partition with the highest communication traffic (first element from the list) is assigned to the most powerful worker (first entry from the list).

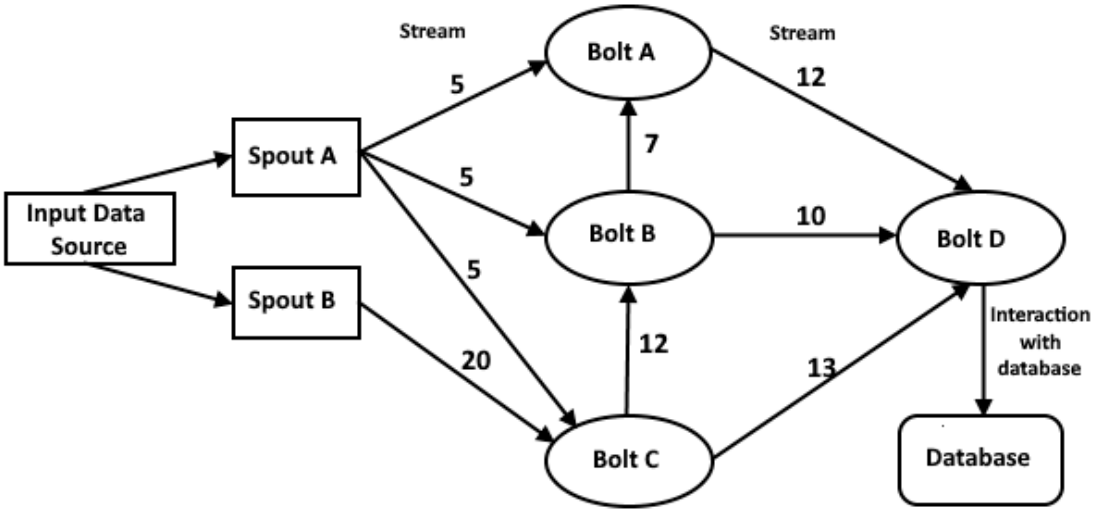



FIGURE 3.7: An Example Topology with Inter-executor Communication Traffic for Gr-Storm.

Partitions are allocated to workers (line 14), once all workers are used then the next machine is occupied (lines 11–13). With this approach, minimum machines are used, and inter-machine communication is also decreased.

To demonstrate how the Gr-Storm algorithm partitions a topology's weighted DAG, consider an example topology t in Fig. 3.7. Spout A, B and Bolt A, B, C and D represent the executors of the topology and the numbers on the edge of the topology DAG represent the communication traffic between the executors. For Graph partitioning of the executors, W_DAG is passed to Max-flow_Min-cut [53] (line 4).

Fig. 3.8 shows different possible options available for graph partitioning. According to Max-flow Min-cut algorithm [53], a minimum cut should be applied to partition the graph having maximum flow. The total flow of the graph is 35 from source to sink. If we break the group with 47 data flows then we have to apply 5 cuts. Only the right most cut with 35 value is fulfilling the algorithm constraints with 3 cuts. All other lines either contain less flow or more cuts. Therefore, we will place Bolt D in one partition and the remaining executors in another partition.

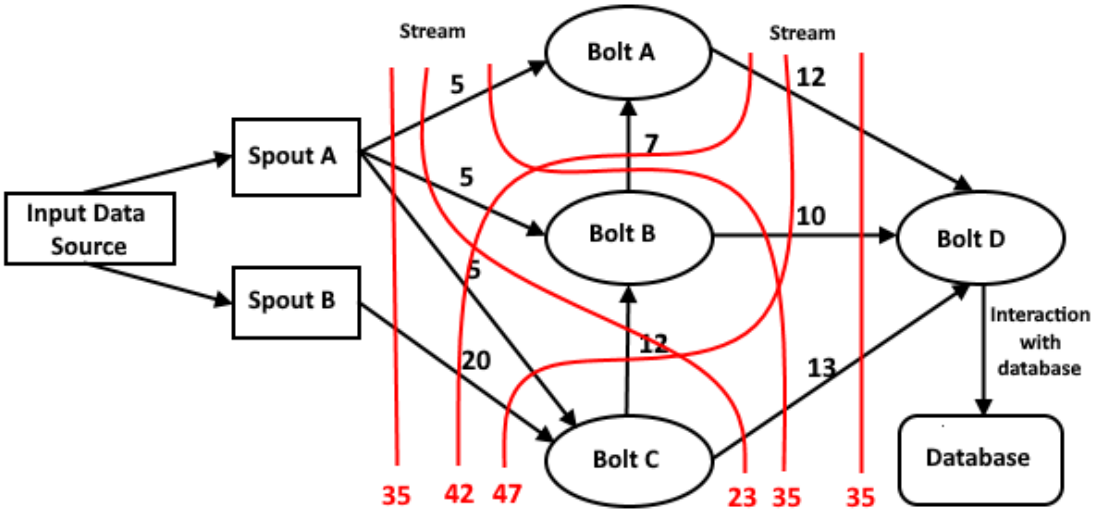



FIGURE 3.8: Graph Partitioning Example of Max-flow Min-cut Algorithm.

## 3.6 Comparison of the Proposed Schedulers

Contemplating the highlighted research gaps, a traffic-aware and resource-aware job scheduling system for SPE in a heterogeneous environment is proposed in this chapter. Based on the proposed system, 4 different dynamic scheduling algorithms using topology's DAG and supervisor node computing power are presented to achieve high throughput and maximize resource provisioning. For ease of reader, a discussion related to comparing the variants of the proposed schedulers is presented in Table 3.11.

- **TOP-Storm** finds the maximum number of executors that can be placed in a single worker process using the executor's connectivity from the topology's DAG. Based on this connectivity, the longest path is calculated and assigned to the most powerful computing machine in the heterogeneous cluster.

  TOP-Storm consists of **two** phases:

  1. **Logical grouping** involves the grouping of executors which is done using DAG.
  2. **Physical mapping** assigns frequently communicating executors group to a node based on the node's computation power starting from most powerful to least.

TABLE 3.11: Comparison of the Proposed Schedulers

| Algorithm Name | Scheduling Mechanism |
|---|---|
| **TOP-Storm** | **Logical grouping** involves DAG-based grouping of executors. |
| | **Physical mapping** based on communication and computation power. |
| **A3-Storm** | **Logical grouping:** Traffic, as well as the topology's DAG, is employed. |
| | **Physical mapping** same as TOP-Storm. |
| **BAN-Storm** | **Logical grouping** same as A3-Storm. |
| | **Physical mapping** relies on bandwidth, FLOPs, and installed memory. |
| **Gr-Storm** | **Logical grouping:** Max-flow min-cut algorithm partitions executors. |
| | **Physical mapping** same as A3-Storm. |

The time complexity of first phase is $O(e_u \times e_{ps})$ where $e_u$ is all unassigned executors for a topology and $e_{ps}$ is maximum executors per slot. For second phase, time complexity is $O(n)$ where $n$ is the number of unassigned nodes in the cluster. The overall time complexity to generate a scheduler for a given topology $t$ will be $O(e_u \times e_{ps}) + O(n)$.

- **A3-Storm** is an extension of **TOP-Storm** Scheduler. In this scheme, traffic, as well as the topology structure, is employed while preparing the execution plan. A3-Storm reads inter-executor traffic and sorts them in descending order. It starts assigning executors to the worker process according to their traffic. For the next executor assignment, traffic communication of unassigned executors is compared with assigned executors and the executor with the highest communication with the already assigned executor is selected. After all executors assignment to slots, the physical assignment is made starting from the most powerful machine and so on. The time complexity for A3-Storm is same as TOP-Storm because of the fact that only communication traffic has been introduced which will take constant time $O(1)$.

- **BAN-Storm** is an extension of **A3-Storm** in which logical grouping remains same but the physical mapping is performed based on the node's bandwidth along FLOPs and installed memory. The time complexity for

BAN-Storm is similar to A3-Storm because both are sharing same logical grouping phase and bandwidth is employed int he physical mapping phase.

- **Gr-Storm** further extends **A3-Storm** scheduler. The first phase formulates the topology's DAG with intra-executor computation requirements and inter-executor communication requirements to generate a modelled DAG, which is then given to the max-flow min-cut algorithm to produce a partitioned graph of executors. The second phase which belongs to physical mapping remains same.

  The logical grouping phase of Gr-Storm is based upon max-flow min-cut algorithm. The worst-case time complexity for finding the maximum flow and minimum cut in a flow network is $O(VE^2)$, where V is the number of nodes and E is the number of edges in the network. The algorithm may need to run E iterations to find the maximum flow, and in each iteration, it needs to find an augmenting path from the source to the sink. For second phase, time complexity is $O(n)$ where $n$ is the number of available nodes in the cluster. Therefore, the overall time complexity for a given topology $t$ will be $O(VE^2) + O(n)$.

# Chapter 4

# Experimental Evaluation and Results

This chapter describes the experimental setup including software/hardware configurations, three benchmarked scheduling heuristics (employed for experimental evaluation), and the two topologies used to evaluate the system throughput and resource usage for each experimented scheduler.

## 4.1 Experimental Setup

A real heterogeneous Apache Storm cluster is configured with one nimbus node, one ZooKeeper node, and nine supervisor nodes to evaluate the performance of the proposed schedulers. Ubuntu 19.04 64-bit is installed on each node with network connectivity of either 100 or 1000 Mb/s. Each node has a Java OpenJDK 13 executing on top of the Ubuntu operating system and uses Apache Storm 2.0.0 as the base framework orchestrated by Apache Zookeeper 3.4.13, with dependent libraries i.e., zeromq 4.3.2 and JZMQ 3.1.1 Python 3.7.3 used for scripting purposes.

A heterogeneous system configuration has been adapted that comprises high-capacity nodes (in terms of computation) equipped with 8 cores whereas the low-capacity nodes have 4 cores. This configuration is used to achieve heterogeneity.

As a fair comparison, the proposed schedulers are built as an extension of Apache Storm 2.0.0 and the comparable approaches (default scheduler, resource-aware scheduler, and isolation scheduler) are also directly extracted from this release.

Table 4.1 contains experimental environment hardware configurations. In this table, the first column is representing the name of the machine followed by the installed processor on that machine. In the next column, floating point operations per cycle are written. GFLOPs are calculated with the help of equation 3.1 which requires *number_of_cores*, *cycle/second* which is the clock frequency of a core (in Hz), and *flops/cycle*. RAM installed on each node is also mentioned here because it can play an important role in topology execution. That's why equation 3.2 is devised to highlight the installed RAM on each node. Next, the computation power of each node is calculated based on the output of the equation 3.1 and installed RAM. As a result, a numeric value is received which helps in assigning an index number to each node. This index number is used later on at the time of task assignment. A smaller number represents more powerful nodes as compared to a larger number. During task assignment, nodes with a smaller number are used first which ensures the usage of powerful nodes as compared to less powerful nodes to achieve better throughput. The main reason for this selection is that in a dedicated cluster with different computing nodes selection of more powerful nodes ultimately enhances throughput. This node selection technique is a major difference between the proposed schedulers and the state-of-the-art schedulers (default scheduler and isolation scheduler).

## 4.2 Benchmarked Scheduling Heuristics

The Apache Storm ships with multiple schedulers [98] for example Default Scheduler, Isolation Scheduler, Resource-aware Scheduler, and Multitenant Scheduler. The first three follow single-tenant architecture while the last one uses multitenant

architecture. Multitenancy refers to a software architecture in which a single instance of software serves multiple customers. Systems designed in such a manner are often called shared. All four proposed schedulers are derived from the default scheduler which is based on single-tenant architecture. Therefore, the proposed schedulers are compared with the following three state-of-the-art single-tenancy schedulers:

1. The **default Apache Storm** scheduler assigns tasks to pre-configured worker processes and then assigns these worker processes to machines in a round-robin fashion. This leads to an even distribution of workload among available machines in the cluster [89].

2. The **Resource-Aware Storm** [22] scheduler, is a system that implements resource-aware scheduling within Storm. RA-Storm is designed to increase overall throughput by maximizing resource utilization while minimizing network latency. When scheduling tasks, RA-Storm can satisfy both soft and hard resource constraints as well as minimize network distance between components that communicate with each other. It consists of two main parts, task selection, and node selection. First, RA-Storm obtains a list of unassigned tasks using breadth-first traversal. Second, a node is selected for each task based on physical distance with bandwidth consideration.

3. The **Isolation** [99] scheduler makes it easy and safe to share a cluster among many topologies. The isolation scheduler lets you specify which topologies should be isolated, meaning that they run on a dedicated set of machines within the cluster with no co-executing topologies. These isolated topologies are given priority in the cluster, so resources will be allocated to isolated topologies if there's competition with non-isolated topologies. Once all isolated topologies are allocated, the remaining machines on the cluster are shared among all non-isolated topologies. The isolation scheduler solves the resource contention problem between topologies by providing full isolation between topologies.

Table 4.1: Hardware Configurations of the Employed Experimental Cluster.

| Hostname | Processor | FLOPs / Cycle | GFLOPs (Equation 3.1) | RAM (GB) | Computation Power (Equation 3.2) | Index |
|---|---|---|---|---|---|---|
| Nimbus<br>Zookeeper<br>Node-A<br>Node-B<br>Nodes-C | Intel Core i7-6700 CPU @ 3.40GHz x 8 | 6816 | 185395 | 3.7 | 148317 | 1 |
| Nod-D<br>Node-E<br>Node-F | Intel Core i5-4460 CPU @ 3.20GHz x 4[1] | 6385 | 81728 | 7.7 | 65384 | 2 |
| Node-G<br>Node-H<br>Node-I | Intel Core i5 CPU M 460 @ 2.53GHz x 4 | 5054 | 51147 | 3.7 | 40918 | 3 |

[1]Core i5 processors are considered mid-range CPUs with a moderate number of cores and threads. Their impact on results can vary depending on the requirements. In cases where substantial computational power or parallel processing is necessary, a Core i5 system may face limitations and performance constraints. However, any negative effects are expected to be consistent across all scheduling heuristics employed in the experiments.

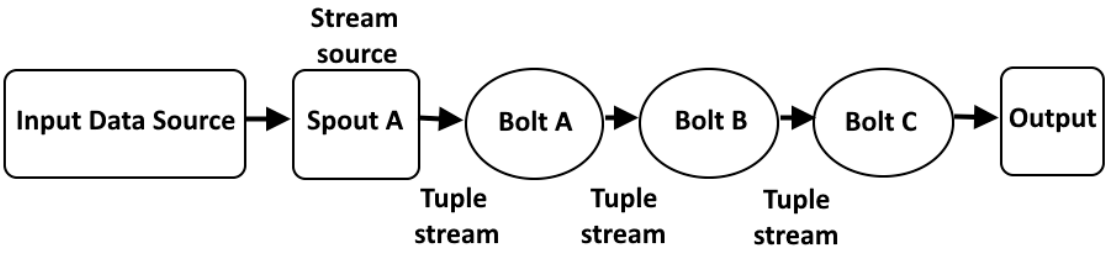



FIGURE 4.1: Layout of Linear Topology.

