Guard: Attack-Resilient Adaptive Load Balancing in Distributed Streaming Systems

Anas Daghistani, Mosab Khayat, Muhamad Felemban, Walid G. Aref, Arif Ghafoor

Abstract—The performance of distributed streaming systems relies on how even the workload is distributed among their machines. However, data and query workloads are skewed and change rapidly. Therefore, multiple adaptive load-balancing mechanisms have been proposed in the literature to rebalance distributed streaming systems according to the changes in their workloads. This paper introduces 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. This paper proposes *Guard*, a component that detects and blocks attacks that target the adaptive load balancing of distributed streaming systems. Guard uses an unsupervised machine-learning technique to detect malicious users that are involved in the attack. Guard does not block any user unless it detects that the user is malicious. Guard does not depend on a specific application. Experimental evaluation for a high-intensity attack illustrates that Guard improves the throughput and the availability of the system by 85% and 86%, respectively. Moreover, Guard improves the minimum availability that the attacker achieves by 325%.

Index Terms—Attack-Resilient, Malicious Activity, Adaptive Load Balancing, Distributed Streaming Systems

1 Introduction

Data generated every second is rapidly increasing daily. This is due to the ubiquity of smart devices and the Internet of Things (IoT), e.g., smartphones, smart watches, health monitors, traffic sensors, and connected vehicles. Moreover, social networks generate a huge amount of data, e.g., 500 million tweets are created daily [1]. This growth of data has led to the spread of new services, e.g., smart homes, autonomous vehicles, traffic control, social network analysis, and online video games. Supporting these services has raised the demand on developing real-time, efficient, and scalable systems for processing queries. The current scale of the data being generated cannot be handled by using centralized environments. Therefore, multiple distributed streaming systems have been developed to provide scalable and real-time processing solutions. Hence, there is an increasing number of applications that are being implemented using these systems. Examples include Storm [2], Twitter Heron [3], and SparkStreaming [4].

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 can change rapidly. In addition, generally the distribution of data and queries is skewed. This skewness changes with time and user interest. The use of static load-balancing techniques does not make the system fully utilize its machines, which can lead to low throughput and high response time. Therefore, various approaches have been proposed to adaptively balance the load in distributed streaming systems according to data and query workload, e.g., STAR [5], Tornado [6], Ameoba [7].

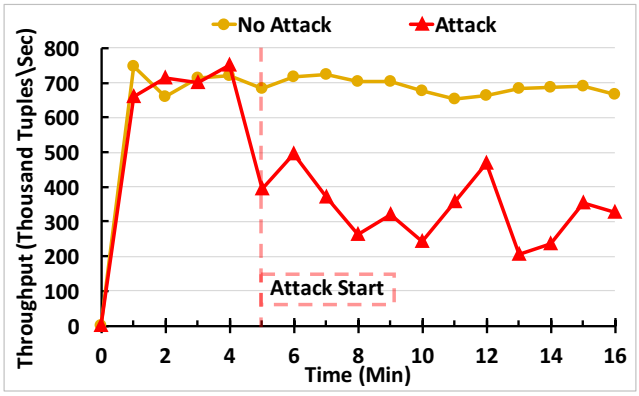


Fig. 1. The effect of an attack on a distributed streaming system

Although using adaptive load-balancing techniques significantly improves the performance of distributed streaming systems, such techniques make the system vulnerable to attacks. Attacks on the adaptive load-balancing techniques mainly target the availability of the system. To illustrate, attacks on availability can be initiated using the knowledge that the system uses an adaptive load-balancing technique to redistribute workload across the machines based on the changes in the workload [8]. In distributed streaming systems, many applications demand a querying system that returns answers in a timely manner to utilize the information at the right time. Attackers can target specific locations

- *A. Daghistani, M. Khayat are with the Department of Computer Engineering, Umm Al-Qura University, Makkah, Saudi Arabia and the Elmore Family School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN. E-mail: ahdaghistani, maakhayat@uqu.edu.sa*
- *A. Ghafoor is with the Elmore Family School of Electrical and Computer Engineering and Purdue's Center for Education and Research in Information Assurance and Security (CERIAS), Purdue University, West Lafayette, IN.E-mail: ghafoor@purdue.edu*
- *M. Felemban is with the Center for Intelligent Secure Systems and the Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia. E-mail: mfelemban@kfupm.edu.sa*
- *W. Aref is with the Department of Computer Science and Purdue's Center for Education and Research in Information Assurance and Security (CERIAS), Purdue University, West Lafayette, IN. E-mail: aref@purdue.edu*

with their queries to force the system to waste its processing power by engaging in a continuous state of load-balancing. Hence, delaying the answers in these applications can lead to misleading information. For example, law enforcement officers can predict criminal activities by querying Twitter data in real-time [9], [10], which can affect their patrolling schedule. In such cases, attackers could intentionally degrade the performance of the system, which can lead to delivery of outdated information resulting in erroneous decision making process.

Figure 1 illustrates a timeline of the throughput of a distributed streaming system with adaptive load-balancing. The results are collected while running Apache Storm [2] and SWARM adaptive load-balancing technique [11] on 6 Amazon EC2 instances. These instances are divided to form one cluster that consists of 41 virtual machines. Please refer to Section 5.2 for more details on the cluster setup. The system processes a real workload from Twitter and serves one million continues queries. Section 5.1 shows more details about the application and the dataset used for the experiment. The adaptive load-balancing has been targeted by an attack starting from Minute 5. Notice that the attack reduces the minimum throughput by 70%. The main objective of the techniques in this paper is to block the attacks and make the throughput of the system as close as possible to the throughput when there is no attack.

This paper describes a new type of attacks that forces adaptive load-balancing mechanisms of distributed streaming systems into a continuous state of rebalancing. Furthermore, this paper proposes *Guard*, a component that detects and responds to malicious attacks on adaptive load-balancing mechanisms of distributed streaming systems. The paper 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. The design of 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.

The rest of this paper proceeds as follows. Section 2 explains the way adaptive load-balancing mechanisms redistribute the workload. Section 3 presents the attack model on adaptive load-balancing mechanisms. Section 4 introduces Guard, its collected features that model the users, its detection mechanism, and its response mechanism. Section 5 studies the performance of Guard under various attack scenarios. Section 6 discusses the related work. Section 7 concludes the paper.

