ATTACK-RESILIENT ADAPTIVE LOAD-BALANCING IN DISTRIBUTED SPATIAL DATA STREAMING SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Anas Daghistani

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Walid G. Aref, Co-Chair

Computer Science Department

Dr. Arif Ghafoor, Co-Chair

School of Electrical and Computer Engineering

Dr. Charlie Hu

School of Electrical and Computer Engineering

Dr. Sunil Prabhakar

Computer Science Department

Approved by:

Dr. Dimitrios Peroulis

Head of the School Graduate Program

To my parents Hazim and Dunya, for their continuous love and sacrifices.

To my wife Nada Alnoory, for her love and support.

ACKNOWLEDGMENTS

All praise and thanks are due to the Almighty Allah who always guides me to the right path and has helped me to complete this dissertation.

I am extremely grateful to my advisor, Prof. Walid Aref, for the continuous support, enthusiasm, and guidance during my PhD. journey. His experience and our discussions have developed my research ideas that led to this dissertation. I would like to express my sincere gratitude to my advisor Prof. Arif Ghafoor, for his continuous encouragement, recommendations and valuable advice. I would also like to thank the rest of my committee members: Prof. Charlie Hu and Prof. Sunil Prabhakar, for serving in my committee and their insightful comments.

My sincere thanks to my parents, Hazim Daghistani and Dunya Daghistani, for their utmost love, support, and warm prayers. No words can express my gratitude to my parents. Also, I would like to thank my brother Mustafa, and my sister Sawsan, for their continuous love and encouragement. My deepest thanks to my beloved wife, Nada Alnoory. Your continuous love, encouragement, and support kept me know that there is a light at the end of the tunnel. Thank you for your patience and help during my PhD journey.

Also, I thank my fellow labmates and friends for the stimulating discussions and the helpful support especially: Mosab Khayat, Muhamad Felemban, Ahmed Mahmood, Ahmed Abdelhamid, Abdulellah Alsaheel, Thaimer Qadah, Albraa Alsaati, Yahya Javed, Nader Alawadhi, Amgad Madkour and Tawfeeq Shawli. I would like to extend my deepest gratitude to Prof. Saleh Basalamah for his friendship and guidance.

I would like to thank Umm Al-Qura University and the Government of Saudi Arabia for giving me the opportunity to continue my studies with sponsoring my scholarship.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT xi

1 INTRODUCTION 1

1.1 Organization 7

2 RELATED WORK 8

2.1 Spatial Data Streaming Systems 8

2.2 Adaptive Distributed Processing Systems 9

2.3 Workload Estimations in Distributed Processing Systems 13

2.4 Security in Distributed Streaming Systems 14

2.5 Unsupervised Machine Learning 16

3 TrioStat: ONLINE WORKLOAD ESTIMATION IN DISTRIBUTED SPATIAL DATA STREAMING SYSTEMS 19

3.1 The Cost Model 19

3.2 Collecting and Maintaining Statistics 21

3.2.1 Required Statistics 22

3.2.2 Maintaining the Statistics 24

3.2.3 Correctness of the Statistics 27

3.3 Estimating the Workload 33

4 SWARM: ADAPTIVE LOAD BALANCING IN DISTRIBUTED SPATIAL DATA STREAMING SYSTEMS 36

4.1 SWARM Architecture 36

4.2 Indexing and Initialization 39

4.2.1 The Global Index 39

Page

4.2.2 The Local Index 42
4.3 Decentralised Adaptive Load Balancing Protocol 44
4.4 Workload Reduction by Repartitioning 48
4.4.1 Searching for the Best Subset of Partitions to Move 48
4.4.2 Searching for Best Split for a Partition to Move 49
4.5 Preserving System Integrity 51
4.5.1 Correctness During Load Balancing 51
4.5.2 Correctness of Query Execution 52
5 GUARD: DETECTION AND RESPONSE FOR ATTACKS TARGETING ADAPTIVE LOAD BALANCING IN DISTRIBUTED STREAMING SYSTEMS 55
5.1 Malicious Attacks on Adaptive Load Balancing in Distributed Streaming Systems 55
5.1.1 The Attack Model 56
5.2 Guard Architecture 59
5.3 Real-Time Feature Engineering 61
5.3.1 Collecting Raw Information about Hotspots 65
5.4 Unsupervised Attack Detector 66
5.5 Response to Malicious Users 72
6 EXPERIMENTS 73
6.1 Experimental Setup 73
6.1.1 Application and Dataset 73
6.1.2 Cluster Setup 74
6.2 Performance of the Online Workload Estimations 75
6.3 Performance of the Adaptive Load Balancing 80
6.3.1 Capability and Execution Latency 80
6.3.2 Reaction to Hotspots 82
6.3.3 Overhead of the Adaptive Load Balancing 87

Page

6.4 Performance of Detecting and Blocking Attacks Targeting Adaptive Load Balancing Mechanisms 90
6.4.1 Attack Effect on Throughput 91
6.4.2 Attack Effect on Availability 96
6.4.3 Detection and Recovery 98
6.4.4 Overhead of Attack Detection 101
7 CONCLUSION AND FUTURE WORK 103
7.1 Summary of Contribution 103
7.2 Future Work 105
REFERENCES 107
VITA 114

LIST OF TABLES

Table Page

5.1 Description of the features used by Guard 62

LIST OF FIGURES

Figure Page

1.1 Heatmap of tweets during different times 2

1.2 The effect of an attack targeting the adaptive load-balancing mechanism of a distributed streaming system 4

1.3 Abstract design of an attack-resilient adaptive load-balancing mechanism for distributed spatial streaming systems 6

2.1 Adaptive load-balancing in a spatial distributed streaming system 12

3.1 Example for dividing the space into a grid of small cells 22

3.2 TrioStat statistics for partition p_{11} 23

3.3 Updating partition p_{11}'s *Statistics Collectors* 26

3.4 Partition with k cells (rows) 28

3.5 Estimating the workload when partition p_{11} split into two sub-partitions p_a and p_b 34

4.1 The architecture of SWARM 38

4.2 SWARM's index for GlobalIndex machines 40

4.3 SWARM with the HashMap of m_1 and m_5 43

4.4 Workflow of load balancing decision 45

4.5 Decision mechanism for load balancing 45

4.6 Workflow of rebalancing 47

5.1 Example of attack model on a distributed streaming system that contains three machines. 57

5.2 Architecture of Guard and its connections with the distributed Streaming System 59

5.3 Flow chart for Guard's unsupervised attack detection mechanism 68

6.1 Overhead of TrioStat in executor machines 75

6.2 Network overhead of TrioStat statistics 76

Figure Page

6.3 Total Storage for the statistics while varying the number of partitions . . . 77
6.4 Total Storage for the statistics while varying the grid size 79
6.5 Capability and execution latency 81
6.6 Uniform distribution hotspot with normal distribution data intensity . . . 82
6.7 Normal distribution hotspot with normal distribution data intensity 84
6.8 Uniform distribution hotspot with step data intensity 84
6.9 Two overlapping hotspots (H1 and H2) in different locations 85
6.10 Two consecutive hotspots (H1 and H2) in different locations 86
6.11 CPU and network utilization 86
6.12 Overhead of SWARM operations in GlobalIndex machines 87
6.13 Overhead of SWARM operations in executor machines 88
6.14 The effect of an attack with malicious activity rate 150 queries/second on the system's throughput 91
6.15 The effect of an attack with malicious activity rate 300 queries/second on the system's throughput 92
6.16 The effect of an attack with malicious activity rate 600 queries/second on the system's throughput 93
6.17 The effect of an attack with malicious activity rate 1200 queries/second on the system's throughput 94
6.18 Average system's throughput while varying the additional activity 95
6.19 Availability of the system while varying malicious activity 97
6.20 Average detection and blocking times while varying malicious activity . . . 98
6.21 Average detection and blocking times while varying the number of malicious users involved in the attack 99
6.22 Average recovery time after blocking the attack 100
6.23 Overhead of Guard's operations 101

ABSTRACT

Daghistani, Anas Ph.D., Purdue University, August 2020. Attack-Resilient Adaptive Load-Balancing in Distributed Spatial Data Streaming Systems. Major Professors: Walid G. Aref, Arif Ghafoor.

The proliferation of GPS-enabled devices has led to the development of numerous location-based services. These services need to process massive amounts of spatial data in real-time with high-throughput and low response time. The current scale of spatial data cannot be handled using centralized systems. This has led to the development of distributed spatial streaming systems. The performance of distributed streaming systems relies on how even the workload is distributed among their machines. However, the real-time streamed spatial data and query follow non-uniform spatial distributions that are continuously changing over time. Therefore, Distributed spatial streaming systems need to track the changes in the distribution of spatial data and queries and redistribute their workload accordingly. This thesis addresses the challenges of adapting to workload changes in distributed spatial streaming systems to improve the performance while preserving the system's security.

The thesis proposes *TrioStat*, an online workload estimation technique that relies on a probabilistic model for estimating the cost of partitions and machines of distributed spatial streaming systems. TrioStat has a decentralised technique to collect and maintain the required statistics in real-time with minimal overhead. In addition, this thesis introduces *SWARM*, a light-weight adaptive load-balancing protocol that continuously monitors the data and query workloads across the distributed processes of spatial data streaming systems, and redistribute the workloads soon as performance bottlenecks get detected. SWARM uses TrioStat to estimate the workload of the system's machines. Although using adaptive load-balancing techniques signif-

icantly improves the performance of distributed streaming systems, they make the system vulnerable to attacks. In this thesis, we introduce a novel attack model that targets adaptive load-balancing mechanisms of distributed streaming systems. The attack reduces the throughput and the availability of the system by making it stay in a continuous state of rebalancing. The thesis proposes *Guard*, a component that detects and blocks attacks that target the adaptive load balancing of distributed streaming systems. Guard is deployed in SWARM to develop an attack-resilient adaptive load balancing mechanism for Distributed spatial streaming systems.

1. INTRODUCTION

The recent growth in spatial data has been phenomenal due to the proliferation of GPS-enabled devices, e.g., smartphones, smart watches, health monitors, and connected vehicles. Also, social networks generate huge deluge of spatial data, e.g., 500 million tweets are created daily, and they can be geotagged [1]. This growth leads to the development of location-based services, e.g., Internet search engines that return results based on user location, self-driving cars, video games (e.g., Pokemon GO), and ride-sharing services. Five billion Google search queries are generated every day [1]. Supporting these services places a huge demand on developing real-time, efficient, and scalable systems for processing location-based queries. Therefore, there is a growing demand to develop new systems that are optimized to process big spatial data instead of using general-purpose systems that are not tunable for the needs of spatial data [2].

Distributed data streaming systems have the potential to provide real-time scalable solutions. There is an increasing number of spatial applications that are being implemented using these systems. Examples include Storm [3], Twitter Heron [4], and SparkStreaming [5]. Spatial applications require extending the capabilities of general distributed data streaming systems to support spatial operations and spatial query processing. In particular, spatial partitioning and indexing techniques are needed to support efficient processing of spatial data [6–14]. Distributed spatial streaming systems distribute the workload across machines by making each machine responsible for some data partitions. The partitions are generated by dividing the underlying space into spatial rectangles. Data points and queries are directed according to their locations to the machines that handle the overlapping partitions.

Motivation A key challenge to improve the performance of a distributed system is to ensure workload balancing across its machines. However, the workload can change rapidly in spatial data applications. The challenge in load balancing stems from the

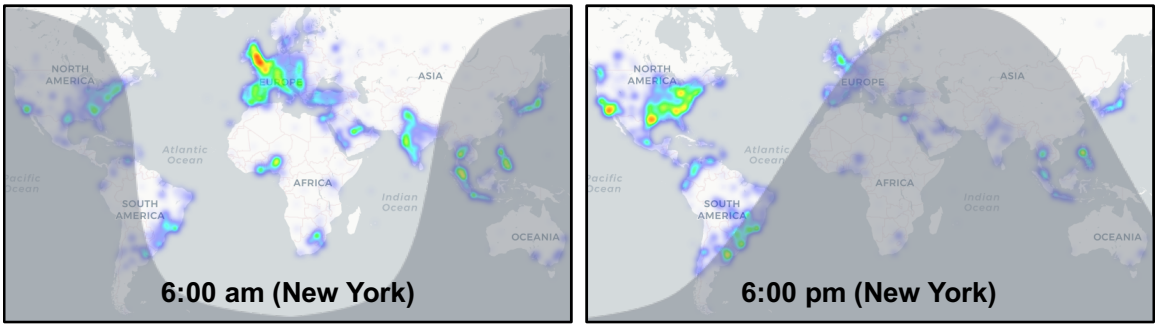

Fig. 1.1. Heatmap of tweets during different times

fact that the spatial distributions of data and queries are skewed, and this skewness changes with time and with users' interests. For example, different time-zones can lead to significant changes in the spatial distribution of the data being generated throughout the day. In addition, a major event in a specific location can also lead to more generation of new data and queries in the partitions that overlap this event. Figure 1.1 illustrates heatmaps of tweets generated during one hour at various times of the day. Notice that at 6AM (EDT), Europe and Asia are more active than the Americas. The opposite happens at 6PM (EDT). Moreover, events related to sports, politics, natural disaster, etc., can cause a huge change in the distribution of tweets and queries generated as a large number of users get interested in these events.

Most of the existing cluster-based data streaming systems use static data-partitioning schemes to distribute the workload among machines. For this purpose, a limited history of the collected data is used. However, static data-partitioning schemes are not effective in spatial applications due to the rapid changes in spatial data and query distributions as mentioned above. Distributed spatial streaming systems often do not maintain a global system workload state because the data and queries are distributed across their machines. Hence, it is challenging to estimate the workload of spatial distributed streaming systems. Existing systems, e.g., [15, 16] address the issue of distributing the workload by using adaptive mechanisms to update the data-partitioning plan as the workload changes. These systems use a centralized approach

to keep statistics and decisions about changing the plan. Deploying a new partitioning plan requires temporary halting the query processor until repartitioning takes place. Therefore, the solutions in [15, 16] are not viable for distributed data streaming systems because streams cannot be stopped until repartitioning takes place, and processing should happen in real-time. Moreover, collecting and maintaining statistics about the workload of every machine in a centralized machine introduces high network, storage, and processing overheads.

The main objective of this dissertation is to introduce an efficient and secure adaptive load-balancing mechanism for distributed spatial streaming systems. This mechanism dynamically re-balances the workload among the system's machines to improve its throughput, and execution latency, while maintaining low overhead and being resilient to attacks. The success of an adaptive load-balancing mechanism relies on the accuracy and speed of estimating the workloads of the system's machines. **This dissertation introduces *TrioStat* to address the challenges of workload estimations in distributed spatial streaming systems.** TrioStat is an online workload estimation technique that relies on a probabilistic model to estimate the workload of partitions and machines in a distributed spatial data streaming system. TrioStat introduces a new statistics structure that requires minimal storage overhead. TrioStat uses a decentralized technique to collect and maintain the required statistics in real-time locally in each machine. Thus, TrioStat introduces negligible network overhead. Moreover, TrioStat has an efficient algorithm to collect the statistics with very localized overhead to process every newly received data point or query. TrioStat enables distributed spatial data streaming systems to compare the workloads of both the machines and the data partitions.

This dissertation introduces SWARM to addresses the challenges associated with adaptively changing the workload distribution among the machines of a distributed spatial streaming system. SWARM is a **S**patial **W**orkload-aware **A**daptive **R**outing **M**anager. SWARM is a layer that can be integrated into any distributed data streaming system that processes spatial data.

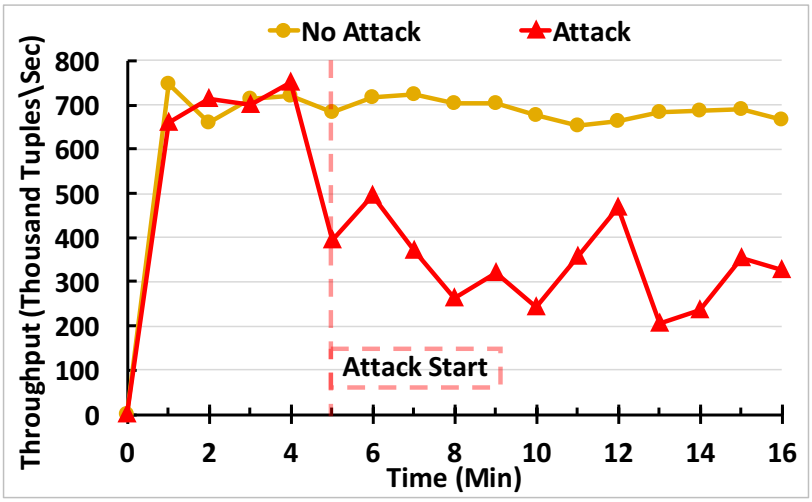

Fig. 1.2. The effect of an attack targeting the adaptive load-balancing mechanism of a distributed streaming system

SWARM adaptively load balances the workload among the available machines. TrioStat is integrated in SWARM to provide the necessary workload estimations. This makes it possible to decentralize load balancing, maximize local decision making, and reduce the communication overhead. SWARM proposes efficient algorithms to find the best partitioning plan while avoiding unnecessary repartitioning. SWARM maintains the integrity of the system while re-balancing the system without halting the system. SWARM is generic and does not depend on a specific spatial application. It can be used directly with minimal changes to the code of the spatial application. SWARM does not require prior knowledge about the distribution of data or the queries. SWARM achieves high machine utilization. This leads to high performance, low response time, and the handling of larger volumes of spatial data and queries.

Although using adaptive load-balancing techniques significantly improves the performance of distributed streaming systems, they make the system vulnerable to attacks. Attacks can be initiated using the knowledge that the system is using an adaptive load-balancing technique to redistributes workload across the machines base on changes of the workload. This type of attacks limits the availability and the

throughput of the system. Another objective of the attack can be to draw the attention of the system from focusing on serving real events. A different type of attacks on adaptive load-balancing mechanisms tries to leak protected information to malicious users. Figure 1.2 illustrates a timeline of the throughput of a distributed streaming system with adaptive load-balancing. The adaptive load-balancing has been targeted by an attack starting from Minute Five. Notice that the attack reduces the minimum throughput by 70%.

This dissertation introduces *Guard* to address the challenges of detecting and blocking attacks that target adaptive load-balancing mechanisms of distributed streaming systems. The dissertation reveals a new type of attacks that forces adaptive load-balancing mechanisms of distributed streaming systems into a continuous state of rebalancing. Guard is a component that detects and responds to malicious attacks on adaptive load-balancing mechanisms of distributed streaming systems. The main objective of Guard is to block the attacks and make the throughput of the system as close as possible to the throughput when there is no attack. Guard is used to make SWARM resilient to attacks. Guard introduces new features that are collected to characterize the behavior of the users and their relationships with hotspots. Guard collects the features with minimal overhead. Guard adopts an unsupervised machine learning technique that uses the collected features to detect and block the attack. Moreover, it allows Guard to differentiate between malicious and legitimate hotspots. Guard detects and blocks malicious users even when they coordinate in performing a single attack on the system. Guard does not block users until it is certain that they are malicious. Guard is general in the sense that it does not depend on a specific adaptive load-balancing mechanism nor a specific distributed streaming system. Guard requires minimal changes to the original code of the application.

Figure 1.3 illustrates the integration of the solutions that this dissertation proposes to develop an attack-resilient adaptive load-balancing mechanism for distributed spatial streaming systems. SWARM adds a routing layer composed of multiple machines

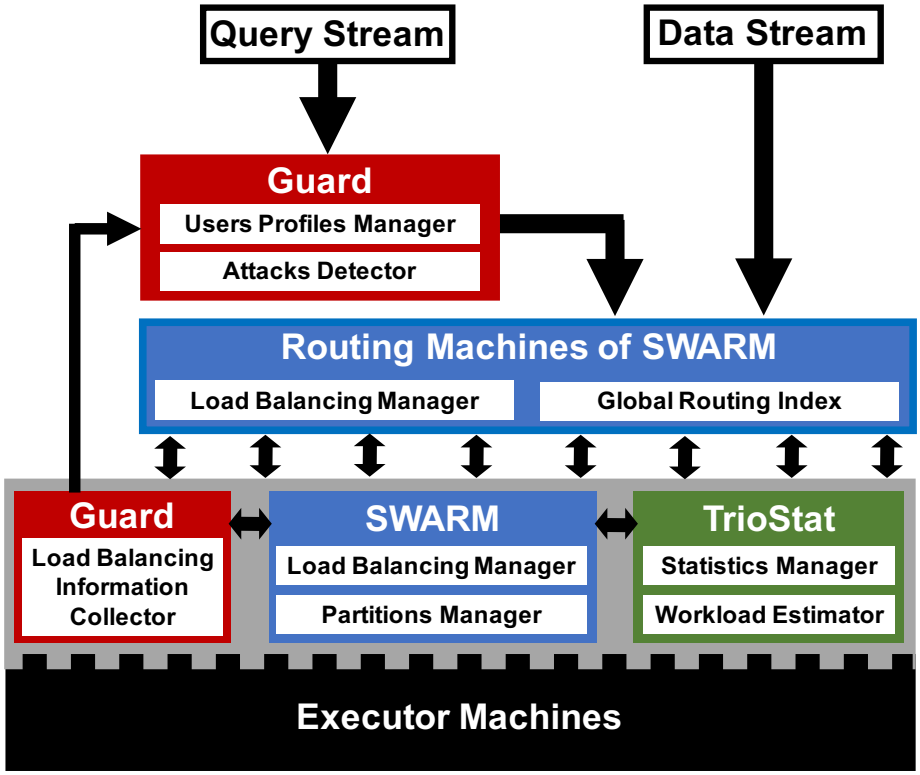

Fig. 1.3. Abstract design of an attack-resilient adaptive load-balancing mechanism for distributed spatial streaming systems

that distribute the workload among the executor machines based on a spatial index. Moreover, SWARM, TrioStat, and Guard add local components to every executor machine. SWARM's component in an executor machine manages load-balancing and the partitions that the machine holds. TrioStat collects and maintains statistics locally to provide the workload estimates that SWARM requests. SWARM adaptively re-partitions and/or redistributes partitions based on the changes in the workload estimates. Guard uses a separate machine to build and maintain profiles about the users of the system and detect attacks based on the profiles. Guard receives the query stream before forwarding it to the routing machines of SWARM. This gives Guard the ability to collect statistics about users' queries and to block attacks. Guard's compo-

nent in an executor machine collects information about every re-balancing operation that SWARM performs.

1.1 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides relevant background concepts and related work. A solution to the problem of online workload estimation for distributed spatial streaming systems (TrioStat) is presented in Chapter 3. Chapter 4 discusses the design and the implementation of the proposed adaptive load balancing protocol, SWARM. Chapter 5 addresses the problem of detection and response for attacks targeting adaptive load-balancing mechanisms in distributed streaming systems (Guard). Chapter 6 studies the performance of proposed solutions. Chapter 7 concludes the dissertation with suggestions for future work.

2. RELATED WORK

This chapter presents the work related to adaptive distributed spatial streaming systems and its security. We classify the work related into the following categories: (1) Big data streaming systems, (2) Adaptive distributed processing systems, (3) Workload estimations in distributed processing systems, (4) Distributed streaming systems security, and (5) Unsupervised machine learning.

2.1 Spatial Data Streaming Systems

Centralized spatial data streaming systems have been developed to answer spatial queries over spatial streams, e.g., PLACE [17], SINA [18], SEA-CNN [19], and Gpac [20]. However, these systems are not scalable and cannot handle the current scale of streamed spatial data.

General-purpose big data systems provide an infrastructure to scale up the batch and real-time processing. General-purpose big data systems are either batch-oriented or stream-oriented. Examples of batch-oriented include Hadoop [21] and Spark [22]. Batch-oriented systems require minutes or even hours to process data and are not suitable for real-time processing. Yahoo S4 [23], Apache Samza [24], Apache Storm [3], Twitter Heron [4], and Spark Streaming [5] are examples of stream-oriented systems that can process data in real-time with latencies ranging between milliseconds up to few seconds.

Distributed stream-oriented systems can be categorized based on their processing model into micro-batch and tuple-at-a-time systems. Spark Streaming [5], M3 [25], Comet [26], and Google DataFlow [27] are examples of micro-batch systems that accumulate small batches of data before distributing them for processing. In contrast, tuple-at-a-time systems process data tuple once it arrives to produce results with

low latency. Therefore, tuple-at-a-time systems are more suitable for real-time applications. Examples of tuple-at-a-time systems are: Apache Storm [3] and Twitter Heron [4]. The performance of these systems relies on how evenly they distribute their workload among their machines. Moreover, these systems are not optimized for spatial data processing and are not adaptive. In this dissertation we focus on addressing the challenges of adaptive load-balancing in tuple-at-a-time spatial systems.

To enable the scalable processing of big spatial data, several general-purpose big data systems have been extended with spatial partitioning and indexing techniques. e.g., [6–11, 14]. HadoopGIS [28], SATO [29], and SpatialHadoop [30] are big spatial processing systems on top of Hadoop. LocationSpark [31], Cruncher [6], Simba [32], SparkGIS [33] are spatial extensions to Spark. All these systems do not offer real-time big spatial data processing. Most of the existing big spatial data streaming systems use static data partitioning schemes to distribute the workload among machines. Zhang et al. [34] extends Storm with static spatial partitioning to enable real-time spatial data processing. However, these techniques are not effective in spatial applications due to the rapid changes in data and query distributions. This dissertation proposes SWARM that enables adaptive spatial processing over any distributed streaming system that works in a tuple-at-a-time systems, including Storm.

2.2 Adaptive Distributed Processing Systems

The workload of distributed streaming systems is skewed and is continuously changing. Hence, static partitioning is not suitable for distributing the workload among the system's machines. This has led to the development of adaptive load-balancing mechanisms that achieve higher throughput and lower response time. Data Skewness has been addressed to improve the performance of distributed systems, e.g., [35], [36], [37], [38, 39]. The work in [35] proposes an approach to detect the degree of distribution in spatial data and choose the best partitioning strategy accordingly. However, this approach works only offline with Hadoop. Therefore, it is

not suitable for streaming systems, where the processing cannot be halted. Fang, et al. [36] introduce a key-based workload partitioning strategy to rebalance the workload with minimum migration overhead. The rebalancing problem is posed as an optimization problem that considers the skewness and variance of the workload. SIMOIS [37] is a distributed join system that reduces the imbalance of workload skewness by identifying the set of workload-heavy (hotspots) keys and optimizes the join query accordingly. PKG2 and PKG5 [38, 39] are stream partitioning schemes that evenly distribute the received workload for each key among a limited number of the system's machines. The work in [36–39] focus on key-based applications. Therefore, their techniques are not optimized to work efficiently with processing spatial data. This dissertation has the leverage to change the spatial boundaries of partitions to distribute the workload of a hotspot. Unlike PKG2 and PKG5, our proposed solution is not forced to distribute the workload of a hotspot over a specific number of machines. It can distribute the workload among all executor machines, if necessary.

Several adaptive batch and streaming management systems have recently been proposed to handle any variabilities in the underlying workload of different applications. AQWA [15] is an adaptive spatial processing system on top of Hadoop. AQWA distributes new batches of data into HDFS files offline before starting to process the queries. This cannot work for processing streams in real-time. Cruncher [6] is a proposal for adaptive spatial stream processing on top of Spark. However, Cruncher works only on micro-batch stream processing that has relatively high latency, i.e., seconds. However, our approach is able to adaptively process spatial data in real-time with minimal latency. Amoeba [40, 41] introduce a distributed streaming system for general multi-dimensional workloads with adaptive rebalancing. Amoeba does not consider real-time stream processing. STAR [42] is a distributed streaming warehouse for spatial data that supports low-latency and up-to-date data analytical applications by adapting to workload changes. Tornado [12, 13] is a distributed in-memory streaming system for spatio-textual data that extends Storm. Tornado includes an adaptive indexing layer for dynamic re-distribution of processes across the system according to

changes in the data distribution and/or query workload. PS^{2}Stream [14] is a publish-subscriber system for spatio-textual data that supports dynamic load adjustments to adapt to the changes of the workload. Tornado and PS^{2}Stream are designed to work only with spatio-textual data and continuous spatial-keyword filter queries. In addition, they only support publish/subscribe applications, which result in removing data points as soon as they are processed. However, our approach is more general as it works with any spatial application. Moreover, it can work over any distributed streaming system that processes spatial data in a tuple-at-a-time manner.

Existing adaptive load-balancing mechanisms from the literature differ in the types of applications that they support and the models used for measuring their workload. They distribute the workload depending on the distinctive key features of the applications they serve, e.g., spatial regions, text topics, or hash values. They are common in the way they rebalance their workload by repartitioning the responsibilities of each machine according to the changes in data and query workloads. All the mechanisms monitor the workload of each machine in the system by computing, for each machine, a score that represents the amount of data and query workload that have processed. The rebalancing is achieved by moving some responsibilities of the machine with the highest workload to the machine with the lowest workload. Other mechanisms move the workload to a subset of the under-loaded machines instead of to the lowest machine alone. Most distributed streaming systems support continuous queries, and return their results to the users in real-time. Continuous queries are queries that get registered and stored in the system for a period of time that is predetermined by the user. Every time a new tuple arrives, the system checks if this tuple qualifies as a result for any of the registered continuous queries. Moving some of the workload of a machine includes moving some of its continuous queries.