## 4.3 Test Topology Selection and Scenario Used

The Storm has various types of topology structures for example, linear topology layout (Fig. 4.1), diamond topology layout (Fig. 4.2), and star topology layout (Fig. 4.3). For homogeneous systems, a symmetric topology that evenly distributes the workload across cores is suitable to leverage the full processing power of the multicore processors. In contrast, heterogeneous systems benefit from an asymmetric topology that assigns tasks based on core capabilities, optimizing resource utilization across different cores. The performance of the proposed schemes is evaluated with the help of 2 linear benchmark topologies. Those topologies are selected that can process data and make decisions. In this context, the topologies which are shipped with Apache Storm suit [100] and used by researchers as a benchmark are the following:

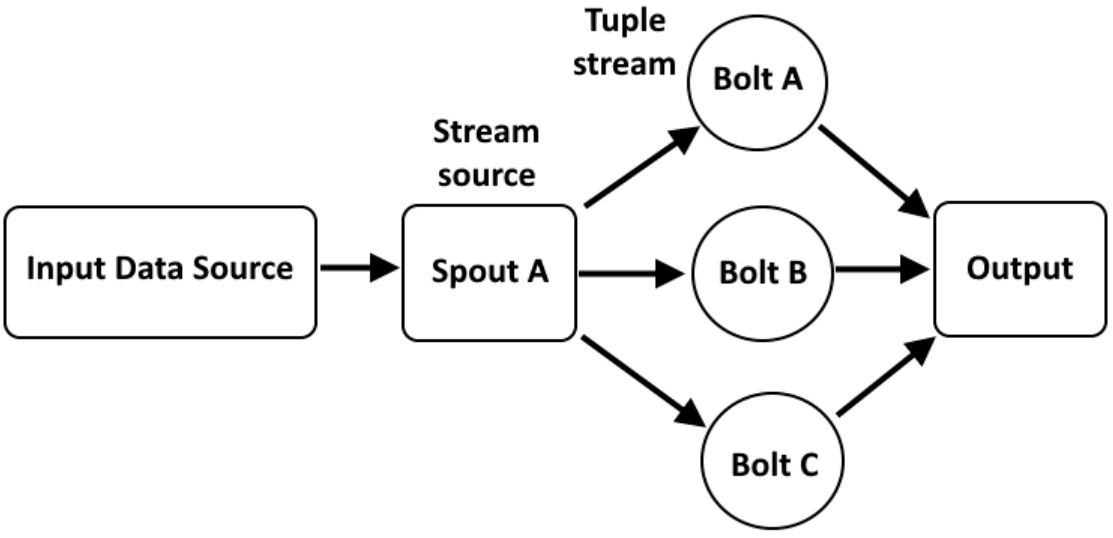



FIGURE 4.2: Layout of Diamond Topology.

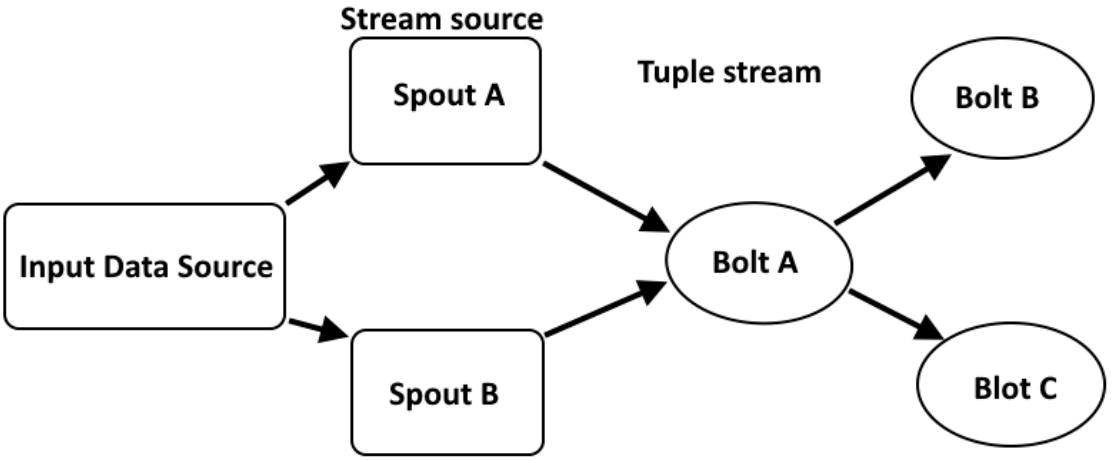



FIGURE 4.3: Layout of Star Topology.

1. **Word Count** Topology [101] breaks sentences into words and then counts the number of occurrences of each word as shown in listing 4.1.

2. **Exclamation Topology** [14] breaks sentences into words and then appends three exclamation marks (!!!) to the words as shown in listing 4.2. It uses a spout that generates random words and two bolts that just append three exclamation marks to each word. As a result, we get words with six exclamation marks. For example, the word "Ruzainah", passes through 2 bolts which at the end produce "Ruzainah!!!!!!"[102].

For the evaluation of the proposed schemes using 2 topologies with 3 state-of-the-art schedulers different scenarios are designed to test the suitability of the proposed

TABLE 4.2: Different Scenarios for the Experiments.

| | Scenario 1 | Scenario 2 | Scenario 3 | Scenario 4 | Scenario 5 | Scenario 6 |
|---|---|---|---|---|---|---|
| **Total Supervisor Nodes** | 9 | | | | | |
| **Number of Available Slots on Each Supervisor Node** | 3 | | | 4 | | 5 |
| **Total Available Slots for Topology Execution** | 27 (9 nodes x 3 slots) | | | 36 (9 nodes x 4 slots) | | 45 (9 nodes x 5 slots) |
| **Slots Required for Topology Execution** | 3 | 4 | 5 | 4 | 5 | 5 |

```
public static void main(String[] args) throws Exception {
TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("spout", new RandomSentenceSpout(),
   5);
builder.setBolt("split", new SplitSentence(),
   8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(),
   12).fieldsGrouping("split", new Fields("word"));

Config conf = new Config();
conf.setDebug(true);

if (args != null && args.length > 0){
        conf.setNumWorkers(3);
        StormSubmitter.submitTopology(args[0], conf,
   builder.createTopology());
}
else{
        conf.setMaxTaskParallelism(3);

        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("word-count", conf,
   builder.createTopology());

        Thread.sleep(10000);
        cluster.shutdown();
        }
}
```

LISTING 4.1: Word count topology

schemes. Overall 6 different scenarios are designed as shown in Table 4.2. Each scheduler for word count as well as for exclamation topology is executed under these scenarios and results are recorded. For example, the number of slots required for topology execution varied from 3 to 5 for both topologies. Tuple processing by each scheduler is recorded with a varying number of worker processes to analyze the impact on throughput. Similarly, the number of available worker processes for each node also varied from 3 to 5 (for details see Table 4.2). For example, Scenario 4 states that we have 4 slots per supervisor node and we have 9 supervisor nodes in the given cluster. As a total, we have 36 (4 slots x 9 nodes) slots available for topology scheduling. For scenario 4, 4 slots are required by topology for execution. Therefore, 4 slots from 36 available slots will be selected for topology execution

```
1  public static void main(String[] args) throws Exception {
2  TopologyBuilder builder = new TopologyBuilder();
3
4  builder.setSpout("word", new TestWordSpout(), 10);
5  builder.setBolt("exclaim1", new ExclamationBolt(),
       3).shuffleGrouping("word");
6  builder.setBolt("exclaim2", new ExclamationBolt(),
       2).shuffleGrouping("exclaim1");
7
8  Config conf = new Config();
9  conf.setDebug(true);
10
11 if (args != null && args.length > 0) {
12         conf.setNumWorkers(3);
13         StormSubmitter.submitTopology(args[0], conf,
       builder.createTopology());
14 }
15 else {
16         LocalCluster cluster = new LocalCluster();
17         cluster.submitTopology("test", conf,
       builder.createTopology());
18         Utils.sleep(10000);
19         cluster.killTopology("test");
20         cluster.shutdown();
21         }
22 }
```

Listing 4.2: Exclamation topology

by the scheduler. This selection of slots will vary from the scheduler to scheduler based on their algorithm. For a fair comparison, each experiment is performed for 20 minutes time slots while reading (of the employed performance metrics) is recorded at a 1-minute interval.

## 4.4 Comparison with State-of-the-art Schedulers

### 4.4.1 Word Count Topology

Using the word count topology, a performance comparison of the employed scheduling mechanisms (see Fig. 4.4-4.9), average throughput (see Fig. 4.10), average throughput per node (see Fig. 4.12), and the number of supervisor nodes used (i.e.,

resource usage as shown in Fig. 4.11) is presented. This comparison is performed using scenarios 1–6 (see Table 4.2).

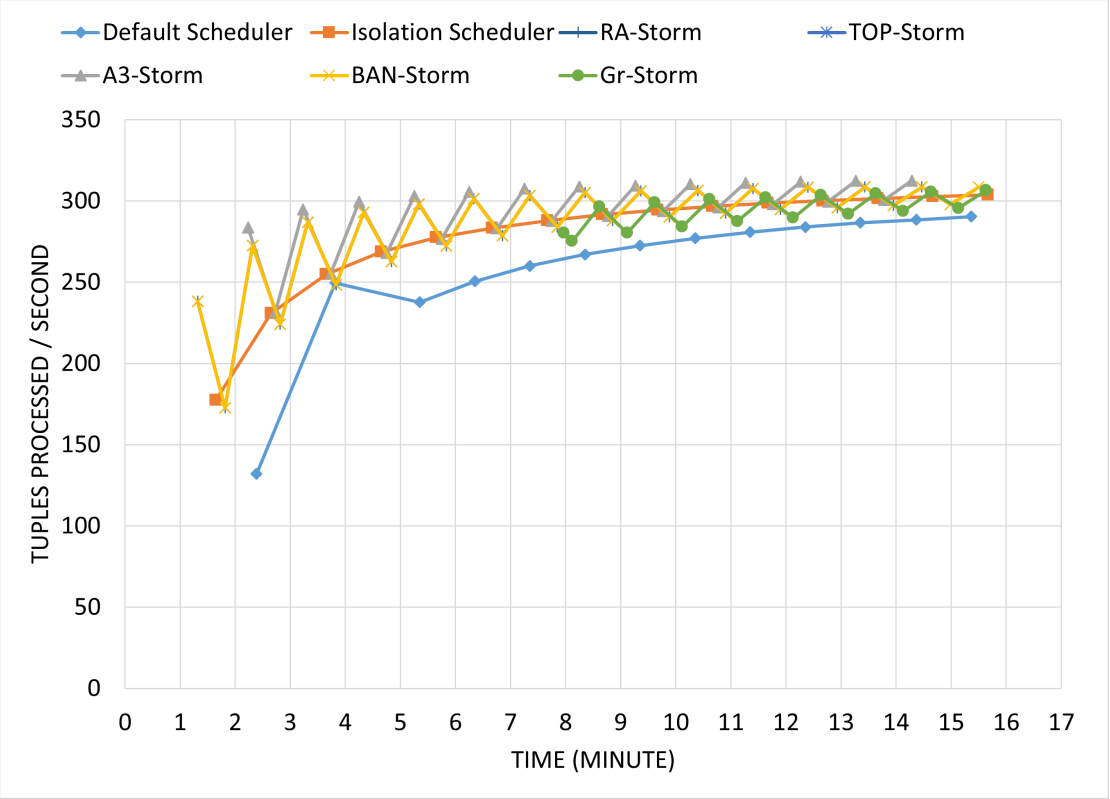



FIGURE 4.4: Performance Comparison of the Scheduling Algorithms for Word Count Topology using Scenario 1.

For scenario 1, we have 3 slots per supervisor (a total of 27 slots as we have 9 supervisor nodes) and 3 slots are required by the word count topology for execution. Fig. 4.4 shows inter-executor traffic produced using scenario 1. Here, the TOP-Storm, BAN-Storm and Gr-Storm improve approximately 6% on average throughput and the A3-Storm improves approximately 8% as compared to the default scheduler (see results in Fig. 4.10). The proposed schedulers attain commendable results while using only 60% of computational resources (i.e., 3 supervisor node of the employed cluster) with respect to the default scheduler which acquires 5 nodes of the cluster for execution (see Fig. 4.11). The reason for this assignment is that the proposed schedulers consume all available slots of a selected node and then employ the next node for scheduling. On the other hand, the default scheduler makes such an assignment in a round-robin fashion.

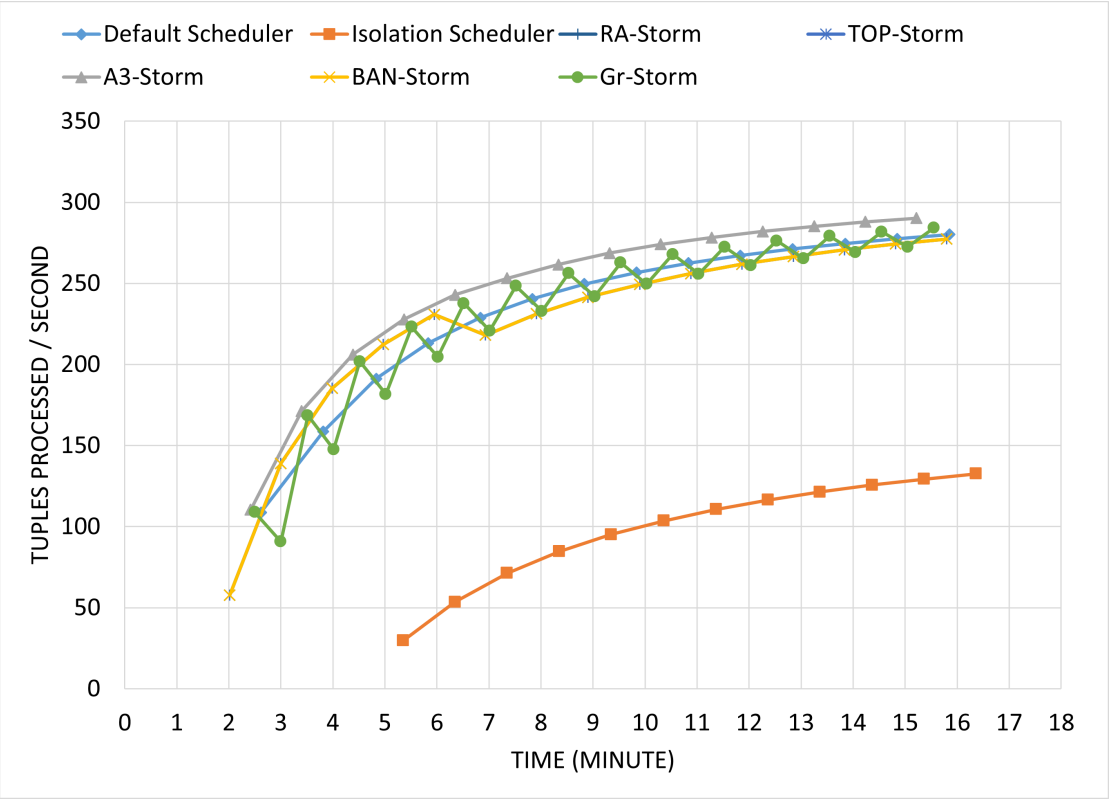



FIGURE 4.5: Performance Comparison of the Scheduling Algorithms for Word Count Topology using Scenario 2.

The performance comparison between the proposed schedulers and 3 baseline scheduling algorithms executed using scenario 2 is shown in Fig. 4.5. Improvement in average throughput (see Fig. 4.10) is observed at 4% and 1% for A3-Storm and Gr-Storm, respectively (with 33% less computational resources (as shown in Fig. 4.11)).

Fig. 4.6 shows the inter-node traffic-related results for the schedulers. From Fig. 4.6, it is observed that the TOP-Storm outperforms all the other scheduling heuristics for different performance metrics. Average throughput (shown in Fig. 4.10) attained up to 5% for the TOP-Storm scheduler using 4 supervisor nodes while the default scheduler consumed 7 supervisor nodes (as shown in Fig. 4.11).

As shown in Fig. 4.7, similar performance results were attained for the scheduling algorithms executed using scenario 4 with a single computing node (see Fig. 4.11). Gr-Storm attained up to 28% improvement (in Fig. 4.10). In scenario 5, we have 4 slots per node and 5 slots are required by the word count topology.