2 Adaptive Load-Balancing in Distributed Streaming Systems

Typically, the workload of distributed streaming systems is skewed and is changing continuously. Therefore, 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. STAR [5], Tornado [6], Ameoba [7], [12], SWARM [11], PKG2 [13], PKG5 [14], and PS2Stream [15] are examples of adaptive load-balancing mechanisms that are used in distributed streaming systems.

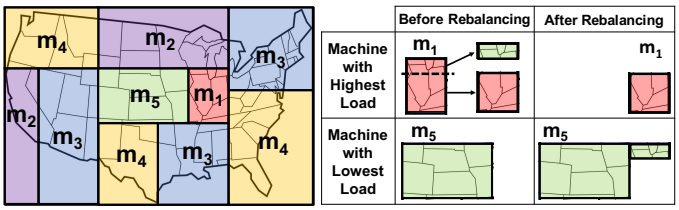


Fig. 2. Adaptive load-balancing in a spatial distributed streaming system

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. These mechanisms 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 a score, for each machine, that represents the amount of data and query workload that is 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 under-loaded machines instead of only the lowest machine. 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 parts of the workload of a machine includes moving some of its continuous queries.

Figure 2 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 its spatial dimension. Consequently, the adaptive load-balancing mechanism divides the whole space (USA map) into spatial partitions among five machines: m_1, m_2, m_3, m_4, and m_5. Figure 2 illustrates that m_1 is responsible for one partition only. This partition includes a city that gains increased user interest 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. In this scenario, the adaptive load-balancing mechanism identifies that m_1 is overloaded while m_5 has the lowest workload. Therefore, the adaptive load-balancing mechanism divides m_1's partition, which 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.

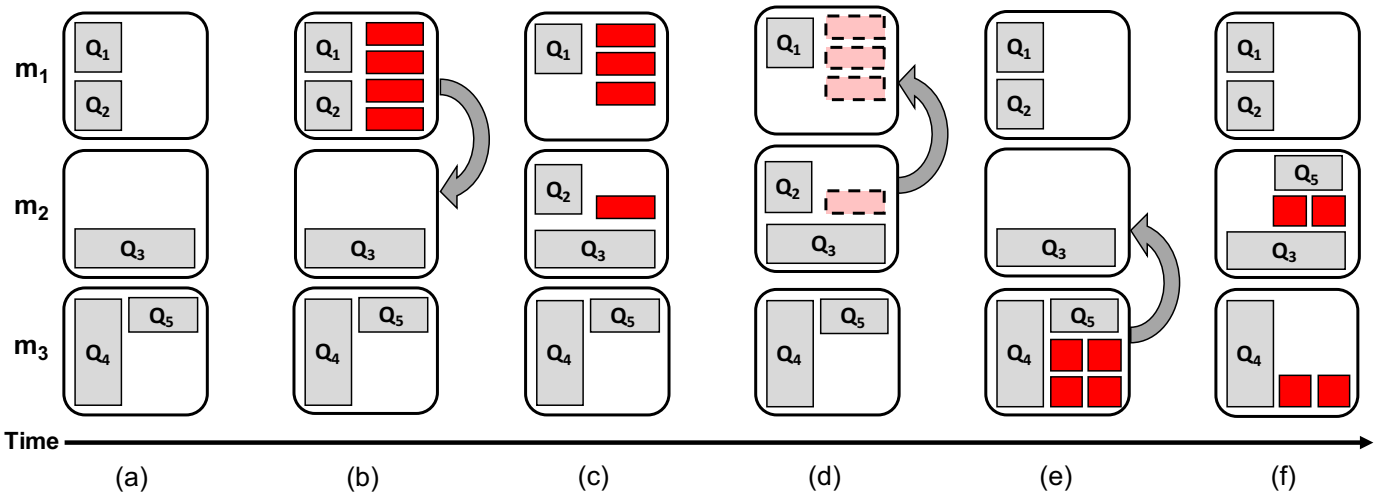


Fig. 3. Example of attack model on a distributed streaming system that contains three machines.

As long as the system is unbalanced, the adaptive load-balancing mechanism continues partitioning and moving workloads across machines.

3 Attack Model

The goal of adaptive load-balancing mechanisms in distributed streaming systems is to maintain a high level of availability. In this paper, 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 re-balancing. In particular, an attacker can trigger numerous rebalancing operations by submitting a carefully designed sequence of queries that results in creating malicious hotspots. In addition to reducing the system throughput and availability, this type of attacks can divert the system from serving real major events. This attack model is similar in concept to the stealthy Denial of Service (DoS) attack model presented in [16].

To carry out an attack, the attacker needs to sample the data points looking for the next *attack region*, which entails observing an increase in data tuples on regions where data hotspots can emerge. We assume that the attacker has a limited computational power and cannot receive/digest all the data points received by the distributed streaming system. The attacker creates an overlay grid over the spatial region. We represent the grid using an $n \times n$ array (cells) denoted by G, where g_{ij} represents the number of sampled points in the region associated with the cell. The size of each cell can be chosen based on the attackers resources. A small cell size provides a more accurate view of the next attack region. Assume that R is the rate at which the attacker can submit queries. Let δ represents the time interval used by the attacker to move its attack to a different attack region. In other words, δ represents the time taken by the attacker to create the next malicious hotspot. Note, δ is computed as $\delta = Q/R$, where Q is the total number of malicious queries submitted by the attacker. The attacker creates the array G every δ seconds. Let $\overline{G}$ be the array created in the previous interval. The attacker's goal is to find the cell that contains a potentially emerging data hotspot and to register malicious queries that creates a malicious hotspot which triggers unnecessary load balancing. Emerging data hotspots can be detected by finding the difference between the cells in G and $\overline{G}$. We denote the location of the next attack region at time k with h_k, which corresponds to the cell that observes that maximum increase in data tuples. Formally, h_k can be found by solving the following equation:

$$h_k = \underset{g_{ij}}{\operatorname{argmax}} |g_{ij} - \overline{g}_{ij}| \tag{1}$$

The cost of the attack can be expressed as a sum of two terms. The first term, C_G, corresponds to the cost (time) of sampling the data stream and populating G. The second term, C_{h_k}, represents the time for computing h_k. Formally, we model the cost of the attack as the following:

$$Cost = C_G(\delta, \lambda) + C_{h_k}(\delta, n) \tag{2}$$

where λ is the sampling rate of data tuples from the stream. Note, small δ implies more frequent attempts to create malicious hotspots that can trigger more unnecessary load balancing operations. This unwanted behavior can lead to more sever degradation in the performance of the system. However, frequent attempts require more computational power from the attacker. Large value of n leads to more accurate estimation of emerging hotspots. However, larger n increases the required computational power for C_{h_k} and increases the memory storage required to hold G and $\overline{G}$.