Figure 2.1 gives an example of how an adaptive load-balancing mechanism redistributes the workload of a spatial distributed streaming system. The distinctive key feature of this application is being spatial. Therefore, the adaptive load-balancing mechanism divides the whole space (USA map) into spatial partitions among five

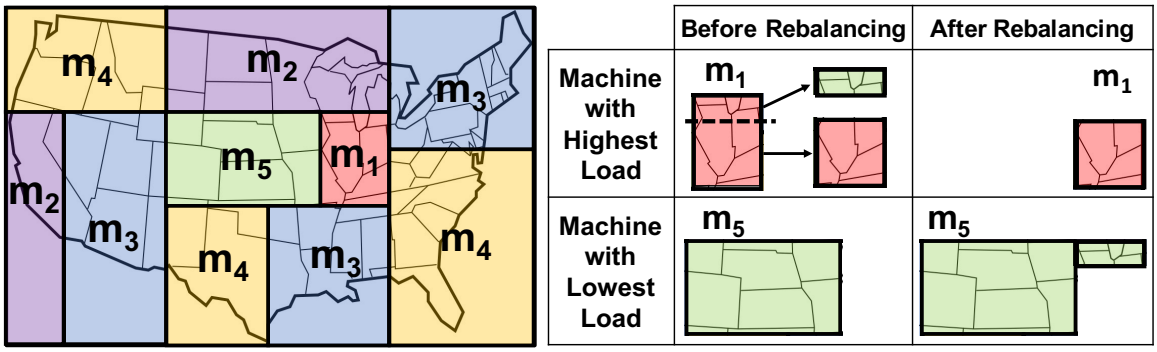

Fig. 2.1. Adaptive load-balancing in a spatial distributed streaming system

machines: m_1, m_2, m_3, m_4, and m_5. Figure 2.1 illustrates that m_1 is responsible for only one partition. This partition includes a city that is gaining an increase in users interests because of an event. This creates a hotspot that requires more processing from m_1 because of the increased query and data workload on this city. The adaptive load-balancing mechanism identifies that m_1 is overloaded while m_5 has the lowest workload. Hence, the adaptive load-balancing mechanism divides m_1's partition that contains the hotspot into two partitions based on the workloads of m_1 and m_5. One of the partitions is moved to m_5 to evenly distribute the workload between the two machines. As long as the system is unbalanced, the adaptive load-balancing mechanism will continue to repartition and move workloads across the machines.

Using adaptive load-balancing mechanisms increases the machine utilization, which leads to improved throughput and execution latency. However, a new type of attacks can target the adaptive load-balancing mechanisms to keep the system in a continuous state of re-balancing. This type of attacks leads to lower throughput and availability of the system. This dissertation reveals the specifications of this type of attacks. Moreover, we proposes Guard, which can be deployed on distributed streaming systems to detect and block attacks on their adaptive load-balancing mechanisms.

2.3 Workload Estimations in Distributed Processing Systems

Adaptive load-balancing mechanisms improve the performance and scalability by trying to even the workload across all machines. One obstacle is to estimate the workload of spatial distributed streaming systems because spatial data and query workloads are skewed and rapidly changing. Distributed spatial streaming systems often do not maintain a global system workload state because the data and queries are distributed across their machines. Collecting and maintaining statistics about the workload of every machine in a centralized location introduces high network, storage, and processing overheads.

Several techniques exist to track data changes and adapt distributed systems accordingly. Belussi, et al. [35] propose an approach for SpatialHadoop [30] to detect the skewness degree in spatial data distribution using box-counting functions [43]. They choose the best partitioning strategy using a heuristic sketch and the detected skewness degree. However, they cannot track online changes in skewness in a distributed streaming setup. Also, they estimate the workloads based on the data distribution without considering the query workloads. Identifying hotspot keys in SIMOIS [37] is performed using an exponential counting scheme. However, SIMOIS technique works only with distributed join system. PKG2 and PKG5 [38,39] use only the key frequencies as an estimate for the workload and do not consider queries. Thus, they are not suitable for tracking changes in spatial data distribution and estimating the workload of their partitions.

Several techniques have been proposed for efficient spatial data aggregation and summarization. Ho, et al. [44] introduce a technique to answer range-sum queries of the number of points in a window by maintaining prefix sums in a grid. Riedewald, et al. [45] generalize the idea of prefix sums to count the number of rectangles (queries) and support OLAP queries. Maintaining aggregates and summarizations for spatial regions is challenging because the counting could result in duplicate counting of some queries. Euler histograms [46] count rectangles that intersect a given region without

duplicates. Euler histograms help estimate the selectivity of spatial joins [47, 48]. AQWA [15] adaptively changes the partitioning of Hadoop [21] by maintaining statistics using the prefix sum technique and a variant of the Euler histogram. AQWA introduces a cost model to estimate both the data and query workloads. However, it is centralized and hence is not viable for distributed streaming systems because data and statistics are distributed on different machines.

Some distributed streaming systems use adaptive load-balancing that redistribute the workload based on AQWA's cost model for estimating the workload, e.g., STAR [42], Tornado [12,13], PS^{2}Stream [14], and Amoeba [40,41]. However, these techniques are relatively slow when updating the statistics and updating the workload cost model when the statistics change. The reason is that they consider only the history of data and queries without considering how persistent these estimates could be in the future. Most distributed streaming process data in real-time and do not store the data for a long durations. Systems with adaptive load-balancing need a technique that can accurately predict the workload fast with minimum network and processing overheads. This dissertation proposes TrioStat, which estimates the workload for distributed spatial streaming systems with minimum overhead. TrioStat is applicable to tuple-at-a-time systems that are suitable for real-time processing.

2.4 Security in Distributed Streaming Systems

Data security implies three aspects: Confidentiality, Integrity, and Availability [49]. Confidentiality assures that confidential information is not disclosed to unauthorized personnel. On the other hand, integrity ensures that data do not undergo unauthorized changes either intentionally or accidentally. Availability guarantees that the system is functioning and data can be accessed whenever it is requested. In this dissertation, we focus on distributed stream processing systems availability in the presence of attacks. The availability can be ensured by providing fault-tolerance techniques [50] and Intrusion Detection Systems (IDS) [51, 52].

In general, fault-tolerance techniques aim at avoiding service failures in the presence of system faults [53]. There are three mechanisms of fault-tolerance in distributed systems: (1) cold restart, (2) check-pointing, and (3) replication. Both cold restart and check-pointing use restarting operators to correct and recover from transient failures. The restart operation can happen either directly (in case of cold restart) or via a checkpoint (in case of check pointing restart) [54]. In replication-based fault-tolerance, multiple instances of the same operators are running concurrently to transparently take over processing from a faulty operator.

Huang, et al. [55] proposes AF-Stream systems that addresses the natural trade-off between performance and fault-tolerance. In particular, AF-Stream approximates fault tolerance by mitigate backup overhead using adaptive issuing backups, while ensuring that the errors upon failures are bounded with theoretical guarantees. Knasmüller, et al. [56] presents the Pathfinder framework that enables functional redundancy at the level of stream processing operator paths. In particular, Pathfinder reacts to failures in the main path by switching into alternative failure-free paths. Liu, et al. [57] proposes E-Storm, a replication-based state management system. The objective of E-Storm is to maintain multiple back-ups for the state on multiple nodes. Load-balancing mechanisms can be considered as a replication-based fault-tolerance technique. Fang, et al. [58] proposes combining fault-tolerance and load-balancing mechanisms with the objective to reduce the overall resource consummations while keeping high-throughput, highly-available system. The workload is balanced based on data-level strategy that considers data skewness and node failure.

On the other hand, Intrusion Detection System (IDS) is a crucial technique that is integrated with any comprehensive security solution for high-assurance database security. The main goal of IDS is to monitor and detect illicit accesses and malicious actions in the system. The existing methodologies of IDS can be broadly classified into two groups: *signature-based detection* and *anomaly detection*. In the signature-based detection approach, IDS looks for attack patterns in access logs using data mining techniques [59–62]. This approach works against well-known attacks. However, it

is incapable of detecting emerging types of attacks. On the other hand, IDS that uses the anomaly detection approach looks for deviations in normal user behavior. Thus, it is capable of detecting unexpected emerging attack patterns. Although IDS can protect systems by detecting and rejecting future accesses of attackers, it is not designed to mitigate the risk of intrusion. Moreover, IDS suffers from long detection delay as well as high false-alarm rate that can cause negative impact on the availability of the systems.

In this dissertation, we address the availability degradation of the distributed streaming systems caused by attacks. The proposed methodology employs an intrusion detection system to detect and prevent attackers from registering malicious queries. Several security threats to distributed streaming and load-balancing systems have been addressed in the literature. Existing security mechanisms in big data streaming systems focus on authentication, access control, and auditing [63] to maintain data confidentiality. However, in big data streaming systems maintaining high availability is critical. Work related to security attacks that compromise the availability in streaming and load-balancing systems is under-explored. Ledlie, et al. [64] proposes an algorithm, *k-choice*, to balance workload in systems vulnerable to Sybil attacks that can affect the skewness of query distribution over the workload. Kang, et al. [65] introduces *sensitivity attack*, a new type of attacks on data plane systems. In that attack, a malicious user can articulate a query to "flip" the expected behavior of the data plane systems (including load-balancing mechanisms in streaming systems).

2.5 Unsupervised Machine Learning

In data analysis and machine learning, unsupervised learning is a type of statistical modeling which mainly has the objective of clustering data points into groups. Such grouping is performed without the need of training labeled data, which distinguish the unsupervised approach from the supervised approach. The input of unsupervised clustering techniques is a set of n unlabeled data points that ought to be grouped

according to a certain similarity metric. The output of the clustering techniques is an assignment of the n data points to a set of k clusters.

All of the clustering techniques rely, in one way or another, on a similarity metric to define groups and groups' members who are assumed to be similar. Some common similarity metrics used in the literature include Euclidean distance, Mahalanobis distance, and Pearson correlation distance. Other metrics that can be used with non-quantitative data points include Hamming distance and Jaccard distance. Choosing the right distance metrics depends on the type of data points to be clustered and the type of similarity to be found in a given clustering task.

There are numerous techniques to perform unsupervised clustering, which can be categorized into nine groups [66]: partitioning-based clustering, density-based clustering, distribution-based clustering, fuzzy clustering, hierarchical clustering, clustering based on graph theory, clustering based on grid, clustering based on fractal theory, and model-based clustering. In partitioning-based clustering, the number of clusters is predefined as a hyper-parameter. K-Means [67] leads this category of clustering techniques. On the contrary, density-based clustering techniques like DBSCAN [68] determine the number of clusters according to the number of local communities that can be found from the density of the points in certain locations in the feature space. Distribution-based clustering [69] assigns data points to clusters according to their belonging to the same distribution, assuming that the whole data points came from multiple distributions. In fuzzy clustering [70], a data point can belong to more than one cluster, which makes the output clusters non-mutually exclusive. Hierarchical clustering [71] allows defining clusters at different levels of granularity, starting from a single cluster containing all data points to a level where each data point constitutes a different cluster. When the data points and their relationships are represented by a graph, they can be clustered using techniques such as CLICK [72]. Grid-based clustering such as STING [73] relies on transforming the feature space that represents the data points into a grid structure and cluster data points in each cell of the grid.

Finally, In model-based clustering [74], each cluster are assumed to have a model that fits the data points that belong to that cluster.

K-Means [67] is one of the mostly applied unsupervised clustering techniques. As a partitioning-based technique, K-Means requires to assign a predefined number of clusters to appear in the output assignments, i.e., the number of partitions. This number is used to determine the number of centroids, i.e., the clusters' centers, which are initially assigned randomly then are updated iteratively during the procedure of K-means. Data points are clustered in K-means according to the distance from the centroids measures based on one of the distance metrics aforementioned. The advantage of K-Means appear in its low time complexity, simplicity, and scalability. On the other hand, K-Means has some shortcomings, including outliers sensitivity, the necessity of knowing the number of clusters in the data upfront, and the bad performance in partitioning clusters within clusters. In this dissertation, we proposes an unsupervised attack detector for detecting attacks that target adaptive load-balancing mechanisms in distributed streaming systems. This detector uses K-Means during its analyses to cluster users based on their behaviors.

3. TrioStat: ONLINE WORKLOAD ESTIMATION IN DISTRIBUTED SPATIAL DATA STREAMING SYSTEMS

This chapter introduces TrioStat, an online workload estimation technique for estimating the workload of partitions and machines in a distributed spatial data streaming system. TrioStat provides estimations with minimum network and storage overheads. The cost model that TrioStat uses to estimate workloads is presented in the Section 3.1. Section 3.2 introduces a novel mechanism for collecting and maintaining the required statistics for online workload estimation. Finally, Section 3.3 presents the way TrioStat uses its statistics to efficiently provide workload estimations.

3.1 The Cost Model

Distributed spatial streaming systems divide their whole space that the application serves into partitions. The partitions are distributed across the machines of the system. The system rebalances the workload across its machines by repartitioning and/or redistributing the partitions. TrioStat estimates the workload of a partition by computing its potential processing cost on the system in relation to all other partitions. Moreover, the workload of a machine is estimated according to the partitions served by the machine. TrioStat introduces a probabilistic cost model that relies on three terms (Trio). The main factor of the cost model is the amount of data points that are received by each partition. The cost model gives higher weight to partitions having a high number of queries. This is because the increase of number of queries indicates to an increase in the required number of query checks against every new data point. Moreover, the cost model predicts the future workload of each partition based on its workload history. This prediction serves as a scale factor for the overall cost and workload of each partition. Assume that we have a distributed spatial streaming

system, say S, that has a set of executor machines M. Each machine $m \in M$ holds some partitions P_m, where $|P_m| = n_m$, n_m is the number of partitions in Machine m. Each partition $p \in P_m$, locally maintains some statistics. The cost estimate $C(p)$ of a partition p is computed as follows:

$$C(p) = N(p) \times Q(p) \times Prob(p) \tag{3.1}$$

$N(p)$ is the number of points received by Partition p, $Q(p)$ is the number of queries that overlap p, and $Prob(p)$ is the probability that new data and queries land in p. $Prob(p)$ depends on the amount of data and queries that arrived during the last round of repartitioning. Note that the workload history is captured via N and Q while $Prob$ is a weighting factor to the cost of this history. The effect of old data can fade with time as Section 3.2.2 discusses. $Prob(p)$ is estimated as follow:

$$Prob(p) = \frac{R(p)}{R(S)} \tag{3.2}$$

where $R(p)$ and $R(S)$ are the number of data points and queries received by p and all of S, respectively, during the last round of repartitioning. $R(S)$ is computed as follows.

$$R(m) = \textstyle\sum_{i=1}^{n_m} R(p_i) \tag{3.3}$$

$$R(S) = \textstyle\sum_{i=1}^{|M|} R(m_i) \tag{3.4}$$

By substituting Eqn. 3.2 into Eqn. 3.1, then:

$$C(p) = \frac{N(p)Q(p)R(p)}{R(S)} = \frac{Num(C(p))}{R(S)} \tag{3.5}$$

where $Num(C(p))$ is the numerator of Partition p's cost formula. The workload of Machine m is computed according to the set of partitions P_m that m holds by:

$$C(m) = \textstyle\sum_{i=1}^{n_m} C(p_i) \tag{3.6}$$

Using Eqn. 3.5,

$$\begin{aligned} C(m) =& \frac{N(p_1)Q(p_1)R(p_1)}{R(S)} + \cdots + \frac{N(p_{n_m})Q(p_{n_m})R(p_{n_m})}{R(S)} \\ C(m) =& \frac{\sum_{i=1}^{n_m}\{N(p_i)Q(p_i)R(p_i)\}}{R(S)} = \frac{Num(C(m))}{R(S)} \end{aligned} \tag{3.7}$$

where $Num(C(m))$ is the numerator of Machine m's cost formula. $Num(C(m))$ can be computed locally. In contrast, computing $R(S)$ requires information from all the machines in S. $R(S)$ is the same for all machines, and hence it is computed once using Eqn. 3.4 that requires only one number ($R(m)$) from each executor machine. Thus, comparing and ranking the machines according to their costs is the same as comparing and ranking them using only $Num(C(m))$. Moreover, computing $Num(C(p))$ for the partitions of a machines is enough to compare and rank the partitions locally in their machine by cost.

3.2 Collecting and Maintaining Statistics

Collecting statistics in distributed streaming systems is challenging because the data arrives in high continuous volumes. Moreover, most applications need real-time processing for each data point with minimum latency. Thus, a feasible technique for collecting and maintaining statistics in distributed streaming systems should require minimum number of updates. Also, each partition should maintain its statistics locally without the need to communicate with other machines. TrioStat achieves this by maintaining minimum local statistics that are enough to estimate the workloads of partitions and machines using the cost model that Section 3.1 discusses. TrioStat uses a hash table in every executor machine to link the ID of every partition in the machine with its statistic structure. TrioStat maintains the statistics of every partition in a simple multidimensional array in memory. The statistics of each row (or column) is located next to each other in memory. Therefore, TrioStat can provide workload estimations for a partition or a part of a partition fast by taking advantage of cache prefetching. Reading the first needed statistic to compute a workload estimation from a row (or column) result on having the remaining needed statistics in cache. The remaining of this section discusses in details the process of collecting and maintaining statistics in TrioStat.

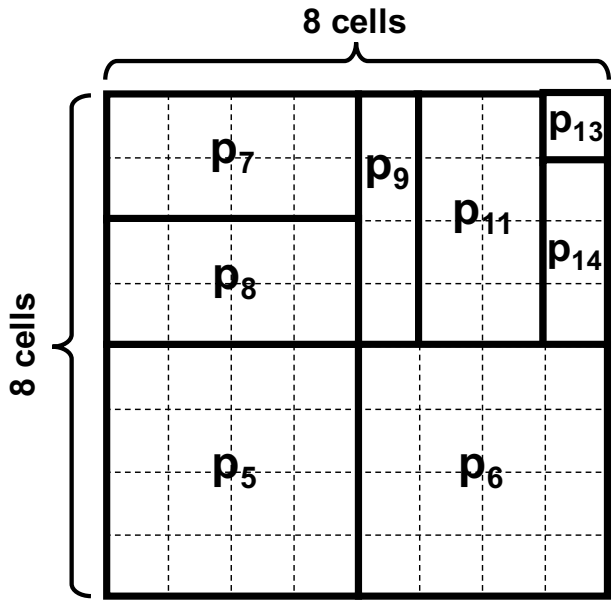

Fig. 3.1. Example for dividing the space into a grid of small cells

3.2.1 Required Statistics

TrioStat maintains the minimum statistics needed for using the cost model. The whole space of the application is divided into a grid of small cells, which should be aligned with the boundaries of the partitions. Figure 3.1 shows an example for dividing the space into a grid of 8X8 small cells. The arrangement of cells that cover a partition is passed to TrioStat with the partition ID of the executor machine that holds this partition. Increasing the number of cells that divide the space results in increasing the storage and processing overhead of TrioStat and increasing the granularity of workload estimation. We use Partition p_{11} in Figures 3.1 to illustrate how TrioStat maintains the statistics.

Systems periodically evaluate their performances and check if repartitioning could improve the performance. Therefore, systems ask TrioStart to provide the needed workload estimates by the end of every repartitioning round. Figure 3.2 gives the maintained statistics in p_{11} after asking TrioStat to prepare the statistics and be ready

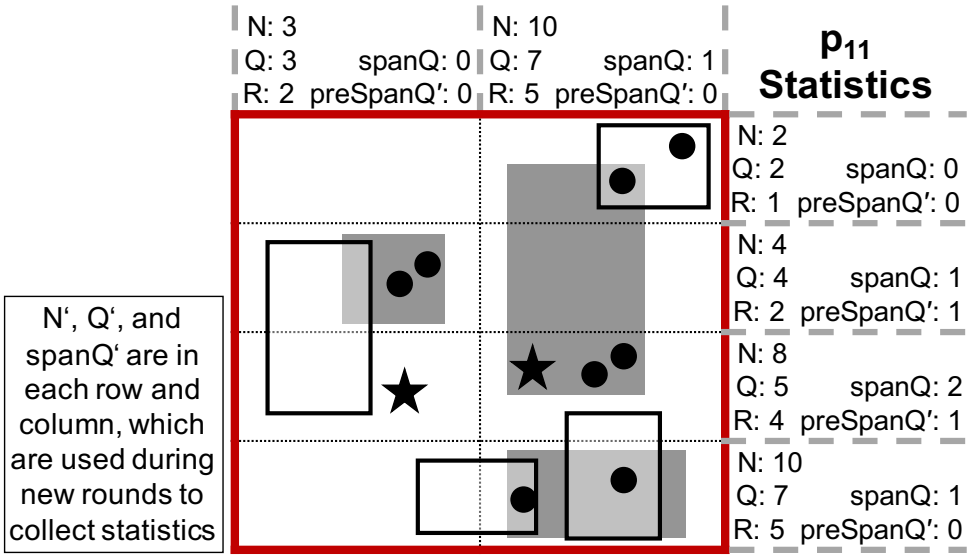

Fig. 3.2. TrioStat statistics for partition p_{11}

to provide workload estimations. The dots and rectangles represent the positions of the data points and the query ranges in p_{11}, respectively. The stars and the gray rectangles mark the data points and the queries received in the last round of repartitioning, respectively. p_{11} has a 4X2 cell matrix. The indexes of the rows start form top to bottom, while it is from left to right for the columns. TrioStat maintains in each row and column 5 statistics, 3 of which are cumulative. Row i's (Column j's) cumulative statistics represent the total from the uppermost row (leftmost column) until Row i (Column j), respectively. The 5 maintained statistics in each row and column are: (1) N: the cumulative number of data points, (2) Q: the cumulative number of queries, (3) R: the cumulative number of data points and queries received during the last round of repartitioning, (4) $spanQ$: the number of queries whose ranges span from the previous row/column, and (5) $preSpanQ'$: the number of queries received during the last repartitioning round whose ranges span from the previous row/column. To illustrate, refer to Row 3 of p_{11} in Figure 3.2. All cumulative statistics reflect the objects in the first three rows. There are 8 data points (N)

and 5 queries (Q). Two data points and two queries are received during the last round, hence $R = 4$. Two queries span from the second row ($spanQ = 2$). However, only one of them is received during last round ($preSpanQ' = 1$).

TrioStat uses these statistics for estimating the workload of each part of a partition. The overall statistics of a partition p ($N(p)$, $Q(p)$, and $R(p)$) are the ones in the last row/column. The statistics are only updated at the end of every repartitioning round to avoid the overhead of updating almost all the statistics whenever a new data point or a query arrives. Additional three statistics, termed *Statistics Collectors*, for each row and column are introduced, namely N', Q', and $spanQ'$. *Statistics Collectors* are used to update the statistics at the end of a round. They reduce the number of updates per received data point or query. The next section presents how these *Statistic Collectors* are updated and used for maintaining the statistics.

3.2.2 Maintaining the Statistics

TrioStat needs to have a small number of updates when receiving a data point or query. When a new data point arrives, TrioStat updates only two of a partition's *Statistics Collectors*. However, when a new query arrives, TrioStat updates the *Statistics Collectors* of the rows and columns that overlap the query. Having more statistics to update will not affect the performance because the arrival rate of data is much higher than that of queries in distributed streaming applications.

Three *Statistics Collectors*, N', Q', and $spanQ'$, are used in each row/column to count different types of received objects during the most recent round of repartitioning. N' and Q' count the new data points and queries, respectively. $spanQ'$ counts the number of queries that their ranges span from the previous row/column. When a new data point arrives, TrioStat increments N' of the row and the column containing the data point. When a new query arrives, TrioStat increments both Q' of the row and the column that overlap the top-left corner of the query, and $spanQ'$ of the remaining rows and the columns that overlap the query.

To conclude a repartitioning round, TrioStat uses the *Statistics Collectors* to update all remaining statistics as follows. Let $i \geq 0$ be a row/column index. Then, the statistics are updated as follows:

$$
\begin{aligned}
N(i) &= N(i) + \textstyle\sum_{j=0}^{i} N'(j) \\
Q(i) &= Q(i) + \textstyle\sum_{j=0}^{i} Q'(j) \\
R(i) &= \textstyle\sum_{j=0}^{i} N'(j) + \sum_{j=0}^{i} Q'(j) \\
spanQ(i) &= spanQ(i) + spanQ'(i) \\
preSpanQ'(i) &= spanQ'(i)
\end{aligned}
$$

Algorithm 1: updateStat(*PartitionID*, *rowOrColumn*)

```
1  stat[ ][ ] = partitionsHashMap.get(PartitionID)
       .statistics(rowOrColumn)            ▷ Multidimensional array
2  int sumN' = 0
3  int sumQ' = 0
4  for i = 0 to Num of rowOrColumn in PartitionID do
5      sumN' += stat[N'][i]
6      sumQ' += stat[Q'][i]
7      stat[N'][i] = 0                     ▷ Reset current N'
8      stat[Q'][i] = 0                     ▷ Reset current Q'
9      stat[N][i] += sumN'
10     stat[Q][i] += sumQ'
11     stat[preSpanQ'][i] =stat[spanQ'][i]
12     stat[spanQ][i] += stat[spanQ'][i]
13     stat[spanQ'][i] = 0                 ▷ Reset current spanQ'
14     stat[R][i] = sumN' + sumQ'
15 end
```

The naive way to compute the cumulative statistics requires computing the summations from the beginning each time. Its time complexity to update the statistics of a partition is $O(k^2)$, where k is the number of rows and columns of the partition's statistics. However, TrioStat utilizes the fact that the summations in the equations can be carried out from one row/column to another. Hence, there is no need to compute the summations from scratch each time. With only one addition, we produce the statistics of the next row/column from these of the previous row/column. Algorithm 1 illustrates how to update the statistics of a partition by passing once through the partition's rows and columns. The time complexity of using Algorithm 1 to update the statistics of a partition is $O(k)$. This algorithm runs as a separate task in the background. Note that all *Statistics Collectors* are reset to 0 to be ready for collecting the statistics of the next round of repartitioning.

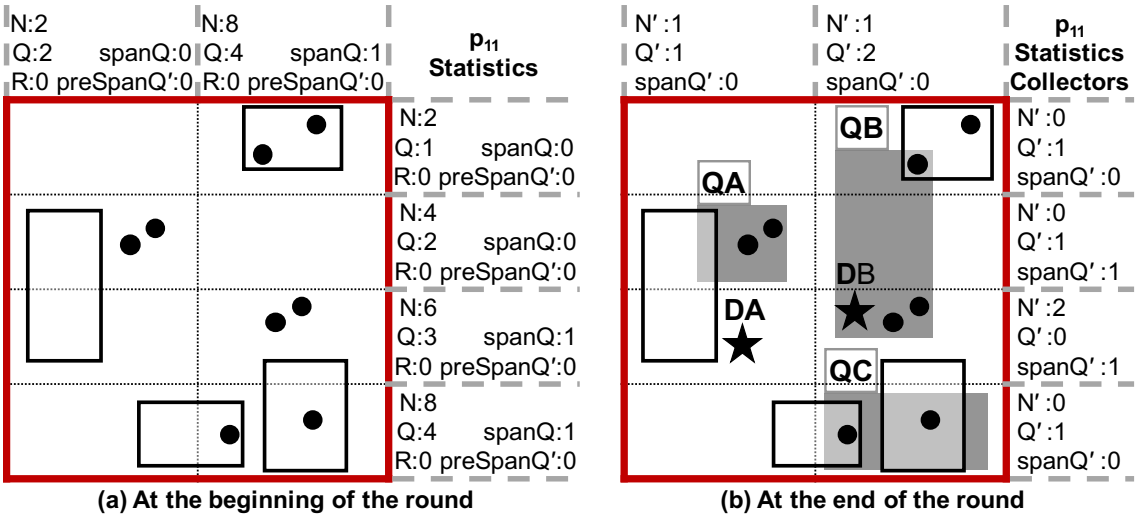

Fig. 3.3. Updating partition p_{11}'s *Statistics Collectors*

Figure 3.3 illustrates the statistics of Partition p_{11} while receiving new data points and queries. Figure 3.3a illustrates the positions of the data points and the ranges of the queries in p_{11} at the beginning of a new repartitioning round. Also, it shows the current state of the maintained statistics as Section 3.2.1 discusses. The *Statistics*

Collectors are all set to 0 at the beginning of the round. Figure 3.3b shows the *Statistics Collectors* at the end of the round after receiving 2 new data points and 3 new queries. During the repartitioning round, the two data points D_A and D_B are received first. Both data points are in the third row (Row_2), but one of them is in the first column (Col_0) while the other is in the second column (Col_1). Hence, $N'(Row_2)$ is incremented twice while $N'(Col_0)$ and $N'(Col_1)$ are each incremented once. Then, Queries Q_A, Q_B, and Q_C arrive into p_{11} in this order. The upper-left corner of Q_A is in the cell that overlaps Col_0 and Row_1. Also, the range of Q_A is contained within one cell. Thus, only $Q'(Row_1)$ and $Q'(Col_0)$ are incremented. Q_B starts in Row_0 and spans through Row_1 and Row_2. Thus, $spanQ'(Row_1)$ and $spanQ'(Row_2)$ are incremented in addition to the increment of $Q'(Row_0)$ and $Q'(Col_1)$. At the end of the round, *Statistics Collectors* are used to update the statistics using Algorithm 1. The results of the updated statistics are given in Figure 3.2.