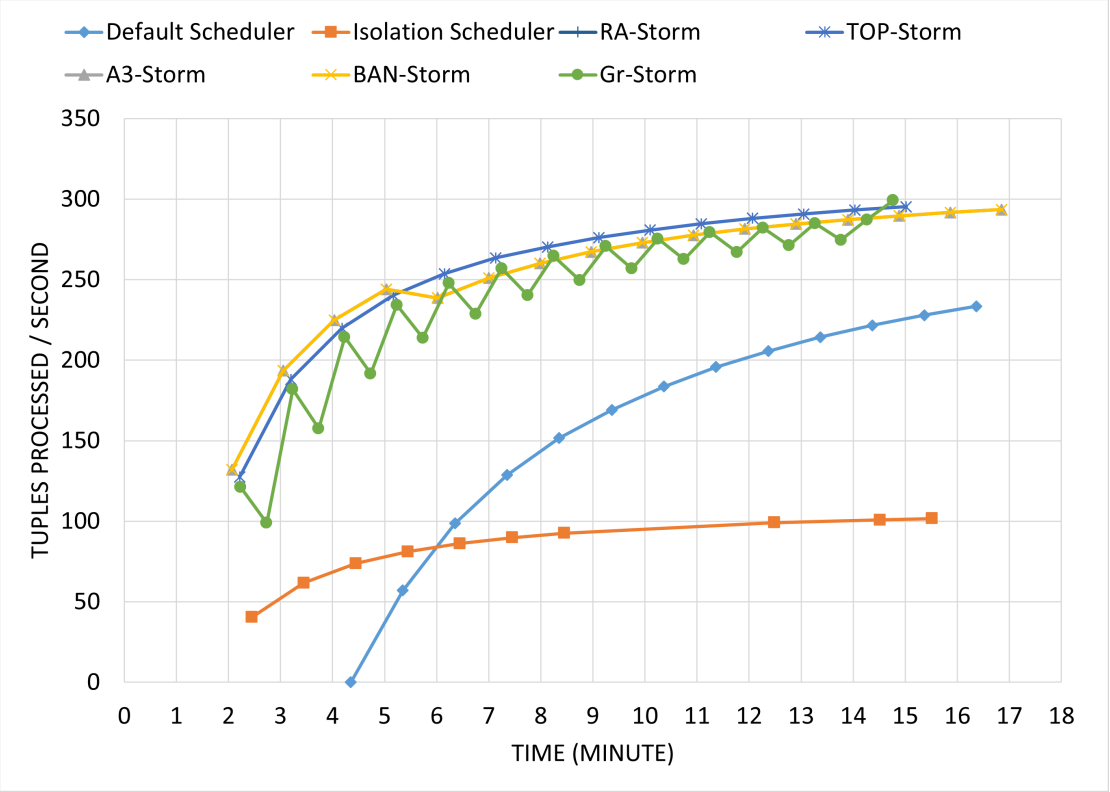



FIGURE 4.6: Performance Comparison of the Scheduling Algorithms for Word Count Topology using Scenario 3.

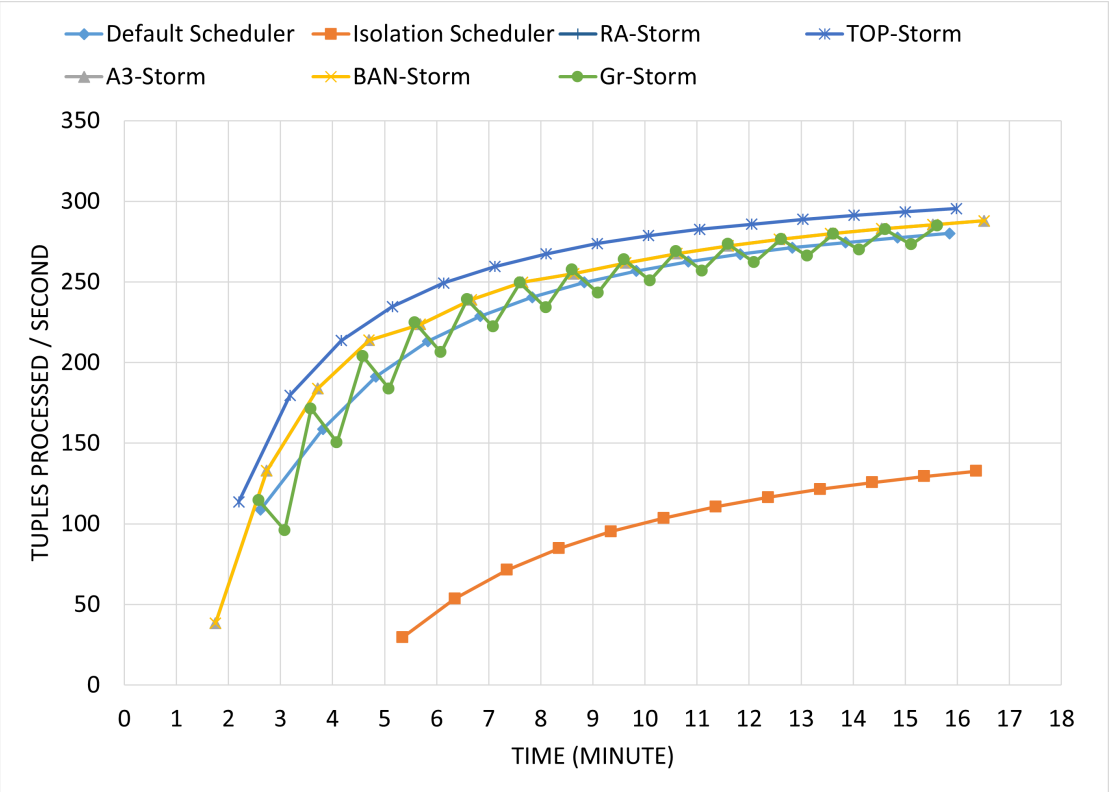



FIGURE 4.7: Performance Comparison of the Scheduling Algorithms for Word Count Topology using Scenario 4.

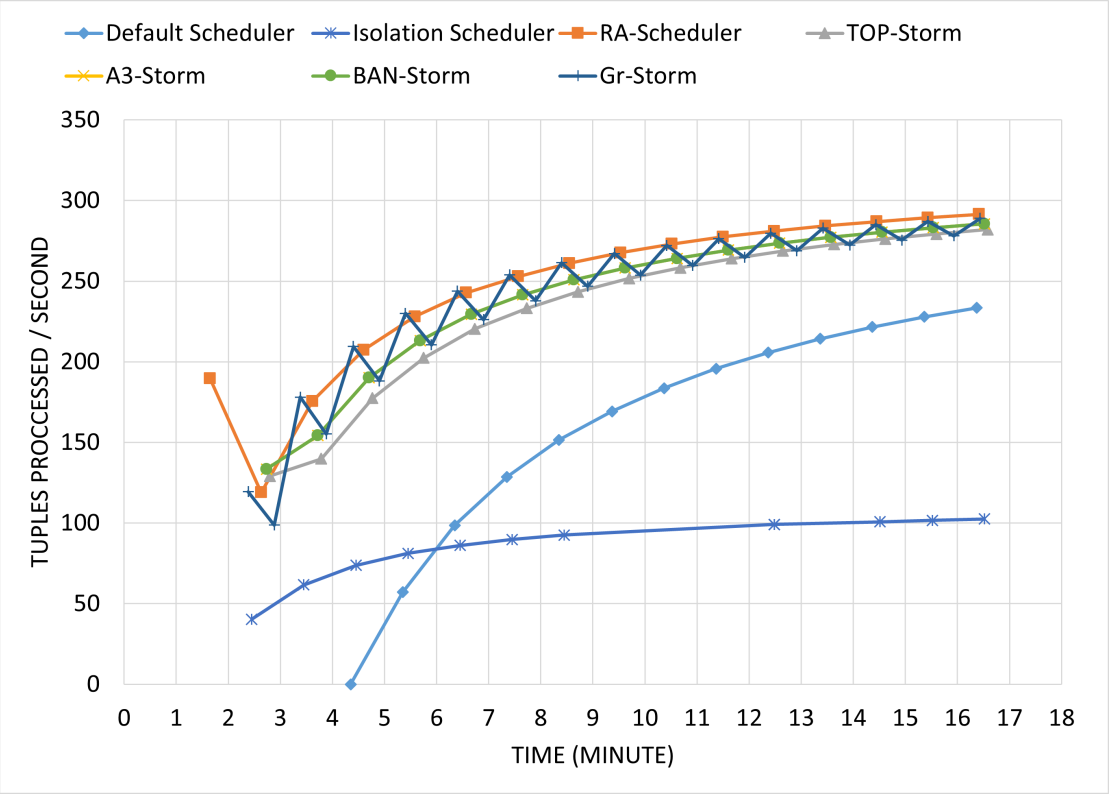



FIGURE 4.8: Performance Comparison of the Scheduling Algorithms for Word Count Topology using Scenario 5.

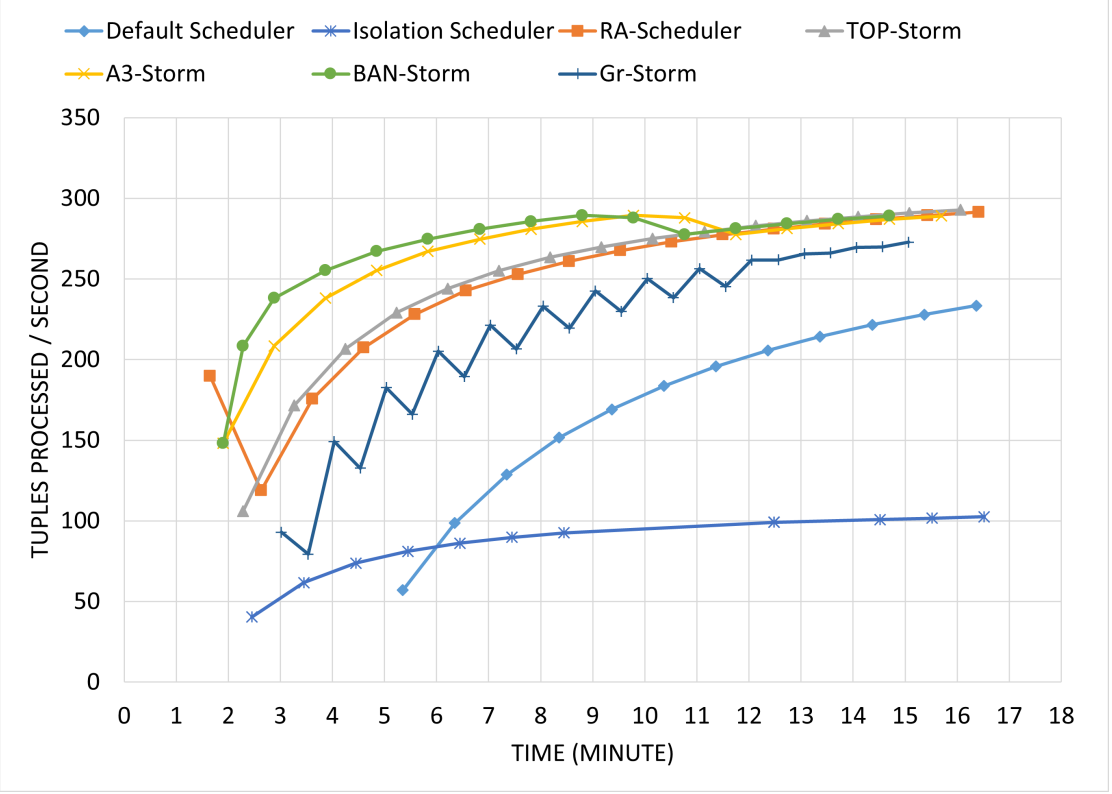
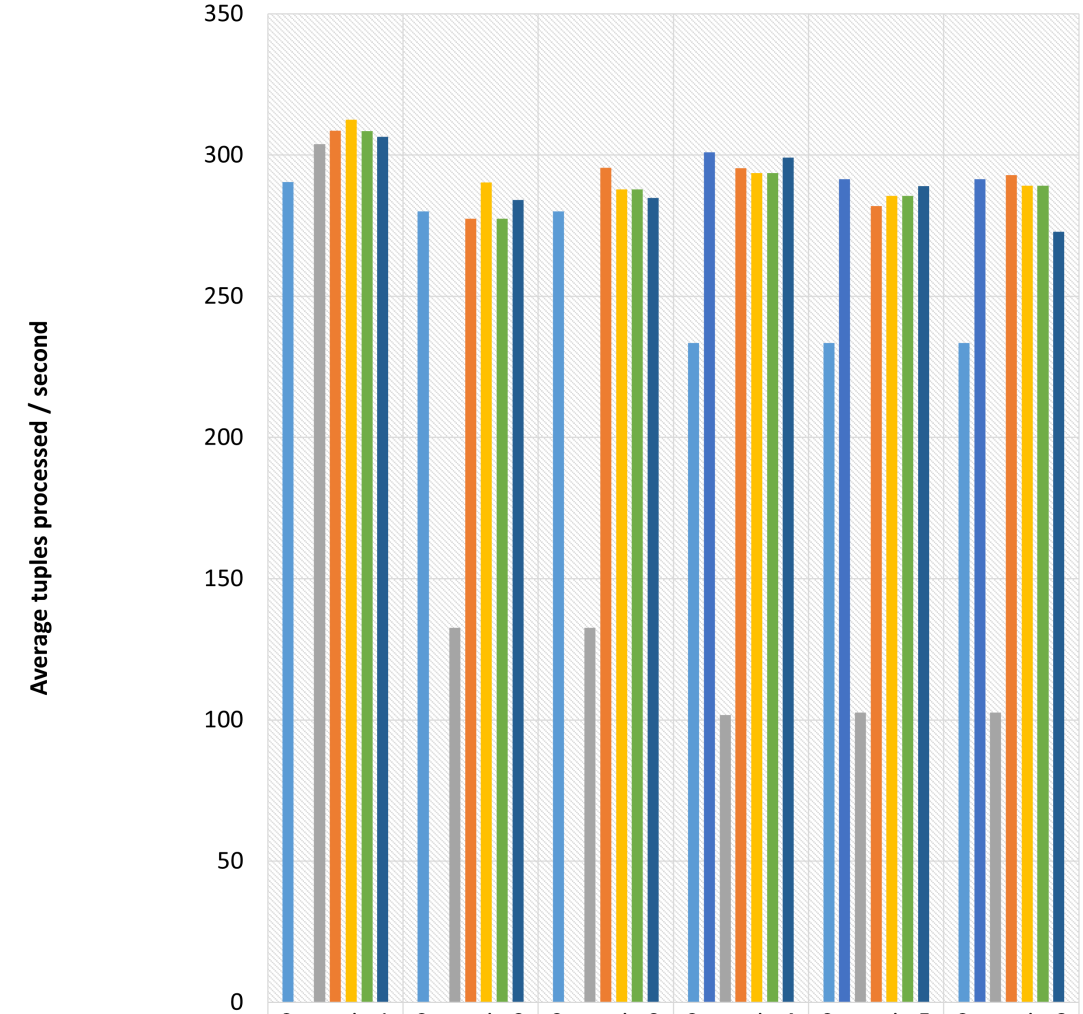



FIGURE 4.9: Performance Comparison of the Scheduling Algorithms for Word Count Topology using Scenario 6.




| | Scenario 1 | Scenario 2 | Scenario 3 | Scenario 4 | Scenario 5 | Scenario 6 |
|---|---|---|---|---|---|---|
| Default Scheduler | 290 | 280 | 280 | 233 | 233 | 233 |
| RA-Storm | 0 | 0 | 0 | 301 | 291 | 291 |
| Isolation Scheduler | 304 | 133 | 133 | 102 | 103 | 103 |
| TOP-Storm | 309 | 277 | 295 | 295 | 282 | 293 |
| A3-Storm | 312 | 290 | 288 | 294 | 286 | 289 |
| BAN-Storm | 309 | 277 | 288 | 294 | 286 | 289 |
| Gr-Storm | 307 | 284 | 285 | 299 | 289 | 273 |

FIGURE 4.10: Average Throughput Calculated using Equation 1.1 for Word Count Topology.