To illustrate dynamics of an attack, consider the following scenario in Figure 3. Initially, there are 5 registered continuous queries (Q_1, Q_2, Q_3, Q_4 and Q_5) that are distributed among three machines, m_1, m_2, and m_3, as illustrated in Figure 3(a). Assume that this query distribution among three machines is balanced. In this attack, the attacker can access the data stream that is processed by the system by means of continuous queries. From the supported queries, the attacker can conveniently learn about the data attributes (dimensions) used in partitioning/balancing the data in the system. For example, if the system serves queries about Twitter data, attackers can speculate that the data dimensions would include spatial attributes (latitude and

longitude), textual attributes (topics, and hashtags), among others. After gathering the necessary information, the attacker needs to identify and focus the malicious queries to the location of the attack region that increases the cost in Equation 2.

For example, in Figure 3(b), the attacker submits four malicious continuous queries (in red) to m_1, which serve the attacking region. Each malicious query is submitted once and is required to be checked against every new overlapping data tuple. As a result, the system detects that m_1 becomes overloaded, and decides to rebalance the workload by migrating Q_2 and one of the malicious queries to m_2 (Figure 3(c)). Thus, the attacker successfully creates a malicious hotspot that triggers a rebalance operation by submitting malicious queries. The effectiveness of the malicious hotspot can be increased by choosing a key that overlaps with large amount of data. For example, in a spatial application, inquiring about different restaurants in Chicago has a higher probability for creating a hotspot than querying about restaurants in the middle of an ocean.

Next, the attacker terminates the malicious continuous queries as shown in Figure 3(d). In this case, the system has to migrate Q_2 back to m_1 to restore the balanced state. Note, this attack results in triggering two rebalance operations. If both migrated queries were malicious at Figure 3(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 3(e) that in turn result in one more rebalance operation shown in Figure 3(d). As long as the attacker continues to succeed in creating malicious hotspots, the system persists in wasting its processing cycles by engaging in continous 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 to which he/she does not have an access permission. 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 one user attack, it is easier to detect the attacker because this user has 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 participating malicious user contributes a little to the attack. Consequently, the activity of malicious users becomes similar to the activity of legitimate users. However, collectively, this group of users perform a distributed malicious attack that is harder to detect.

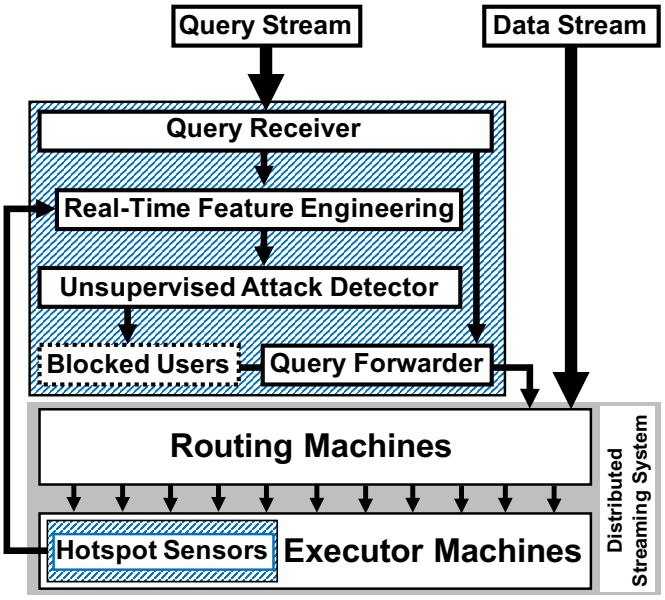


Fig. 4. Architecture of Guard and its connections with the distributed Streaming System

4 Guard: Attack Detection and Response

There are many challenges when designing a defense mechanism that prevents the type of attacks explained in the previous section. In general, characterizing the behavior of normal traffic while coping with changes due to some unusual events is a challenge. These changes in the behavior might be abnormal but still can be legitimate and should not be considered an attack since it is the nature of human to get attracted to unusual events (genuine hotspots). To solve this issue of detecting attacks in this context, it is essential to engineer a set of features that can differentiate between benign and malicious behaviors in the feature space. These features should take into consideration the dynamic nature of streaming systems. Thus, these features should consider the current behavior of users as well as the past behavior and use such information in the attack detection process. Moreover, the absence of training data sets prevents transferring knowledge while modeling the malignant behavior. Therefore, this rules out the option of using any supervised machine learning techniques. These challenges steer us toward designing Guard as an unsupervised machine learning system that detects malignant behavior in real-time without the need of any training data. The novelty of Guard lies in its features that hold a fading memory about the behavior of users as groups as well as individuals as discussed in Section 4.2.

4.1 Architecture

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 4 in a blue striped pattern. Guard's first and main component is a 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 for analysis.

Guard has a generic design 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, and 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 ID number that uniquely identifies a detection round. Each new query is tagged with the detection *RoundID*. Gaurd increments *RoundID* once a detection round ends. *HotspotID* is a global ID number that uniquely identifies the next hotspot that triggers 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*.

As illustrated in Figure 4, 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*. This detector is the main process that finds if there is an attack on the system and detects the malicious users involved in the attack. This 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. 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 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.

4.2 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. Guard collects measurements of these features periodically every detection round as a tumbling window [17]. Guard maintains the features in a hash table with *UserID* as the key. Table 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*.

The first set 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 set 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. Subsequently, 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 4.2.1

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 triggers 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 1 to reduce the distance among users involved in a multi-users coordinated attack. Their numbers are compressed representations for the sequence of hotspots in which the user appears. 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 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 triggers rebalancing during the current round. The weight of a query depends on the time it is requested. Queries requested in older rounds have lower weight. Let Q_t be the number of users' queries

TABLE 1
A brief 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
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

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:

$$\text{Feature x12} = Q_t + \frac{Q_{t-1}}{2} + \frac{Q_{t-2}}{4} + \frac{Q_{t-3}}{8} + \frac{Q_{t-4}}{16} \quad (3)$$

Notice that the rounds used in Equation 3 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 used in Equation 3, their weights do not have a big effect on the value of Feature x12. Equation 3 uses a weighting method similar to the fading method that is explained in the next paragraph. Hence, Feature x13 in Table 1 is computed by dividing x12 by the faded number of new queries (x6).