Notice that the target of TrioStat is not to count the actual number of data points but rather to track the change in the spatial data workload. To diminish the effect of old data gradually, the number of data points N is divided by 2 before it is updated in each round of repartitioning. This is to reduce the effect of old data points on the current spatial distribution. In distributed streaming systems that support historical queries, TrioStat needs to be informed about data expiration to update N accordingly.

3.2.3 Correctness of the Statistics

In this section, we prove the correctness of the statistics that TrioStat collects and maintains about data points and queries. To show that, we need to prove that the maintained statistics always represent the true number of data points and queries without any over- or under-counting.

First, we prove the correctness of the statistics for data points. Assume that we have a partition that has k rows and only one column. This results in k cells in total as in Figure 3.4.

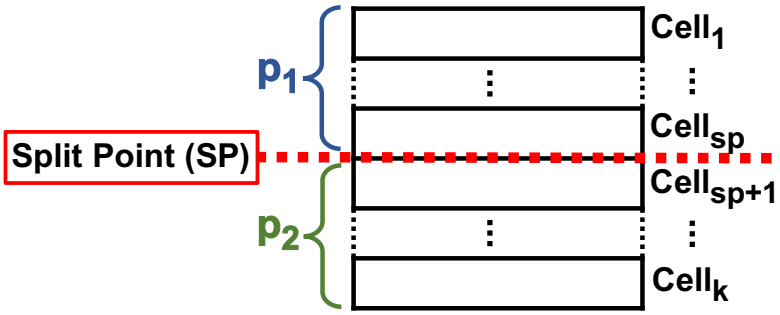

Fig. 3.4. Partition with k cells (rows)

Let i be the row number of a cell, where $1 \leq i \leq k$, $n(i)$ be the true number of data points in $cell_i$, and $N(i)$ be the cumulative number of data points that TrioStat maintains in row_i. $n(i)$ can be obtained by simply counting the number of data points within $cell_i$. As mentioned before, the cumulative number $N(i)$ is computed from top to down for horizontal divisions. Therefore, $N(i)$ can be computed as follows:

$$N(i) = \sum_{j=1}^{i} n(j)$$

In the initial case where $k = 1$, there is only one cell with $n(1)$ data points, hence $N(1) = n(1)$. For $k = 2$, $N(2) = n(1)+n(2)$. We can derive the number of data points in $cell_2$ as $n(2) = N(2) - N(1)$. In general, let us assume a partition as illustrated in Figure 3.4 that has a split point sp, where $1 <= sp <= k$, that divides the partition into two partitions, say p_1 and p_2. $n(p_i)$ represents the true number of data points in Partition p_i. Therefore, the number of data points in each partition can be computed as follows:

$$\begin{aligned}
& n(p_1) = n(1) + n(2) + ... + n(sp) \\
\therefore\ & n(p_1) = \sum_{j=1}^{sp} n(j) = N(sp) \\
& n(p_2) = n(sp+1) + n(sp+2) + ... + n(k) = \sum_{j=sp+1}^{k} n(j) \\
\therefore\ & n(p_2) = \sum_{j=1}^{k} n(j) - \sum_{z=1}^{sp} n(z) = N(k) - N(sp)
\end{aligned}$$

Therefore, this shows that the computed statistics, i.e., N, is equal to the true number of data points, i.e., n. An analogous proof can be used to show that N is also correct when dividing cells vertically, and N is cumulatively computed from left to right.

Now, we prove the correctness of the maintained statistics about queries by using the same setup of Figure 3.4. Given the input query boundaries, we can extract the exact count of all queries in each grid $cell_i$ by maintaining four variables, namely, q_s, q_e, q_{se} and q_o (s stands for start, e stands for end, se stands for start and end, and o stands for overlap, as explained below). Let $q_s(i)$ be the number of queries whose upper boundary intersects $cell_i$ and whose lower boundary intersects another cell. Furthermore, let $q_e(i)$ be the number of queries whose lower boundary intersects $cell_i$ and whose upper boundary intersects another cell. Let $q_{se}(i)$ be the number of queries whose upper and lower boundaries intersect $cell_i$. Finally, let $q_o(i)$ be the number of queries whose upper and lower boundaries do not intersect $cell_i$ but their ranges overlap $cell_i$. Therefore, the true number, $q(i)$, of queries that intersect $cell_i$ is the sum of these four variables, i.e.,

$$q(i) = q_s(i) + q_e(i) + q_{se}(i) + q_o(i) \tag{3.8}$$

Now, we extend the formula above to compute the true number of queries that overlap a sub-partition, i.e., one column of cells that starts from Row u and ends in Row $l > u$. Let that number of $q(u, l)$. We need to avoid double counting of a query that overlaps multiple cells. $q(u, l)$ is equal to the true number of queries in Row u and only queries that start in any row from row $u + 1$ up to row l, no matter where these queries end. For rows after u, only counting queries that start in any cell will exclude recounting any query that span over multiple cells. Therefore, $q(u, l)$ can be computed using the four variables as follow:

$$q(u, l) = q(u) + \sum_{j=u+1}^{l}(q_s(j) + q_{se}(j)) \tag{3.9}$$

We need to demonstrate that the statistics gathered by TrioStat when counting the number of queries is equal to the true number, i.e., $q(u, l)$. Refer to Figure 3.4

for illustration. As in the figure, we have a partition that has k cells starting from Cell 1 at the top down to Cell k at the bottom. To maintain query statistics, for each Row i, where $1 \leq i \leq k$, of a partition, TrioStat maintains only two statistics per row, namely, $Q(i)$ and $Q_{span}(i)$. $Q(i)$ is the cumulative number of queries from the first row of the partition, i.e., Row 1, to Row i. Therefore, $Q(i)$ directly represents the number of queries that start at any row from the beginning of the partition until Row i. Recounting of queries can happen by considering queries that only end or overlap any of the cells as they are already counted where they started. Thus, they are excluded from $Q(i)$ as follows:

$$Q(i) = \sum_{j=1}^{i}(q_s(j) + q_{se}(j)) \tag{3.10}$$

Let $Q_{span}(i)$ be the number of queries that extend (span) from an upper row, say Row $(i-1)$, to Row i. Thus, $Q_{span}(i)$ represents the number of queries that overlap or start without ending in Row $(i-1)$. Note that $Q_{span}(1)$ will always equal 0 because there are no queries that extend from outside the partition to the first row. Thus, $Q_{span}(i)$ can be formulated from the true numbers as follows:

$$Q_{span}(i) = \begin{cases} 0, & \text{if } i = 1 \\ q_s(i-1) + q_o(i-1), & \text{otherwise} \end{cases}$$

Although $Q_{span}(i)$ depends on the variables of the previous row $(i-1)$ in TrioStat, there is another equivalent way of computing $Q_{span}(i)$ with the variables from Row i that makes the proof easier to follow. Note that any query that overlap Row $(i-1)$ or starts without ending in Row $(i-1)$ definitely extends to Row i and this query's range either ends at Row i or overlaps Row i and continues to the next row below Row i. This can be reflected in the formula for calculating $Q_{span}(i)$ as follows:

$$\begin{aligned} \because q_s(i-1) + q_o(i-1) &= q_e(i) + q_o(i) \\ \therefore Q_{span}(i) &= q_e(i) + q_o(i) \end{aligned} \tag{3.11}$$

This equivalent equation for calculating $Q_{span}(i)$ is correct also in the case when $i = 1$ because both $q_e(1)$ and $q_o(1)$ are always equal 0.

In the initial case, i.e., when $k = 1$, and there is only one cell in the partition with $q(1)$ queries, $Q(1) = q_s(1) + q_{se}(1) = q(1)$ and $Q_{span}(1) = 0$. This is correct because $q_s(1)$, $q_e(1)$ and $q_o(1)$ are all equal 0 as $cell_1$ covers the whole partition and every query definitely starts and ends in this cell. For $k = 2$, using Eqns. 3.10 and 3.11, Q and Q_{span} for Rows 1 and 2 are computed as follows:

$$\begin{aligned} Q(1) &= q_s(1) + q_{se}(1) \\ Q(2) &= Q(1) + q_s(2) + q_{se}(2) \\ Q_{span}(1) &= q_e(1) + q_o(1) = 0 \\ Q_{span}(2) &= q_e(2) + q_o(2) \end{aligned}$$

Notice that when $k = 2$, there are only the following three possible sub-partitions: a partition that has $cell_1$ only, $cell_2$ only, or $cell_1$ and $cell_2$. The computation of the true numbers can be computed using Eqn. 3.9 as follows:

$$\begin{aligned} q(1,1) &= q(1) = q_s(1) + q_e(1) + q_{se}(1) + q_o(1) \\ \therefore &= q_s(1) + 0 + q_{se}(1) + 0 = Q(1) \\ q(1,2) &= q(1) + q_s(2) + q_{se}(2) \\ &= q_s(1) + q_e(1) + q_{se}(1) + q_o(1) + q_s(2) + q_{se}(2) \\ &= q_s(1) + 0 + q_{se}(1) + 0 + q_s(2) + q_{se}(2) \\ \therefore &= Q(1) + q_s(2) + q_{se}(2) = Q(2) \\ q(2,2) &= q(2) = q_s(2) + q_e(2) + q_{se}(2) + q_o(2) \\ \therefore &= Q(2) - Q(1) + Q_{span}(2) \end{aligned}$$

Note that the maintained statistics are enough to compute the true number of queries in all possible sub-partitions when $k = 2$. Refer to Figure 3.4 for illustration. For cases $k > 2$, assume that a partition has a split point sp, where $1 <= sp <= k$, that

divides the partition into two sub-partitions, say p_1 and p_2. TrioStat's query statistics can be computed using the Eqns. 3.10 and 3.11 as follows:

$$\begin{aligned} Q(sp) &= \textstyle\sum_{j=1}^{sp}(q_s(j) + q_{se}(j)) \\ Q(k) &= \textstyle\sum_{j=1}^{k}(q_s(j) + q_{se}(j)) \\ Q_{span}(sp+1) &= q_o(sp+1) + q_e(sp+1) \end{aligned}$$

The true number of queries in the partition p_1 is computed using the Eqns. 3.9 and 3.8 as follow:

$$\begin{aligned} q(p_1) = q(1, sp) &= q(1) + \textstyle\sum_{j=2}^{sp}(q_s(j) + q_{se}(j)) \\ &= q_o(1) + q_e(1) + q_{se}(1) + q_s(1) + \textstyle\sum_{j=2}^{sp}(q_s(j) + q_{se}(j)) \\ &= q_o(1) + q_e(1) + \textstyle\sum_{j=1}^{sp}(q_s(j) + q_{se}(j)) \\ \therefore &= 0 + 0 + \textstyle\sum_{j=1}^{sp}(q_s(j) + q_{se}(j)) = Q(sp) \end{aligned}$$

Notice that the computed statistic $Q(sp)$ is exactly equal the true number of queries in $p1$, i.e., $q(p_1)$. The true number of queries in the Partition p_2 is computed as follows:

$$\begin{aligned} q(p_2) = q(sp+1, k) &= q(sp+1) + \textstyle\sum_{j=sp+2}^{k}(q_s(j) + q_{se}(j)) \\ &= q_o(sp+1) + q_e(sp+1) + q_{se}(sp+1) + q_s(sp+1) \\ &\quad + \textstyle\sum_{j=sp+2}^{k}(q_s(j) + q_{se}(j)) \\ &= q_o(sp+1) + q_e(sp+1) + \textstyle\sum_{j=sp+1}^{k}(q_s(j) + q_{se}(j)) \\ \therefore q(p_2) &= Q_{span}(sp+1) + \textstyle\sum_{j=sp+1}^{k}(q_s(j) + q_{se}(j)) \\ &= Q_{span}(sp+1) + \textstyle\sum_{j=1}^{k}(q_s(j) + q_{se}(j)) \\ &\quad - \textstyle\sum_{j=1}^{sp}(q_s(j) + q_{se}(j)) \\ \therefore q(p_2) &= Q_{span}(sp+1) + Q(k) - Q(sp) \end{aligned}$$

Therefore, TrioStat's statistics (Q and Q_{span}) are necessary and sufficient to compute the true number of queries. The same proof can be used to show that TrioStat's

statistics are correct by using the computed cumulative number Q from left to right when dividing partitions vertically.

For TrioStat's Statistic R, notice that R represents the cumulative number of the newly received data points and queries. Thus, the proof of correctness for R is that same as the ones for the data points and the queries proofs explained above. However, in the proofs, $Q_{preSpan}$ is to used instead of Q_{span}, where the former represents the span of only the new queries.

3.3 Estimating the Workload

Periodically, distributed spatial streaming systems evaluate the effectiveness of their partitioning and workload distribution. Hence, they should inform TrioStat by the end of every repartitioning round to prepare the statistics to be used for workload estimations as Section 3.2 discusses. After that TrioStat becomes ready to provide workload estimations in $O(1)$ for partitions, part of a partition, and machines.

According to Eqn. 3.5 and Eqn. 3.7, there is no need to divide by $R(S)$ to compare the workload of partitions and machines because $R(S)$ is a common factor in all equations. Therefore, TrioStat provides the workload estimation for a Partition p to be $W(p) = Num(C(p))$, and the workload estimation for a Machine m to be $W(m) = Num(C(m))$. Let $N(i)$, $Q(i)$, $R(i)$, $spanQ(i)$, and $preSpanQ'(i)$ be the statistics of p at row (or column) index i. Also, let L be the index of the last row (or column) of p's statistics. TrioStat provides the workload estimation of p by using p's statistics as follow:

$$W(p) = N(L) \times Q(L) \times R(L)$$

For example, the workload estimation of Partition p_{11} in Figure 3.5 is, $W(p_{11}) = 10 \times 7 \times 5 = 350$.

TrioStat estimates the workload of the sub-partitions that could result after splitting p into two sub-partitions p_a and p_b. Figure 3.5 shows an example of splitting p_{11} vertically or horizontally. The split point (sp) is the index of the row (or col-

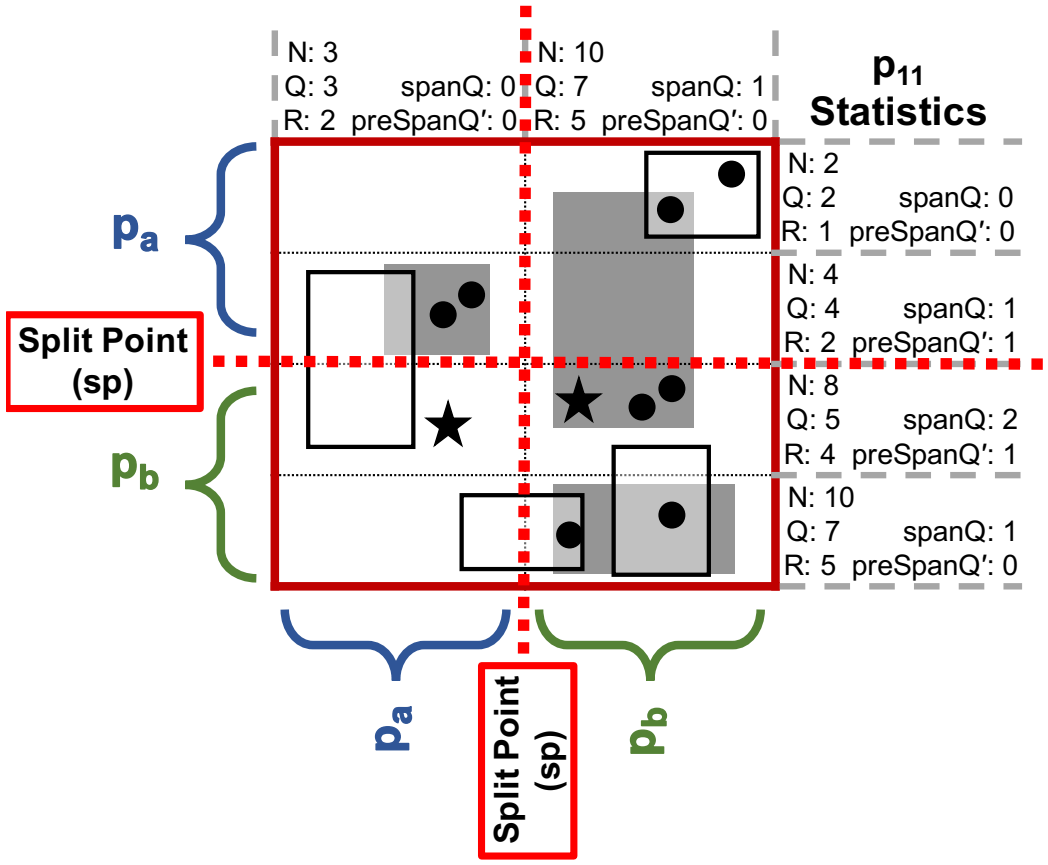

Fig. 3.5. Estimating the workload when partition p_{11} split into two sub-partitions p_a and p_b

umn) where the partition is split. TrioStat uses the maintained statistics directly to estimate the workloads of the sub-partitions as follow:

$$W(p_a) = N(sp) \times Q(sp) \times R(sp)$$

$$Q(p_b) = Q(L) - Q(sp) + spanQ(sp+1)$$

$$R(p_b) = R(L) - R(sp) + preSpanQ'(sp+1)$$

$$W(p_b) = [N(L) - N(sp)] \times Q(p_b) \times R(p_b)$$

For example, when P_{11} is split horizontally on the second row as in Figure 3.5: $W(p_a) = 32$, $Q(p_b) = 5$, $R(p_b) = 4$ and $W(p_b) = 120$. Notice that $W(p)$ is always greater than or equal to $W(p_a) + W(p_b)$ because the queries got distributed

among the sub-partitions. The sum of the sub-partitions' workload estimations is usually smaller than the original partition's workload estimation. This is because the total required number of query checks against every newly received data point is decreased. Moreover, the probability of receiving new objects can be different for each sub-partition. The sum of the sub-partitions' workload estimations can be equal to the original partition's workload estimation only when the split point (sp) cuts all the queries of the original partition and the probability of receiving new objects is equal for both sub-partitions.

TrioStat estimates the workload of a machine $W(m)$ by summing the workload estimations of the partitions that m holds. To compare the machines according to their workloads, the machine that is going to perform the comparison should ask all machines to share their workload estimations. Hence, TrioStat requires minimal network overhead. Moreover, $R(S)$ can be computed in a common machine by getting and summing $R(m)$ of every machine. $R(S)$ is a useful measurement that is used to monitor the throughput of the system because it represents the number of objects (data points and quires) that have been served by the system during the last round of repartitioning.

4. SWARM: ADAPTIVE LOAD BALANCING IN DISTRIBUTED SPATIAL DATA STREAMING SYSTEMS

This chapter addresses the challenges of adaptive load-balancing in distributed streaming systems that processes spatial data. The chapter introduces SWARM, a lightweight adaptivity protocol that continuously monitors the data and query workloads across the distributed processes of the spatial data streaming system, and redistribute and rebalance the workloads soon as performance bottlenecks get detected. SWARM is able to handle multiple query-execution and data-persistence models. A distributed streaming system can directly use SWARM to adaptively rebalance the system's workload among its machines with minimal changes to the original code of the underlying spatial application. SWARM tracks the changes of the workload's spatial distribution based on the workloads estimations provided by TrioStat.

This chapter proceeds as follows. Section 4.1 introduces the architecture of SWARM. Section 4.2 discusses the spatial indexes that are used by SWARM. Section 4.3 explains in details SWARM's decentralised adaptive load-balancing protocol. Section 4.4 presents SWARM's algorithms to find the best way to repartition the workload when re-balancing is needed. Section 4.5 describes how SWARM preserves system integrity while re-balancing the system.

4.1 SWARM Architecture

SWARM works with any distributed streaming system that processes spatial data using a data processing pipeline. SWARM is designed for tuple-at-a-time systems (e.g., Apache Storm [3]) that target milliseconds latency and not for micro-batched systems (e.g., Spark Streaming [5]) that target sub-second latency. SWARM does not require changing the original code of the system's executor machines, e.g., their

indexes, their way of handling data or processing queries. However, the data and the query streams should be redirected to SWARM first. To stress the performance of the system, we assume that the maximum arrival rate of the data stream is higher than the processing capability of the system. This means that the application is trying to fully utilize its machines to process as much data as possible. The main requirement for the data points is to have geo-locations, e.g., Twitter is a good source of geo-tagged tweets that are generated every second. SWARM supports snapshot and continuous queries. A continuous query progressively reports the query results, mainly the data points that satisfy the query's spatial range and its other predicates. Some applications are interested in the recent portion of the data, e.g., the most recent hour. This interest can be expressed as a sliding or a tumbling window. Data expires once it exits the window. SWARM will need to update TrioStat's statistics accordingly.

Figure 4.1 shows the architecture of SWARM and its connections with the distributed streaming system. SWARM is composed of two layers, the routing layer and the execution load-balancing layer. SWARM replaces the partitioning layer of spatial streaming applications (that have a partitioning layer). SWARM is placed on top of the original executor machines $m_1, \cdots, m_n$ and directly receives the incoming streamed data and queries. The routing layer accepts new data points and queries, and routes them to appropriate executor machines. The routing layer has multiple *GlobalIndex* machines to avoid bottlenecks. GlobalIndex machines can communicate with each other and with any executor machine. Each GlobalIndex machine has a spatial grid index that divides the whole space into rectangular partitions. Each executor machine is responsible for one or more partitions. Every new data point, say x, or query is received by only one GlobalIndex machine that uses the index to identify the partition, say p_x, that spatially contains x, then routes x to p_x's executor machine. One GlobalIndex machine, termed the Coordinator, has an additional role other than routing. Section 4.2 explains that further. Having GlobalIndex machines reduce the processing overhead, memory usage, and communication among executor

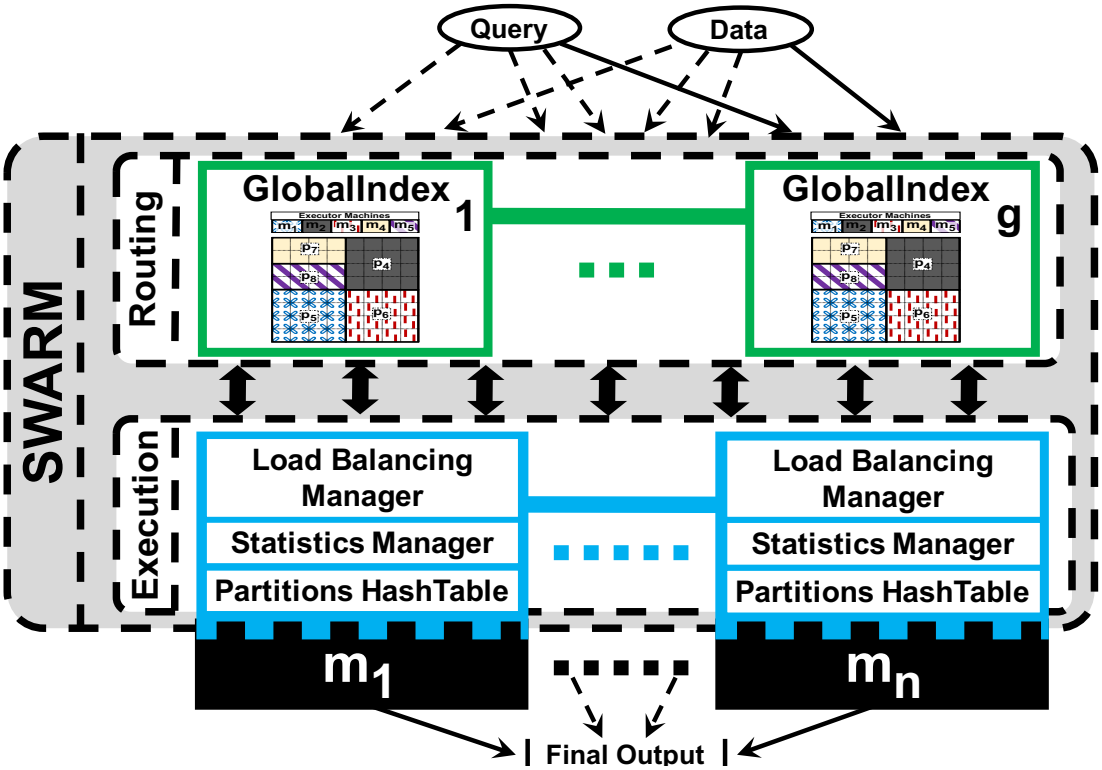

Fig. 4.1. The architecture of SWARM

machines. The reason is that data points and queries of a partition will be localized in one executor machine. A query will not be sent to all executor machines. Also, communication among executor machines to aggregate the results is reduced. SWARM uses GlobalIndex machines to adaptively load-balance the workload among executor machines, as in Section 4.3.

SWARM's second layer contains load-balancing units above the system's original executor machines $m_1, \cdots, m_n$. Each unit has a load-balancing manager, a statistics manager, and a HashTable to index the partitions. Each unit communicates with all other load-balancing and GlobalIndex machines. The unit receives data points and queries only for the partitions that are under its control. SWARM uses TrioStat(introduced in Chapter 3) as its statistics manager. The unit's statistic manager updates its statistics given the new object, as discussed in Section 3.2. Moreover, the

unit uses a HashTable to identify the partition(s), say p, that overlap the received object, as in Section 4.2. Then, the original application code in the executor machine processes this object on p. The load-balancing manager estimates the workload cost of its executor machine periodically using the cost model in Section 3.3 using local statistics. It shares this cost with one GlobalIndex machine. Executor machines with the highest and lowest workloads, say m_H and m_L, respectively, are identified. m_H moves part of its workload to m_L, as explained in Section 4.3 and Section 4.4.

4.2 Indexing and Initialization

SWARM does not require prior knowledge about the distribution of the incoming data or queries. Initially, SWARM divides the whole spatial area evenly among all executor machines. This section introduces the global and local indexes used by SWARM.

4.2.1 The Global Index

SWARM uses a 2D spatial grid index in each GlobalIndex machine to divide the space into grid cells of a predefined size $C_1 \times C_2$ (refer to Figure 4.2). This global index replaces the partitioning index of spatial applications that have a partitioning layer. As in Figure 4.2a, each cell points to a partition that covers this cell. A partition has a unique ID, partition borders, and the ID of the executor machine that handles the partition. Thus, it takes $O(1)$ operations to route an object. Figure 4.2b gives an example for initial configuration of the index in the GlobalIndex machines of a system with 5 executor machines $m_1, \cdots, m_5$ and 5 partitions $p_1, \cdots, p_5$. The patterns (colours) of the partitions link them to the executor machines that control them.

This index routes the received queries and data points to the responsible executor machine(s). However, one of the GlobalIndex machines, termed the Coordinator, has higher responsibilities for load balancing alongside routing incoming objects. Ini-

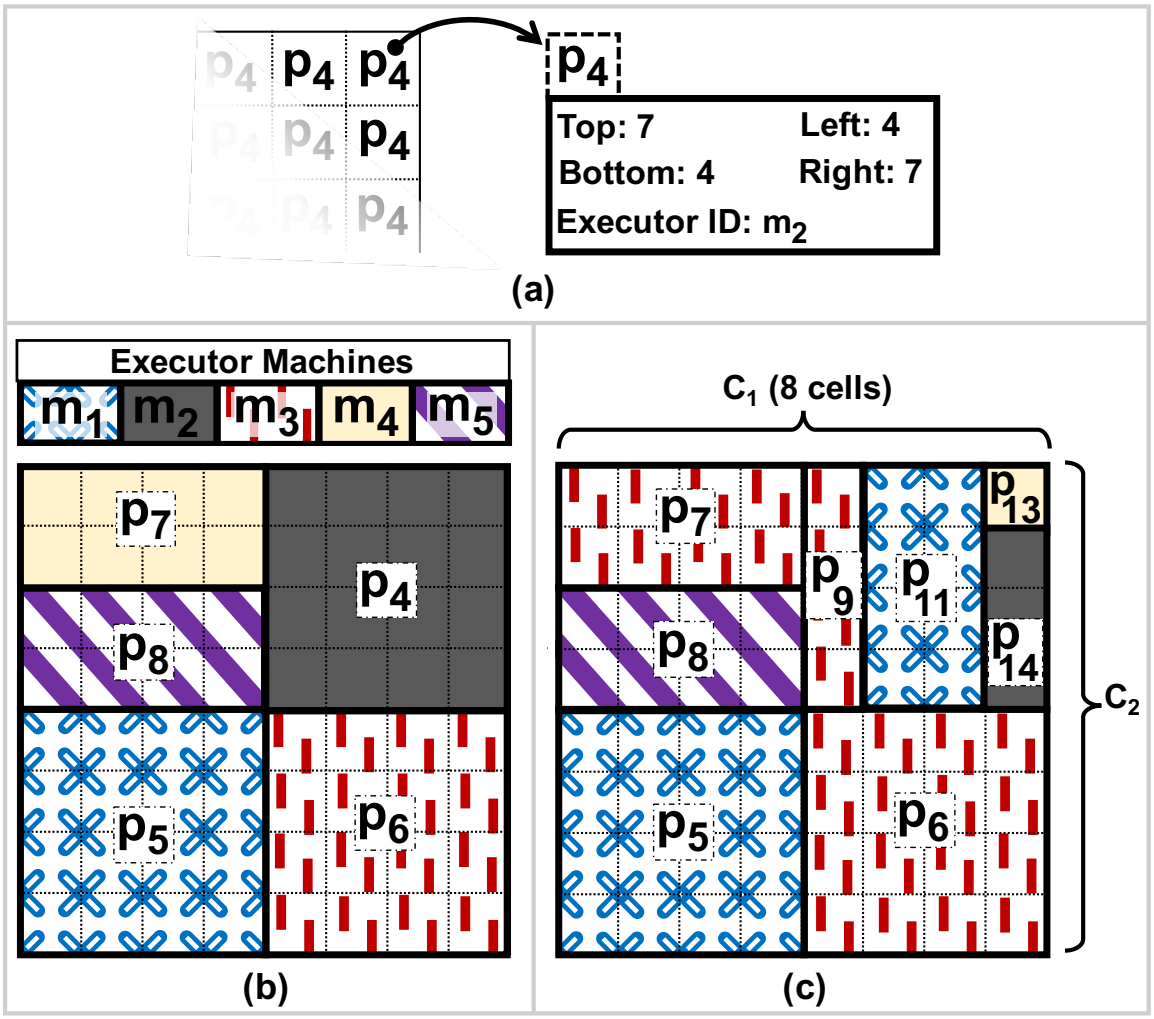

Fig. 4.2. SWARM's index for GlobalIndex machines

tially, the whole space is contained in one partition. The Coordinator creates the initial index by recursively splitting the partition with the largest area into two equal sub-partitions, until each executor machine has one partition. The splitting can be horizontal or vertical depending on which side length is longer. Every time a partition is split, say p_1, the sub-partitions are given new unique IDs, say p_2 and p_3. Splitting of a partition stops if a resulting sub-partition is smaller than a cell. Then, the index is shared with all the GlobalIndex machines. Moreover, the Coordinator sends the information about each partition to the executor machine responsible for that

partition. The Coordinator is also responsible for identifying the two machines with highest and lowest workloads in each load-balancing round. Every round, the Coordinator receives from each executor machine only two numbers that help determine the cost of all executor machines, as explained in Section 4.3. If the Coordinator fails, another GlobalIndex machine takes over as the new Coordinator. This prevents a single point of failure in the system. The decision of choosing a GlobalIndex machine to be the new Coordinator is made by using the Byzantine agreement protocol among all executor machines.