Fig. 4.8 shows inter-executor traffic produced using this scenario. Here, the Gr-Storm improves performance by approximately up to 24% on average throughput. When scenario 6 is deployed (Fig. 4.9) then the TRA-Storm computes up to 25% more tuples in the given time (Fig. 4.10) with 58% fewer resources (Fig. 4.11).

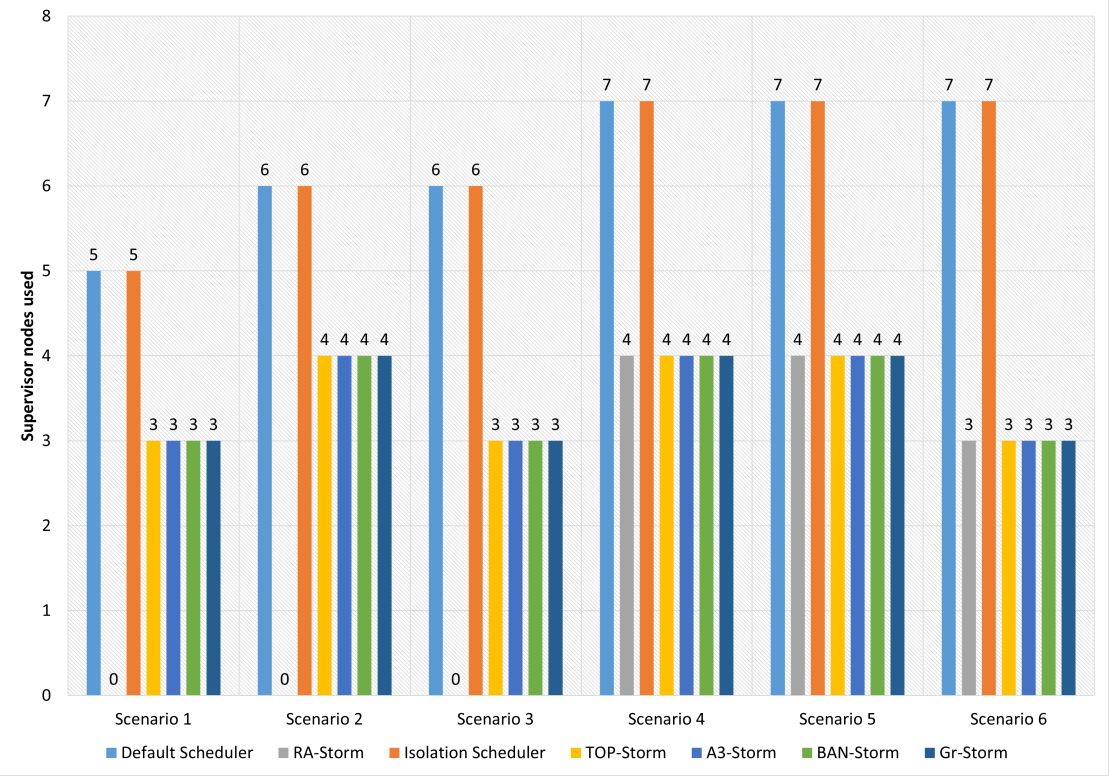


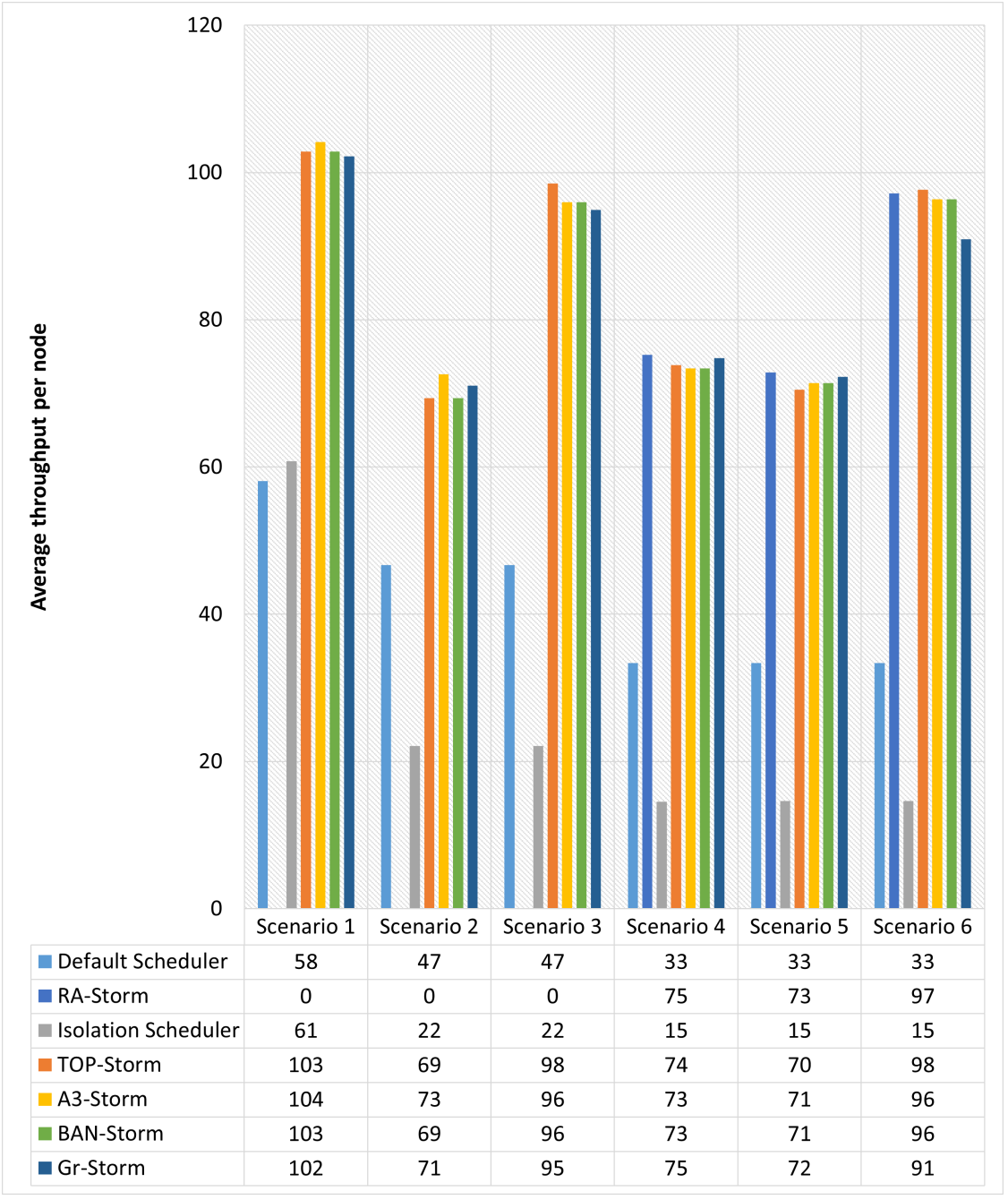

FIGURE 4.11: The Number of Nodes Used for Word Count by Scheduling Algorithms.

### 4.4.2 Word Count Topology - Result Analysis

Figure 4.10 shows the average throughput produced using scenarios 1 to 6 with the y-axis representing average tuples processed (throughput). Experiments run for hours, however, the performance after around fifteen minutes remains nearly the same.

In figure 4.10, we can see that in all 6 scenarios the proposed schedulers performed better as compared to the state-of-the-art schemes. A3-Storm processed maximum tuples for scenarios 1 and 2. TOP-Storm remains the top performer for scenarios 3 and 6. RA-Storm performed best in scenarios 4 and 5. These scenarios are based on a different number of slots available on each supervisor node and the number of slots required by a topology to run. As a result, the scheduling algorithms generate different execution plans which ultimately affect the throughput.

According to Figure 4.10, it is observed that some results are identical. This is due to the scheduling mechanism of the benchmarked algorithms. To understand




| | Scenario 1 | Scenario 2 | Scenario 3 | Scenario 4 | Scenario 5 | Scenario 6 |
|---|---|---|---|---|---|---|
| Default Scheduler | 58 | 47 | 47 | 33 | 33 | 33 |
| RA-Storm | 0 | 0 | 0 | 75 | 73 | 97 |
| Isolation Scheduler | 61 | 22 | 22 | 15 | 15 | 15 |
| TOP-Storm | 103 | 69 | 98 | 74 | 70 | 98 |
| A3-Storm | 104 | 73 | 96 | 73 | 71 | 96 |
| BAN-Storm | 103 | 69 | 96 | 73 | 71 | 96 |
| Gr-Storm | 102 | 71 | 95 | 75 | 72 | 91 |

FIGURE 4.12: Average Throughput per Node for Word Count Topology Calculated using Equation 1.2.

their working, let's take an example. Suppose we have to schedule a topology using 3 slots. In the given cluster, if we have 3 supervisor nodes with a single slot on each node. Using round-robin technique, topology will be scheduled, and all 3 slots are being used. Furthermore, if number of slots are increased from 1 to 2 on each node and try to regenerate new execution plan with 3 required slots then still same execution plan will be generated by the default storm scheduler because the default scheduler occupy slot from one node and move to another node for next slot in round-robin fashion. Therefore, if 3 slots are required and in the cluster of 3 nodes same execution plan will be generated whether each node has 1 slot or 3 slots. From Table 4.2, it is obvious that number of available slots are same for scenarios 1 – 3, scenarios 4 - 5 and scenario 6 and required number of slots are variable for all these scenarios. That's why for the default storm scheduler, we have same results for scenario 1, scenarios 2 – 3, and scenarios 4 – 6.

In Figure 4.11, the total number of nodes used by each scheduling algorithm is presented. Due to the compact allocation scheme, the processed schedulers employed a minimum number of supervisor nodes in each case. As a result, the average throughput per node for word count topology under scenarios 1 to 6 (see Figure 4.12) is almost double.

### 4.4.3 Exclamation Topology

The same procedure is repeated for exclamation topology. Average throughput (Fig. 4.19), and resource usage (Fig. 4.20) for exclamation topology are shown here. This comparison is also performed using the scenarios given in Table 4.2. For scenario 1, we have 3 slots per supervisor, also we need 3 slots to execute the topology. Fig. 4.13 shows inter-executor traffic produced using scenario 1. Gr-Storm attain a modest 2% improvement in average throughput as compared to the default scheduler (see Fig. 4.19). However, Gr-Storm achieves a 2% improved throughput with 40% less computational resource acquisition as compared to the default scheduler (4.20). The default scheduler uses a round-robin approach that

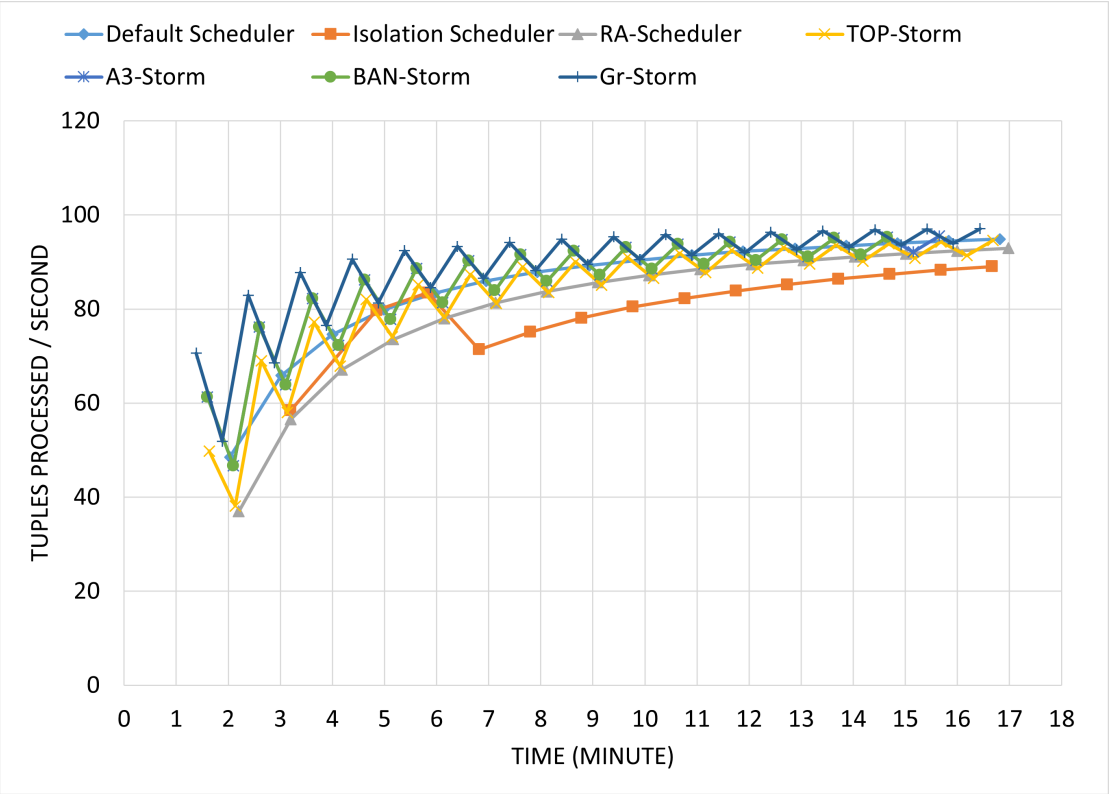



FIGURE 4.13: Performance Comparison of the Scheduling Algorithms for Exclamation Topology using Scenario 1.

engages maximum supervisor nodes in the cluster. As a result, inter-node traffic is increased, and throughput is decreased.

The performance comparison of the algorithms under scenario 2 is shown in Fig. 4.14. Improvement in average throughput (Fig. 4.19) is 23% for BAN-Storm respectively with 66% resources (Fig. 28).

Fig. 4.15 shows a performance comparison between schedulers in terms of inter-node traffic. From Fig. 4.15, it is observed that A3-Storm outperforms other scheduling algorithms in every performance metric. Average throughput (shown in Fig. 4.19) is improved to 23% for the A3-Storm scheduler using 4 supervisor nodes while the default is using 7 supervisors (Fig. 4.20).