Features x1 to x5, x12, and x13 in Table 1 are measurements for the current detection round. The remaining features in Table 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 behavior over older behavior. 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 behavior in previous rounds. Feature x11 is an accumulated number. Feature x14 is an averaged number over all rounds that the user is involved in their hotspots. Feature x14 is useful to detect abnormal behaviors. It is important to view Feature x14 over previous rounds to examine whether the abnormality persists or 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 [18] to transform the features to the range from 0 to 1. The normalized features are passed to the *Unsupervised Attack Detector* as a multi-dimensional array.

$$x' = \frac{x - min(x)}{max(x) - min(x)} \quad (4)$$

4.2.1 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 4.1, Guard attaches to every query a *RoundID* to determine the round of when the query is requested. The attached list breaks down the count of the user's queries that are found in the hotspot based on the queries' *RoundID*s. The attached list includes the count of the queries that are requested during the latest five rounds only. Therefore, users who request 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 significant effect on the features that are calculated based on this list, as discussed in Section 4.2.

Guard's main component maintains the true current *RoundID*, as shown in Section 4.1. However, every *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 any 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*.

4.3 Unsupervised Attack Detector

Guard's *Unsupervised Attack Detector* is triggered periodically at 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 behavior and interaction using the K-Means++ algorithm [19]. K-Means is a well-known unsupervised clustering technique [20] 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