A *hotspot* is a region with a large amount of queries and high probability to receive a lot of new data and queries. Figure 4.2c gives a possible configuration of the index when a hotspot appears in the top-right corner of the space. Each load-balancing round, the Coordinator requests from the machine with the highest cost to move some of its partitions' responsibilities to the machine with the lowest cost. As in Figure 4.2c, the hotspot leads to splitting some partitions and moving others to different machines. An executor machine can be responsible for any number of partitions, e.g., m_3 handles 3 partitions. p_{13} has a hotspot, but cannot be split because its size equals a cell's size. Thus, SWARM has an executor machine (m_4) responsible for only p_{13}. As the hotspot migrates, m_4 might become responsible for other partitions.

Routing a data point is fast as it overlaps only one cell. However, a range query can overlap multiple cells. A naive algorithm for finding which partitions overlap a query can be visiting all cells that overlap this query. Algorithm 2 efficiently determines which partitions overlap a query. The algorithm uses SWARM's index and partition structure.

Algorithm 2 can skip cells by using the partitions' borders. It adds the coordinates of the cell that overlaps the query's top-left corner to the *checkCell* stack. If the cell's coordinates (c) taken from *checkCell* overlaps the query, the cell's partition (p) is added to the *result*. Also, two cells are added to *checkCell*: 1) the one after the right border of p on the same row as c, and 2) the one below the bottom border of

Algorithm 2: queryOverlap(Query q)

```
1  Stack<CellCoordinate> checkCell
2  List<Partition> result
3  indexQuery iq = mapQueryToIndex(q)
4  checkCell.push(CellCoordinate(iq.left, iq.top))
5  while !checkCell.isEmpty do
6      CellCoordinate c = checkCell.pop
7      Partition p = gridIndex[c.x][c.y]
8      if (p not in result) && (c overlaps iq)
9          checkCell.push(CellCoordinate(p.right+1, c.y))
10         checkCell.push(CellCoordinate(c.x, p.bottom-1))
11         result.add(p)
12     end
13 end
14 return result
```

p and on the same column as c. The algorithm recursively takes and adds cells to *checkCell* until the stack is empty.

4.2.2 The Local Index

SWARM adds a local index in each executor machine. Notice that this local index is separate from any other index used in the user's application code. Also, SWARM does not interfere with the application logic in any other way. For example, a user's application for evaluating spatio-textual queries might be using a grid index for maintaining data points or an R-Tree index for storing continuous queries. In other words, user applications can be used as is in executor machines while still using SWARM for load balancing. SWARM's new local index in executor machines receives data points and queries from the GlobalIndex machines. Then, it identifies

the required partition for processing the received data points or queries, and forwards them to the user's application for evaluation.

SWARM adds a HashMap index to each executor machine to easily find the required partition when processing newly received data or queries. Each executor machine uses its HashMap to index the partitions that are under its control. The HashMap key is PartitionID that is the unique identifier of each partition.

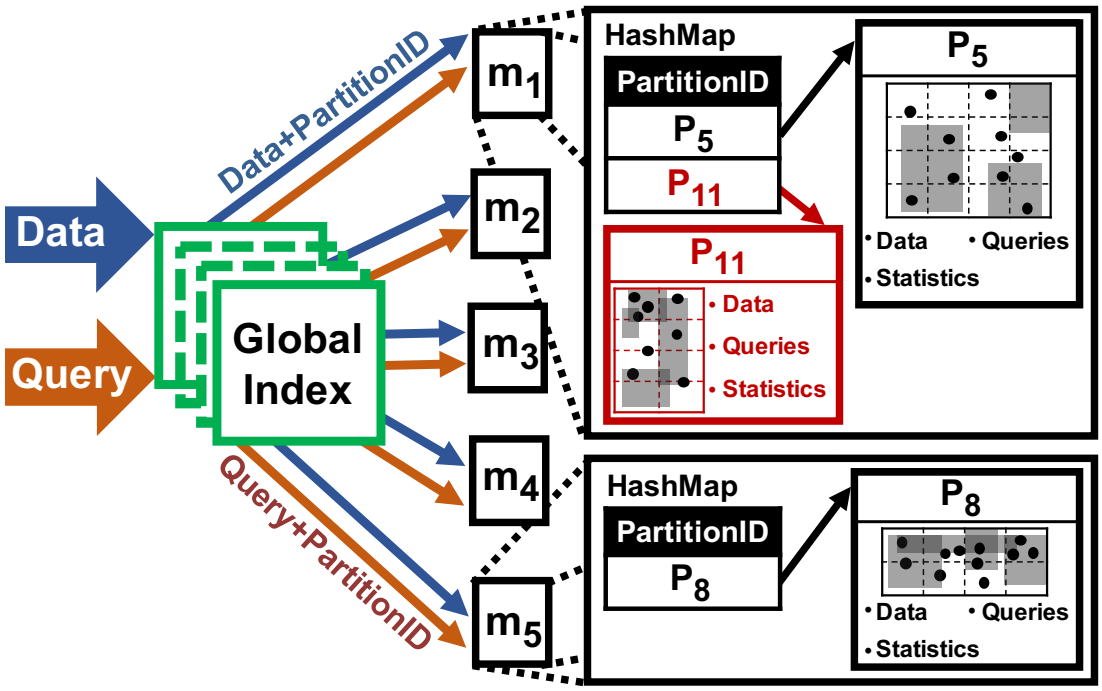

Fig. 4.3. SWARM with the HashMap of m_1 and m_5

Figure 4.3 illustrates the HashMap of the executor machines m_1 and m_5. The GlobalIndex machines attach to every received data point or query the PartitionID of the partition that overlaps it. This received data point or query is routed with the PartitionID to the executor machine that is responsible for that partition. The HashMap of m_5 contains partition p_8 while m_1 has partitions p_5 and p_{11}. Each time an executor machine receives a data point or query from a GlobalIndex machine, it directly uses the attached PartitionID with the HashMap to find the structure of

corresponding partition in $O(1)$. The structure of a partition maintains its metadata (PartitionID, position in space, and size), data points, queries, and statistics. The partition's data points and queries are maintained using the code of the original application.

4.3 Decentralised Adaptive Load Balancing Protocol

SWARM adopts a lazy repartitioning mechanism to balance the workload. It does not over-react to transient changes in the workload so as not to overwhelm the system by excessive load balancing activities. It rebalances the workload only if rebalancing will enhance the system's throughput and reduce its execution latency. Each round of load balancing, SWARM considers only the machine, say m_H, with the highest cost for workload reduction. Moreover, workload reduction can only happen by moving some of the partitions' responsibilities to the machine, say m_L, with the lowest cost.

Most applications that use distributed streaming systems heavily depend on the network bandwidth and connection availability between machines. One of the main objectives of SWARM is to minimize communication. SWARM achieves this objective by breaking the process of load balancing into stages of local decision making. The amount of local computations increases with each stage while the number of machines performing the computations decreases significantly. SWARM uses an asynchronous approach for communicating and applying new partitioning plan among executor machines. Because SWARM uses TrioStat to estimate the workloads, all workload estimations are computed locally using the local statistics in each machine, which minimizes the network overhead.

Figure 4.4 illustrates communications, local computations, and local decision-making between one of the executor machines, say m_i, and the Coordinator. Load balancing is triggered periodically in all executor machines, e.g., every 15 seconds. At the beginning, each executor updates its statistics since the last round of load balancing using Algorithm 1. Then, the numerator part of the cost equation, i.e.,

$Num(C(m_i))$, is calculated using local statistics. $Num(C(m_i))$ is sent to the Coordinator along with $R(m_i)$. The Coordinator computes $R(S)$ using Eqn. 3.4 after receiving $Num(C(m_i))$ and $R(m_i)$ from all executor machines. Then, the Coordinator decides the best load balancing action.

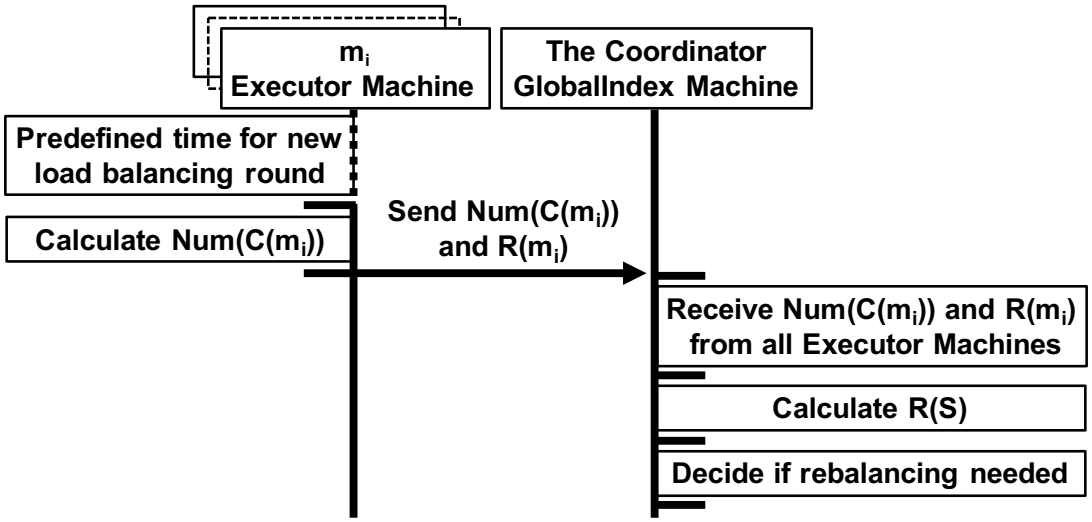

Fig. 4.4. Workflow of load balancing decision

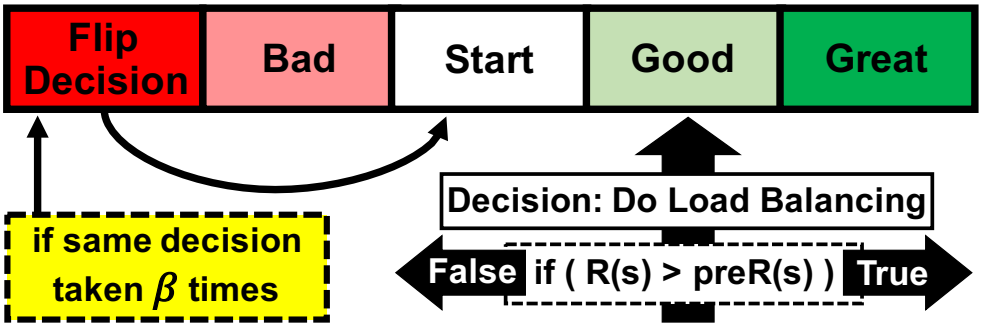

Fig. 4.5. Decision mechanism for load balancing

Load balancing in SWARM is an iterative process. As in Figure 4.5, the decision mechanism has five stages, and has a structure that points to the current stage, and that stores the decision made in the previous round. In each round, the Coordinator

makes one of two possible load-balancing decisions, either to rebalance the workload, or to simply do nothing. The Coordinator applies the previous load-balancing decision each round unless the *Flip Decision* (leftmost) stage is reached. The leftmost stage flips the decision and resets the stage pointer to the *Start* stage. Initially, the *Start* (middle) stage is selected, and the previous decision is set to "Do Nothing".

In every load balancing round, the Coordinator moves the pointer to the right if the throughput has enhanced ($R(S) > preR(S)$). Otherwise, it moves the pointer to the left. Thus, moving to the right indicates that the overall throughput is enhancing and the decision performed in the previous round (iteration) has proven correct. This mechanism insures that the current decision continues to be carried over into future rounds until it is ineffective, and in this case, it is flipped. This avoids over-reacting to the system's transient fluctuations in performance.

When the same decision was taken for β number of times (e.g., 20), the stage pointer is forced to move to the *Flip Decision* stage. This is to avoid sticking with one decision for a long time while this decision is making the system stay in a sub-optimal partitioning plan, e.g., if the Coordinator decides to do nothing each round because the throughput keeps increasing in one round and decreasing in the next. Forcing the system to try rebalancing may lead to better throughput. Otherwise, the decision will be flipped again to do nothing after two rounds. In most situations, SWARM will not stay in the same workload state for a long time because the workload is continuously changing. Hence, the value of β is not critical for the performance of SWARM.

Figure 4.6 shows the workflow when the Coordinator decides to rebalance. It sorts all executors based on their costs, and identifies the machine with the highest and lowest costs, m_H and m_L, respectively. The Coordinator requests from m_H to reduce its workload by migrating portions of it to m_L. This message contains three numbers, the cost of m_L ($C(m_L)$), $R(S)$, and a unique un-used partition ID that m_H can use to create new partitions, if needed. m_H tries to move portions of its partitions to m_L, as discussed in Section 4.4. If m_H finds partitions to move, a new background task

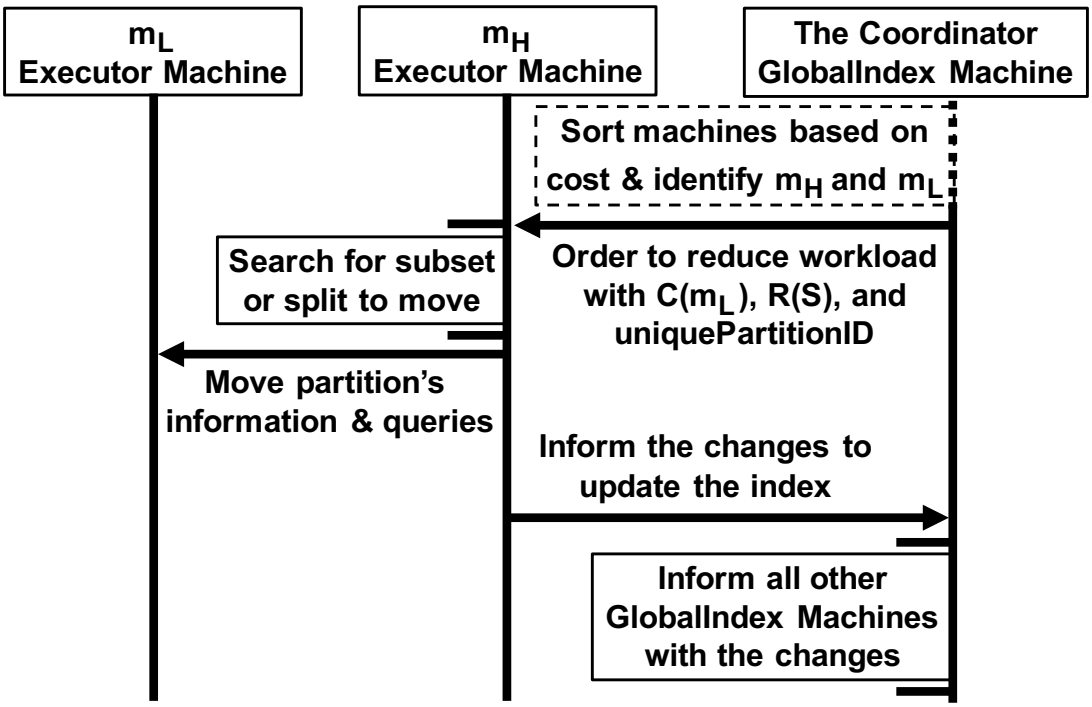

Fig. 4.6. Workflow of rebalancing

is created to send the partitions' information and their continuous queries to m_L. m_L adds these partitions and continuous queries to its workload. After the move, m_H reports the changes to the Coordinator that forwards the changes to all other GlobalIndex machines. They update their indexes using a latch-free background task. The cells that point to the old partition will gradually point to the new partitions that have new unique IDs. This allows the index to concurrently route new data during the update. Section 4.5 discusses how the integrity of data and queries' results is preserved while migrating the workload between machines and while updating the index. If m_H cannot find a feasible workload reduction, it informs the Coordinator. The latter identifies the next highest workload machine and treats it as m_H, and repeats the process.

When the distribution of the workload changes, SWARM might move some old split partitions to different machines. This may result in having an executor machine

controlling partitions that are adjacent, i.e., sharing a boundary. Hence, there is no benefit in keeping them separate. Moreover, they increase the overhead of maintaining separate small partitions. Adjacent partitions are combined using a background task that is triggered occasionally in all executor machines, e.g., every 5 hours. Every executor machine merges any of its partitions that form a connected rectangle. When an executor machine finds a possible merge, it creates the larger partition using a unique unused partition ID that it requests from the Coordinator. Then, the executor machine reports the changes to the Coordinator that, in turn, forwards the changes to all the other GlobalIndex machines.

4.4 Workload Reduction by Repartitioning

m_H can reduce its workload in one of two ways, and applies them in this order: 1) move a subset of its partitions to m_L. 2) split one of its partitions into two, and moves one of them to m_L. First, m_H tries to reduce its workload using the first technique because it requires less overall overhead. If the first technique does not succeed, m_H tries the second technique. The search for workload reduction is performed as a background task.

Let P_{m_H} be the set of partitions that m_H controls. The numerator part of the partitions' cost ($Num(C(p_i))$) is already computed by TrioStat, where $p_i \in P_{m_H}$. From Section 4.3, $C(m_L)$, $R(S)$, and a new unique partition's ID are made available to m_H. The remaining of this section discusses the challenges of each case and proposes efficient algorithms.

4.4.1 Searching for the Best Subset of Partitions to Move

Finding the best subset of P_{m_H} to move means that after moving this subset from m_H to m_L both machines will have approximately equal costs and workloads. Let C_{max} be the maximum cost of partitions that m_H can move to m_L without overloading m_L. Hence, $C_{max} = (C(m_H) - C(m_L))/2$. C_{max} serves as a guide to ensure that the

new workload plan will be better than the current one. Finding a subset of the partitions that their total costs equals C_{max} will result in equal workload for m_H and m_L after moving the subset. m_H searches for the subset that maximizes the total cost of the partitions to be moved without exceeding C_{max}. This is a direct application of the *Subset-Sum Problem* (SSP), which is a special case of the *0-1 Knapsack Problem*, where the value of each item is equal to its weight [75]. In our case, the cost of each partition j is used as the weight of each item j in SSP and C_{max} is used as the capacity of the knapsack. Although SPP is NP-Hard, there is an *Approximate Greedy Algorithm* that guarantees a worst-case performance ratio of $\frac{1}{2}$ with a time complexity of $O(n)$ [75]. SWARM applies the algorithm after sorting the partitions in descending order of their costs. This increases the time complexity to $O(klog(k))$, where k is the number of partitions in m_H. However, sorting can result in better performance on average without affecting the worst-case performance. Moreover, this extra sorting step is necessary for the splitting algorithm that might be applied if a subset is not found. Probing larger partitions first minimizes the number of moved partitions, and hence, reducing the amount of information to be sent through the network.

The procedure used by m_H is presented in Algorithm 3, where C_{m_L} and C_{m_H} are the costs of the machines with the lowest and the highest costs, respectively. m_H calls this algorithm after receiving a request from the Coordinator (with Parameters C_{m_L} and R_S) to reduce m_H's workload. C_{m_H}, P_{m_H} ($controlledPartitionsList$), and the cost of each partition p in m_H (C_p) are available for use in the function.

4.4.2 Searching for Best Split for a Partition to Move

If m_H fails to reduce its workload using the above technique, it tries to make m_H and m_L costs approximately equal using the splitting technique. m_H chooses a partition $p \in P_{m_H}$ and splits it into two sub-partitions p_1 and p_2. m_H calculates the expected cost difference (C_{diff}) between m_H and m_L when p_1 is moved and p_2 is

Algorithm 3: findSubset(C_{m_L}, R_S)

```
1  totalMoveCost = 0
2  moveSubset = empty list of partitions
3  C_max = (C_{m_H} − C_{m_L})/2
4  Sort controlledPartitionsList based on partitions' cost from largest to lowest
5  for each partition "p" in controlledPartitionsList do
6      if (C_p + totalMoveCost <= C_max)
7          totalMoveCost += C_p
8          moveSubset.add(p)
9          if (totalMoveCost == C_max)
10             break
11         end
12     end
13 end
14 return moveSubset
```

kept. m_H tries to find the best split point for p that will make $C_{diff} = 0$. This is an NP-Hard problem. Thus, we use an approximate Greedy Algorithm.

SWARM considers the partition with largest cost (p) for splitting. If m_H cannot split p because it has reached the size of one cell, m_H will try splitting the next largest partition in cost. The list of partitions in m_H can be directly used since it has been sorted during the subset technique. m_H searches for the best splitting point that results in the minimum absolute cost difference C_{diff} between m_H and m_L. Hence, m_H considers all possible vertical and horizontal split lines in p. The algorithm used in SWARM reduces the number of possible split lines by performing a binary search on the rows and columns of the statistics covering p that contain all numbers needed to calculate C_{diff}. C_{diff} is computed as follows:

$$C_{diff} = [(C(m_H) - C(p)) + C(p_2)] - [C(m_L) + C(p_1)]$$

Before the search, m_H already has $C(m_H)$, $C(m_L)$, and $C(p)$. This makes the search depends only on $C(p_1)$ and $C(p_2)$. Let sp be the statistic's row/column index of a split point on p. $C(p_1)$ and $C(p_2)$ can be computed using Eqn. 3.5 and the maintained local statistics in p as follows:

$$C(p_1) = N(sp) \times Q(sp) \times R(sp)/R(S)$$
$$Q(p_2) = Q(p) - Q(sp) + spanQ(sp+1)$$
$$R(p_2) = R(p) - R(sp) + preSpanQ'(sp+1)$$
$$C(p_2) = [N(p) - N(sp)] \times Q(p_2) \times R(p_2)/R(S)$$

The search ends when the algorithm finds a split point that causes $C_{diff} = 0$. Otherwise, the search continues till the end and the split point that achieves minimum absolute value of C_{diff} is used. In the worst case, m_H performs 4 binary searches: two searches on horizontal splitting points while considering the moved partition p_1 to be the upper or lower sub-partition, and two searches on vertical splitting points while considering p_1 to be the right or left sub-partition.

4.5 Preserving System Integrity

4.5.1 Correctness During Load Balancing

SWARM does not stop receiving and processing new data points and queries during the process of load balancing. The critical point of losing a data point or processing a data point twice can happen after identifying m_H and m_L. m_H continues to receive and process new data points and queries while searching for workload reduction. Moreover, if m_H decides to split a partition, it continues to use the old partition while creating the two new sub-partitions. After m_H finds either a subset of partitions or a good split, m_H sends the metadata of the moved partition/s and their continuous queries to m_L. After the move, m_H informs the Coordinator about the changes. Then, the Coordinator informs the remaining GlobalIndex machines. Whenever a GlobalIndex machine receives the new changes from m_H, it runs a latch-

free background task that updates the index according to the changes while using the index for routing, as discussed in Section 4.3. During the update of GlobalIndex machines, m_H forwards new incoming objects that overlap the moved partition/s to m_L. m_H keeps the metadata of the moved partition/s until their data are expired and the next load balancing round starts. Starting a new load balancing round implies that all GlobalIndex machines have finished updating their indexes, i.e., GlobalIndex machines route all new objects that overlap the moved partition/s to m_L. This mechanism ensures that no objects get lost or processed twice during load balancing.

4.5.2 Correctness of Query Execution

Most applications of distributed streaming systems are focused either on the current state or some limited extended state of the data. Limited extended state could be based on a time window (e.g., sliding or tumbling window) or based on data item count (count window) or some storage size (e.g., predefining the size of stored data in the window). Every distributed streaming system has a specific form of data expiration policy. Because data will eventually expire, **SWARM reduces communication overhead by not moving data**. SWARM needs to know whenever data points are expired to stop tracking on which machine they are stored and to update the statistics. SWARM moves the partitions with only their continuous queries. In applications that support stateful operators (e.g., aggregate operators), the state is stored in the query not in the partition. Hence, SWARM moves the queries and their states with the migrated partition to their new executor before redirecting the stream.

As partitions are split and are moved to other machines, SWARM keeps a record in the metadata of every sub-partition that links the sub-partition to its previous responsible machine and its parent partition. Before moving a partition to m_L, m_H adds its machine ID as the previous responsible machine and the previous $PartitionID$ as the parent partition to the metadata of the moved partition. m_H continues to hold parent partitions and their data until all data become expired. Splitting or moving a

partition multiple times before the data expires might lead to a chain of partitions, where each of them is linked to the previous one. Mostly, the chain of partitions will remain short because In-memory systems tend to support short windows that make data expire quickly.

A query may only need a subset of the chain of partitions to be involved in the final result. When a partition, say p, is to answer a query q, the machine responsible for p will check if its parent partition, say p_p, exists and needs to be involved. If p_p is found, its responsible machine is asked to process q. All involved machines send their answers directly to the machine that has q, say m_q. m_q waits for the next involved machine in the chain to send the results of q. Every involved machine consults the next involved machine in the chain to answer q. Depending on whether an involved partition is expired or not, there are two ways to respond: 1) If the partition is expired, its machine acknowledges m_q and the previous involved machine in the chain that the partition is expired. Hence, the previous involved machine in the chain breaks the chain by cleaning the record of the previous responsible machine in the metadata of the partition and becoming the last machine in the chain. 2) If the partition is not expired, q's results are sent to m_q. Every involved machine sends an acknowledgement message to m_q after it is done sending q results. Acknowledgement messages contain the number of result messages that were sent and the next involved machine ID in the chain. m_q keeps track of all involved machine IDs and the status of their results. m_q produces the final output after receiving all result messages and acknowledgement messages from every involved machine. To produce the final output, m_q combines all received answers with the answers from its partition that overlaps q.

For example, assume that a partition p_1 in executor machine m_1 is split into p_2 and p_3. m_1 adds to the metadata of both p_2 and p_3 the previous responsible machine m_1 and the parent partition p_1. m_1 keeps p_2 and moves p_3 to m_2. Now, assume that m_2 receives a query q related to p_3, and it needs old data. m_2 finds that the previous responsible machine in the metadata of p_3 is m_1. Therefore, m_2 sends q to m_1 with p_1's ID as the target parent partition. If p_1 is not expired, m_1 applies q on

p_1, and sends the results to m_2 and an acknowledgement message. Otherwise, m_2 acknowledges that p_1 is expired to break the chain of partitions associated with p_3. After receiving the acknowledgement, m_2 produces the final results by combining p_3's results with any received results.

5. GUARD: DETECTION AND RESPONSE FOR ATTACKS TARGETING ADAPTIVE LOAD BALANCING IN DISTRIBUTED STREAMING SYSTEMS

This chapter reveals a new type of attacks that forces adaptive load-balancing mechanisms of distributed streaming systems into a continuous state of rebalancing. Furthermore, the chapter proposes Guard, a solution that detects and responses to this new type of attacks. Guard is general as it does not depend on a specific adaptive load-balancing mechanism nor a specific distributed streaming application. Guard requires minimal changes form the distributed streaming application. Guard uses an unsupervised machine learning technique to separate malicious users from normal users based on their behaviors. Guard does not block users until it is certain that they are malicious.

The rest of this chapter proceeds as follows. Section 5.1 presents the attack model on adaptive load-balancing mechanisms. Section 5.2 introduces Guard's architecture and its main component. Section 5.3 presents Guard's collected features of the users. Section 5.4 discusses Guard's unsupervised detection mechanism. Finally, Section 5.5 presents Guard's response mechanism.