As shown in Fig. 4.16, the same performance metric curve of these scheduling algorithms under scenario 4 has a different magnitude (see Fig. 4.19). TOP-Storm achieved 3% (in Fig. 4.19) and the reason for having a minor improvement in average throughput is that we have a total of 18 executors for exclamation

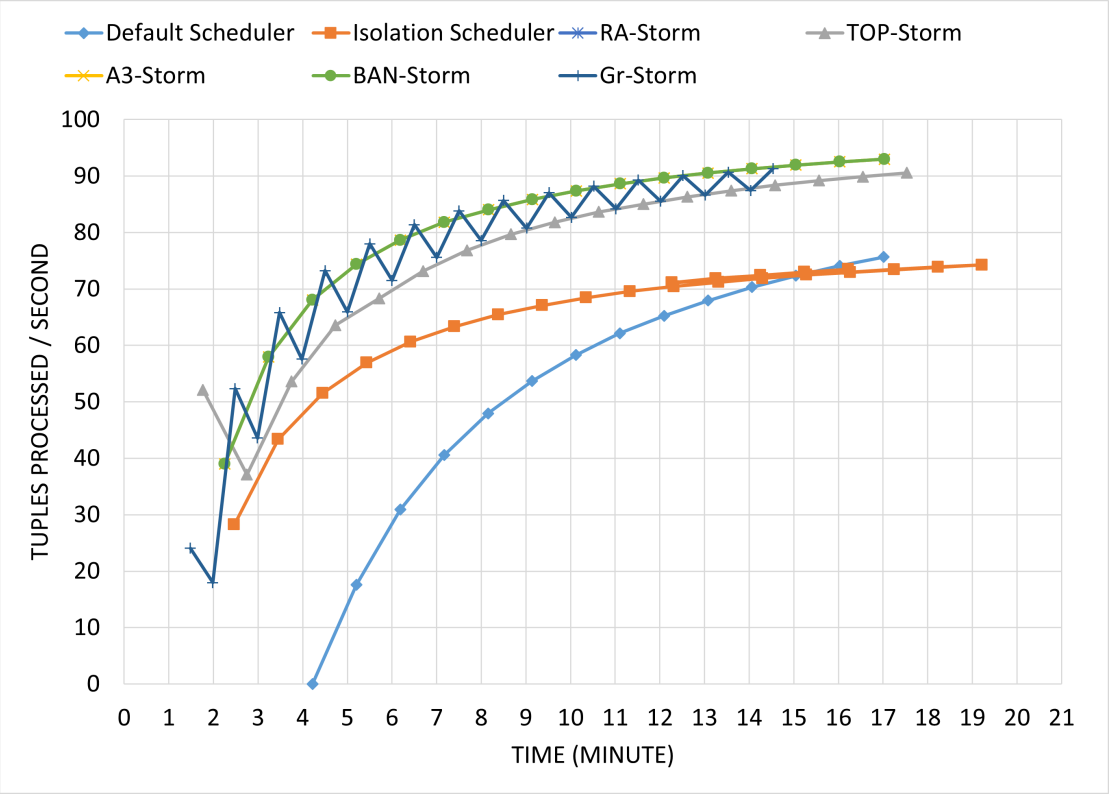



FIGURE 4.14: Performance Comparison of the Scheduling Algorithms for Exclamation Topology using Scenario 2.

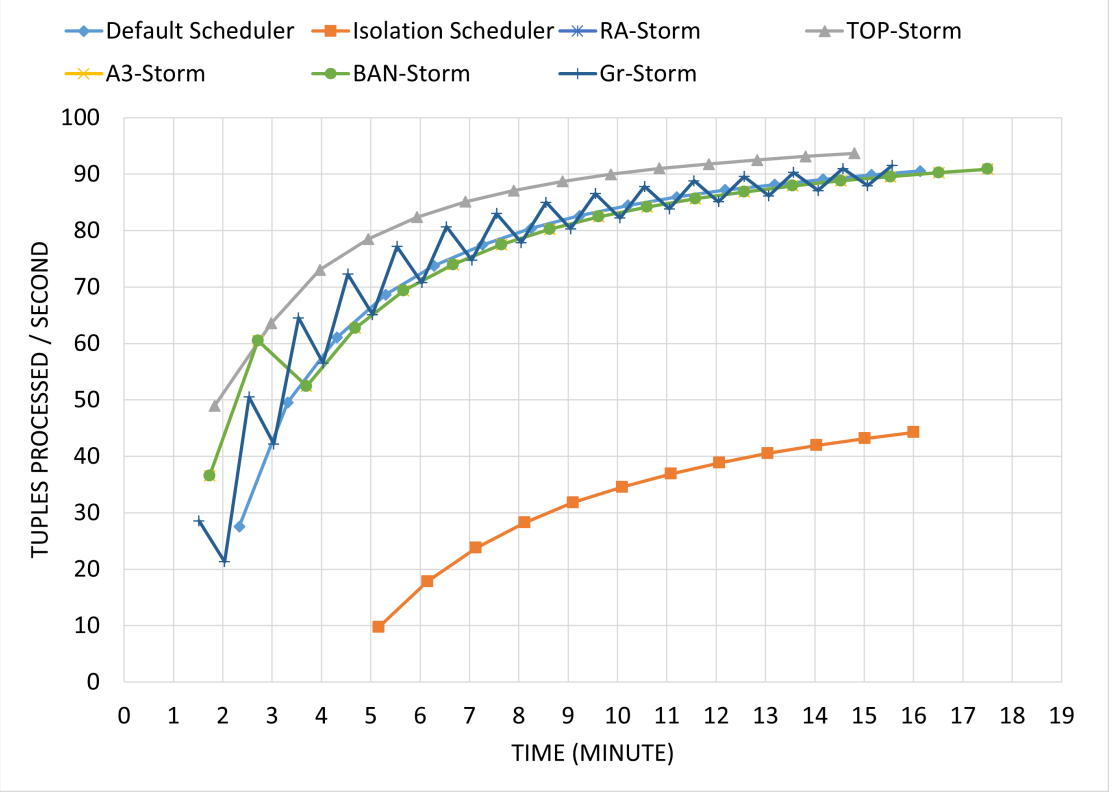



FIGURE 4.15: Performance Comparison of the Scheduling Algorithms for Exclamation Topology using Scenario 3.

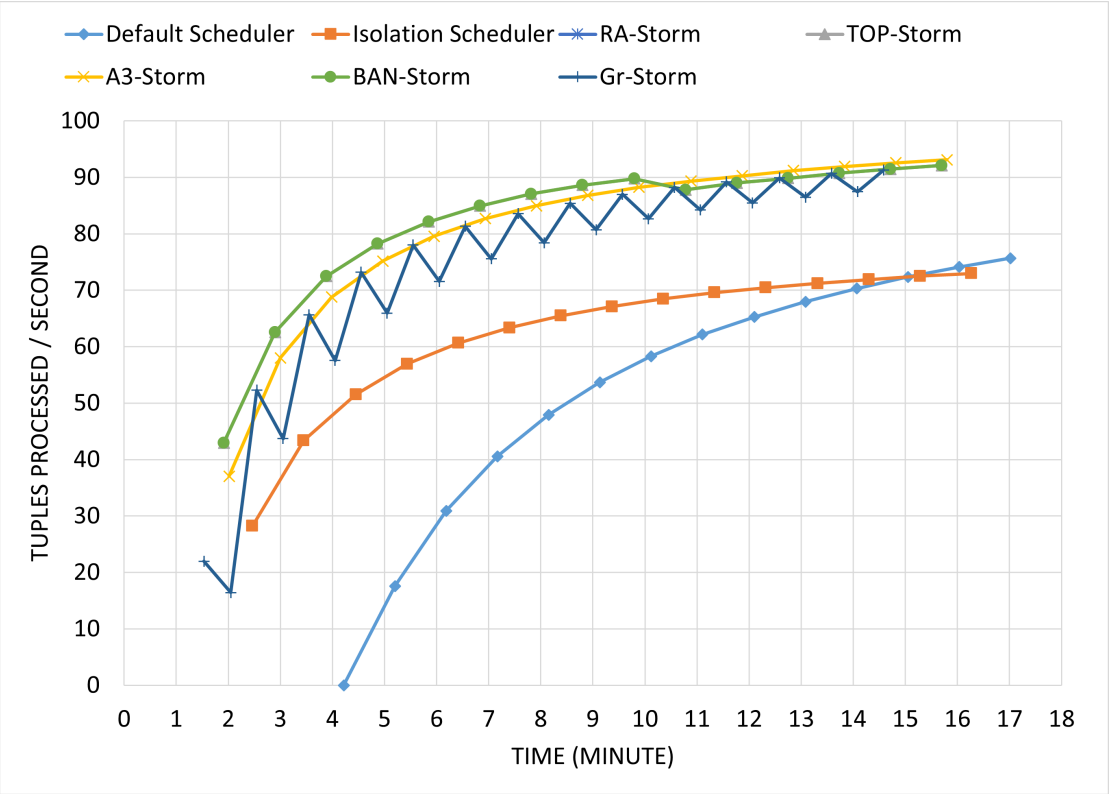



FIGURE 4.16: Performance Comparison of the Scheduling Algorithms for Exclamation Topology using Scenario 4.

topology. As the number of slots is increasing then inter-executor communication is also increasing. For scenarios with fewer slots, we have better results.

For the execution of scenario 5, we have 4 slots per node and 5 slots in total are required for the execution of exclamation topology. Fig. 4.17 shows inter-executor communication produced using scenario 5. Both TOP-Storm and TRA-Storm improved around 3% (as shown in Fig. 4.19) with only 4/7 resources when compared to the default scheduler (see Fig. 4.20). When scenario 6 is deployed (Fig. 4.18) then the A3-Storm processed 2% more tuples in the given time (Fig. 4.19) with the consumption of 20% resources (Fig. 4.20).

### 4.4.4 Exclamation Topology - Result Analysis

In figure 4.19, we can see that in all 6 scenarios the proposed schedulers performed better as compared to the state-of-the-art schemes. Gr-Storm processed maximum

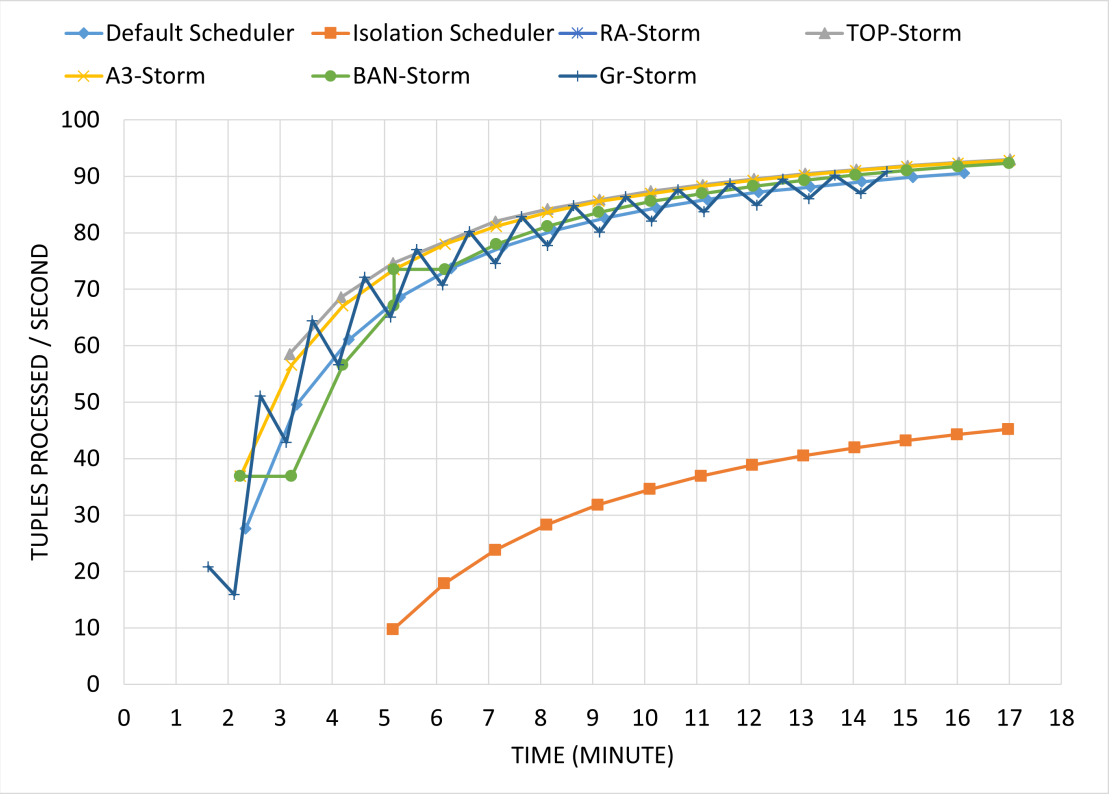



FIGURE 4.17: Performance Comparison of the Scheduling Algorithms for Exclamation Topology using Scenario 5.

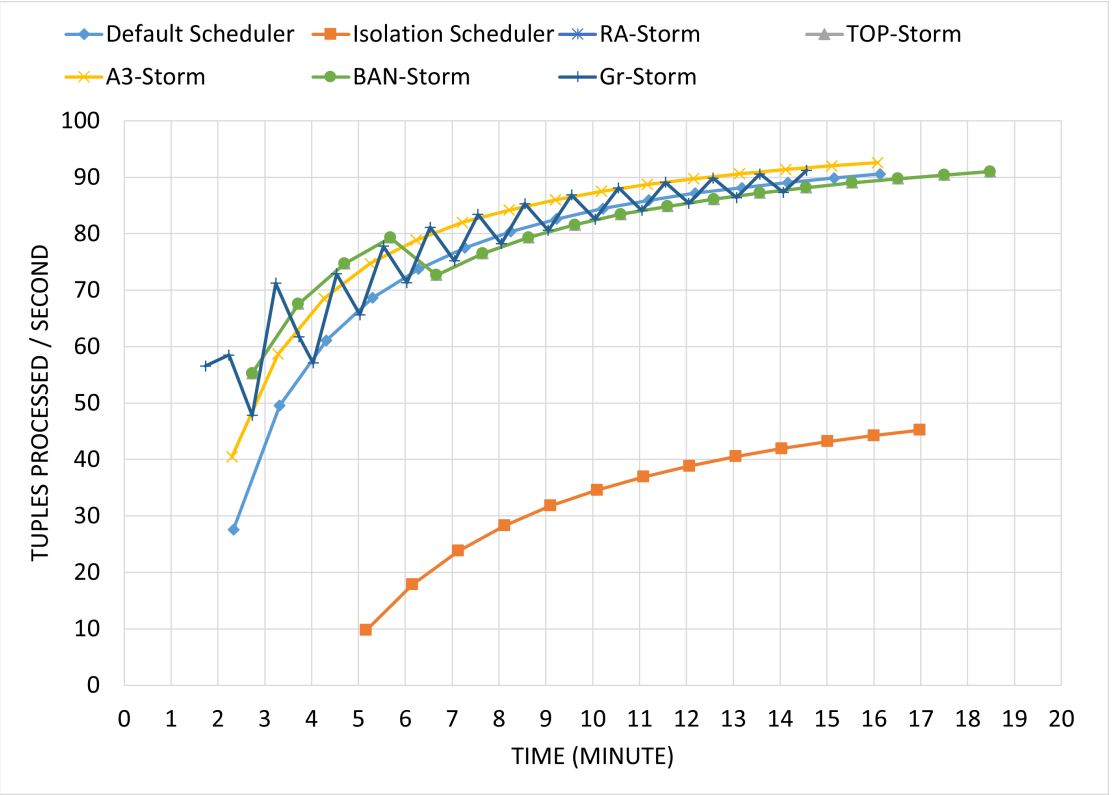
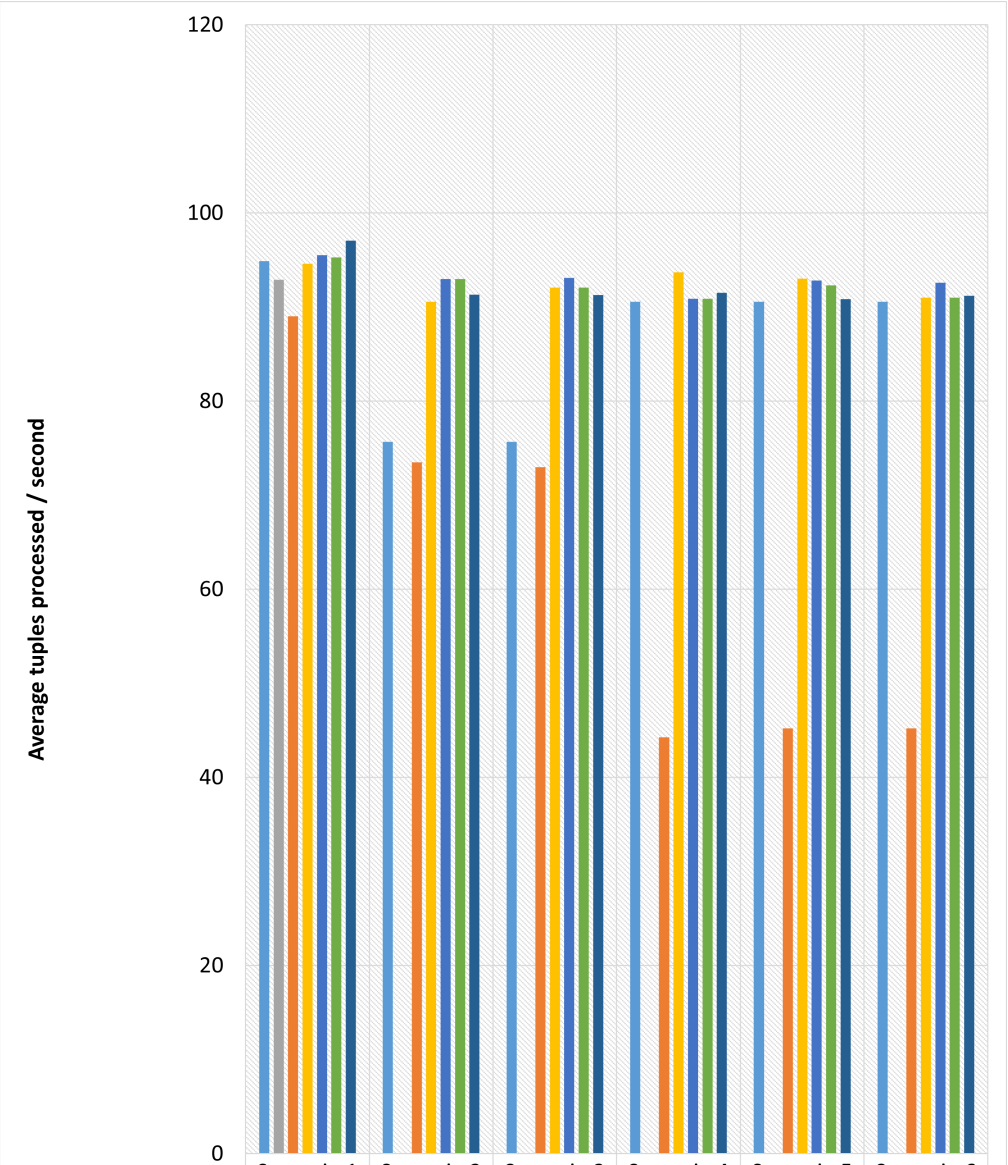