Algorithm 1: 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
```

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 1 lists the pseudocode for Guard's unsupervised attack detection mechanism. The detection mechanism is triggered every time the *Real-Time Feature Engineering* component passes the normalized features of the users to the detection mechanism. Guard creates a cluster, termed the *Suspicious Cluster (SC)* that initially includes all the 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 can 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 4.2. 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. Subsequently, 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 ensures that all the users in the final SC are involved in the exact hotspots and no normal users are 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 results 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 recursively re-clusters 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 contains 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. β 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 β 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. However, there is a risk to falsely consider some normal users as malicious when β is small. Notice that ALQR of the cluster that contains the malicious users increases each time tese users create a malicious hotspot. Therefore, the attacks are eventually blocked regardless of the value of β. Empirically, we find that setting the value of β around 3 is a good rule of thumb.

The next check performed on SC is to check 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 either adds SC to the *Suspicious Clusters List* or confirm that its users are malicious if the exact cluster is added before to the *Suspicious Clusters List* in a previous round. The users who are 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 between their creation. 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 for 30 minutes between the creation of consecutive malicious hotspots, the system does not stay in a continuous state of rebalancing and the attack fails. 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 ensure that all the users of this cluster coordinate to launch the attack. Therefore, Guard can significantly reduce the possibility of blocking normal users by mistake with other malicious users.

4.4 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 are 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 eventually expires 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 recovers after rebalancing its workload that includes old queries of blocked users. Not removing blocked users' old queries allows Guard to work with any distributed streaming applications. The effect of the old queries of blocked users on the system might be reduced by developing a mechanism that triggers 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.

5 Experiments

The proposed work is realized as a proof-of-concept in Apache Storm [2] and SWARM [11] in the form of its adaptive load-balancing mechanism in the context of spatial data streams.

5.1 Application and Dataset

The application is a location-aware publish-subscribe. The input stream is geotagged tweets from Twitter. Users can subscribe to get tweets in a specific spatial range by submitting continuous range queries. A tweet is geo-tagged by a point in the space. A tweet qualifies a user's query if the tweet lies inside the query's spatial range. The continuous queries are stored in an R*-Tree index [21]. Each tweet is directed to only one executor machine, and is checked against all queries using the local R*-Tree.

A real dataset from Twitter is used in the experiments. The dataset is of size 140 GB and it contains one Billion geotagged tweets in the US. The tweets are collected from January 2014 to March 2015. To form a continuous and infinite spatial data stream, the 1 Billion tweets are streamed repeatedly from the beginning each time they finish. The query workload is synthesized from Twitter's dataset to compose continuous range queries. The focal points of the queries are determined using the locations of real tweets.

5.2 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 [22] 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. 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.

5.3 Specifications of the Application and the Attacks

Experiments are all performed from a cold start. 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 grid index that divides the whole space is of size 1000 x 1000. This size allows small cities in the US to be covered by multiple cells. The spatial side lengths of queries are 0.16% of the whole space's side length (about the size of a university campus).

The system is pre-loaded with one million continuous queries coming from 110 users. When there is an attack on the system, 10 of the 110 users are malicious users while the remaining 100 are normal users. Also, malicious users are responsible for 100,000 queries when there is an attack while normal users are responsible for 900,000 queries. Continuous queries are triggered every time a new tuple arrives to allow the system to check if this tuple qualifies as a result for these continuous queries. To prevent saturating the system and to roll out the adverse effect of increasing query processing on the throughput, we artificially create a fix query processing load by registering new continues queries with a probability equal to the probability of deleting existing continues queries. We execute such behavior for both classes of users (normal users and malicious users) to void any potential detection of attackers due to controlling the query processing load variable. In this setup, the total activities (registering new queries/second + removing old queries/second) performed by each class of users (normal users and malicious users) equals 600 queries/second (i.e., 300 queries/second + 300 queries/second). At this rate, the total number of one million continues queries are kept the same but the queries themselves keep on changing at every second (due to removing and registering queries with equal probabilities).

Queries of the normal users are determined based on the locations of real-tweets as described in Section 5.1. This creates natural hotspots in the system based on real Twitter activity. Real-life workloads are often skewed [23]. Normal Users' activities are assigned to normal users using a Zipfian distribution [24], [25], [26], [27]. We configure the Zipfian distribution to follow the 80/20 rule [28], [29]. This results in almost 80% of the activities coming from 20% of normal users. Such workload puts the system in a real-life situation where it has users with varying activity levels. On the other hand, malicious users involved in the attack follow a uniform distribution for their malicious activity to maximize the attack effect while maintaining low activity rate per user to match the rate of most of the normal users (the 80% normal users that has low activity rate). This behavior makes it hard for detection system to detect attacker by looking at activity rate alone. To match the attack model we present in Section 3, malicious users register their queries to a targeted attack region while removing their older queries from the previous attack region. Therefore, the rate of the malicious activity (i.e., $2R$) determine how fast malicious hotspots are created and how fast the attack region changes (i.e., δ). The attacker chooses the location of the next malicious hotspot based on the locations of a tweets sample taken from the current data stream.