5.1 Malicious Attacks on Adaptive Load Balancing in Distributed Streaming Systems

The performance of a distributed streaming system is directly affected by how balanced the workload among its machines. Data and query workload of distributed streaming systems change rapidly. Moreover, the distribution of data and queries are skewed, and this skewness changes with time and user interest. The usage of static load-balancing techniques does not make the system fully utilize its machines, and this

leads to low throughput and high response time. Therefore, various approaches have been proposed to adaptively load balance distributed streaming systems according to data and query workload, e.g., STAR [42], Tornado [13], Ameoba [41].

Although using adaptive load-balancing techniques significantly improves the performance of distributed streaming systems, they make the system vulnerable to attacks. Attacks can be initiated using the knowledge that the system is using an adaptive load-balancing technique to redistributes workload across the machines base on changes of the workload. This type of attacks limits the availability and the throughput of the system. Another objective of the attack can be to draw the attention of the system from focusing on serving real events. A different type of attacks on adaptive load-balancing mechanisms tries to leak protected information to malicious users. The next section reviles the attack model that can be initiated on the adaptive load-balancing mechanisms of distributed streaming systems.

5.1.1 The Attack Model

Adaptive load-balancing mechanisms in distributed streaming systems strive to maintain a high level of availability. In this chapter, we consider a new type of attacks that aims to limit the availability of distributed streaming systems. The essence of the attack is to force the load-balancing mechanism into a continuous state of rebalancing. In particular, an attacker can trigger multiple rebalancing operations by submitting a carefully designed sequence of queries that create malicious hotspots. In addition to reducing the system throughput and availability, this type of attacks can divert the system from serving real major events. Moreover, the attack can be concentrated to make the system leak protected information.

To illustrate the attack, consider the following scenario in Figure 5.1. Initially, there are 5 registered continuous queries (Q_1, Q_2, Q_3, Q_4 and Q_5) that are distributed among three machines as illustrated in Figure 5.1(a). Assume that this query distribution among three machines, m_1, m_2, and m_3, is balanced. In this attack, the

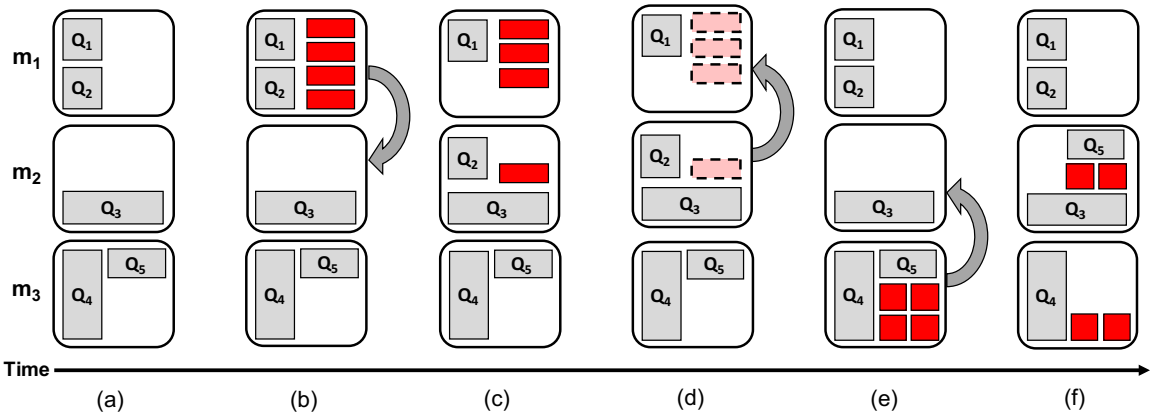

Fig. 5.1. Example of attack model on a distributed streaming system that contains three machines.

attacker can access the data stream that is being processed by the system. Moreover, the attacker knows the data dimensions that are used to partition and distribute the workload in the system. The dimensions can be speculated from the queries that the application supports, e.g., spatial location, text topics, or hash values for more general applications.

In Figure 5.1(b), the attacker submits four malicious continuous queries (in red) to m_1. Each malicious query is submitted once and is required to be checked against every new data tuple that it overlaps with. As a result, the system detects that m_1 becomes overloaded, and decides to rebalance the workload distribution by migrating Q_2 and one of the malicious queries to m_2 (Figure 5.1(c)). Thus, the attacker has successfully created a malicious hotspot that has triggered a rebalance operation by submitting malicious queries. The effectiveness of the malicious hotspot can be strengthen by choosing a specific key that has a high probability to overlap with large amount of data. For example, in a spatial application, asking about different restaurants in Chicago have higher probability for creating a hotspot than asking about restaurants in the middle of the ocean.

Next, the attacker terminates the malicious continuous queries as shown in Figure 5.1(d). In this case, the system has to migrate Q_2 back to m_1 to restore the

balanced state. Note that this attack has resulted in triggering two rebalance operations. If both migrated queries were malicious at Figure 5.1(c), the system would still be in a balanced state when the attacker terminates the queries. Moving on, the attacker submits four more queries in Figure 5.1(e) that in turn result in one more rebalance operation (Figure 5.1(d). As long as the attacker continues to succeed in creating malicious hotspots, the system wastes its processing cycles continuously rebalancing. Similarly, this attack can be performed by submitting snapshot queries instead of continuous queries. However, the successful creation of malicious hotspots requires submitting a large number of snapshot queries with a high rate. Therefore, creating malicious hotspots by submitting snapshot queries requires consuming higher amounts of resources from the attacker side than by submitting continuous queries. In contrast, it is easier to detect the attack that sends a large number of queries with a high rate.

This attack can be also used towards data exfiltration. In particular, an adversary can create a malicious hotspot in a region that he/she does not have access permission to. The load-balancing mechanism is oblivious to the access-control mechanism. The load-balancing mechanism redistributes the data and queries by moving some of them to another machine. Meanwhile, the adversary can perform eavesdropping attack (sniffing or snooping) on the network to exfiltrate the data.

The attack can be performed by submitting malicious queries from one machine (user) or multiple machines (users). In the case of using one user, it is going to be easier to detect the attacker because this user will have activity that is much higher than normal users. On the other hand, the total activity of the attack can be hidden by initiating a coordinated and distributed attack where every involved malicious user contributes a little to the attack. Hence, the activity of malicious users becomes similar to the activity of normal users. However, collectively, this group of users perform a distributed malicious attack that is harder to detect.

5.2 Guard Architecture

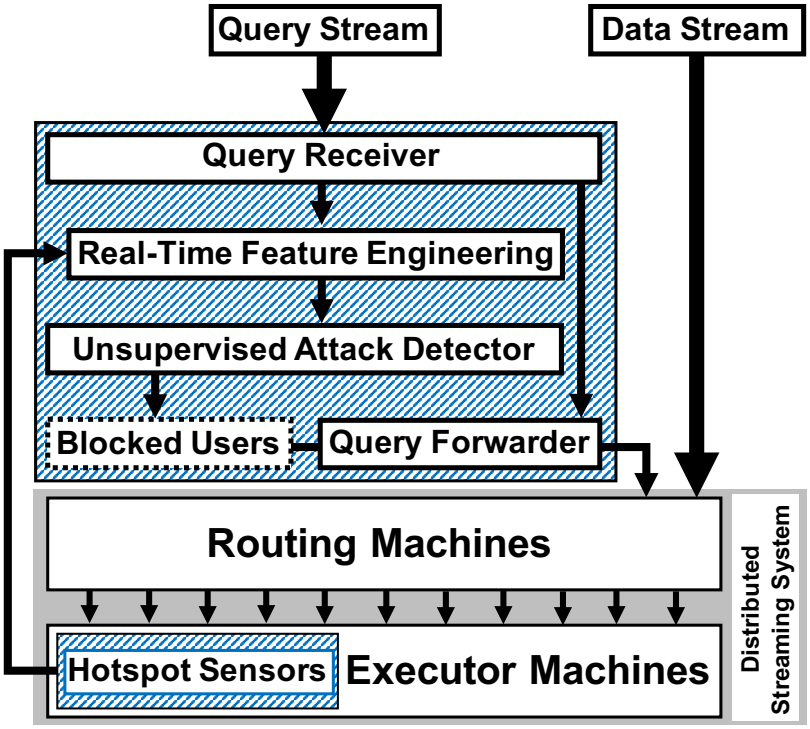

Fig. 5.2. Architecture of Guard and its connections with the distributed Streaming System

Guard is our proposed solution to detect and respond to attacks that aim to affect the performance of distributed streaming systems through deceiving their adaptive load-balancing mechanisms. Guard is composed of two components that are illustrated in Figure 5.2 in a blue striped pattern. Guard's first and main component is a separate unit that contains all the detection and response processes. The second component, termed, *Hotspot Sensor*, is located inside every executor machine of the distributed streaming application. *Hotspot Sensors* sense that a hotspot has triggered the rebalancing mechanism, and collect raw information about the hotspot's queries. Then, *Hotspot Sensors* send the hotspot's raw information to the main component to be analyzed.

Guard is general in the sense that it does not require changes to the load-balancing mechanism. Guard requires the following changes to the original code of the distributed streaming application: 1) Redirecting the query stream to be sent to the main component of Guard. 2) Adding the *Hotspot Sensors* to every executor machine. 3) Calling the *Hotspot Sensors* before rebalancing, and giving them access to the partition containing the hotspot. 4) Whenever rebalancing happens, moved information should be encrypted to prevent data exfiltration.

Guard's detection mechanism is triggered periodically every detection round to check if the system is under attack. A short detection round introduces higher overhead on the system but it reduces the time to stop an attack as soon as one happens. However, Guard requires the detection round to be long enough to allow the adaptive load-balancing mechanism to identify more than one hotspot during the detection round and re-balance the workload accordingly. For example, the detection round in our experiments is one minute, and it allows having a maximum of three hotspots to be re-balanced.

Guard maintains two counters, namely *RoundID* and *HotspotID*. *RoundID* is a global serial number that uniquely identifies a detection round. *RoundID* is used to track at which round every query were requested. Gaurd increments *RoundID* once a detection round ends. *HotspotID* is a global serial number that uniquely identifies the next hotspot that will trigger the adaptive load-balancing mechanism to redistribute the workload. *HotspotID* is initialized to 1 and it is incremented by one every time a hotspot's raw information is received from *Hotspot Sensors*.

Refer to Figure 5.2 for illustration. Guard's main component is composed of multiple processes. The *Query Receiver* receives every query request in the query stream, e.g., newly issued snapshot queries, requests to register new continuous quires, and requests to terminate old continuous quires. The *Query Receiver* attaches to every query request the current *RoundID* and a *UserID*. The *UserID* is an identification for the user's machine that submitted the query request. The IP address of the machine submitting a query request is used as the *UserID* for the request. The

Query Receiver passes the queries to the *Real-Time Feature Engineering* and the *Query Forwarder*. The *Real-Time Feature Engineering* is the process that builds the feature space in real-time by collecting and analyzing raw information from the *Query Receiver* and the *Hotspot Sensors*. At the end of every detection round, the *Real-Time Feature Engineering* finalizes the features and passes them to the *Unsupervised Attack Detector*. The *Unsupervised Attack Detector* is the main process that finds if there is an attack on the system and detects the malicious users involved in the attack. The *Unsupervised Attack Detector* uses an unsupervised machine learning technique to cluster users based on their behaviors, their interactions with each other, and their relationships with the created hotspots. Then, it uses a rule-based technique to decide if there is a cluster involved in an attack on the load-balancing mechanism of the system. The *Unsupervised Attack Detector* stores malicious user IDs in a hash table, called *Blocked Users*. The *Query Forwarder* forwards the queries that it gets from the *Query Receiver* to the routing machines of the distributed streaming system. However, it only forwards user queries that are not in the *Blocked Users* hash table. The *Query Forwarder* distributes the query messages among the routing machines by sending them in a round-robin fashion.

5.3 Real-Time Feature Engineering

Guard's unsupervised detection technique relies on a set of features representing the users of the system. These features are engineered to define a space that enables separating normal users from the malicious ones. As time-series, Guard collects measurements of these features periodically every detection round as a tumbling window [76, 77]. Guard maintains the features in a hash table with *UserID* as the key. Table 5.1 lists all the features used by Guard, along with a brief description of each feature. We categorize these features into two groups according to the source of the data used to measure them: features from raw queries and features from *Hotspot Sensors*.

Table 5.1.: Description of the features used by Guard

ID	Feature Name	Description
x1	New_Queries	Number of requested queries by the user during the current round
x2	Deleted_Queries	Number of deleted queries by the user during the current round
x3	LRB_Involvement	Number of hotspots that have triggered **L**oad **R**e-**B**alancing during the current round and that contain at least one of the user's queries
x4	LRB_Contribution	Number of user's queries found in any hotspot that has triggered re-balancing during the current round
x5	Hotspot_Seq	Sum of *HotspotID*s that the user has been involved in during the current round. This number is a compressed representation for the sequence of hotspots that the user has been involved in
x6 - x10	Historical features	Similar to x1 - x5 but with a time fading function that captures the measurements of previous rounds
x11	Hotspot_Seq_New	Sum of *HotspotID*s that the user has been involved in during all rounds with queries requested in the same round as the hotspot creation round

Continued on next page

Table 5.1.: continued from previous page

ID	Feature Name	Description
x12	LRB_Weighted_Contribution	Similar to x4 but it is calculated using weights of queries based on how old they are. Queries requested in older rounds have lower weights
x13	LRB_Queries_Rate	The percentage of weighted user's queries found in hotspots that triggered re-balancing during the current round (x12) out of the faded number of all user's queries that have been requested during recent rounds (x6)
x14	Avg(LRB_Queries_Rate)	The average of x13 values of the user, for all the rounds that have a value different than zero

The first subset of the features is engineered from the queries requested from the system. These queries are forwarded to the feature-engineering process by the *Query Receiver*. Guard measures these features as statistics for each user in a given detection round. These features are calculated by counting the number of new requested queries and the number of terminated queries for each user.

The second subset of features is engineered from the information collected by the *Hotspot Sensors*. These sensors collect information about the users who have queries in the partition that contains the hotspot. Then, the sensors forward the collected information to the *Real-Time Feature Engineering* that uses the data to generate hotspot-related features. Collecting information about hotspots by *Hotspot Sensors* is explained in Section 5.3.1

Most of the significant features are hotspot-related features. These features are designed to enable separating normal users from both *single- and multi-user coordinated attacks*. For example, measuring the number of times the user's queries appear in hotspots that have triggered rebalancing is useful to detect single-user attacks. On the other hand, Guard utilizes the engineered features Hotspot_Seq and Hotspot_Seq_new in Table 5.1 to reduce the distance between users involved in a multi-users coordinated attack. Their numbers are compressed representations for the sequence of hotspots that the user has appeared on. Every time the queries of the user appear in a hotspot, the *HotspotID* is added to the previous value of Hotspot_Seq. Hotspot_Seq_new is calculated in a similar way but it only gets updated if at least one of the queries is requested during the same round of the hotspot. These features allow observing similarities among users according to their behavior of causing load rebalancing.

Feature x12 in Table 5.1 is important to differentiate between the behavior of normal users and malicious users in relation to hotspots. It represents the weighted number of user's queries found in any hotspot that have triggered rebalancing during the current round. The weight of a query depends on when it has been requested. Queries requested on older rounds have lower weight. Let Q_t be the number of users' queries found in hotspots during the current round t. Let $t-i$ be the i^{th} round before t. Feature x12 is calculated using the following equation:

$$\begin{aligned} \text{Feature x12} &= Q_t + \frac{Q_{t-1}}{2} + \frac{Q_{t-2}}{4} + \frac{Q_{t-3}}{8} + \frac{Q_{t-4}}{16} \\ &= \sum_{i=0}^{4} \frac{Q_{t-i}}{2^i} \end{aligned} \tag{5.1}$$

Notice that the rounds used in Equation 5.1 are limited to the most recent five rounds to reduce the network overhead. Moreover, queries of older rounds have a small weight because their denominator increases exponentially. Therefore, if queries of older rounds are to used in Equation 5.1, their weights will not have a big effect on the value of Feature x12. Equation 5.1 uses a weighting method similar to the fading

method that is explained in the next paragraph. Hence, Feature x13 in Table 5.1 is computed by dividing x12 by the faded number of new queries (x6).

Features x1 to x5, x12, and x13 in Table 5.1 are measurements for the current detection round. The remaining features in Table 5.1 are historical features that capture the measurements of the previous rounds. The historical features x6 to x10 are designed to have a fading effect that signifies recent behaviors over older behaviors. The fading is applied by halving the measurement of the previous round and adding it to the current round's measurement. The purpose of the historical features is to increase the robustness of the users' representation by including their behaviors in previous rounds. Feature x11 is an accumulated number. Feature x14 is an averaged number over all rounds that the user has been involved in their hotspots. Feature x14 is useful to detect abnormal behaviors. It is important to view Feature x14 over previous rounds to examine if the abnormality persists or if it is just noise.

The last step performed in the feature engineering process is *feature normalization*. This step is essential to prepare the features for Guard's *Unsupervised Attack Detector* by ensuring equal ranges for the features. Guard uses min-max normalization [78] to transform the features to the range from 0 to 1, refer to Equation 5.2. The normalized features are passed to the *Unsupervised Attack Detector* as a multi-dimensional array.

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{5.2}$$

5.3.1 Collecting Raw Information about Hotspots

For every hotspot, *Hotspot Sensors* create a summary that includes *UserID*s of the users that have queries in partitions containing the hotspot and a list attached to every *UserID*. As mentioned in Section 5.2, Guard attaches to every query a *RoundID* to determine the round of when the query was requested. The attached list breaks down the count of the user's queries that have been found in the hotspot based on the queries' *RoundID*s. The attached list includes the count of the queries that have

been requested during the latest five rounds only. Hence, users that have requested all their hotspot queries in rounds older than five have an empty list. Limiting the number of rounds to five is chosen to reduce the overhead of sending large summaries. Moreover, the count of queries that are older than five rounds do not have large effect on the features that are calculated based on this list, as discussed in Section 5.3.

Guard's main component maintains the true current *RoundID*, as shown in section 5.2. However, e very *Hotspot Sensor* maintains its own current *RoundID* that gets updated based on the *RoundID* of the newest query that the *Hotspot Sensor* reads. Therefore, Guard does not add an overhead to the system to synchronize Guard's main component and all *Hotspot Sensors*. The summary gets filtered while being sent to the *Real-Time Feature Engineering* to include only the five most recent rounds based on the *Hotspot Sensor*'s local current *RoundID*. While receiving the summary, the *Real-Time Feature Engineering* discards the information of extra rounds based on the true current *RoundID*.

5.4 Unsupervised Attack Detector

Guard's *Unsupervised Attack Detector* gets triggered periodically by the end of every detection round. The detector learns how to separate malicious users from normal users in an unsupervised manner, i.e., without the need to be trained with a pre-labeled data. The *Unsupervised Attack Detector* clusters users based on their behaviors and interactions using the K-Means++ algorithm [79]. K-Means is a well-known unsupervised clustering technique [80] that partitions n data points into a desired number of clusters (K). The centroids of these clusters, i.e., initially, the K means are assigned randomly, then they are adjusted iteratively according to the available data. Subsequently, the data points can be clustered according to their distances from the K centroids. K-Means++ is an implementation of the traditional K-Means with a fast technique to choose the seeds, i.e., the initial centroids that, on average, can produce more accurate results compared to other K-Means implementations.

Algorithm 4: attackDetector(Users_Features U)

```
1  Cluster SC = All_Users(U)                          ▷ Suspicious Cluster
2  Cluster[2] C
3  do
4      C = K-Means++(SC, 2)                           ▷ Cluster SC into 2
5      if ALQR(C[0]) > ALQR(C[1])
6          SC = C[0]
7      else
8          SC = C[1]
9      end
10     sd = StandardDeviation(SC.Hotspot_Seq_New)
11 while sd is large
12 Cluster N = All_Users(U) − SC                      ▷ Normal Cluster
13 if ALQR(SC) > ALQR(N)×β
14     if SC involved in hotspots during current round
15         if SuspiciousClustersList.contains(SC)
16             BlockedUsers.addUsersOf(SC)
17         else
18             SuspiciousClustersList.add(SC)
19         end
20     else
21         No Attack
22     end
23 else
24     No Attack
25 end
```

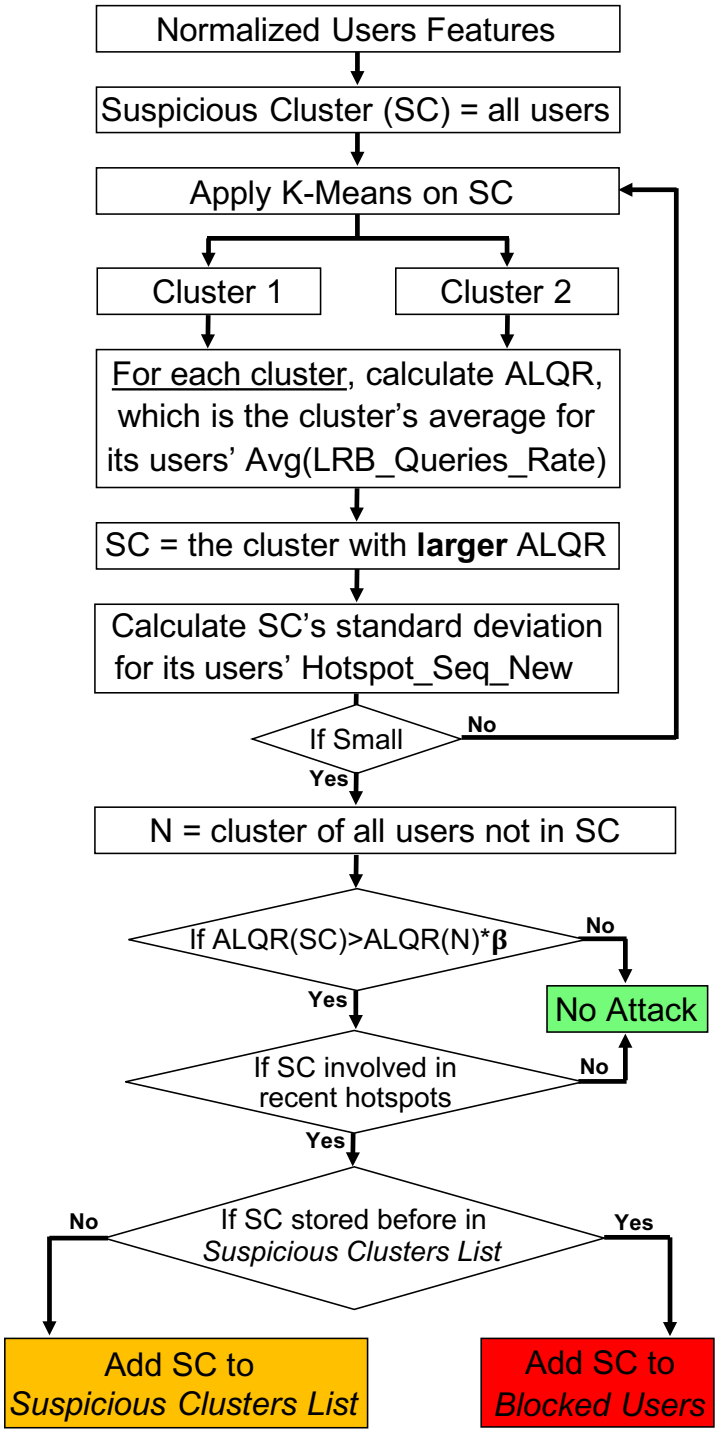

Fig. 5.3. Flow chart for Guard's unsupervised attack detection mechanism

Algorithm 4 lists the pseudocode for Guard's unsupervised attack detection mechanism. Moreover, Figure 5.3 illustrates the internal processes of the algorithm in a flow chart. The detection mechanism starts every time the *Real-Time Feature Engineering* passes the normalized features of the users to the detection mechanism. Guard creates a cluster, termed the *Suspicious Cluster (SC)* that initially includes all users with their normalized features. Guard clusters SC into two clusters using the K-Means++ algorithm, where its K is equal to 2, and it uses the Euclidean distance. The anomalous behavior of attackers should be detected in a variety of normal behaviors that are not predictable with a particular training set, and this prevents the applicability of using supervised machine learning techniques. The behavior of normal users is always changing due to the dynamic workload of real-time streaming applications. For example, the normal behavior of users during a worldwide event is different than when there is only a local event. Worldwide events can lead to higher activity rates of data, queries, and hotspots.

After SC is divided into two clusters, one of them contains only normal users while the other cluster requires further investigation. To differentiate between the two clusters, Guard uses a feature (ALQR) for clusters to measure the collaborative intention of the cluster's users to create hotspots with their new queries. ALQR of a cluster is computed by calculating the average for the Avg(LRB_Queries_Rate) of the users in the cluster. Feature Avg(LRB_Queries_Rate) is explained in Section 5.3. ALQR of a cluster is the cluster's average for its users' percentage of weighted queries that are found in hotspots over all queries requested during recent rounds. Recall that queries being requested in older rounds have lower weights.

To identify the cluster that requires further investigation, Guard computes ALQR of each of the two clusters that K-Means++ produces. For a cluster to have a small ALQR, it means that the users of this cluster do not intend to create hotspots with their new queries. Hence, the cluster that has smaller ALQR contains normal users. Therefore, Guard updates SC to be the cluster with the larger ALQR. Then, Guard calculates the standard deviation for the Hotspot_Seq_New feature of SC's users. The

standard deviation measures the similarity of SC's users in their involvement in the same hotspots. Having a small standard deviation (near zero) indicates that SC's users has been involved many times in the creation of the same hotpots. On the other hand, a large standard deviation means that not all of the users in SC are coordinating to create hotspots. When SC's standard deviation is large, SC needs to be clustered again using K-Means++ as before. Re-clustering continues until the standard deviation becomes near zero. If re-clustering continues until the standard deviation reaches exactly zero, it will ensure that all the users in the final SC are involved in the exact hotspots and no normal users will be blocked by mistake. However, this might make Guard block portion of the malicious users during one round while leaving the remaining to be blocked in a future round. This can happen when malicious users target a spot where a genuine hotspot is already going to form. The hotspot might get created fast, even before all coordinated malicious users contribute to its creation.

Multiple re-clusterings can be required in some situations when the behavior of malicious users is similar to some normal users and more than one group represent the behavior of normal users. In these situations, having K-Means++ to produce two clusters will result in having normal users in one of them and a mix of normal and malicious users in the other cluster. The best K parameter for the K-Means cannot be known beforehand because it changes continuously as a result of the changes in the real-time workload and the change in the behavior of users. To overcome this issue, Guard uses the recursive way of re-clustering the users into two clusters, where every time one of the clusters can be excluded because it contains only normal users.

After the detection algorithm exits the recursive clustering phase, SC will contain the users that might be malicious. Guard creates a new cluster (N) that contains all the users except the ones that are in SC. Therefore, N contains only normal users. ALQR of N is computed to represent the average behavior of normal users in relation to hotspots. Unlike the query activities of normal users, a large percentage of malicious users' query activities are found in hotspots. Thus, ALQR of a cluster

containing malicious users is larger than ALQR of normal users. $\boldsymbol{\beta}$ is a constant factor that represents the allowed distance from the average behavior of normal users (ALQR(N)), so that a user is considered a normal user. When ALQR(N) times $\boldsymbol{\beta}$ is larger than ALQR(SC), it means that the behavior of the users in SC is within the allowed distance from the average behavior of normal users. Thus, users of SC are normal and there is no attack. Otherwise, it is still possible that SC contains malicious users. Guard blocks the attacks faster for lower $\boldsymbol{\beta}$ assignments. However, there is a risk to falsely consider some normal users as malicious when $\boldsymbol{\beta}$ is small. Notice that ALQR of the cluster that contains the malicious users increases each time they create a malicious hotspot. Therefore, the attacks will eventually be blocked regardless of the value of $\boldsymbol{\beta}$. Empirically, we find that setting the value of $\boldsymbol{\beta}$ around 3 is a good rule of thumb.

The next check performed on SC is to make sure whether or not its users are involved in a hotspot during the current detection round. If they are not involved, then there is no attack. Otherwise, SC is confirmed to be a cluster of suspicious users. Guard will either add SC to the *Suspicious Clusters List* or confirm that its users are malicious if the exact cluster has been added before to the *Suspicious Clusters List* in a previous round. The users that have been confirmed to be malicious are added to the *Blocked Users* hash table. If any cluster in the *Suspicious Clusters List* does not become suspicious again by a specific number of detection rounds, it gets removed form the list. A different random removal time is attached to every cluster in the *Suspicious Clusters List*. The lower limit for the randomly generated removal time should allow at least 5 detection rounds to pass. The attack is guaranteed to fail when it does not harm the system by creating consecutive hotspots with a long time between them. The upper limit for the randomly generated removal time should be a long time that guarantees the failure of attacks. For example, the detection round lasts for 1 minute in our experiments, while the removal time is generated randomly between 5 and 30 minutes. If the attacker waits 30 minutes between the creation of consecutive malicious hotspots, the system will not stay in a continuous

state of rebalancing and the attack will fail. This randomization makes the detection mechanism resilient for the attack to bypass because it makes it harder to predict the assigned removal time for malicious users in the *Suspicious Clusters List*. Also, it ensures that the *Suspicious Clusters List* is always short and fast to search. Adding a cluster to the *Suspicious Clusters List* before blocking its users gives Guard the ability to make sure that all the users of this cluster coordinate to perform the attack. Therefore, Guard can significantly reduce the possibility of blocking normal users by mistake with other malicious users.