FIGURE 4.18: Performance Comparison of the Scheduling Algorithms for Exclamation Topology using Scenario 6.




| | Scenario 1 | Scenario 2 | Scenario 3 | Scenario 4 | Scenario 5 | Scenario 6 |
|---|---|---|---|---|---|---|
| Default Scheduler | 95 | 76 | 76 | 91 | 91 | 91 |
| RA-Storm | 93 | 0 | 0 | 0 | 0 | 0 |
| Isolation Scheduler | 89 | 73 | 73 | 44 | 45 | 45 |
| TOP-Storm | 95 | 91 | 92 | 94 | 93 | 91 |
| A3-Storm | 96 | 93 | 93 | 91 | 93 | 93 |
| BAN-Storm | 95 | 93 | 92 | 91 | 92 | 91 |
| Gr-Storm | 97 | 91 | 91 | 92 | 91 | 91 |

FIGURE 4.19: Average Throughput Calculated using Equation 1.1 of the Scheduling Algorithms for Exclamation Topology.

tuples for scenario 1. However, Ban-Storm performed well in scenario 2. A3-Storm remains the top performer for scenarios 3, 5, and 6. TOP-Storm performed best in scenario 4.

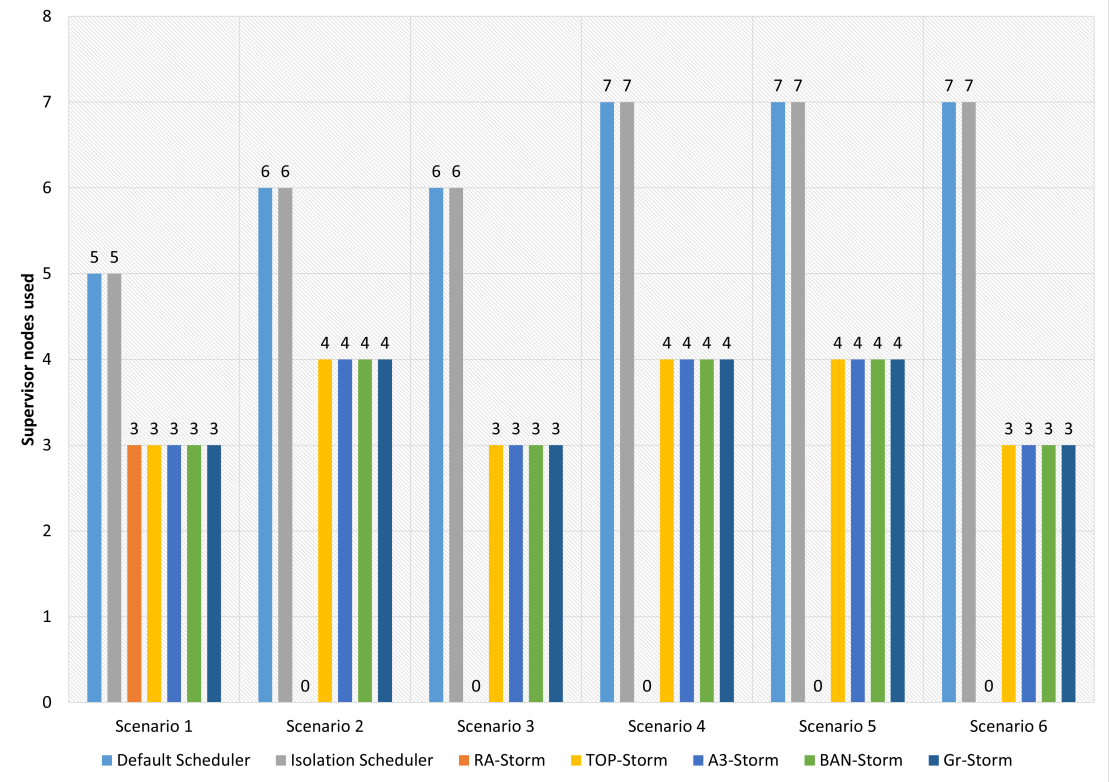
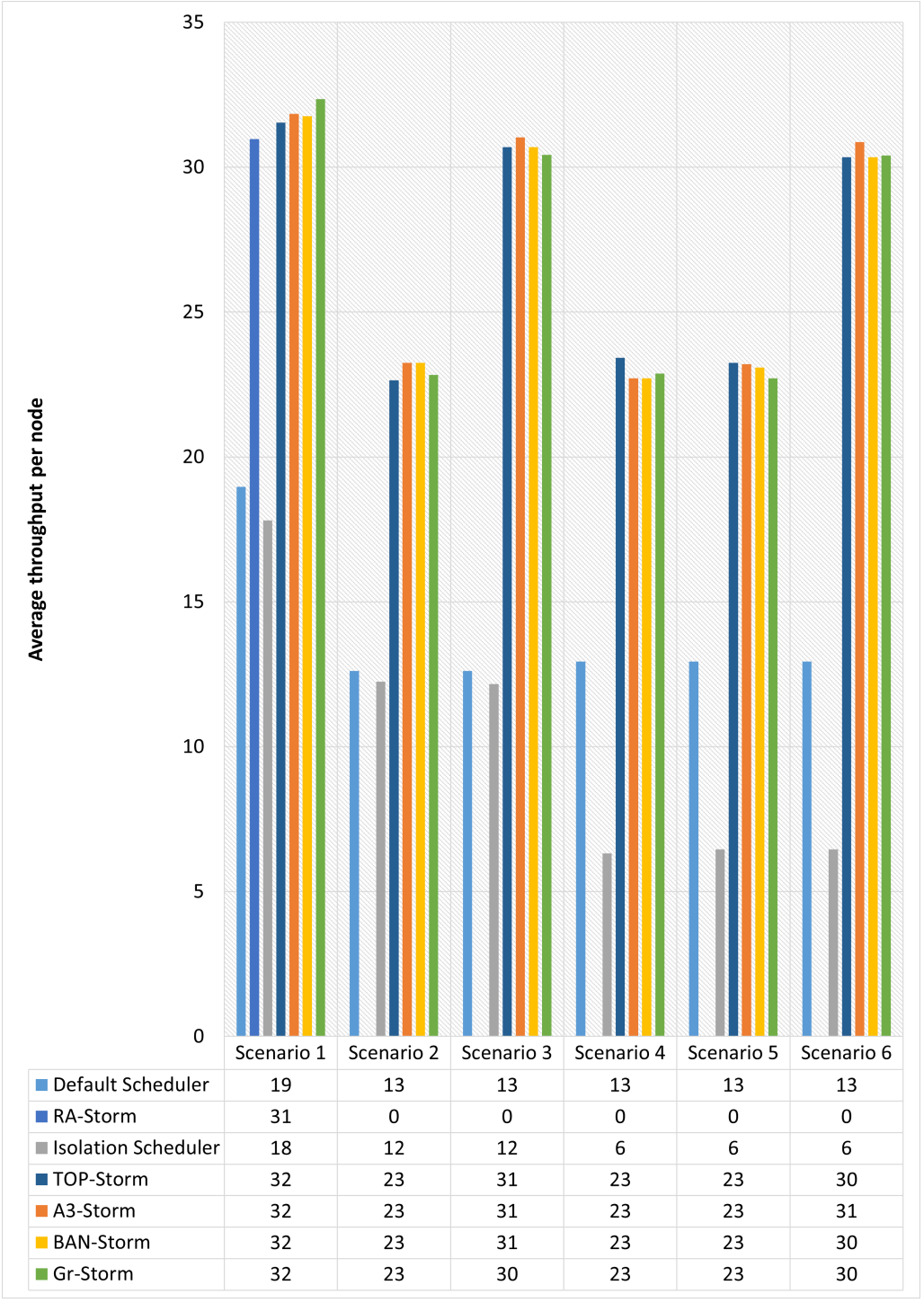



FIGURE 4.20: The Number of Nodes Used for Exclamation Topology by Scheduling Algorithms.

With the help of Equation 1.2, the average throughput per node for the proposed schedulers and baseline scheduling algorithms are calculated as shown in Figure 4.21. significant Improvement in average throughput per node is achieved by the proposed schedulers with respect to the default scheduler and isolation scheduler with fewer resources concerning the other schedulers (as shown in Figure 4.20). The reason is that the proposed schemes consume all available slots of a machine and then use the next machine. Conversely, the default Storm uses a round-robin approach.

## 4.5 Result Analysis

It is evident from the results that the proposed schedulers are more efficient (in terms of throughput and resource utilization) as compared to state-of-the-art scheduling heuristics. Results show the importance of considering inter-tasks communication for stream-based scheduling heuristics. These results will be difficult




| | Scenario 1 | Scenario 2 | Scenario 3 | Scenario 4 | Scenario 5 | Scenario 6 |
|---|---|---|---|---|---|---|
| Default Scheduler | 19 | 13 | 13 | 13 | 13 | 13 |
| RA-Storm | 31 | 0 | 0 | 0 | 0 | 0 |
| Isolation Scheduler | 18 | 12 | 12 | 6 | 6 | 6 |
| TOP-Storm | 32 | 23 | 31 | 23 | 23 | 30 |
| A3-Storm | 32 | 23 | 31 | 23 | 23 | 31 |
| BAN-Storm | 32 | 23 | 31 | 23 | 23 | 30 |
| Gr-Storm | 32 | 23 | 30 | 23 | 23 | 30 |

FIGURE 4.21: Average Throughput per Node for Exclamation Topology Calculated using Equation 1.2.

to achieve if the available resources or communication patterns between executors are ignored.

The default scheduler uses a round-robin approach which may increase inter-node

traffic and decrease throughput. Similarly, it engages maximum supervisor nodes in the cluster. Conversely, the proposed schedulers consume all available slots of a node and then move to the next node if required. In this way, a compact slots assignment is achieved which is directly proportional to higher throughput. In summary, up to 2 times for word count and 1 and half times for exclamation topologies higher average throughput has been attained with reduced resource acquisition (i.e., 60% fewer resources). For configurations where fewer resources are employed, the difference in average throughput is also minimal and vice versa.

The Isolation scheduler executes topologies in isolation on a dedicated cluster. The isolation scheduler generates different mappings as compared to the default scheduler. Instead of throughput, this scheduler focuses to improve maximum cluster utilization which increases inter-node traffic and adversely affects throughput. It has been shown (in Fig. 4.12 and Fig. 4.21) that for both topologies (Word Count Topology [101] and Exclamation Topology [14]) isolation scheduler performed worst for cluster configurations (see Table 4.2).

R-Storm is designed to increase throughput by maximizing resource utilization while minimizing network latency. R-Storm considers memory and CPU as well as minimizing the network distance between components that communicate with each other. R-Storm ignores the number of required slots provided by the user at the time of topology submission. Based on resources required by a topology, R-Storm distributes executors among different slots and then packs slots to nodes as compact as possible. If the desired number of slots is not available in the cluster, then R-Storm does not execute the topology. Similarly, the user cannot force R-Storm to use fewer/more slots than calculated by R-Storm. Hence, over-provisioning or under-utilization is a common scenario in this case. For example, we have 28 executors for word count topology. When R-Storm generates mapping then it divides them into 5 slots having 6 executors each. 2 extra executors are generated where each executor occupies 768 MB space in memory. That's why we have attained R-Storm's results for scenarios 4–6 (see Fig. 4.10) only for word count topology where 5 slots are used. With over-provisioning, R-Storm performs slightly better than the proposed schemes. The same is the case for exclamation

topology where we have a total of 18 executors. R-Storm divided them into 3 slots (6 executors each). That's why we attained R-Storm's results for scenario 1 only (see Fig. 4.19) only for exclamation topology where 3 slots are used. In this case, when equal resources are used then A3-Storm outperforms R-Storm by attaining 9% on average higher throughput.

Overall, these results demonstrate the ability of the proposed schedulers to efficiently place more communicating executors closer considering the topology's traffic resulting in overall higher tuple execution throughput. Additionally, the state-of-the-art schedulers (i.e., default and isolation) cannot also consolidate computational resources while occupying the resources even if they are not required.

### 4.5.1 Summary

From the results, it is obvious that topology mapping with respect to resources and communication patterns is a better scheduling choice. It performs better in terms of resource usage and throughput as compared to other scheduling algorithms. These results will be difficult to achieve if the available resources or communication patterns between executors are ignored.

# Chapter 5

# Conclusion and Future Work

This chapter concludes the thesis by summarizing the research findings pertaining to the proposed system and schedulers. Moreover, this Chapter provides an insight into promising open issues for future work in this area.

## 5.1 Research Findings

Through an extensive literature survey, it was identified that existing SPEs can benefit from improved scheduling techniques. The primary objective of this research was to propose a scheduling scheme for SPEs that achieves near-maximum throughput while employing a minimum number of resources.

In order to meet the research objective, a traffic-aware and resource-aware job scheduling system for SPE in a heterogeneous environment is proposed. Based on this system, four dynamic schedulers were presented to optimize task assignment and resource utilization by considering communication patterns and computational requirements. The first scheduler, **TOP-Storm**, calculates the longest path in the topology's DAG and assigns it to the most powerful computing machine. The second scheduler, **A3-Storm**, prioritizes tasks based on inter-task communication and assigns them to computing nodes accordingly. It compares unassigned tasks'

communication with already assigned tasks to make optimal assignments. Similarly, **BAN-Storm**, focuses on regularly communicating tasks and maps them closer to each other. This results in a smaller number of provisioned nodes in the cluster. Lastly, **Gr-Storm**, uses the max-flow min-cut algorithm to partition the graph and then intelligently assigns executors to workers based on computation power and communication needs. These schedulers aim to enhance throughput and resource utilization by optimizing task assignment in a heterogeneous cluster.