5.4 Throughput

Figure 5 shows a timeline for the system's throughput in terms of thousand tweets processed per second during every minute. Figure 5 compares the throughput while there is an attack or no attack on the system. Figures 5(a), 5(b), 5(c), and 5(d) present the effect of the attack on the system while setting 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 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 graphs in Figure 5 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 in the system and how the system can adapt to the hotspots created by normal users. 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 5(a) is small. Hence, the attacker does not succeed most of the time in creating strong malicious hotspots that force the system in a continuous state of rebalancing. Note, Guard completely blocks the attack after the creation of its second successful malicious hotspot. This complies with the Guard's design that requires at least two malicious hotspots to be created before blocking malicious users. Guard places a 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 as compared to the normal users.

Increasing malicious activity to 300 queries/second in Figure 5(b) shows more successful attack when Guard is absent. 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.

Figure 5(c) shows the results when malicious activity is 600 queries/second. The increase in malicious activity decreases the throughput more on the average and results in lower minimum throughput. Note that during the experiments of Figure 5(c), the activities of malicious users are similar to those of normal users that have average activity. In this case, Guard detects and blocks any attack faster than when the malicious activity is smaller. However, the attack achieves lower minimum throughput momentarily. The reason is that a higher malicious activity results 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.

Figure 5(d) presents the results for the case when the malicious users issue 1200 queries/second. In this scenario,

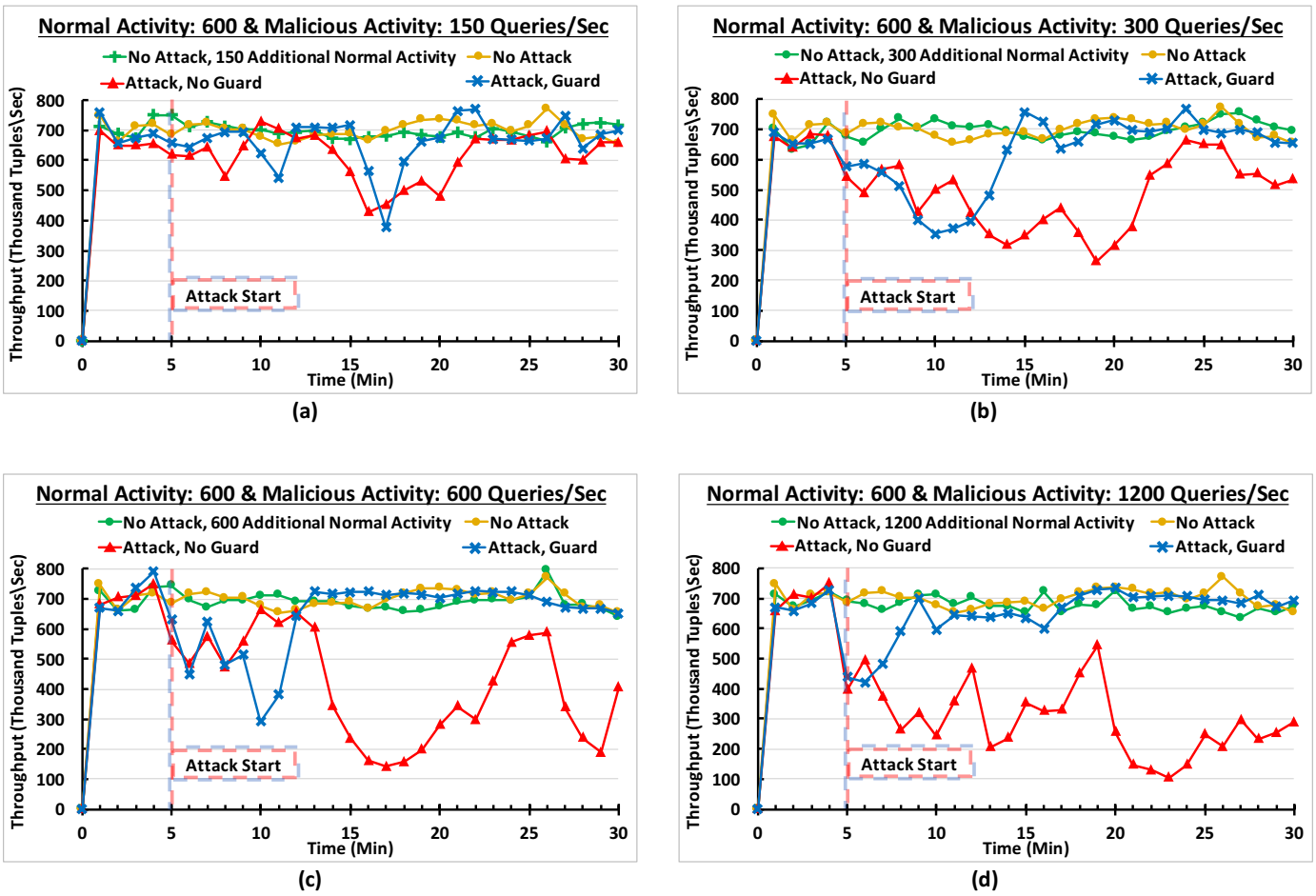


Fig. 5. Timeline showing the effect of the attack on the system's throughput while varying the malicious activity rate

the activity of malicious users is similar to the activity the active normal users. Therefore, the attack succeeds during the whole experiment as the system gets into a continuous state of rebalancing while chasing malicious hotspots. Guard detects all malicious users and blocks them faster than any of the attacks that have lower malicious activity rates. Note, the cost of the attack is higher in Figure 5(d) as compared to (a) due to the increase in the malicious activity rate. This increase in R results in reduced δ, which increases the attack cost as discussed in Section 3.

When the attacker does not succeed in creating a new malicious hotspot, the system correctly balances the workload in a timely manner and recovers its performance. Such performance improvement is equivalently achievable by using Guard for discovering and blocking a successful attack in a timely manner. Note, the job of Guard is to detect and prevent an attack that has the potential to succeed which results in performance degradation exhibited by the dips in the red curves in Figure 5.

Figure 6 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,

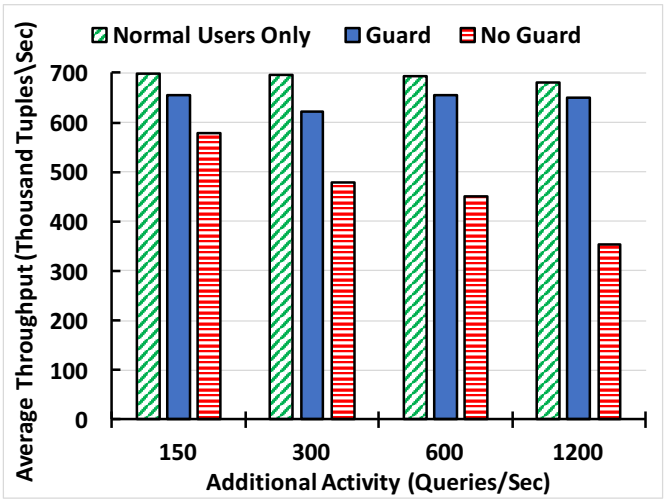


Fig. 6. Average system's throughput while varying the additional activity

the overhead of registering and terminating more queries causes a small decrease in the average throughput as the activity rate increases. When there is an attack and Guard is absent from the system, the average throughput decreases as the malicious activity rate 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 indeed malicious before blocking them. In other words, when malicious users have a similar behavior to that of normal users, Guard takes a passive policy to reduce false-positive by allowing malicious users to stay in the system while collecting more evidence to justify blocking them. This passive policy does not harm the overall performance of the system since it only applies to the scenarios in which the attackers behave normally to some extent, thus do not burden the system with extra malicious hotspots. As discussed in Section 4.3, Guard adds the attackers to the *Suspicious Clusters List* the first time they create a malicious hotspots and wait for them to create another malicious hotspot to block them. Guard enhances the average throughput between 14% and 85% when there is an attack. Note, if the average throughput is computed over more than 30-minute periods, it increases when the system has Guard and decrease when Guard is absent. The reason is that the effect of the small period before Guard blocks the attack fades with time.

To summarize Figures 5 and 6, Guard succeeds in detecting and blocking the malicious users that are involved in creating malicious hotspots. In addition, Guard differentiates between malicious and normal users regardless of which normal user the attacker is imitating. As the attacker increases the malicious activity, a decreased average throughput and a lower minimum throughput are observed. On the other hand, Guard detects and blocks attacks faster as their malicious activity increases. Therefore, Guard puts the attacker in a dilemma of not knowing at what rate the malicious activity should be sent. Recall that the objective of the attacker is to degrade the system performance by increasing the malicious activity. However, this increase results in detecting and blocking the attack faster.

5.5 Availability

The box plot in Figure 7 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 forces the system to 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 7 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 corrosponds to 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.

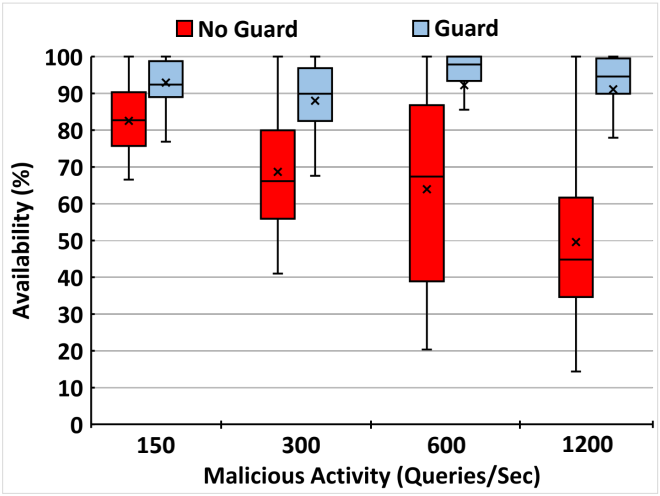


Fig. 7. Availability of the system while varying malicious activity

As shown in Figure 7, Guard achieves around 90% average availability in all experiments. In addition, 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 the experiments start without the attack for five minutes. We notice that when Guard is present, the minimum availability is much higher than when Guard is absent. Moreover, the attack manages to reach this minimum availability just for a short period of time. When Guard is absent, Figure 7 shows that the attack reduces the average and the minimum availability while increasing its malicious activity. Moreover, the time spent while the system has a low availability is increased with attacks that have a higher malicious activity as indicated by the positions of the boxes in Figure 7. The reason is that higher malicious activity allows the attack to succeed in drawing 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.