5.5 Response to Malicious Users

Guard responds to malicious users by simply blocking their new query requests from reaching the distributed streaming application. Guard's *Unsupervised Attack Detector* adds the IDs of malicious users that have been detected to the *Blocked Users* hash table. Every time the *Query Forwarder* receives a new query request, it verifies that the sender of the request is not in the *Blocked Users* hash table before forwarding the request to the distributed streaming application. Furthermore, new queries received from blocked users are discarded.

Old continuous queries of the blocked users will eventually expire and get removed from the system. Therefore, Guard does not remove old queries of the blocked users. The attack affects the system only when it removes old queries and adds new ones to make the system in a continuous state of rebalancing. Therefore, the system will recover after rebalancing its workload that includes old queries of blocked users. Not removing blocked users' old queries allows Guard to be general in working with any type of distributed streaming applications. The effect of the old queries of blocked users on the system might be reduced by developing a mechanism that asks all executor machines to immediately remove these queries. The feasibility to develop this mechanism depends on the application's implementation and whether the benefits of finding and removing the queries justifies the added overhead.

6. EXPERIMENTS

This chapter presents the performance evaluations of the proposed solutions: TrioStat, SWARM, and Guard. In Section 6.1, we show the details of the experimental setups and the used application. Section 6.2 analyzes the performance of online workload estimations, TrioStat. Section 6.3 evaluates the performance of our adaptive load-balancing mechanism, SWARM. Section 6.4 conducts an extensive evaluation of Guard's performance in detecting and blocking attacks that target adaptive load-balancing mechanisms.

6.1 Experimental Setup

We realize TrioStat, SWARM, and Guard in Apache Storm [3]. However, the proposed solutions can be used with any other distributed streaming systems that process spatial data streams in tuple-at-a-time manner.

6.1.1 Application and Dataset

The application is a location-aware publish-subscribe. The input stream is geo-tagged tweets from Twitter. Users can subscribe to get tweets in a specific spatial range by submitting continuous range queries. A tweet is geo-tagged as a point, say d, in space, where s qualifies for a user's subscription, say Query q, if d lies inside q's spatial range. Typically, before SWARM, each query gets replicated into all executor machines in an R*-Tree [81]. Each point is directed to only one executor machine, and is checked against all queries using the replicated R*-Tree. All the experiments are performed from a cold start. The grid index of SWARM and TrioStat that divides the whole space is of size 1000×1000. This size allows small cities in the US to be

covered by multiple cells. There are one million queries that are pre-loaded to every system. The spatial side lengths of queries are 0.16% of the side length of the whole space (about the size of a university campus).

Experiments are performed using a real dataset from Twitter and a synthetic query workload. The used dataset is composed of 1 Billion geotagged tweets of size 140 GB in the US. The tweets are collected from January 2014 to March 2015. The spatial data stream is made continuous and infinite by streaming the 1 Billion tweets repeatedly from the beginning each time they finish. The query workload is composed of continuous range queries. The focal points of the queries are determined using the locations of the real tweets.

6.1.2 Cluster Setup

Experiments are performed on 6 Amazon EC2 instances. Every instance runs Apache Storm 1.0.0 over Ubuntu 18.04.2. The Nimbus of Storm and a Zookeeper server [82] are installed in one of the instances that is of type m5.xlarge with 4 vCPU and 16 GB of memory. The remaining five instances are of type m5.2xlarge. Every one of the five instances has 8 vCPU and 32 GB of memory. Every instance is divided into 8 virtual machines, where each virtual machine has one vCPU and 4 GB of memory. Hence, they create a total of 40 virtual machines. The Tweets and queries streams are produced by 10 virtual machines that act as Storm spouts. In Guard's experiments, one of these virtual machines also runs Guard as it does not require continuous processing power. The remaining 30 virtual machines are divided as 8 routing machines (GlobalIndex in SWARM) and 22 Executor machines. The network bandwidth of the cluster is up to 10 Gbps.

6.2 Performance of the Online Workload Estimations

This section presents and analyzes the performance of TrioStat. TrioStat is realized in the application to provide workload estimations. In the applications, the routing machines distribute the workload among the executor machines according to the partitions that each machine holds. The application starts a new repartitioning round every 15 seconds. By the end of every round, the application asks TrioStat to update the statistics, and request from all executor machines to send their workload estimations ($W(m)$) and the number of newly received objects ($R(m)$) to one of the routing machines.

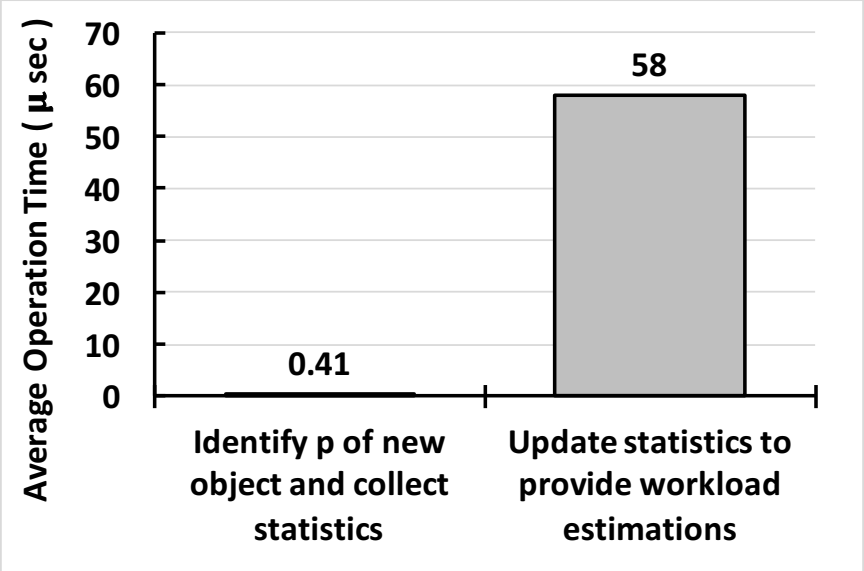

Fig. 6.1. Overhead of TrioStat in executor machines

Figure 6.1 illustrates the overhead of TrioStat operations by showing the average time each operation takes in microseconds after running the system for an hour. Figure 6.1 illustrates that TrioStat adds 0.41 microseconds to the processing time of a new object to identify its partition and collect statistics about it. This demonstrates TrioStat's success in minimizing the added overhead to the processing of each new object. At the end of every repartitioning round, TrioStat needs 58 microseconds on

average to update the statistics of all the partitions that an executor machine holds. After this update, TrioStat can estimate the workloads in $O(1)$, as in Section 3.3.

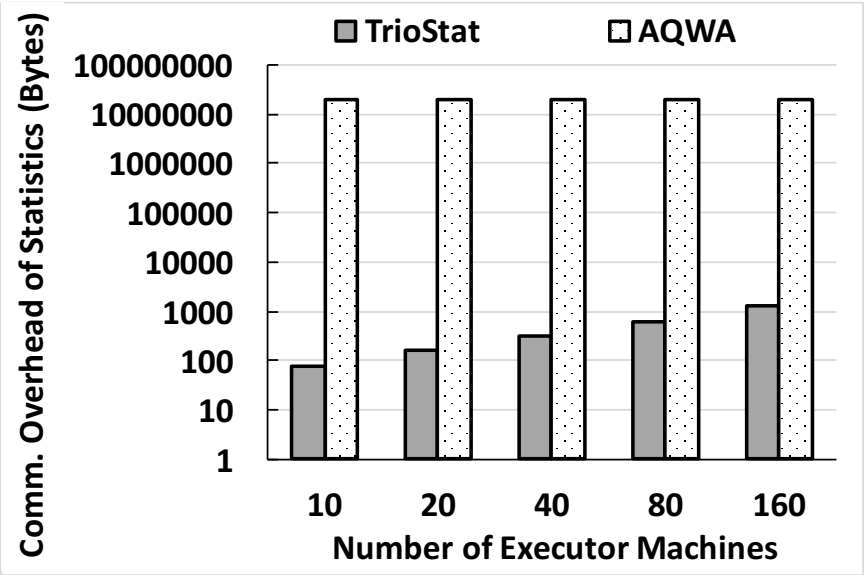

Fig. 6.2. Network overhead of TrioStat statistics

Figure 6.2 gives the network overhead of TrioStat's decentralized statistics compared to AQWA's centralized statistics approach [15]. Notice that the results are given in logarithmic scale. AQWA's statistics require maintaining one number per cell to count the data points, and four numbers per cell to count the queries. The four numbers in each cell are required to use the Euler Histogram [46] to count queries in a partition without re-counting queries that overlap multiple cells. Thus, AQWA collects the 5 statistics for every cell in the machine that holds the cell. By the end of every repartitioning round, all the collected statistics should be sent to one machine in order to be combined and used for workload estimation. Figure 6.2 compares the two approaches by measuring the number of bytes needed to be sent to one of the machines to monitor the performance of the system, compare the workload of all machines, and decide accordingly if repartitioning is needed. TrioStat's decentralized approach outperforms the centralized approach because TrioStat requires sending

only two statistics **per executor machine**. The two numbers are the workload estimate of the machine ($W(m)$) and the number of the newly received objects by the machine ($R(m)$). In contrast, the centralized approach requires sending five statistics **per cell** in the system, i.e., five million statistics for the 1000×1000 grid that divides the space. TrioStat will always outperform the centralized approach because each machine can hold at least one cell sized partition, i.e., TrioStat's machines will send 2 statistics per machine while the centralized machines will send 5. However, having every machine holds only one partition with one cell is not practical. Hence, the grid size will be increased and that will increase the amount of statistics that the centralized approach have to send.

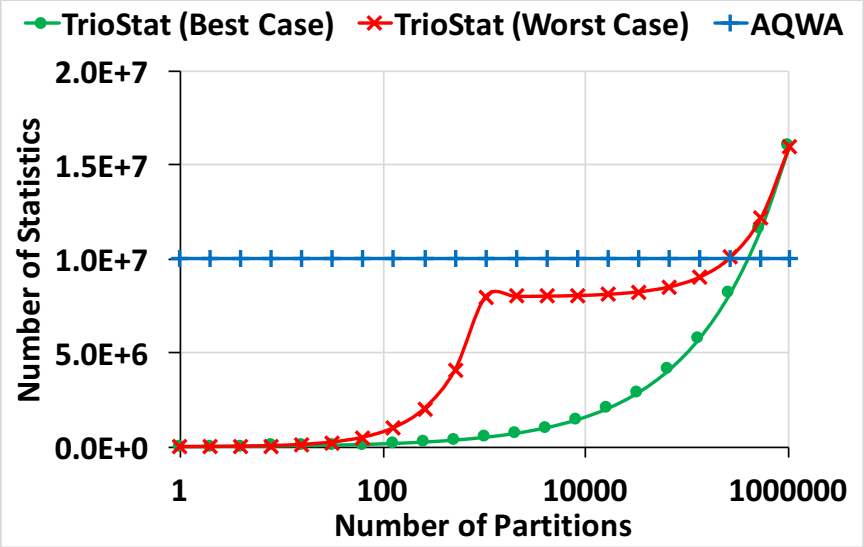

Fig. 6.3. Total Storage for the statistics while varying the number of partitions

TrioStat requires having 8 numbers stored for every row and column of every partition. 5 of them are for the required statistics and 3 for the statistics collectors. Therefore, the storage that TrioStat requires is distributed across the machines according to the distribution of the partitions. On the other hand, AQWA requires storing a total of 10 numbers per cell. 5 of the numbers are used to collect the

statistics and they are stored across the machines according to the distribution of the partitions. However, the remaining 5 numbers are all stored in a centralized machine to aggregate the collected statistics and can be used for workload estimation. This results in high storage overhead in one of the machines. Figure 6.3 gives the results of analyzing the total storage that TrioStat requires for statistics in comparison to AQWA while varying the number of partitions that divide the whole space. The grid that divides the space is 1000×1000 resulting in 1 million cells. Figure 6.3 gives the number of partitions in logarithmic scale between having 1 partition and 1 million partitions (all partitions are composed of a single cell). The required storage for TrioStat's statistics depends on the number of partitions and their shapes. Since TrioStat maintains statistics for every row and column of every partition, the total storage of TrioStat increases with the increase in the total circumferences of the partitions. Therefore, TrioStat requires the minimum storage when all partitions are squares (Best Case). On the other hand, The maximum storage (Worst Case) happens when the maximum number of partitions is of size 1 X grid side length. There is a fast increase in the worst case of TrioStat in Figure 6.3 before having 1000 partitions (equivalent to grid side length) because all partitions can be of size 1 X grid side length except one partition. In this worst case scenario, any increase in the number of partitions up to 1000 will result in increasing the total circumferences of the partitions by two times the grid side length. The increase in the worst case slows down after having more than 1000 partitions because any split after having 1000 partitions of size 1 X grid side length results in increasing the total circumferences of the partitions by exactly two. When the number of partitions become very large, each partition will be formed of very few cells. In this case, TrioStat requires higher storage than AQWA. However, it is not practical to have a large number of small partitions in the system. This is because having a large number of small partitions results on duplicating queries in a large number of partitions that needs to communicate to produce queries' final results. Usually, neighboring partitions in the same machines gets combined to reduce the overhead of query execution, which will result

on having medium sized partitions. Hence, having a large number of partitions that is close to the number of cells in the grid may never happen.

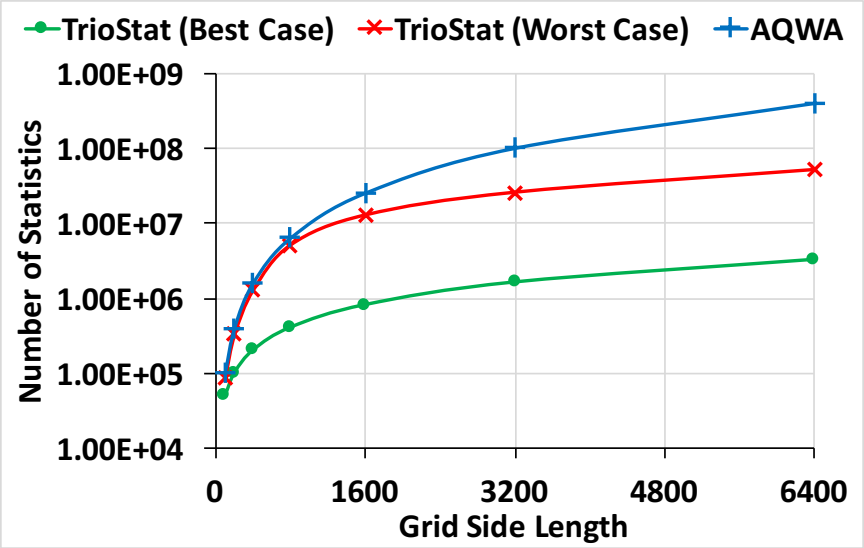

Fig. 6.4. Total Storage for the statistics while varying the grid size

Figure 6.4 gives the results of analyzing the total storage that TrioStat requires for statistics in comparison to AQWA while varying the size of the grid that divides the whole space. The grid side length varies between 100 and 6400 cells. Notice that the number of statistics is represented in logarithmic scale. The number of partitions is fixed to be 1000 partitions. TrioStat outperforms AQWA in all cases. The gap between TrioStat in the best case and the worst case is small when the side length of the grid is less than 1000 (the number of partitions) for the same reason explained in the previous paragraph.

6.3 Performance of the Adaptive Load Balancing

This section presents the results of testing SWARM against 3 other approaches: 1) *Replicated*: New queries are replicated into all executor machines, each covering the whole space. In contrast, a new data point is sent to only one executor machine in a round-robin fashion. 2) *Static Uniform Grid*: The whole space is evenly partitioned among all executor machines. 3) *Static Grid Based on History*: The partitioning of the whole space is determined based on observing a limited history of the data and query workloads. The whole space of *Static Grid Based on History* is partitioned offline based on 400K data points and 200K queries taken from the dataset. TrioStat's cost model is used in the third approach with the limited history to partition the workload. Hence, the costs of all executor machines are almost equal.

As mentioned, our cluster is composed of 40 virtual machines. 10 virtual machines are Storm spouts that produce the tweets stream and the queries. The remaining 30 virtual machines are executor machines in the *Replicated* approach. In SWARM and the other two static grid approaches, the 30 virtual machines are divided as 8 routing machines (GlobalIndex in SWARM) and 22 Executor machines. The machines' ratio is chosen after conducting an empirical study. We find that a good starting ratio is 1:3 (GlobalIndexes:Executors). All the used approaches store continuous queries in the R*-Tree index. SWARM starts a new load balancing round every 15 seconds.

6.3.1 Capability and Execution Latency

One performance measure that we use is the *Units of Work* measure, which is the number of tuple checks per second against queries. *Units of Work* is calculated by multiplying the number of queries in the system by the number of processed tuples per second. We use *Units of Work* in place of the throughput of the system as the former provides a fairer comparison because it reflects the overall amount of checks (work) that is conducted by the system regardless of the *selectivities* of the queries. Figure 6.5a gives the average *Units of Work* after running the system for

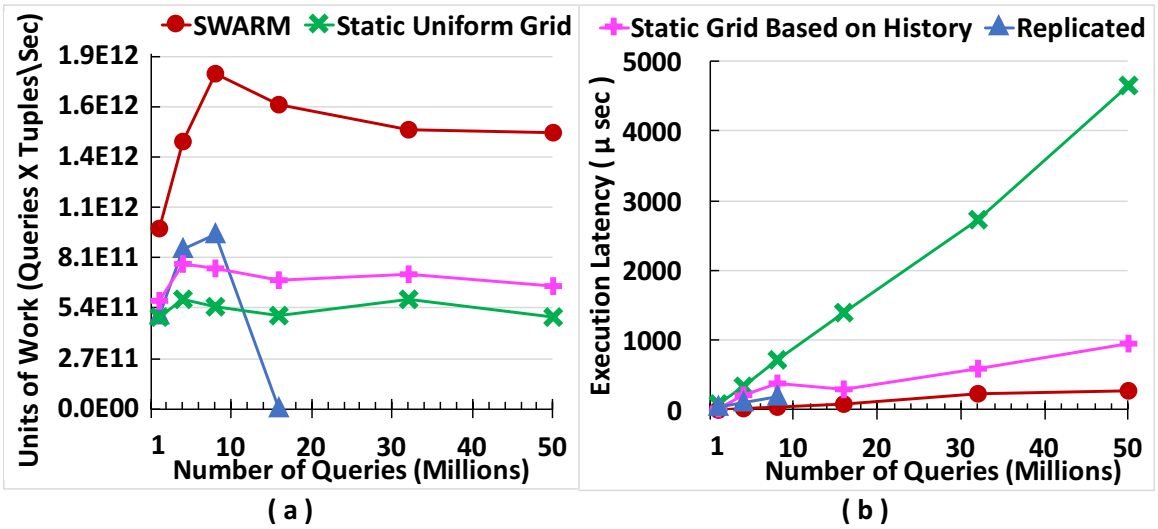

Fig. 6.5. Capability and execution latency

an hour. SWARM outperforms the other approaches while varying the number of continuous queries in the system. On average, SWARM achieves 200% improvement over *Static Grid Based on History*. SWARM performance saturates after 32 million queries as the system reaches its peak capacity for the used tweets distribution. The *Replicated* achieves better performance than both static grid approaches between 8 and 16 million queries because of better workload distribution among the system's machines. However, *Replicated* fails to support more than 16 million queries due to high memory overhead as all queries and indexes are replicated on all machines.

Figure 6.5b shows the average execution latency in microseconds while varying the number of queries after running the system for an hour. SWARM achieves the lowest average execution latency compared to other approaches. *Replicated* achieves lower average execution latency than both static grid approaches. However, it fails to support more than 16 million queries. The incremental rate of SWARM's average execution latency is very small compared to the other approaches. SWARM reduces execution latency on average 4x compared to *Static Grid Based on History*.

6.3.2 Reaction to Hotspots

We use the term *Hotspot* to refer to a spatial region that receives a large amount of queries and/or data points that is likely to persist for some duration of time. This definition of a *hotspots* excludes spikes of heavy workloads that do not persist for a significant duration of time. Twitter real dataset contains thousands of hotspots, where lots of tweets overlap in location and in time. We are interested in observing how SWARM reacts to various types of hotspots in contrast to the approaches. Hence, several scenarios of hotspots are created by synthetically redirecting a percentage of the data spouts to a specific location in the US. The normal Twitter dataset with its hotspots is used at the beginning and at the end of the experiments' time line. By default, the locations of the data points and queries that compose a hotspot's are generated inside a square range with a spatial side length of 15% of the whole space using a uniform distribution. All queries in a hotspot are instantiated during one minute of the hotspot's start time. This is to test how fast SWARM reacts to extreme hotspot situations.

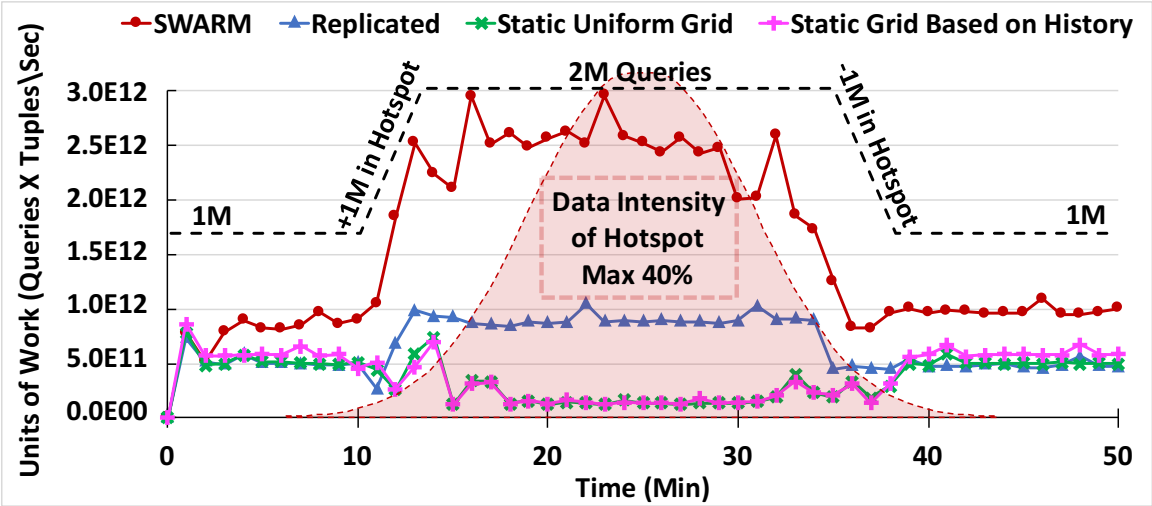

Fig. 6.6. Uniform distribution hotspot with normal distribution data intensity

Every Figure from 6.6 to 6.10 shows a timeline for the average *Units of Work* per minute. The axis scale of the *Units of Work* is the same across all experiments. The dashed line represents the number of queries in the system. Figure 6.6 compares the performance of all approaches with the appearance of one hotspot. The data intensity of the hotspot follows a normal distribution (the shaded area in the figure). The hotspot is created by redirecting up to 40% of the data spouts to the lowerleft corner of the US. As the figure indicates, SWARM outperforms all other approaches. SWARM achieves higher performance during the hotspot than before and after the hotspot. SWARM's higher performance is due to having a better chance to redistribute the uniformly distributed hotspot. Both of the static grid approaches suffer during the hotspot because only a small set of their executor machines become overloaded with the hotspot. During the regular Twitter hotspots, *Static Grid Based on History* achieves better performance than both *Static Uniform Grid* and *Replicated*. During the synthetic hotspot, it has the worst performance because its partitions are predetermined using a limited history of Twitter's normal dataset. The sudden increase and drop in processing performance is due to the back-pressure of the spouts that periodically makes the spouts try to increase the data injection rate. The performance of all approaches return to normal after the disappearance of the hotspot.

Figure 6.7 gives the performance when the hotspot's data points are generated using normal distribution inside the hotspot's region instead of using a uniform distribution. The normal distribution's variance is 20% of the hotspot's spatial side length. SWARM outperforms all the other approaches. SWARM has lower performance in the case of the normal-distribution hotspot in contrast to the uniform-distribution hotspot. The reason is that there are higher levels of spatial overlap among the data points and the queries inside the normal distribution hotspot that makes it harder for SWARM to find a new partitioning that distributes the workload evenly.

Figure 6.8 gives the performance when a uniform distribution hotspot appears directly with maximum data intensity. The data intensity of the hotspot follows a step function. Although this type of hotspots is uncommon, SWARM manages

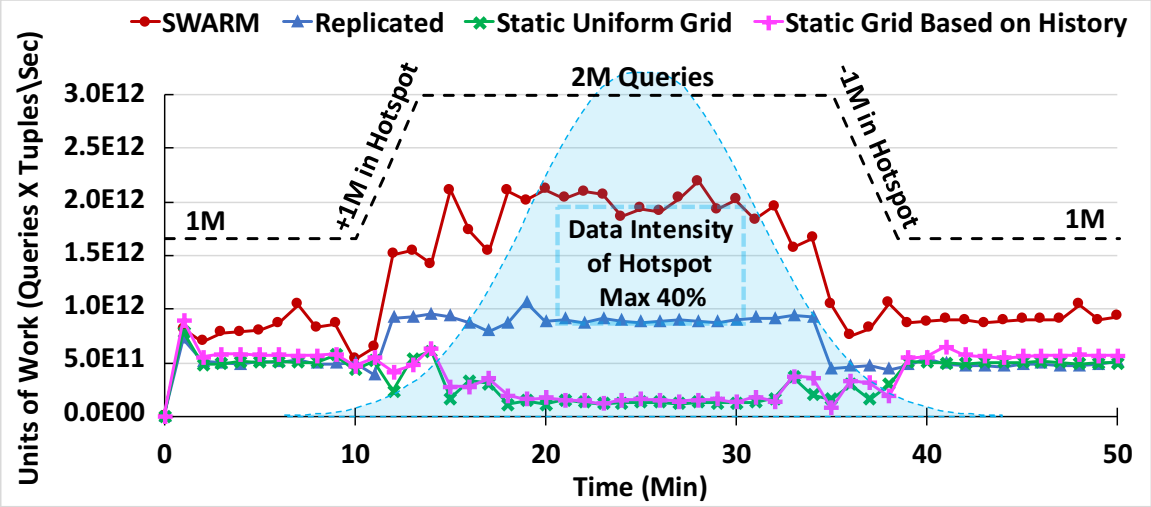

Fig. 6.7. Normal distribution hotspot with normal distribution data intensity

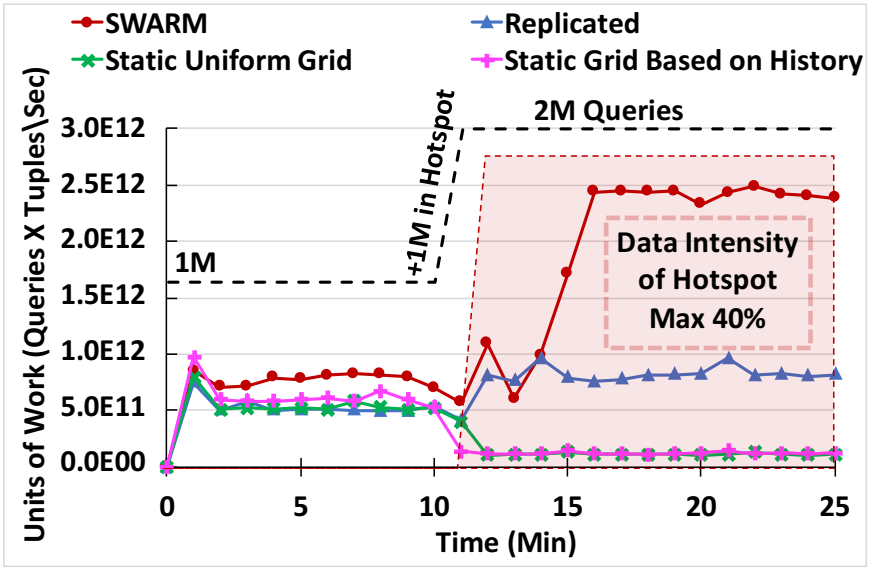

Fig. 6.8. Uniform distribution hotspot with step data intensity

to overcome the drop in performance. SWARM experiences a sudden drop in performance immediately after the hotspot starts because SWARM does not complete redistributing the partitions while some of the machines become overloaded, and this

triggers a backpressure from the Storm spouts to reduce their data injection rates. Once SWARM completes the redistribution of the partitions, the spouts' backpressure re-increases the data injection rate slowly.