These proposed schedulers were evaluated using two benchmark topologies (word count topology and exclamation topology) with three state-of-the-art schedulers (default scheduler, R-Storm, and isolation scheduler). Results showed that the proposed schedulers outperformed the default scheduler in average throughput by 28% (which means that it can process 56 more tuples per second) with up to 40% fewer resources. Similarly, as compared to the isolation scheduler, improvement in throughput is up to 200% (which is 197 more tuples per second) with 60% of resources used. Moreover, R-Storm achieved the same throughput with over-provisioning by employing two extra slots for word count topology execution.

Experiments with the exclamation topology showed up to 23% (that is 17 additional tuples in a second) and 112% (50 extra tuples per second), enhanced throughput as compared to default and isolation scheduler. When equal resources are used then this work outperforms R-Storm by 4% (which is 4 tuples per second) on average throughput for exclamation topology and yields a significant amount of resource savings through consolidation. This accomplishment aligns with the main research objective of the study, which aimed to achieve near-maximum throughput in a way that a minimum number of resources are employed.

## 5.2 Future Directions

In this thesis, topology, traffic and resource-aware schedulers with enhanced throughput and improved resource utilization are presented. There are several potential

research directions which could be explored and investigated. These research directions are as follows:

- A data stream processing application can be represented as DAG. Therefore, different graph traversal algorithms for example, Breadth-First Search, Depth-First Search etc. have been used in the literature for the scheduling of streaming applications. Similarly, we have applied a graph-algorithm Max-flow Min-cut [53] in Gr-Storm. As a comparative analysis, topological sort can also be applied. Topological sort-based schedulers are already found in the literature [15, 103–105] as well but we want to investigate how differently our proposed scheduler adds value in this domain.

- Load-awareness is another important aspect to be considered while scheduling. Load-awareness means the actual workload on the computing node should be considered while assigning jobs instead of using a round-robin technique. In this way, resource under-utilization and over-utilization problems can also be addressed.

- SPEs work on the architecture of distributed computing. If a job is scheduled in such a way that some part is being deployed on one location and the remaining part on the other remote location, then latency may increase. Therefore to reduce latency, the location of each computing node can play an important role in real-time applications. That's why, the computing node's location should be considered while scheduling jobs. These types of schedulers are called location-aware schedulers. This can be a potential direction for the extension of this work.

- For any SPE, different types of jobs exist. Like memory-bound, cpu-bound, io-bound and network-bound etc. In this work, we have employed cpu-bound jobs. As an extension to this work, other job types can be tested to see the suitability of the proposed work. Similarly, in a cluster different machines are suitable for different types of jobs. Therefore, we want to propose a job type-aware scheduler in which computing nodes are selected according to the needs and want of the job.

- In this work, equations 3.2, and 3.3 used $\alpha$, $\beta$, and $\gamma$ with manually assigned values. In future, some machine learning algorithms can be explored to systematically assign values to $\alpha$, $\beta$, and $\gamma$ based on historical communication and computational requirement of the topology.

- Cloud computing is a flexible, low-cost, on-demand service platform used as an infrastructure. Due to a shared environment, service-level agreement (SLA) between customers and service providers is becoming an ingredient to measure customer satisfaction level (CSL), currently lacking in SPEs. To address this need, SLA-based scheduling can also be introduced in the proposed work as well.

- In this work, a physical cluster consisting of 11 machines has been employed for experimentations. Some online cloud services can also be used for performance comparison.

- The comparison done in this dissertation used two metrics: throughput and resource utilization. In future, an enhancement in the performance can be carried out for other metrics, such as complete latency, the amount of memory used etc.

- The system proposed (see Chapter 3) in this dissertation consists of 5 major modules, **Tune Scheduling** is one of them. It is a real-time monitoring module which measures how the streaming application performs under a real workload and make adjustment in the execution plan accordingly. It acquires the computation power of each node as well as the overall inter-executor traffic patterns. Considering the traffic patterns and the resources utilized, the execution plan is re-adjusted and fine-tuned accordingly (if required). To make more reasonable tunning, evolutionary computation models or machine learning models can be employed here as well.

- In this work, the experiments were conducted using two benchmark topologies (for example, Word Count Topology and Exclamation Topology etc.). To check the suitability and performance of the proposed work, other topologies can also be employed.

- The proposed algorithms are making scheduling decisions based on topology's DAG. As an extension of this work, it can be adapted for other SPEs as well such as Apache Spark, Apache Flink etc.

- This work aims to solve a multi-objective scheduling problem. Other objectives can also be considered, such as the energy consumption as well as reducing the cost of operators' deployment.

- Resource elasticity is an important concept in SPEs, as it affects the allocation of resources in a computing cluster. In general, resources with low elasticity are considered to be more critical, as they are less responsive to changes in the cluster. As a future work, the functionality to support elasticity in the proposed work can also be investigated as well.

- This research work employed Ubuntu for experimentation. However, the investigation of other operating systems in future studies can provide valuable insights by assessing their impact on overall performance.

- As a future research direction, it would be valuable to investigate how SPEs can leverage quantum computing to optimize complex tasks that involve handling large volumes of real-time data. Quantum computing's unique ability to perform parallel computations and explore multiple states simultaneously holds the potential to enhance the efficiency of processing algorithms in SPEs.

- **Gr-Storm** maps computational requirements of the topology on the available computation power of workers (JVM) in an optimal way to maximize through and resource utilization. A graph algorithm (max-flow min-cut) is used to produce a partitioned graph of executors. In future work, we intend to investigate the use of other graph partitioning algorithms to classify highly computational tasks in a DAG.

# Bibliography


[1] Rubén Casado and Muhammad Younas. Emerging trends and technologies in big data processing. *Concurrency and Computation: Practice and Experience*, 27(8):2078–2091, jun 2015. ISSN 15320634. doi: 10.1002/cpe.3398.

[2] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:154300–154316, 2019. ISSN 21693536. doi: 10.1109/ACCESS.2019.2946884.

[3] Teng Li, Jian Tang, and Jielong Xu. Performance Modeling and Predictive Scheduling for Distributed Stream Data Processing. *IEEE Transactions on Big Data*, 2(4):353–364, 2016. ISSN 2332-7790. doi: 10.1109/tbdata.2016.2616148.

[4] Apache Storm. Storm Documentation, 2014. URL https://storm.incubator.apache.org/documentation/Home.html. Accessed: 2020-03-19.

[5] Apache Software Foundation. Apache Spark™ - Unified Analytics Engine for Big Data, 2018. URL https://spark.apache.org/. Accessed: 2020-03-19.

[6] Apache. Apache Flink: Stateful Computations over Data Streams, 2022. URL https://flink.apache.org/. Accessed: 2020-04-22.

[7] Twitter. Apache Heron, 2014. URL https://apache.github.io/incubator-heron/. Accessed: 2020-04-22.

[8] Apache Software Foundation. Apache Samza project, 2017. URL http://samza.apache.org/. Accessed: 2020-03-19.

[9] Apache Sqoop - Apache Attic. URL https://attic.apache.org/projects/sqoop.html. Accessed: 2022-06-22.

[10] Xunyun Liu and Rajkumar Buyya. Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions. *ACM Computing Surveys*, 53(3):1–41, 2020. ISSN 15577341. doi: 10.1145/3355399.

[11] Supun Kamburugamuve and Geoffrey Fox. Survey of Distributed Stream Processing. *Bloomington: Indiana University*, (February), 2016. ISSN 0893-133X.

[12] Zbyněk Falt and Jakub Yaghob. Task scheduling in data stream processing. Technical report, 2011.

[13] Nicoleta Tantalaki, Stavros Souravlas, and Manos Roumeliotis. A review on big data real-time stream processing and its scheduling techniques. *International Journal of Parallel, Emergent and Distributed Systems*, 35(5): 571–601, 2020. ISSN 17445779. doi: 10.1080/17445760.2019.1585848.

[14] Github. Default Scheduler, jul 2019. URL https://github.com/apache/storm/blob/2.2.x-branch/storm-server/src/main/java/org/apache/storm/scheduler/DefaultScheduler.java. Accessed: 2020-03-19.

[15] Ali Al-Sinayyid and Michelle Zhu. Job scheduler for streaming applications in heterogeneous distributed processing systems. *Journal of Supercomputing*, 76(12):9609–9628, mar 2020. ISSN 15730484. doi: 10.1007/s11227-020-03223-z.

[16] Leila Eskandari, Jason Mair, Zhiyi Huang, and David Eyers. T3-Scheduler: A topology and Traffic aware two-level Scheduler for stream processing systems in a heterogeneous cluster. *Future Generation Computer Systems*, 89: 617–632, dec 2018. ISSN 0167739X. doi: 10.1016/j.future.2018.07.011.

[17] Jiahua Fan, Haopeng Chen, and Fei Hu. Adaptive task scheduling in storm. In *Proceedings of 2015 4th International Conference on Computer Science and Network Technology, ICCSNT 2015*, volume 1, pages 309–314. IEEE, 2016. ISBN 9781467381727. doi: 10.1109/ICCSNT.2015.7490758.

[18] Gerrit Janßen, Ilya Verbitskiy, Thomas Renner, and Lauritz Thamsen. Scheduling Stream Processing Tasks on Geo-Distributed Heterogeneous Resources. In *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*, pages 5159–5164. IEEE, 2019. ISBN 9781538650356. doi: 10.1109/BigData.2018.8622651.

[19] Xunyun Liu and Rajkumar Buyya. D-Storm: Dynamic resource-efficient scheduling of stream processing applications. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, volume 2017-Decem, pages 485–492. IEEE, 2018. ISBN 9781538621295. doi: 10.1109/ICPADS.2017.00070.

[20] Asif Muhammad and Muhammad Aleem. A3-Storm: topology-, traffic-, and resource-aware storm scheduler for heterogeneous clusters. *Journal of Supercomputing*, 77(2):1059–1093, may 2021. ISSN 15730484. doi: 10.1007/s11227-020-03289-9.

[21] Asif Muhammad, Muhammad Aleem, and Muhammad Arshad Islam. TOP-Storm: A topology-based resource-aware scheduler for Stream Processing Engine. *Cluster Computing*, 24(1):417–431, may 2021. ISSN 15737543. doi: 10.1007/s10586-020-03117-y.

[22] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Middleware 2015 - Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015. ISBN 9781450336185. doi: 10.1145/2814576.2814808.

[23] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. SpanEdge: Towards unifying stream processing over central

and near-the-edge data centers. In *Proceedings - 1st IEEE/ACM Symposium on Edge Computing, SEC 2016*, pages 168–178. IEEE, 2016. ISBN 9781509033218. doi: 10.1109/SEC.2016.17.

[24] Pavel Smirnov, Mikhail Melnik, and Denis Nasonov. Performance-aware scheduling of streaming applications using genetic algorithm. *Procedia Computer Science*, 108:2240–2249, 2017. ISSN 18770509. doi: 10.1016/j.procs.2017.05.249.

[25] Zujian Weng, Qi Guo, Chunkai Wang, Xiaofeng Meng, and Bingsheng He. AdaStorm: Resource efficient storm with adaptive configuration. In *Proceedings - International Conference on Data Engineering*, pages 1363–1364. IEEE, 2017. ISBN 9781509065431. doi: 10.1109/ICDE.2017.178.

[26] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. When computing meets heterogeneous cluster: Workload assignment in graph computation. *Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015*, pages 154–163, 2015. doi: 10.1109/BigData.2015.7363752.

[27] Altaf Hussain, Muhammad Aleem, Abid Khan, Muhammad Azhar Iqbal, and Muhammad Arshad Islam. RALBA: a computation-aware load balancing scheduler for cloud computing. *Cluster Computing*, 21(3):1667–1680, sep 2018. ISSN 15737543. doi: 10.1007/s10586-018-2414-6.

[28] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. *DEBS 2013 - Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, pages 207–218, 2013. doi: 10.1145/2488222.2488267.

[29] Leila Eskandari, Zhiyi Huang, and David Eyers. P-scheduler: Adaptive hierarchical scheduling in Apache Storm. In *ACM International Conference Proceeding Series*, volume 01-05-Febr, pages 1–10. ACM, 2016. ISBN 9781450340427. doi: 10.1145/2843043.2843056.

[30] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. Model-free control for distributed stream data processing using deep reinforcement learning.

*Proceedings of the VLDB Endowment*, 11(6):705–718, 2018. ISSN 21508097. doi: 10.14778/3184470.3184474.

[31] Chunlin Li and Jing Zhang. Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of storm. *Journal of Network and Computer Applications*, 87:100–115, jun 2017. ISSN 10958592. doi: 10.1016/j.jnca.2017.03.007.

[32] Hamid Nasiri, Saeed Nasehi, Arman Divband, and Maziar Goudarzi. A scheduling algorithm to maximize storm throughput in heterogeneous cluster. *Journal of Big Data 2023 10:1*, 10(1):1–27, jun 2023. ISSN 2196-1115. doi: 10.1186/S40537-023-00771-Y.

[33] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *Proceedings - International Conference on Distributed Computing Systems*, pages 535–544. IEEE, 2014. ISBN 9781479951680. doi: 10.1109/ICDCS.2014.61.

[34] Leila Eskandari, Jason Mair, Zhiyi Huang, and David Eyers. Poster: Iterative scheduling for distributed stream processing systems. In *DEBS 2018 - Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, pages 234–237, New York, New York, USA, 2018. ACM, ACM Press. ISBN 9781450357821. doi: 10.1145/3210284.3219768.

[35] Lorenz Fischer and Abraham Bernstein. Workload scheduling in distributed stream processors using graph partitioning. *Proceedings - 2015 IEEE International Conference on Big Data, IEEE Big Data 2015*, pages 124–133, 2015. doi: 10.1109/BigData.2015.7363749.

[36] Javad Ghaderi, Sanjay Shakkottai, and R. Srikant. Scheduling storms and streams in the cloud. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 1(4), sep 2016. ISSN 23763647. doi: 10.1145/2904080.

[37] Saima Gulzar Ahmad, Hikmat Ullah Khan, Samia Ijaz, and Ehsan Ullah Munir. Use case-based evaluation of workflow optimization strategy in real-time computation system. *Journal of Supercomputing*, 76(1):708–725, jan 2020. ISSN 15730484. doi: 10.1007/s11227-019-03060-9.

[38] Dawei Sun, Guangyan Zhang, Songlin Yang, Weimin Zheng, Samee U. Khan, and Keqin Li. Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. *Information Sciences*, 319:92–112, 2015. ISSN 00200255. doi: 10.1016/j.ins.2015.03.027.

[39] PU Lohnstein, J Schipper, Ansgar Berlis, and W Maier. Sonographisch unterstützte computerassistierte chirurgie (sacas) in der orbitachirurgie. *HNO*, 55(10):778–784, 2007.

[40] Martin Illecker. Real-time Twitter Sentiment Classification based on Apache Storm, 2015. URL https://dbis-informatik.uibk.ac.at/real-time-twitter-sentiment-classification-based-apache-storm. Accessed: 2020-03-19.

[41] From Wikipedia. Apache Hadoop, 2019. URL https://hadoop.apache.org/. Accessed: 2021-04-01.

[42] Inc. Apache Foundation. Apache Hadoop Wiki, 2013. URL http://wiki.apache.org/hadoop/. Accessed: 2020-03-19.

[43] Introduction to Apache Storm | Apache Storm Tutorials, . URL https://www.allprogrammingtutorials.com/tutorials/introduction-to-apache-storm.php. Accessed: 2023-02-07.

[44] Apache Zookeeper. Apache ZooKeeper - Home, 2016. URL https://zookeeper.apache.org/.

[45] An Introduction to Apache Storm Architecture, Use Cases, Pros & Cons, . URL https://phoenixnap.com/kb/apache-storm. Accessed: 2023-06-22.