5.6 Detection and Recovery

Figure 8 shows the average time for detecting malicious hotspots and blocking malicious users for various 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 above, 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 is created. Figure 8 shows the time it takes Guard

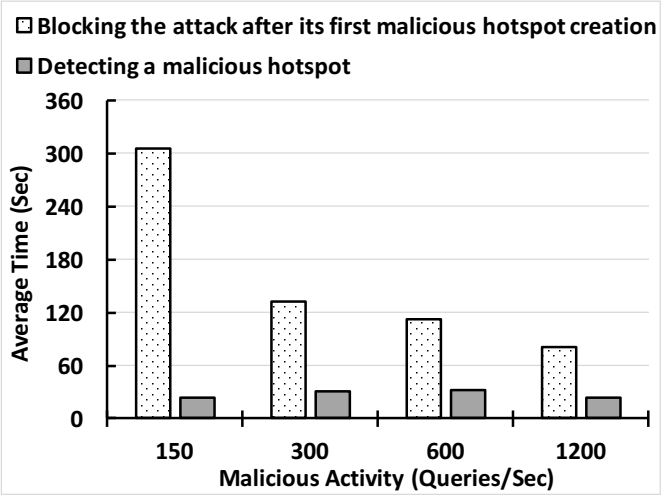


Fig. 8. Average detection and blocking times while varying malicious activity

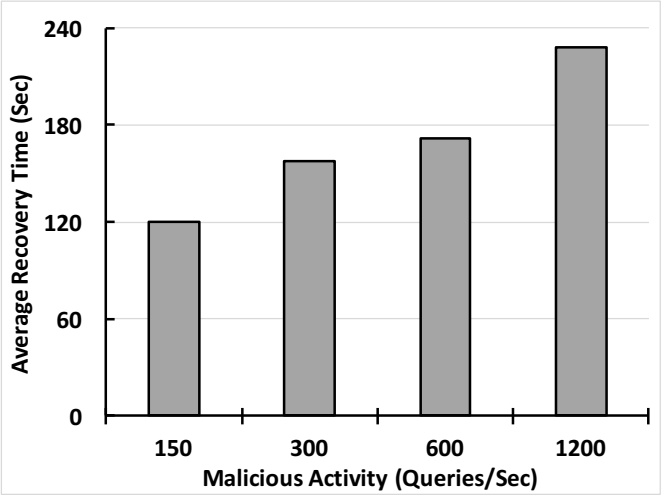


Fig. 10. Average recovery time after blocking the attack

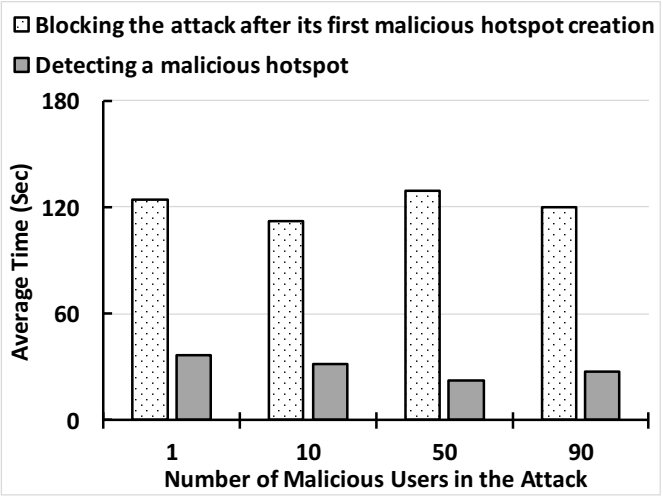


Fig. 9. Average detection and blocking times while varying the number of malicious users involved in the attack

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. This is because 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 quickly. 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 quickly.

Figure 9 shows the average times to detect malicious hotspots and block the attack while varying the number of malicious users in the attack. In the figure, the malicious activity is fixed to 600 queries/second in all experiments. Hence, the activity of every malicious user decreases as the number of malicious users involved in the attack increases. Notice that the average detection time is around 30 seconds in all the cases. Moreover, the average time to block the attack is about two minutes for all cases. The results of Figure 9 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 the variation in the detecting and blocking times, the availability, and the throughput.

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 in later rounds. Therefore, Guard achieves its main objective of blocking malicious users only when it is certain that they are malicious.

Figure 10 illustrates the average recovery time that the system takes after Guard blocks the attack while varying malicious activity. This 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 10 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.

5.7 Overhead

Figure 11 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. In this way, Guard minimizes the additional overhead with the

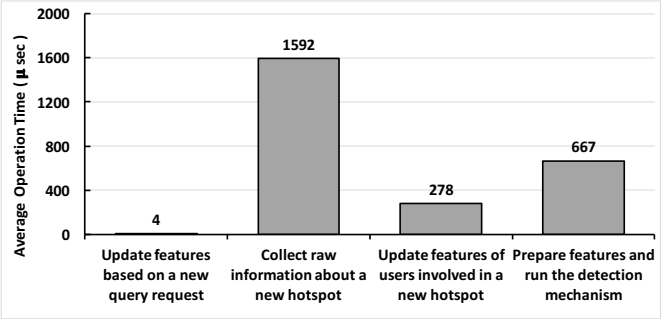


Fig. 11. Overhead of Guard's operations

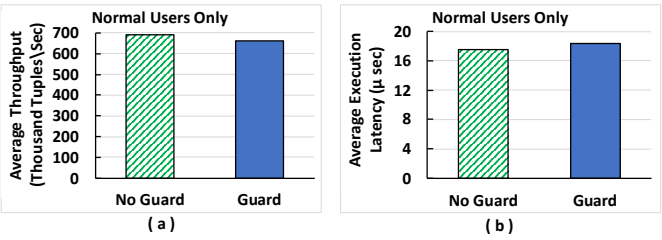


Fig. 12. Overhead of Guard on throughput and execution latency

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. In other word, the total latency added by the loop from the Hotspot sensors to the feature engineering process is about 1.6 millisecond for a workload of one million continuous queries. 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. Hence, the bottleneck is not in Guard itself but in the distributed system protected by Guard. If the system uses a better indexing scheme, the latency introduced by Guard can reduce. Notice that this overhead is incurred only when a hotspot is detected and the load-balancing technique is invoked to redistribute the workload. Hence, the overhead of this operation is negligible as compared to the overhead of redistributing the workload. Moreover, this operation is not performed frequently in a normal situation (no attack). When a *Hotspot Sensor* sends the raw information of a new hotspot, Guard spends 278 microseconds on the 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 is performed by Guard at the end of every detection round.

Figure 12 shows the overhead of Guard in terms of a decrease in the average throughput and an increase in the average execution latency of the system. The average is computed after running the system for 30 minutes with only normal users (no attack). Notice from Figure 12 that the overhead incurred by Guard is small. In particular, the reduction in the average throughput is less than 4% when there is no attack as noted from Figure 12(a). Similarly, Figure 12(b) shows that the average execution latency increases by around 3%. These results show that the price of adding Guard to the system is acceptable given the amount of damage that an attack can do to an unguarded system as discussed earlier in Sections 5.4 and 5.5.

6 Related Work

In this section, we present the work related to security threats in data streaming systems. The related work can be classified in the following categories: (1) big data streaming systems, (2) data skewness in streaming systems, and (3) security threats to streaming systems and load-balancing mechanisms.

Big data streaming systems have been developed to efficiently answer queries over data streams, e.g., PLACE [30], SINA [31], SEA-CNN [32], and Gpac [33]. Several general-purpose big data systems have been proposed to 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 [34] and Spark [35]. Batch-oriented systems require minutes or even hours to process data and are not suited for real-time processing. Yahoo S4 [36], Apache Samza [37], Apache Storm [2], Twitter Heron [3], and Spark Streaming [38] are examples of stream-oriented systems that can process data in real-time with latencies ranging between milliseconds up to few seconds. However, these systems are not optimized for spatial data processing and do not support adaptive workload.

Data Skewness has been addressed to improve the performance of distributed systems. The work in [39] proposes an approach to detect the degree of distribution in spatial data using box-counting functions. Consequently, given the degree of skewness, the best partitioning strategy is obtained using a heuristic sketch. In [40], a key-based workload partitioning strategy is proposed 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 [41] 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. The identification of hotspot key is performed using an efficient exponential counting scheme. PKG2 and PKG5 [13], [14] are stream partitioning schemes that evenly distribute the incoming workload for each key among a limited number of the system's machines.

There exist several distributed streaming systems with adaptive load-balancing for processing different types of workloads. STAR [5] is a distributed streaming warehouse for spatial data that supports low-latency and up-to-date data analytical applications by adapting to the workload changes. SWARM [11] is an adaptive load-balancing mechanism for distributed streaming systems that process spatial data. Tornado [6], [42] is a distributed in-memory streaming system for spatio-textual data that extends Storm. Tornado includes an adaptive indexing layer for dynamic redistribution of processes across the system according to changes in the data distribution and/or query workload.