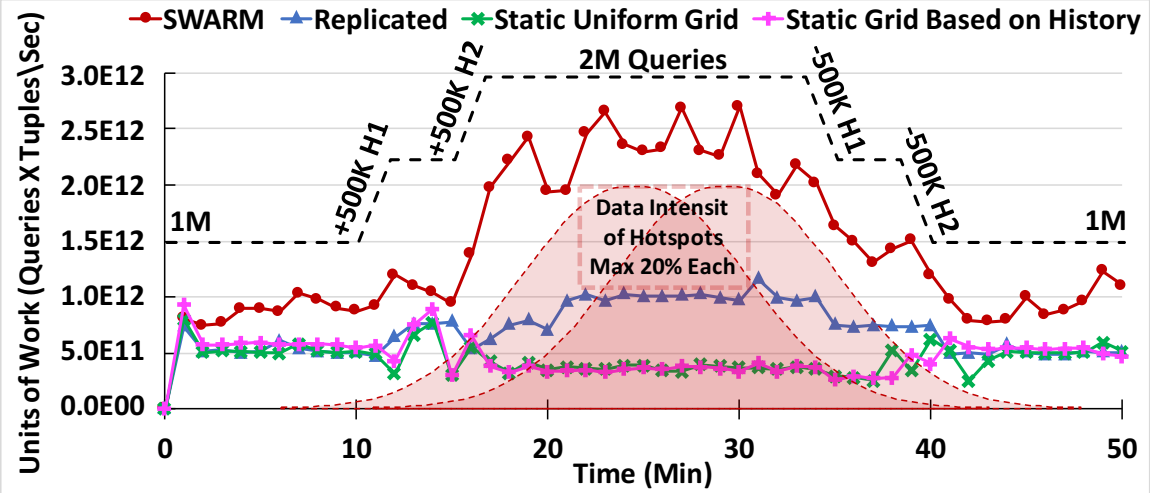

Fig. 6.9. Two overlapping hotspots (H1 and H2) in different locations

Figures 6.9 and 6.10 give the results when having two concurrent hotspots in two different locations. Hotspots H1 and H2 are located in the lowerleft and upperright corners of the US, respectively. Each hotspot is created by redirecting up to 20% of the data spouts, i.e., each hotspot has half the data intinsity of the hotspots in the previous expermints. Figure 6.9 gives the performance results when the two hotspots (H1 and H2) overlapping in time. SWARM achieves similar performance to that in Figure 6.6. Thus, SWARM is not affected by the number of simultaneous hotspots or their spatial locations. The intensity and distribution of the hotspots are the main factors that affect SWARM's rebalancing performance.

Figure 6.10 gives the performance when Hotspot H2 appears directly after Hotspot H1 disappears. This experiment illustrates that SWARM can quickly react to the changes in workload distribution. However, the performance of SWARM slightly drops at the start of H2 because SWARM needs to register the new hotspot queries as well as move some of them to other machines to rebalance the system. This slows

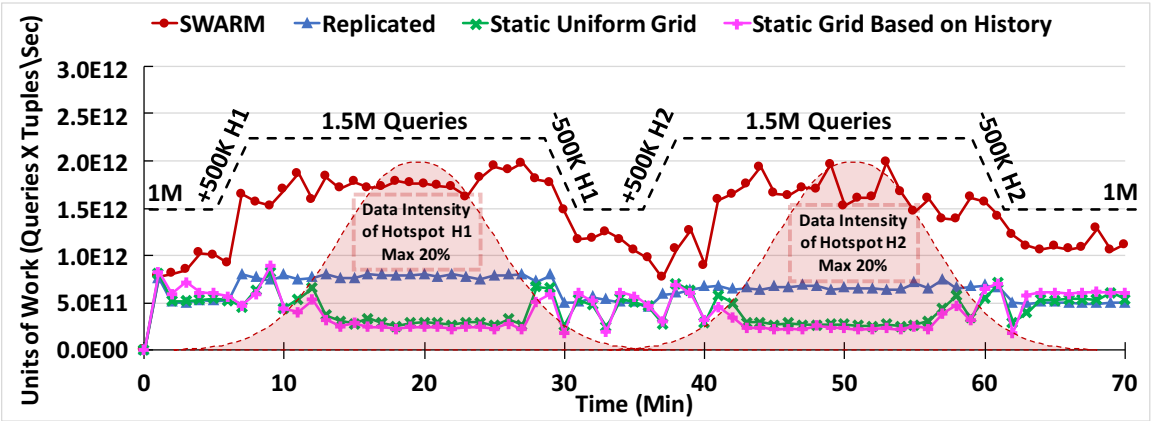

Fig. 6.10. Two consecutive hotspots (H1 and H2) in different locations

down the processing of the new incoming data during that time. After applying the new partitioning plan, SWARM achieves similar performance as the performance during Hotspot H1.

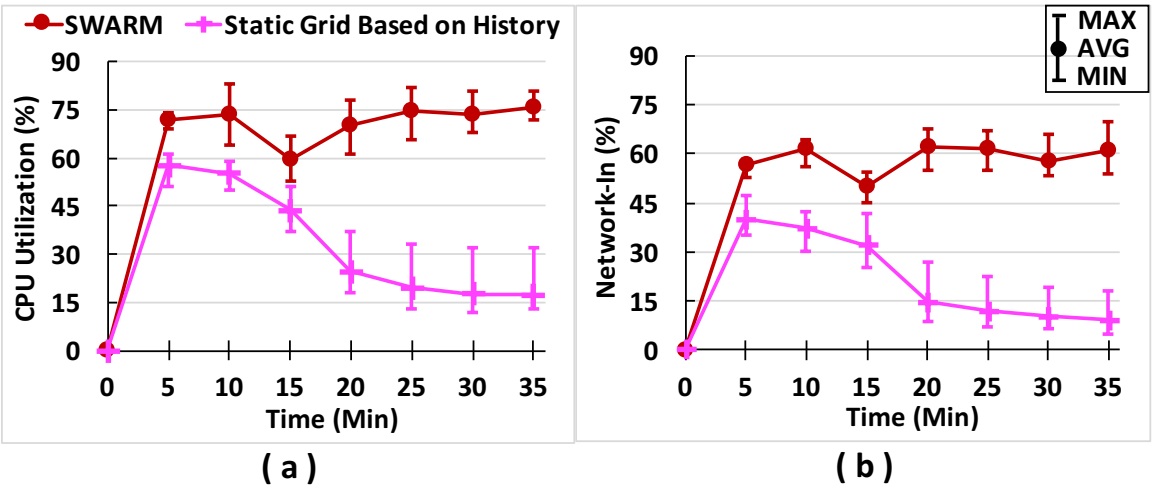

Fig. 6.11. CPU and network utilization

Figure 6.11 gives the CPU and network utilization averages over 5 minutes for the cluster's virtual machines. Also, the minimum and the maximum utilization average achieved by a machine is presented. The utilization is measured during the run of

the experiment presented in Figure 6.6. The drop in CPU and network utilization after 10 minutes of running the experiment happens because the hotspot starts at that moment. Figure 6.11a illustrates that SWARM's average CPU utilization is improved and the gap between the minimum and maximum utilized machines gets smaller as SWARM redistributes the hotspot workload among the other underloaded machines. For *Static Grid Based on History*, the overloaded machines affect the performance of the whole system because the backpressure of the spouts makes them reduce the injection rate to match the processing rate of the slowest machine. Before running the experiment, we test the network utilization to estimate the network's achievable maximum bandwidth, which is found to be 80% of the maximum advertised network bandwidth. Figure 6.11b illustrates that SWARM almost reaches the highest achievable network utilization. This highlights that the bottleneck in SWARM is moved from being in the processing power to being in the network bandwidth.

6.3.3 Overhead of the Adaptive Load Balancing

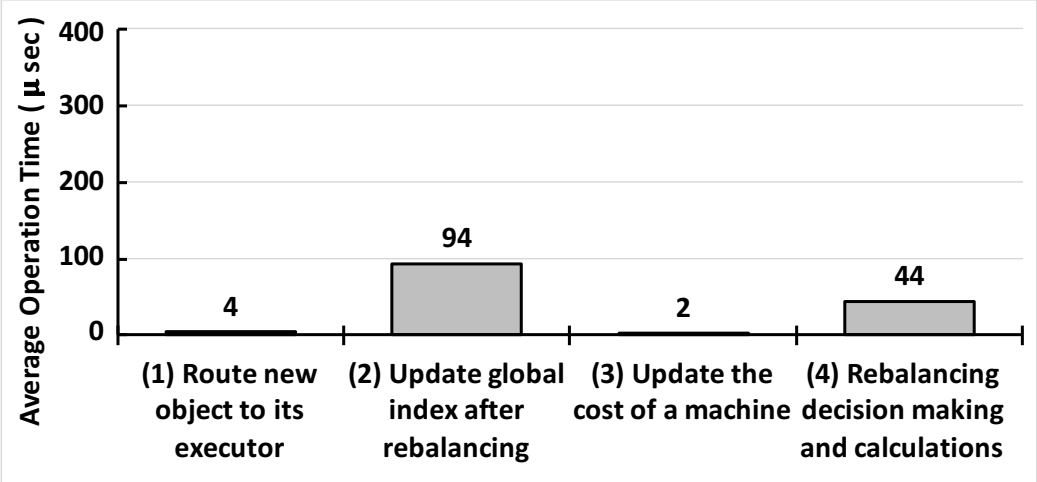

Fig. 6.12. Overhead of SWARM operations in GlobalIndex machines

Figures 6.12 and 6.13 illustrate the overhead of SWARM operations by showing the average time each operation takes in microseconds after running the system for an hour. Figure 6.12 presents the operations of the GlobalIndex machines, while Figure 6.13 presents the operations of the executor machines. Note that as we go from operation (1) to (4) in both figures, the frequency of performing the operation and the number of machines performing it are significantly decreased. Figure 6.12-(1) gives the time it takes for the GlobalIndex machines to find the responsible executor machine for a newly received object and route this object. Figure 6.12-(2) shows the required time to update the index of a GlobalIndex machine according to a moved subset or split. Figure 6.12-(3) and 6.12-(4) shows the operations that get executed only in the Coordinator to receive the executors' cost, finding if rebalancing is needed, and identifying m_H and m_L.

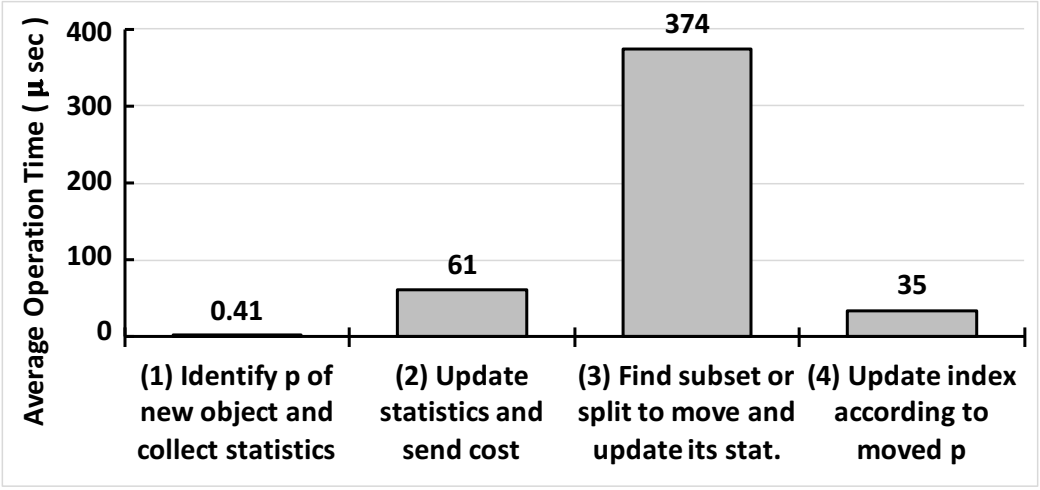

Fig. 6.13. Overhead of SWARM operations in executor machines

Figure 6.13-(1) gives the time added (overhead) to the processing of a new object to identify its partition and collect its statistics. This shows the success of SWARM in minimizing the added overhead to the processing of each new object. At the end of every load-balancing round, the time of Figure 6.13-(2) is needed to update the statistics, compute the cost, and send the cost to the Coordinator. Figure 6.13-(3)

gives the required time for m_H to find a workload reduction, update the statistics, and move the partition(s) to m_L. Figure 6.13-(4) gives the required time for m_L to receive a moved partition and update m_L's index accordingly.

6.4 Performance of Detecting and Blocking Attacks Targeting Adaptive Load Balancing Mechanisms

This section presents the performance of Guard during different attack scenarios. Guard is integrated into SWARM to protect it from attacks. SWARM decides if rebalancing is required every 15 seconds. The duration of Guard's detection round is one minute. This allows a maximum of three hotspots to be re-balanced by SWARM during a detection round.

The system is pre-loaded with 1 million continuous queries coming from 110 users. Users' activities consist of registering new continuous queries and terminating old continuous queries. The default settings of the system have 100 normal users with total activities of 600 Queries/Sec (*Normal Activity*) and 10 malicious users involved in the attack with total activities of 600 Queries/Sec (*Malicious Activity*).

Queries of the normal users are determined based on the locations of real-tweets as described in Section 6.1.1. This creates natural hotspots in the system based on real Twitter activity. Real-life workloads are often skewed [83]. Normal Users' activities are assigned to normal users using Zipf distribution [84–87]. We configure the Zipf distribution to follow the 80/20 rule [88, 89]. This results in almost 80% of the activities coming from 20% of normal users. It puts the system in a real-life situation where it has users with varying activity levels. To maximize the attack effect, malicious users involved in the attack follow a uniform distribution for their malicious activity. Malicious users want to maximize their activity in creating malicious hotspots while maintaining similar activity rate as that of normal users.

At any moment of time, the system is serving on average 1 million continuous queries because the number of new registered queries are almost equal to the number of terminated queries. This is to eliminate the effect of serving higher or lower number of queries on the performance of the system, to focus only on the effect of the attack. Malicious users are responsible for 100 thousand queries when there is an attack while normal users are responsible for 900 thousand queries. When there is no attack,

normal users are responsible for the whole 1 million queries. Malicious users register their 100 thousand queries to a targeted location while removing their older queries from the previous targeted location. Therefore, the intensity of the malicious activity determines how fast malicious hotspots are created and how fast the targeted location changes. The attacker chooses the location of the next malicious hotspot based on the locations of a tweets sample taken from the current data stream.

6.4.1 Attack Effect on Throughput

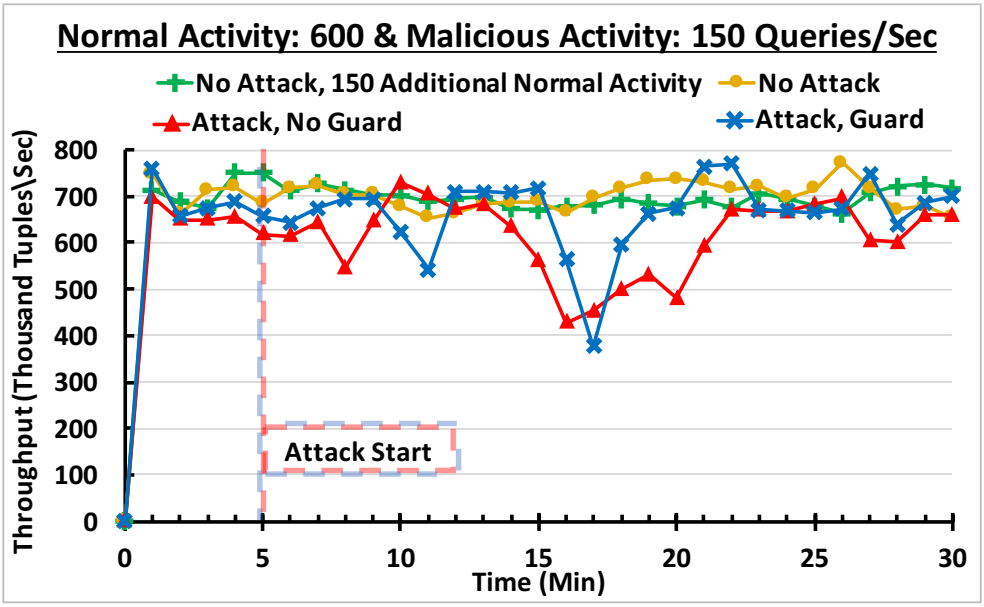

Fig. 6.14. The effect of an attack with malicious activity rate 150 queries/second on the system's throughput

Figures 6.14, 6.15, 6.16, and 6.17 shows a timeline for the system's throughput in terms of thousand tweets processed per second during every minute. The figures compare the throughput while there is an attack or no attack on the system. The figures present the effect of the attack on the system while varying malicious activity rates to 150, 300, 600, and 1200 queries/second, respectively. The attack starts at Minute 5. The figures present the difference that Guard makes when there is an

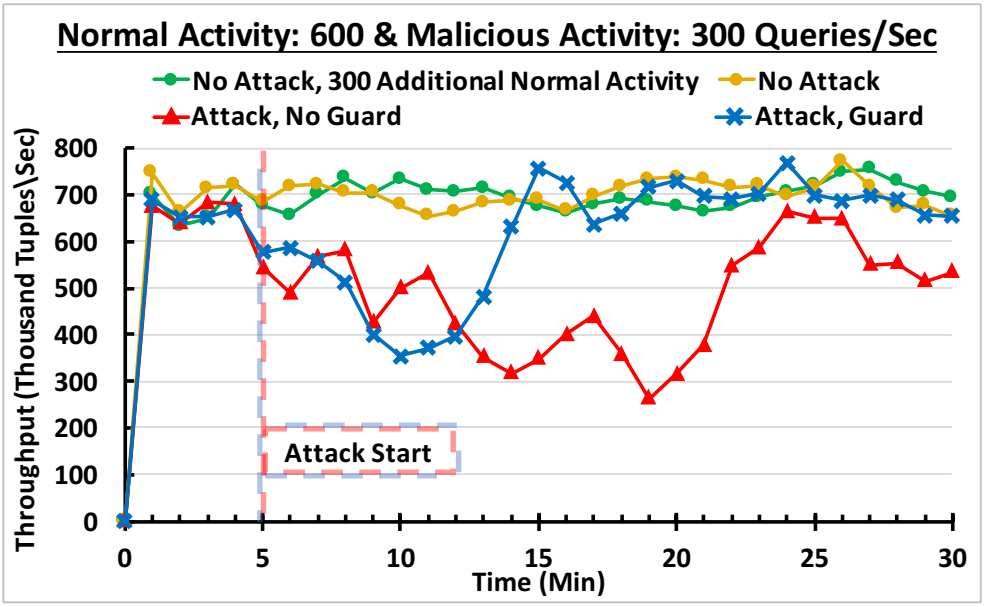

Fig. 6.15. The effect of an attack with malicious activity rate 300 queries/second on the system's throughput

attack on the system. Also, they show how the system recovers from the attack after Guard detects the malicious users and blocks them. All the figures show a common line representing when there is no attack and the total normal users' activity is 600 queries/second. Each sub-figure presents the results when the malicious activity is replaced by an equivalent increase in the total normal users' activity. This is to show the effect of registering and terminating more queries on the system and how the system can adapt to the hotspots that normal users create. Notice that there is a very small decrease in the throughput when there is an extra-normal activity and no attack. Hence, it is clear that the reason for decreasing the throughput is the way malicious users use their extra activity, not the overhead that comes with the extra activity.

The malicious activity in Figure 6.14 is small. Hence, the attacker does not succeed most of the time in creating strong malicious hotspots that make the system in a continuous state of rebalancing. Notice that Guard completely blocks the attack

after the creation of its second successful malicious hotspot. This complies with Guard's design that requires at least two malicious hotspots to be created before blocking malicious users. Guard puts higher penalty on blocking legitimate users than on allowing malicious users to affect the system a little longer before being sure they are malicious and blocking them. Note that during this experiment the activity of the malicious users are very small compared with normal users.

Increasing malicious activity to 300 queries/second in Figure 6.15 shows more success for the attack with the absence of Guard. The attack makes the system lose more throughput most of the time. Guard detects and blocks the attack. The system recovers after Guard blocks the attack and follows a similar performance as the one for the experiment without the attack.

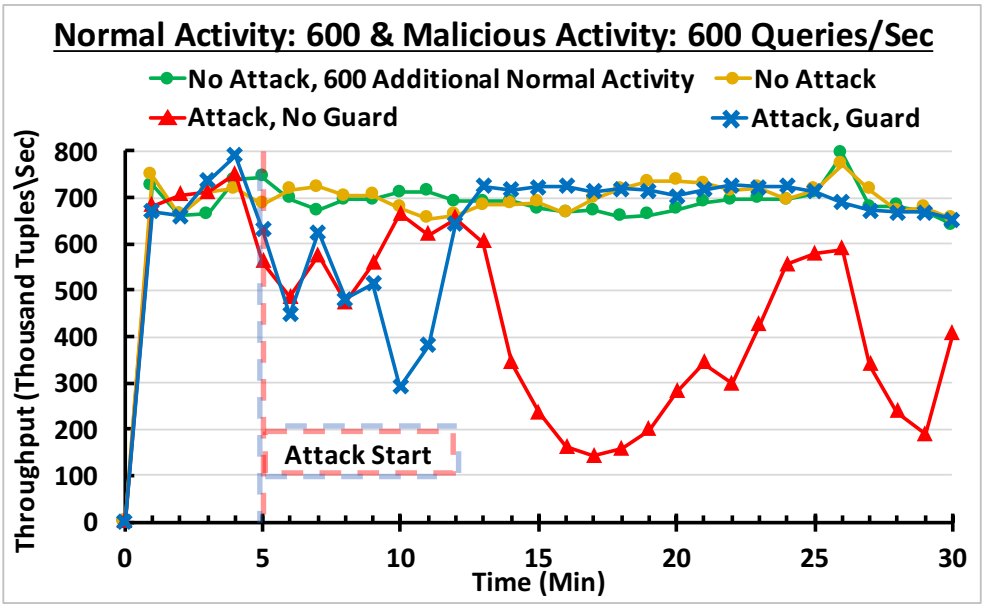

Fig. 6.16. The effect of an attack with malicious activity rate 600 queries/second on the system's throughput

Figure 6.16 shows the results when malicious activity is 600 queries/second. The increase in malicious activity decreases the throughput more on average and makes it reaches lower minimum throughput. Note that during the experiments of Figure 6.16,

the activities of malicious users are similar to those of normal users that have average activity. In this case, Guard detects and blocks the attack faster than when the malicious activity is smaller. However, the attack achieves lower minimum throughput for a moment of time. The reason is that higher malicious activity makes the attacker succeed in creating more malicious hotspots during a shorter time period. Therefore, Guard has the opportunity to collect more raw information about malicious users and be confident to identify them.

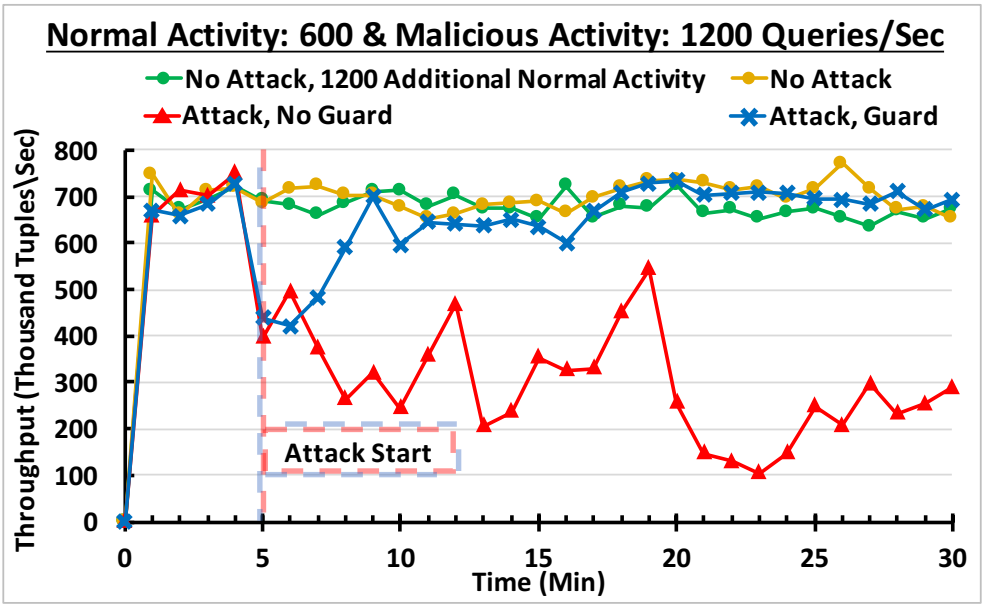

Fig. 6.17. The effect of an attack with malicious activity rate 1200 queries/second on the system's throughput

Figure 6.17 presents the results of having the activity of the malicious users that issue 1200 queries/second. It makes malicious users have an activity similar to that of very active normal users. The attack succeeds during the whole experiment. It makes the system in a continuous state of rebalancing, chasing malicious hotspots. Guard detects all malicious users and blocks them faster than any of the attacks that have lower malicious activity rates.

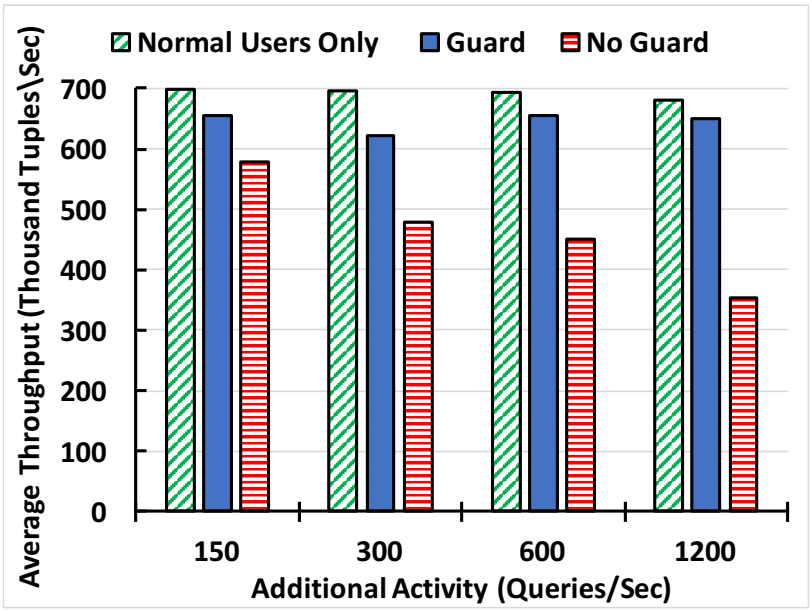

Fig. 6.18. Average system's throughput while varying the additional activity

Figure 6.18 presents the average throughput while varying the additional activity. The average throughput is computed after running every experiment five times for 30 minutes each. The additional activity is added to the total normal users activity when there is no attack. In the case when there is an attack, the additional activity is added to the malicious users activity. When there is no attack, the overhead of registering and terminating more queries causes a small decrease in the average throughput as the activity increases. When there is an attack and Guard is absent from the system, the average throughput decreases as the malicious activity increases. The increase in malicious activity leads to a faster creation of malicious hotspots that causes the system to be in a continuous state of rebalancing. The average throughput decreases by 18% when the malicious activity is 150 queries/second. When the malicious activity is 1200 queries/second, the average throughput decreases by 48%. On the other hand, the attack in all Guard experiments decreases the average throughput approximately by 5%. However, when the malicious activity is 300 queries/second,

the average throughput decreases by 10% because the behavior of malicious users is very similar to the behavior of average normal users that also create some genuine hotspots. When malicious users have a similar behavior to that of normal users, Guard allows malicious users to create more malicious hotspots to make sure they are malicious before blocking them. Guard enhances the average throughput between 14% and 85% when there is an attack. Note that if the average throughput is computed over more than 30-minute periods, it will increase when the system has Guard and decrease when Guard is absent from the system. The reason is that the effect of the small period before Guard blocks the attack fades with time.

To summarize Figures 6.14, 6.15, 6.16, 6.17 and 6.18, Guard succeeds in detecting and blocking the malicious users that are involved in creating malicious hotspots. Guard differentiates between malicious and normal users regardless of which normal users the malicious users are similar to. As the attacker increases the malicious activity, a decreased average throughput and a lower minimum throughput are achieved. On the other hand, Guard detects and blocks attacks faster as their malicious activity increases. Hence, Guard makes the attacker in a dilemma where increasing the malicious activity harms the system more, however, it can risk to be detected and blocked faster.

6.4.2 Attack Effect on Availability

Figure 6.19 is a box plot that illustrates the percentage of system's availability while varying malicious activity. The availability is the percentage of data that gets processed without delay. Since the attack makes the system in a continuous state of load-balancing, it reduces the system's processing power of data. Hence, some data tuples are delayed or dropped because the data stream delivers data in real-time and cannot be slowed. Figure 6.19 shows the results in a box plot form after running each experiment five times. The lowest point in the box plot is the minimum availability achieved during the 30 minutes of all the five experiments while the highest point is

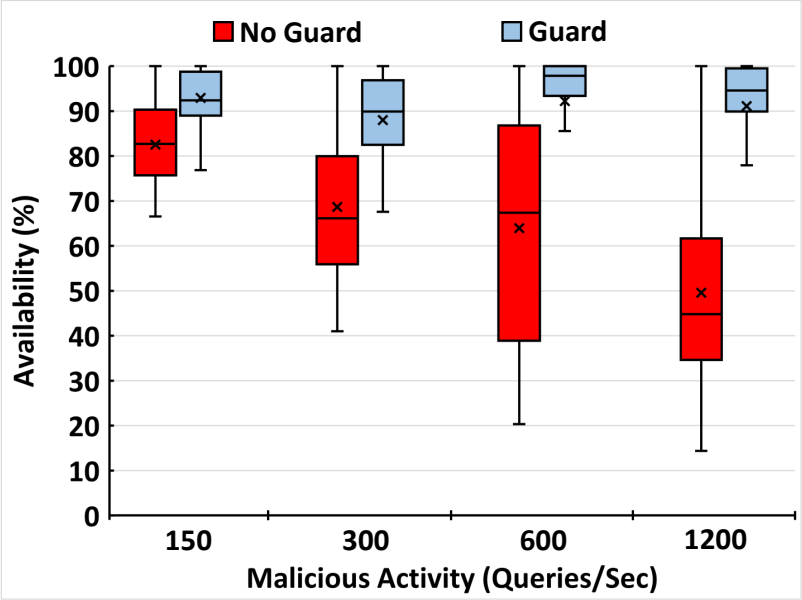

Fig. 6.19. Availability of the system while varying malicious activity

the maximum availability. The box is drawn from the lower quartile (Q1, the median of the lower half of the dataset) to the upper quartile (Q3, the median of the upper half of the dataset) with a horizontal line drawn in the middle to denote the median. Also, the "x" mark represents the average availability.