[46] Enes Şanlıtürk, Ahmet Tezcan Tekin, and Ferhan Çebi. Defect detection in manufacturing via machine learning algorithms. In *Encyclopedia of Data Science and Machine Learning*, pages 212–224. IGI Global, 2023.

[47] Qifeng Lu, Yinchao Zhao, Long Huang, Jiabao An, Yufan Zheng, and Eng Hwa Yap. Low-dimensional-materials-based flexible artificial synapse: Materials, devices, and systems. *Nanomaterials*, 13(3):373, 2023.

[48] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3):45–77, dec 2007. ISSN 07421222. doi: 10.2753/MIS0742-1222240302.

[49] Aline Dresch, Daniel Pacheco Lacerda, and José Antônio Valle Antunes. Design science—the science of the artificial. In *Design Science Research*, pages 47–65. Springer, 2015.

[50] Aline Dresch, Daniel Pacheco Lacerda, and José Antônio Valle Antunes. Design science research. In *Design science research*, pages 67–102. Springer, 2015.

[51] Alan Hevner and Samir Chatterjee. Design Science Research in Information Systems. *Springer*, pages 9–22, 2010. doi: 10.1007/978-1-4419-5653-8_2.

[52] Aline Dresch, Daniel Pacheco Lacerda, and José Antônio Valle Antunes. Proposal for the Conduct of Design Science Research. *Design Science Research*, pages 117–127, 2015. doi: 10.1007/978-3-319-07374-3_6.

[53] Max-flow min-cut theorem - Wikipedia. URL https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem. Accessed: 2020-04-29.

[54] Arwa Z. Selim, I. M. Hanafy, Noha E. El-Attar, and Wael A. Awad. Balanced Schedule on Storm for Performance Enhancement. *International Journal of Advanced Computer Science and Applications*, 13(1):622–632, 2022. ISSN 21565570. doi: 10.14569/IJACSA.2022.0130175.

[55] Jing Zhang, Chunlin Li, Liye Zhu, and Yanpei Liu. The Real-Time Scheduling Strategy Based on Traffic and Load Balancing in Storm. In *Proceedings - 18th IEEE International Conference on High Performance Computing and Communications, 14th IEEE International Conference on Smart City and 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016*, pages 372–379. IEEE, 2017. ISBN 9781509042968. doi: 10.1109/HPCC-SmartCity-DSS.2016.0060.

[56] Kasper Grud Skat Madsen and Yongluan Zhou. Dynamic resource management in a Massively Parallel Stream Processing Engine. In *International Conference on Information and Knowledge Management, Proceedings*, volume 19-23-Oct-, pages 13–22. ACM, 2015. ISBN 9781450337946. doi: 10.1145/2806416.2806449.

[57] Xunyun Liu, Aaron Harwood, Shanika Karunasekera, Benjamin Rubinstein, and Rajkumar Buyya. E-Storm: Replication-Based State Management in Distributed Stream Processing Systems. Technical report, 2017.

[58] Wenjun Qian, Qingni Shen, Jia Qin, Dong Yang, Yahui Yang, and Zhonghai Wu. S-Storm: A Slot-Aware Scheduling Strategy for even Scheduler in Storm. *Proceedings - 18th IEEE International Conference on High Performance Computing and Communications, 14th IEEE International Conference on Smart City and 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016*, pages 623–630, 2017. doi: 10.1109/HPCC-SmartCity-DSS.2016.0093.

[59] D Sun, Y Wang, J Sui, J Rong, and S Gao Available at SSRN …. Lc-Stream: An Elastic Scheduling Strategy with Latency Constrained in Geo-Distributed Stream Computing Environments. *papers.ssrn.com*. URL https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4350234.

[60] Mohammadreza Farrokh, Hamid Hadian, Mohsen Sharifi, and Ali Jafari. SP-ant: An ant colony optimization based operator scheduler for high performance distributed stream processing on heterogeneous clusters. *Expert*

*Systems with Applications*, 191, 2022. ISSN 09574174. doi: 10.1016/j.eswa.2021.116322.

[61] Yangyang Liu. Energy Usage Proling and Topology-Based Scheduling for Clusters. Technical report, 2017. URL http://etd.auburn.edu/handle/10415/6022.

[62] Hamid Hadian, Mohammadreza Farrokh, Mohsen Sharifi, and Ali Jafari. An elastic and traffic-aware scheduler for distributed data stream processing in heterogeneous clusters. *Journal of Supercomputing*, 79(1):461–498, jan 2023. ISSN 15730484. doi: 10.1007/s11227-022-04669-z.

[63] Rizwan Ali, Asif Muhammad, Muhammad Aleem, Omair Shaaq, and Omair Shafiq. WG-Storm Scheduler for Distributed Stream Processing Engines. jun 2023. doi: 10.21203/RS.3.RS-2985470/V1. URL https://www.researchsquare.com/article/rs-2985470/v1.

[64] Mansheng Yang and Richard T.B. Ma. Smooth task migration in Apache Storm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 2015-May, pages 2067–2068. ACM, 2015. ISBN 9781450327589. doi: 10.1145/2723372.2764941.

[65] Jan Sipke Van Der Veen, Bram Van Der Waaij, Elena Lazovik, Wilco Wijbrandi, and Robert J. Meijer. Dynamically scaling apache storm for the analysis of streaming data. In *Proceedings - 2015 IEEE 1st International Conference on Big Data Computing Service and Applications, BigDataService 2015*, pages 154–161. IEEE, 2015. ISBN 9781479981281. doi: 10.1109/BigDataService.2015.56.

[66] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency Computation*, 30(9), may 2018. ISSN 15320634. doi: 10.1002/cpe.4334.

[67] Zhou Zhang, Peiquan Jin, Xiaoliang Wang, Ruicheng Liu, and Shouhong Wan. N-storm: Efficient thread-level task migration in apache storm. In

*2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1595–1602. IEEE, 2019.

[68] Dawei Sun, Yijing Cui, Minghui Wu, Shang Gao, and Rajkumar Buyya. An energy efficient and runtime-aware framework for distributed stream computing systems. *Future Generation Computer Systems*, 136:252–269, nov 2022. ISSN 0167739X. doi: 10.1016/j.future.2022.06.007.

[69] Xiaojun Cai, Feng Li, Ping Li, Lei Ju, and Zhiping Jia. SLA-aware energy-efficient scheduling scheme for Hadoop YARN. *Journal of Supercomputing*, 73(8):3526–3546, aug 2017. ISSN 15730484. doi: 10.1007/s11227-016-1653-7.

[70] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. Cost-efficient enactment of stream processing topologies. *PeerJ Computer Science*, 2017(12), 2017. ISSN 23765992. doi: 10.7717/peerj-cs.141.

[71] Muhammad Hanif, Eunsam Kim, Sumi Helal, and Choonhwa Lee. SLA-based adaptation schemes in distributed stream processing engines. *Applied Sciences (Switzerland)*, 9(6):1045, mar 2019. ISSN 20763417. doi: 10.3390/app9061045.

[72] Kaippilly Raman Remesh Babu and Philip Samuel. Service-level agreement–aware scheduling and load balancing of tasks in cloud. *Software - Practice and Experience*, 49(6):995–1012, jun 2019. ISSN 1097024X. doi: 10.1002/spe.2692.

[73] Altaf Hussain, Muhammad Aleem, Muhammad Azhar Iqbal, and Muhammad Arshad Islam. SLA-RALBA: cost-efficient and resource-aware load balancing algorithm for cloud computing. *Journal of Supercomputing*, 75 (10):6777–6803, 2019. ISSN 15730484. doi: 10.1007/s11227-019-02916-4.

[74] Ahmet Cihat Baktır, Betül Ahat, Necati Aras, Atay Özgövde, and Cem Ersoy. Sla-aware optimal resource allocation for service-oriented networks. *Future Generation Computer Systems*, 101:959–974, 2019.

[75] Hongjian Li, Yuyan Zhao, and Shuyong Fang. CSL-driven and energy-efficient resource scheduling in cloud data center. *Journal of Supercomputing*, 76(1):481–498, jan 2020. ISSN 15730484. doi: 10.1007/s11227-019-03036-9.

[76] Souad Hadjres, Nadjia Kara, May El Barachi, and Fatna Belqasmi. An SLA-aware cloud coalition formation approach for virtualized networks. *IEEE Transactions on Cloud Computing*, 9(2):475–491, 2021. ISSN 21687161. doi: 10.1109/TCC.2018.2865737.

[77] Hongjian Li, Hongxi Dai, Zengyan Liu, Hao Fu, and Yang Zou. Dynamic energy-efficient scheduling for streaming applications in storm. *Computing*, 104(2):413–432, feb 2022. ISSN 14365057. doi: 10.1007/s00607-021-00961-7.

[78] Jingyun Shen, Zheng Luo, Chentao Wu, and Jie Li. BAHS: A bandwidth-aware heterogeneous scheduling approach for SDN-Based cluster systems. In *Proceedings - 15th IEEE International Symposium on Parallel and Distributed Processing with Applications and 16th IEEE International Conference on Ubiquitous Computing and Communications, ISPA/IUCC 2017*, pages 638–645. IEEE, 2018. ISBN 9781538637906. doi: 10.1109/ISPA/IUCC.2017.00101.

[79] Mahendra Bhatu Gawali and Subhash K. Shinde. Task scheduling and resource allocation in cloud computing using a heuristic approach. *Journal of Cloud Computing*, 7(1), dec 2018. ISSN 2192113X. doi: 10.1186/s13677-018-0105-8.

[80] Mahmood Mortazavi-Dehkordi and Kamran Zamanifar. Efficient resource scheduling for the analysis of Big Data streams. *Intelligent Data Analysis*, 23(1):77–102, 2019. ISSN 15714128. doi: 10.3233/IDA-173691.

[81] Stavros Souravlas and Sofia Anastasiadou. Pipelined dynamic scheduling of big data streams. *Applied Sciences (Switzerland)*, 10(14):4796, jul 2020. ISSN 20763417. doi: 10.3390/app10144796.

[82] Jinlai Xu, Balaji Palanisamy, Qingyang Wang, Heiko Ludwig, and Sandeep Gopisetty. Amnis: Optimized stream processing for edge computing. *Journal of Parallel and Distributed Computing*, 160:49–64, 2022.

[83] Tianyu Qi and Maria Rodriguez. A Traffic and Resource Aware Online Storm Scheduler. *ACM International Conference Proceeding Series*, feb 2021. doi: 10.1145/3437378.3444365.

[84] Dawei Sun, Hanyu He, Hongbin Yan, Shang Gao, Xunyun Liu, and Xinqi Zheng. Lr-Stream: Using latency and resource aware scheduling to improve latency and throughput for streaming applications. *Future Generation Computer Systems*, 114:243–258, 2021. ISSN 0167739X. doi: 10.1016/j.future.2020.08.003.

[85] Roman Heinrich, Manisha Luthra, Harald Kornmayer, and Carsten Binnig. Zero-shot cost models for distributed stream processing. *DEBS 2022 - Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, pages 85–90, jun 2022. doi: 10.1145/3524860.3539639.

[86] Qihang Wang, Decheng Zuo, Zhan Zhang, Siyuan Chen, and Tianming Liu. An adaptive non-migrating load-balanced distributed stream window join system. *Journal of Supercomputing*, 79(8):8236–8264, may 2022. ISSN 15730484. doi: 10.1007/s11227-022-04991-6.

[87] Josip Marić, Krešimir Pripužić, Martina Antonić, and Dejan Škvorc. Dynamic Load Balancing in Stream Processing Pipelines Containing Stream-Static Joins. *Electronics (Switzerland)*, 12(7):1613, mar 2023. ISSN 20799292. doi: 10.3390/electronics12071613.

[88] Shun Wang and Guo sun Zeng. Two-stage scheduling for a fluctuant big data stream on heterogeneous servers with multicores in a data center. *Cluster Computing*, pages 1–17, may 2023. ISSN 15737543. doi: 10.1007/s10586-023-04044-4.

[89] Janet Light. Energy usage profiling for green computing. *Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2017*, 2017-Janua:1287–1291, 2017. doi: 10.1109/CCAA.2017.8230017.

[90] NetWell Noise Control. Understanding Hertz, 2020. URL https://www.controlnoise.com/support-tools/about-sound-waves/understanding-hertz/. Accessed: 2023-06-22.

[91] Yasir Noman Khalid, Muhammad Aleem, Radu Prodan, Muhammad Azhar Iqbal, and Muhammad Arshad Islam. E-OSched: a load balancing scheduler for heterogeneous multicores. *Journal of Supercomputing*, 74(10):5399–5431, oct 2018. ISSN 15730484. doi: 10.1007/s11227-018-2435-1.

[92] Romain Dolbeau. Theoretical peak FLOPS per instruction set: a tutorial. *Journal of Supercomputing*, 74(3):1341–1377, mar 2018. ISSN 15730484. doi: 10.1007/s11227-017-2177-5.

[93] FLOPS (Floating Point Operations Per Second) Definition. URL https://techterms.com/definition/flops. Accessed: 2020-03-19.

[94] FLOPS - Wikipedia. URL https://en.wikipedia.org/wiki/FLOPS. Accessed: 2020-03-19.

[95] Usman Ahmed, Muhammad Aleem, Yasir Noman Khalid, Muhammad Arshad Islam, and Muhammad Azhar Iqbal. RALB-HC: A resource-aware load balancer for heterogeneous cluster. *Concurrency and Computation: Practice and Experience*, 33(14):e5606, 2021. ISSN 15320634. doi: 10.1002/cpe.5606.

[96] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016. ISSN 15577317. doi: 10.1145/2934664.

[97] Apache. Apache Flink: Stateful Computations over Data Streams, 2022. URL https://flink.apache.org/. Accessed: 2021-04-01.

[98] Apache Storm Scheduler. URL http://storm.apache.org/releases/2.0.0/Storm-Scheduler.html. Accessed: 2020-07-15.

[99] GitHub. Storm Isolation scheduler, 2014. URL https://storm.incubator.apache.org/2013/01/11/storm082-released.html. Accessed: 2020-03-19.

[100] Example Storm Topologies. URL https://github.com/apache/storm/tree/master/examples/storm-starter. Accessed: 2020-10-23.

[101] Storm Topology Explained using Word Count Topology Example | CoreJavaGuru. URL http://www.corejavaguru.com/bigdata/storm/word-count-topology. Accessed: 2020-03-19.

[102] Sandeep Karanth. Mastering Hadoop, 2014. URL https://books.google.com.pk/books?id=IdEGBgAAQBAJ&pg=PT374&lpg=PT374&dq=Exclamation+Topology&source=bl&ots=lVJVSSfjyR&sig=ACfU3U3AEcjALh6n1V8XaQqheeZOteZ_IQ&hl=en&sa=X&ved=2ahUKEwiolfDIyKHoAhUvSRUIHRyQAQwQ6AEwCXoECC8QAQ#v=onepage&q=ExclamationTopology&f=. Accessed: 2020-03-19.

[103] P. Rizwan and M. RajasekharaBabu. Performance improvement of Data analysis of IoT applications using restorm in big data stream computing platform. *International Journal of Engineering Research in Africa*, 22:141–151, 2016. ISSN 16634144. doi: 10.4028/www.scientific.net/JERA.22.141.

[104] Rizwan Patan and M. Rajasekhara Babu. Re-storm: Real-time energy efficient data analysis adapting storm platform. *Jurnal Teknologi*, 78(10): 139–146, 2016. ISSN 01279696. doi: 10.11113/jt.v78.7672.

[105] Oleksandr Semeniuta and Petter Falkman. EPypes: A framework for building event-driven data processing pipelines. *PeerJ Computer Science*, 2019 (2), 2019. ISSN 23765992. doi: 10.7717/peerj-cs.176.