PS^{2}Stream [15] is a publish-subscriber system for spatio-textual data that supports dynamic load adjustments to adapt to the changes of the workload. Amoeba [7], [12] introduce a distributed streaming system for general multi-dimensional workloads with adaptive rebalancing. Unlike previous work, Guard focuses on addressing security concerns related to load-balancing in distributed streaming systems. The adaptive load-balancing mechanisms of all these previous works are vulnerable to the attack modeled in this paper. Moreover, Guard can be deployed on these systems to detect and block attacks on their adaptive load-balancing mechanisms.

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 [43] 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. In [44], an algorithm, *k-choice*, is proposed to balance workload in systems vulnerable to Sybil attacks that can affect the skewness of query distribution over the workload. Work in [45] 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 systems in streaming systems).

7 Conclusion

This paper investigates 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%. This paper introduces Guard, a security system 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 tested using an application that processes a real dataset from Twitter while facing different attack scenarios. Guard successfully blocks all malicious users without inadvertently 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.

8 Acknowledgements

Walid G. Aref acknowledges the support of the National Science Foundation under Grant Numbers IIS-1910216 and III-1815796.

References

[1] "Internet live stats," https://internetlivestats.com/, 2020.
[2] 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.
[3] 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.
[4] 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.
[5] 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.
[6] 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.
[7] A. Shanbhag, A. Jindal, Y. Lu, and S. Madden, "Amoeba: a shape changing storage system for big data," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1569–1572, 2016.
[8] M. Guirguis, A. Bestavros, I. Matta, and Y. Zhang, "Reduction of quality (roq) attacks on dynamic load balancers: Vulnerability assessment and design tradeoffs," in *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*. IEEE, 2007, pp. 857–865.
[9] L. S. Snyder, Y. S. Lin, M. Karimzadeh, D. Goldwasser, and D. S. Ebert, "Interactive learning for identifying relevant tweets to support real-time situational awareness," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 558–568, 2020.
[10] L. S. Snyder, M. Karimzadeh, C. Stober, and D. S. Ebert, "Situational awareness enhanced through social media analytics: A survey of first responders," in *2019 IEEE International Symposium on Technologies for Homeland Security (HST)*, 2019, pp. 1–8.
[11] A. Daghistani, W. G. Aref, A. Ghafoor, and A. R. Mahmood, "Swarm: Adaptive load balancing in distributed streaming systems for big spatial data," *arXiv preprint arXiv:2002.11862*, 2020.
[12] 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.
[13] 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.
[14] 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.
[15] 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.
[16] M. Ficco and M. Rak, "Stealthy denial of service strategy in cloud computing," *IEEE transactions on cloud computing*, vol. 3, no. 1, pp. 80–94, 2014.
[17] 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.
[18] 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.
[19] 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.
[20] S. Lloyd, "Least squares quantization in pcm," *IEEE transactions on information theory*, vol. 28, no. 2, pp. 129–137, 1982.

[21] 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.

[22] "Apatche Zookeeper," https://zookeeper.apache.org, 2020.

[23] 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.

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

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

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

[27] 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.

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

[29] 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.

[30] M. F. Mokbel and W. G. Aref, "Place: A scalable location-aware database server for spatio-temporal data streams." *IEEE Data Eng. Bull.*, vol. 28, no. 3, pp. 3–10, 2005.

[31] 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.

[32] 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.

[33] 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.

[34] "Apatche Hadoop," http://hadoop.apache.org/, 2020.

[35] "Apatche Sark," http://spark.apache.org/, 2020.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

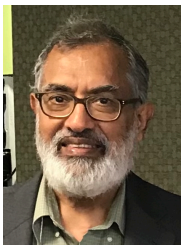
[45] Q. Kang, J. Xing, and A. Chen, "Automated attack discovery in data plane systems," in *12th {USENIX} Workshop on Cyber Security Experimentation and Test ({CSET} 19)*, 2019.

Anas Daghistani is an Assistant Professor in the Computer Engineering Department at Umm Al-Qura University, Makkah, Saudi Arabia. His research interests include databases, distributed systems, big data management, and database security. Daghistani received a Ph.D. from the Elmore Family School of Electrical and Computer Engineering at Purdue University, West Lafayette, USA.

Mosab Khayat is an Assistant Professor in the Computer Engineering Department at Umm Al-Qura University, Makkah, Saudi Arabia. His research interests include modeling and evaluation of visual analytics systems, visualization, and machine learning. Khayat received a Ph.D. from the Elmore Family School of Electrical and Computer Engineering at Purdue University, West Lafayette, USA.

Muhamad Felemban is the director of the Interdisciplinary Center for Intelligent Secure Systems at KFUPM, Dhahran, Saudi Arabia. He is also an assistant professor in the Computer Engineering Department. His research interests include security and privacy of data, the Internet of Things, and distributed systems. Felemban received a Ph.D. from the Elmore Family School of Electrical and Computer Engineering at Purdue University. He is a Member of the IEEE.

Walid G. Aref is a Professor of Computer Science at Purdue. His research interests are in extending the functionality of database systems in support of emerging applications, e.g., spatial,spatio-temporal, graph, biological, and sensor databases. He is the Editor-in-Chief of the ACM Transactions on Spatial Algorithms and Systems (ACM TSAS). He has won several best paper awards including the 2016 VLDB ten-year best paper award. He is a fellow of the IEEE.



Arif Ghafoor is currently a Professor in the Elmore Family School of Electrical and Computer Engineering at Purdue University. His research is in Cyber security, multimedia information systems, and distributed computing. He has served on the editorial boards of various journals and as a guest editor of the various special issues. He has received the IEEE Computer Society 2000 Technical Achievement Award. He is a Life Fellow of the IEEE.