In Figure 6.19, Guard achieves around 90% average availability in all experiments. Moreover, Guard reduces the effect of the attack on the availability to be only for a small period of time as the sizes and positions of Guard's boxes indicate. The maximum availability is 100% in all the experiments because all experiments start without the attack for five minutes. The minimum availability while having Guard is much higher than when Guard is absent. The attack manages to reach this minimum availability just for a moment of the experiment's time. When Guard is absent, Figure 6.19 shows that the attack reduces the average and the minimum availability while increasing its malicious activity. Moreover, the time spent while the system on low availability is increased with attacks that have higher malicious activity as indicated by the positions of the boxes in Figure 6.19. The reason is that higher

malicious activity makes the attack succeed to draw more of the system processing power towards load-balancing. Guard improves the average availability up to 86% depending on the rate of the malicious activity. Also, Guard improves the minimum availability that the attack achieves by 17% up to 325% depending on the malicious activity.

6.4.3 Detection and Recovery

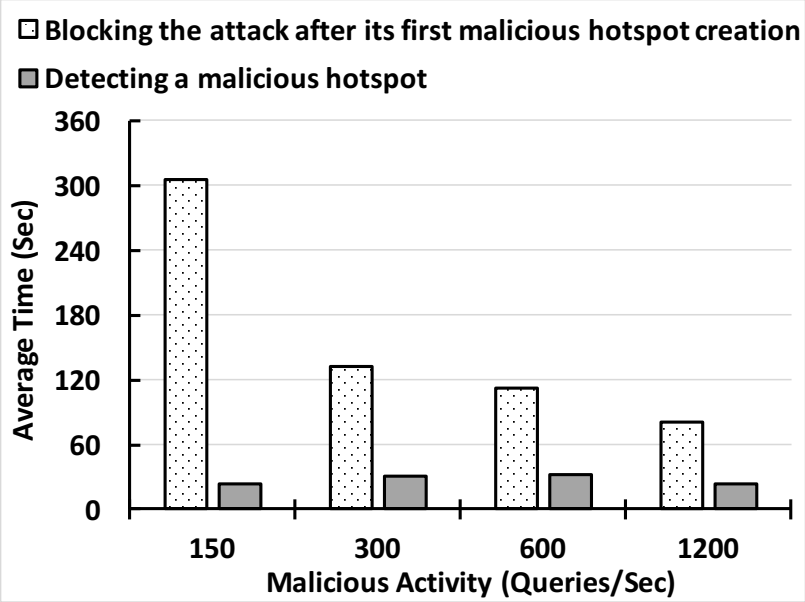

Fig. 6.20. Average detection and blocking times while varying malicious activity

Figure 6.20 shows the average times to detect malicious hotspots and block malicious users while varying malicious activity. The time for detecting malicious users depends on the speed of Guard in detecting malicious hotspots. On average, Guard detects malicious hotspots in 30 seconds. As mentioned, every Guard's detection round lasts for one minute. This indicates that Guard always identifies malicious hotspots during the same detection round, when the hotspot has been created. Fig-

ure 6.20 shows the time it takes Guard to block malicious users starting from their first malicious hotspot creation. The time to block malicious users relies on the speed of their successful creation of malicious hotspots. The reason is that Guard requires the raw information of at least two malicious hotspots to confirm the involvement of malicious users. When malicious activity is low, the time between the successful creation of two malicious hotspots increases. As malicious activity increases, Guard blocks malicious users faster. Guard lets malicious users with low activity rates stay in the system longer. However, their average effect on the throughput and the availability are similar to the average effect of malicious users with high activity rates that get blocked faster.

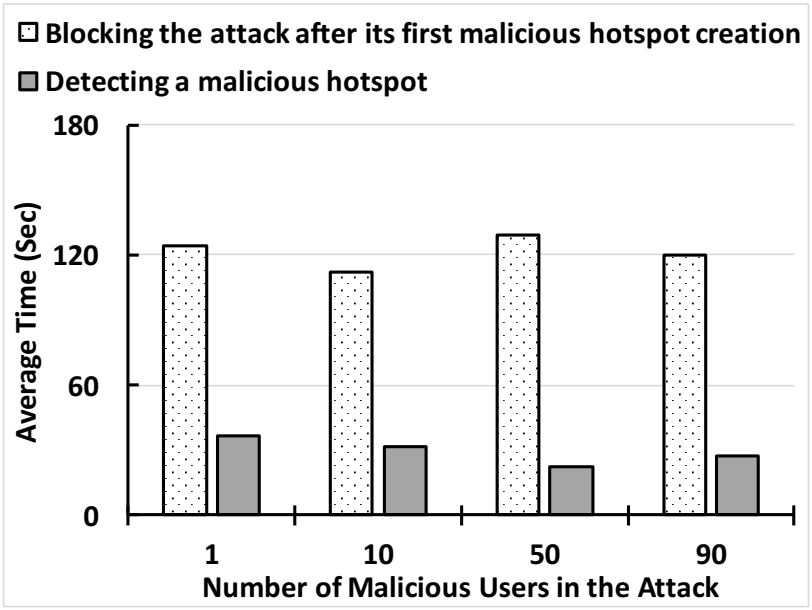

Fig. 6.21. Average detection and blocking times while varying the number of malicious users involved in the attack

Figure 6.21 shows the average times to detect malicious hotspots and block the attack while varying the number of malicious users in the attack. The malicious activity is fixed to 600 queries/second in all experiments of this figure. Hence, the activity of every malicious user decreases as the number of malicious users involved

in the attack increase. Notice that the average detection time is around 30 seconds in all cases. Moreover, the average time to block the attack is around two minutes for all cases. The results of Figure 6.21 indicate that the detection and blocking times of Guard is not affected by the number of malicious users involved in the attack. Hence, Guard's unsupervised machine learning detector succeeds in clustering and identifying all malicious users in the attack, regardless of their number. The rate of the malicious activity is the main contributor to changing the detecting and blocking times, the availability, and the throughput.

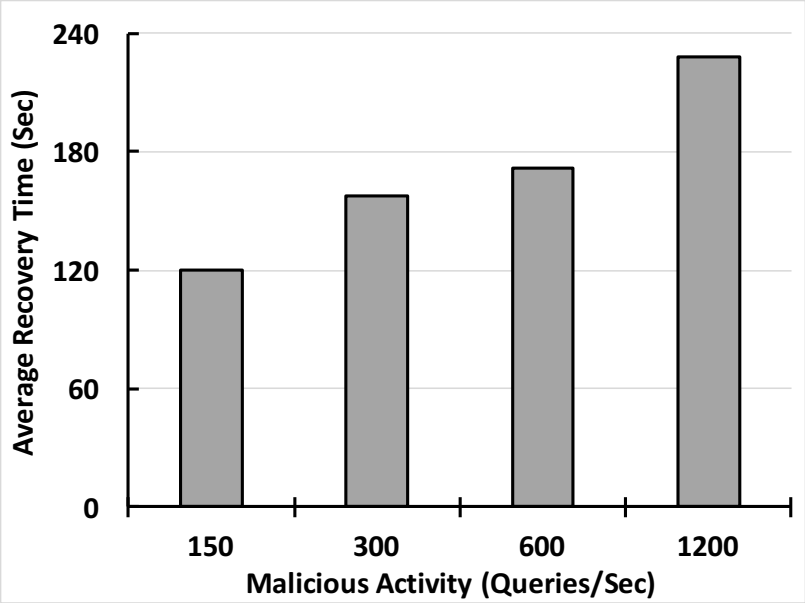

Fig. 6.22. Average recovery time after blocking the attack

During all experiments, Guard manages to detect and block all malicious users without falsely identifying any normal users as malicious. All the experiments demonstrate that Guard's detector does not have any false-negatives or false-positives in identifying malicious users. Although some normal users are added to the suspicious list in some of the experiments, they get removed from the list on later rounds. Therefore, Guard achieves its main objective of blocking malicious users only when being certain that they are malicious.

Figure 6.22 illustrates the average recovery time that the system takes after Guard blocks the attack while varying malicious activity. The average is computed after running each experiment five times. The recovery time starts from the time when the system reaches its lowest throughput due to the attack until it reaches the same throughput of when there is no attack. Figure 6.22 shows that the average recovery time increases with the increase in malicious activity. After Guard blocks the attack, the malicious queries already received stay in the system until they expire. Therefor, higher malicious activities leave higher number of queries that consume more resources to rebalance.

6.4.4 Overhead of Attack Detection

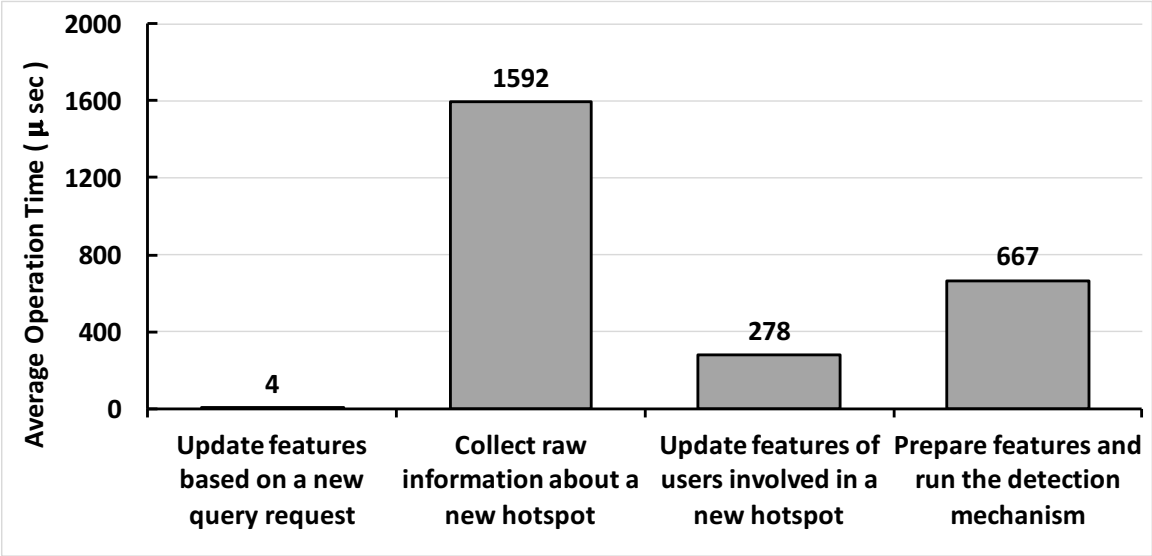

Fig. 6.23. Overhead of Guard's operations

Figure 6.23 illustrates the overhead of Guard's operations by showing the average time each operation takes in microseconds after running the system for an hour. Guard takes 4 microseconds to collect raw information about a new query request and to update its user's features accordingly. This shows the success of Guard in

minimizing the additional overhead with the operations that happen frequently and affect the system's performance. Every time a new hotspot is detected by a *Hotspot Sensor*, it requires approximately 1600 microseconds to collect raw information about the hotspot. The time to collect hotspot raw information depends on the number of queries that overlap with the hotspot. Thus, most of the time is spent to retrieve all queries that overlap with the hotspot, which depends on the speed of the application in accessing queries. Recall that in The experiments, the application uses R*-Tree index. When a *Hotspot Sensor* sends the raw information of a new hotspot, Guard spends 278 microseconds on average to update the features of the involved users. Guard requires 667 microseconds on average to prepare all users' features and to run the unsupervised attack detector. This operation happens in Guard by the end of every detection round.

7. CONCLUSION AND FUTURE WORK

In Section 7.1, the research contributions of this dissertation is summarized. The future research work is discussed in Section 7.2.

7.1 Summary of Contribution

The research contributions of this dissertation are in the area of adaptive load-balancing in distributed spatial streaming systems. First, we introduce TrioStat, an online workload estimation technique that relies on a probabilistic model for estimating the workload of partitions and machines in a distributed spatial data streaming system. We present a cost model that predicts the workload of each machine in the future. The prediction is based on query workload and changes in the distribution of data and queries. TrioStat introduces a new statistics structure that requires minimal storage overhead. TrioStat uses a decentralised technique to collect and maintain the required statistics in real-time locally in each machine. Efficient algorithm is presented to collect statistics without adding much processing overhead with the arrival of every new data point or query. TrioStat is tested and compared against AQWA using an application that processes a real dataset from Twitter. TrioStat enables the application to compare the workload of its machines and data partitions with minimal network, storage, and processing overhead. TrioStat requires sharing only two numbers per machine to compare the machines based on their workloads and to monitor the performance of the system.

The second contribution is to design and implement SWARM, an adaptive load-balancing protocol for distributed spatial streaming systems that rebalance the workload based on the changes in workload estimations provided by TrioStat. SWARM continuously monitors the workload across the distributed processes of the spatial

streaming systems. It adjusts the workload distribution as soon as performance bottlenecks get detected. SWARM can be used directly with minimal changes as a black box with any spatial application. We present a protocol that makes the process of load balancing decentralized. This results in minimizing communications by maximizing local decision making. We introduce two Greedy algorithms for finding the best reduction of the workload for the machine with the highest cost. The reduction of workload can happen by either (1) finding a subset of partitions to move to the machine with the lowest cost, or (2) splitting one of the partitions and moving one of the sub-partitions. We present a design that insures the integrity of the system and the correctness of the query results during load re-balancing. SWARM is tested and compared against other static approaches using an application that processes a real dataset from Twitter. On average, SWARM achieves 200% improvement over a static grid approach that is partitioned based on a limited history of the workload. Moreover, SWARM reduces execution latency on average 4x compared with the other approaches.

The final contribution is to investigate types of attacks that target the adaptive load-balancing mechanisms of distributed streaming applications. High intensity attacks decrease the throughput and availability of the system about 50%. We proposes Guard, a component that continuously monitors the behaviors of the users in the system and their relationships with hotspots. Guard detects and blocks malicious users that try to make the system in a continuous state of load-balancing. Guard is general and requires minimal changes to the distributed streaming systems. Guard uses an unsupervised machine learning technique to detect groups of malicious users that coordinate in attacking the system. Guard is integrated into SWARM. The performance of Guard in protecting SWARM is tested using an application that processes a real dataset from Twitter while facing different attack scenarios. Guard successfully blocks all malicious users without mistakenly blocking any normal users. The system fully recovers after Guard blocks the attacks. Guard improves the throughput by 85% and the availability by 86% as a result of blocking high intensity attacks. Moreover,

Guard reduces the minimum availability that the attacker achieves by 325%. On average, Guard detects the creation of a malicious hotspot in 30 seconds and blocks the attack in two minutes.

7.2 Future Work

The research presented in this dissertation can be extended into various directions. We present the following future research problems that we plan to pursue as an extension of the work presented in this dissertation.

(1) Query-Aware Elastic Distributed Streaming System for Big Data. As discussed in this dissertation, the amount of workload can rapidly vary with time in some applications. Using clusters is expensive and might lead to waste of resources during periods of low required processing. Therefore, these applications require adding and removing resources on demand. We plan to develop an elastic real-time distributed stream processing system that supports continuous queries over big data. We plan to extend TrioStat's cost model to estimate the processing power and storage needed for a given workload. Moreover, we plan to extend SWARM to manage hotspots by efficiently adding/removing resources and changing processing assignments among the system's machines according to the hotspots statistics.

(2) Adaptive Load-Balancing in Micro-Batched Distributed Spatial Streaming Systems. The designs presented in this dissertation are suitable for distributed streaming systems that process spatial data in tuple-at-a-time manner (e.g., Apache Storm [3]). These systems are efficient in providing answers in real-time with minimal execution latency. However, distributed streaming systems that process spatial data in a micro-batch manner (e.g., Spark Streaming [5]) have different advantages that are suitable for different types of applications. They can achieve higher throughput, efficient fault-tolerance, and easier integration with offline data.

Therefore, we plan to redesign TrioStat's cost model and its algorithm that collects and maintains statistics to work on micro-batch distributed streaming systems. Moreover, SWARM can be modified to support micro-batch streaming systems. We plan to investigate the type of attacks that target micro-batch streaming systems.

REFERENCES

REFERENCES

[1] "Internet live stats," https://internetlivestats.com/, 2019.

[2] M. F. Mokbel, "Thinking spatial, ACM SIGMOD Blog," http://wp.sigmod.org/?p=2012, 2016.

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

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

[5] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters." *HotCloud*, vol. 12, pp. 10–10, 2012.

[6] A. S. Abdelhamid, M. Tang, A. M. Aly, A. R. Mahmood, T. Qadah, W. G. Aref, and S. Basalamah, "Cruncher: Distributed in-memory processing for location-based services," in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 2016, pp. 1406–1409.

[7] D. Choi, S. Song, B. Kim, and I. Bae, "Processing moving objects and traffic events based on spark streaming," in *Disaster Recovery and Business Continuity (DRBC), 2015 8th International Conference on*. IEEE, 2015, pp. 4–7.

[8] Y. Lee and S. Song, "Distributed indexing methods for moving objects based on spark stream," *International Journal of Contents*, vol. 11, no. 1, pp. 69–72, 2015.

[9] G. Song, "Parallel and continuous join processing for data stream," Ph.D. dissertation, Université Paris-Saclay, 2016.

[10] Z. Yu, Y. Liu, X. Yu, and K. Q. Pu, "Scalable distributed processing of k nearest neighbor queries over moving objects," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 5, pp. 1383–1396, 2015.

[11] S. Wu, V. Kumar, K.-L. Wu, and B. C. Ooi, "Parallelizing stateful operators in a distributed stream processing system: how, should you and how much?" in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. ACM, 2012, pp. 278–289.

[12] A. R. Mahmood, A. M. Aly, T. Qadah, E. K. Rezig, A. Daghistani, A. Madkour, A. S. Abdelhamid, M. S. Hassan, W. G. Aref, and S. Basalamah, "Tornado: A distributed spatio-textual stream processing system," *PVLDB*, vol. 8, no. 12, pp. 2020–2023, 2015.

[13] A. R. Mahmood, A. Daghistani, A. M. Aly, M. Tang, S. Basalamah, S. Prabhakar, and W. G. Aref, "Adaptive processing of spatial-keyword data over a distributed streaming cluster," in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2018, pp. 219–228.

[14] Z. Chen, G. Cong, Z. Zhang, T. Z. Fuz, and L. Chen, "Distributed publish/subscribe query processing on the spatio-textual data stream," in *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. IEEE, 2017, pp. 1095–1106.

[15] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah, "Aqwa: Adaptive query workload aware partitioning of big spatial data," *Proc. VLDB Endow.*, vol. 8, no. 13, pp. 2062–2073, Sep. 2015.

[16] A. M. Aly, H. Elmeleegy, Y. Qi, and W. Aref, "Kangaroo: Workload-aware processing of range data and range queries in hadoop," in *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, ser. WSDM '16. New York, NY, USA: ACM, 2016, pp. 397–406.

[17] M. F. Mokbel, X. Xiong, W. G. Aref, S. E. Hambrusch, S. Prabhakar, and M. A. Hammad, "Place: a query processor for handling real-time spatio-temporal data streams," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 1377–1380.

[18] M. F. Mokbel, X. Xiong, and W. G. Aref, "Sina: Scalable incremental processing of continuous queries in spatio-temporal databases," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 623–634.

[19] X. Xiong, M. F. Mokbel, and W. G. Aref, "Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2005, pp. 643–654.

[20] M. F. Mokbel and W. G. Aref, "Gpac: generic and progressive processing of mobile queries over mobile data," in *Proceedings of the 6th international conference on Mobile data management*. ACM, 2005, pp. 155–163.

[21] "Apatche Hadoop," http://hadoop.apache.org/, 2019.

[22] "Apatche Sark," http://spark.apache.org/, 2019.

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

[24] "Apatche Samza," http://samza.apache.org/, 2019.

[25] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L.-V. Nguyen-Dinh, W. G. Aref, M. Ouzzani, and A. Ghafoor, "M3: Stream processing on main-memory mapreduce," in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 1253–1256.

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

[27] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[28] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz, "Hadoop gis: a high performance spatial data warehousing system over mapreduce," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1009–1020, 2013.

[29] H. Vo, A. Aji, and F. Wang, "Sato: a spatial data partitioning framework for scalable query processing," in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2014, pp. 545–548.

[30] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 1352–1363.

[31] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: a distributed in-memory data management system for big spatial data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1565–1568, 2016.

[32] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1071–1085.

[33] F. Baig, H. Vo, T. Kurc, J. Saltz, and F. Wang, "Sparkgis: Resource aware efficient in-memory spatial query processing," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2017, p. 28.

[34] F. Zhang, Y. Zheng, D. Xu, Z. Du, Y. Wang, R. Liu, and X. Ye, "Real-time spatial queries for moving objects using storm topology," *ISPRS International Journal of Geo-Information*, vol. 5, no. 10, p. 178, 2016.

[35] A. Belussi, S. Migliorini, and A. Eldawy, "Detecting skewness of big spatial data in spatialhadoop," in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2018, pp. 432–435.

[36] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu, "Parallel stream processing against workload skewness and variance," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 15–26.

[37] F. Zhang, H. Chen, and H. Jin, "Simois: A scalable distributed stream join system with skewed workloads," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 176–185.

[38] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini, "When two choices are not enough: Balancing at scale in distributed stream processing," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 589–600.

[39] M. A. U. Nasir, G. D. F. Morales, D. Garcia-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 137–148.

[40] A. Shanbhag, A. Jindal, S. Madden, J. Quiane, and A. J. Elmore, "A robust partitioning scheme for ad-hoc query workloads," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 229–241.

[41] A. Shanbhag, A. Jindal, Y. Lu, and S. Madden, "A moeba: a shape changing storage system for big data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1569–1572, 2016.

[42] Z. Chen, G. Cong, and W. G. Aref, "Star: A distributed stream warehouse system for spatial data," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2761–2764.

[43] A. Belussi and C. Faloutsos, "Self-spacial join selectivity estimation using fractal concepts," *ACM Transactions on Information Systems (TOIS)*, vol. 16, no. 2, pp. 161–201, 1998.

[44] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant, "Range queries in olap data cubes," *ACM SIGMOD Record*, vol. 26, no. 2, pp. 73–88, 1997.

[45] M. Riedewald, D. Agrawal, and A. El Abbadi, "Flexible data cubes for online aggregation," in *International Conference on Database Theory*. Springer, 2001, pp. 159–173.

[46] R. Beigel and E. Tanin, "The geometry of browsing," in *Latin American Symposium on Theoretical Informatics*. Springer, 1998, pp. 331–340.

[47] N. An, Z.-Y. Yang, and A. Sivasubramaniam, "Selectivity estimation for spatial joins," in *Proceedings 17th International Conference on Data Engineering*. IEEE, 2001, pp. 368–375.

[48] C. Sun, D. Agrawal, and A. El Abbadi, "Selectivity estimation for spatial joins with geometric selections," in *International Conference on Extending Database Technology*. Springer, 2002, pp. 609–626.

[49] G. S. NIST, A. Goguen, and A. Fringa, "Risk management guide for information technology systems," *Recommendations of the National Institute of Standards and Technology*, 2002.

[50] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with sql database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 280–294, Oct 2007.

[51] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on software engineering*, no. 2, pp. 222–232, 1987.

[52] T. F. Lunt, "A survey of intrusion detection techniques," *Computers & Security*, vol. 12, no. 4, pp. 405–418, 1993.

[53] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[54] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[55] Q. Huang and P. P. Lee, "Toward high-performance distributed stream processing via approximate fault tolerance," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 73–84, 2016.

[56] B. Knasmüller, C. Hochreiner, and S. Schulte, "Pathfinder: Fault tolerance for stream processing systems," in *2019 IEEE Fifth International Conference on Big Data Computing Service and Applications (BigDataService)*. IEEE, 2019, pp. 29–39.

[57] X. Liu, A. Harwood, S. Karunasekera, B. Rubinstein, and R. Buyya, "E-storm: Replication-based state management in distributed stream processing systems," in *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 2017, pp. 571–580.

[58] J. Fang, P. Chao, R. Zhang, and X. Zhou, "Integrating workload balancing and fault tolerance in distributed stream processing system," *World Wide Web*, vol. 22, no. 6, pp. 2471–2496, 2019.

[59] Y. Hu and B. Panda, "A data mining approach for database intrusion detection," in *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 2004, pp. 711–716.

[60] C. Y. Chung, M. Gertz, and K. Levitt, "Demids: A misuse detection system for database systems," in *Integrity and Internal Control in Information Systems*. Springer, 2000, pp. 159–178.

[61] K. Ilgun, R. A. Kemmerer, and P. A. Porras, "State transition analysis: A rule-based intrusion detection approach," *IEEE transactions on software engineering*, vol. 21, no. 3, pp. 181–199, 1995.

[62] A. Srivastava, S. Sural, and A. Majumdar, "Database intrusion detection using weighted sequence mining," *Journal of Computers*, vol. 1, no. 4, pp. 8–17, 2006.

[63] Q. Liang, J. Ren, J. Liang, B. Zhang, Y. Pi, and C. Zhao, "Security in big data," *Security and Communication Networks*, vol. 8, no. 14, pp. 2383–2385, 2015.

[64] J. Ledlie and M. Seltzer, "Distributed, secure load balancing with skew, heterogeneity and churn," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 2. IEEE, 2005, pp. 1419–1430.

[65] Q. Kang, J. Xing, and A. Chen, "Automated attack discovery in data plane systems," in *12th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 19)*, 2019.

[66] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Annals of Data Science*, vol. 2, no. 2, pp. 165–193, 2015.

[67] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.

[68] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[69] X. Xu, M. Ester, H.-P. Kriegel, and J. Sander, "A distribution-based clustering algorithm for mining in large spatial databases," in *Proceedings 14th International Conference on Data Engineering*. IEEE, 1998, pp. 324–331.

[70] J. C. Dunn, "A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters," *Journal of Cybernetics*, vol. 3, no. 3, pp. 32–57, 1973.

[71] S. C. Johnson, "Hierarchical clustering schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, 1967.

[72] R. Sharan and R. Shamir, "Click: a clustering algorithm with applications to gene expression analysis," in *Proc Int Conf Intell Syst Mol Biol*, vol. 8, no. 307, 2000, p. 16.

[73] W. Wang, J. Yang, R. Muntz *et al.*, "Sting: A statistical information grid approach to spatial data mining," in *VLDB*, vol. 97, 1997, pp. 186–195.

[74] D. H. Fisher, "Knowledge acquisition via incremental conceptual clustering," *Machine learning*, vol. 2, no. 2, pp. 139–172, 1987.

[75] M. Silvano and T. Paolo, "Knapsack problems: algorithms and computer implementations," 1990.

[76] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '05. Association for Computing Machinery, 2005, p. 311–322.

[77] C. Fahy, S. Yang, and M. Gongora, "Ant colony stream clustering: A fast density clustering algorithm for dynamic data streams," *IEEE Transactions on Cybernetics*, vol. 49, no. 6, pp. 2215–2228, 2019.

[78] J. D. Kelleher, B. Mac Namee, and A. D'arcy, *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT press, 2015.

[79] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. USA: Society for Industrial and Applied Mathematics, 2007, p. 1027–1035.

[80] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[81] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: ACM, 1990, pp. 322–331.

[82] "Apatche Zookeeper," https://zookeeper.apache.org, 2019.

[83] C. Faloutsos, "Next generation data mining tools: power laws and self-similarity for graphs, streams and traditional data," in *European Conference on Machine Learning*. Springer, 2003, pp. 10–15.

[84] G. K. Zipf, "Human behavior and the principle of least effort," *Addison-Wesley Press*, 1949.

[85] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet." *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.

[86] J. Liu, S. Zhang, and Y. Ye, "Agent-based characterization of web regularities," in *Web Intelligence*. Springer, 2003, pp. 19–36.

[87] P. Barford, A. Bestavros, A. Bradley, and M. Crovella, "Changes in web client access patterns: Characteristics and caching implications," *World Wide Web*, vol. 2, no. 1-2, pp. 15–28, 1999.

[88] M. E. Newman, "Power laws, pareto distributions and zipf's law," *Contemporary physics*, vol. 46, no. 5, pp. 323–351, 2005.

[89] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, 1994, pp. 243–252.

VITA

VITA

Anas Daghistani completed his Bachelor degree in Computer Engineering from Umm Al-Qura University (UQU), Makkah, Saudi Arabia in 2011. He received his Master degree in Computer Science from King Abdullah University of Science and Technology (KAUST) in 2013. He received his second Master degree in Electrical and Computer Engineering from Purdue University, West Lafayette, IN, USA in 2019. Before joining Purdue University in Fall 2014, he was a Teacher Assistant in the Computer Engineering Department of UQU. His research interests include databases, distributed systems, big data management, and database security.