Fast Distributed Correlation Discovery Over Streaming Time-Series Data

Tian Guo
EPFL, Switzerland
tian.guo@epfl.ch

Saket Sathe
IBM Research – Australia
ssathe@au.ibm.com

Karl Aberer
EPFL, Switzerland
karl.aberer@epfl.ch

ABSTRACT

The dramatic rise of time-series data in a variety of contexts, such as social networks, mobile sensing, data centre monitoring, etc., has fuelled interest in obtaining real-time insights from such data using distributed stream processing systems. One such extremely valuable insight is the discovery of correlations in *real-time* from large-scale time-series data. A key challenge in discovering correlations is that the number of time-series pairs that have to be analyzed grows quadratically in the number of time-series, giving rise to a quadratic increase in both computation cost and communication cost between the cluster nodes in a distributed environment. To tackle the challenge, we propose a framework called AEGIS. AEGIS exploits well-established statistical properties to dramatically prune the number of time-series pairs that have to be evaluated for detecting interesting correlations. Our extensive experimental evaluations on real and synthetic datasets establish the efficacy of AEGIS over baselines.

Categories and Subject Descriptors

H.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval; H.3.4 [**Distributed systems**]; G.3 [**PROBABILITY AND STATISTICS**]: Correlation and regression analysis

Keywords

Time Series Analysis; Stream Processing; Distributed Computing; Approximate Algorithm

1. INTRODUCTION

Time-series data is dramatically increasing [4, 13]. This trend is observed since the devices producing time-series data have exploded. This has, obviously, led to a demand for accommodating more and more data for efficiently producing answers to diverse problems. One such important problem, which is the focus of this paper, is to discover correlations in real-time from massive-scale time-series data. Different correlation measures, such as the Pearson correlation coefficient, cosine similarity, extended Jaccard coefficient, etc., could be employed for this task.

CIKM'15, October 19-23, 2015, Melbourne, VIC, Australia.
 ISBN 978-1-4503-3794-6/15/10 $15.00.
DOI: http://dx.doi.org/10.1145/2806416.2806440

Real-time correlation detection from large-scale time series data plays an important role in diverse applications. In data center monitoring, correlations between performance counters (e.g. CPU, memory usage, etc.) across large number of servers are typically continuously queried for recognizing the servers with correlated performance patterns [8]. In financial applications, timely discovery of correlations among stock prices can lead stock traders to spot investment opportunities [20]. In online recommendation systems, correlation mining is used to find customers with similar shopping patterns and to provide dynamic recommendations based on how a given customer's behavior correlates with other customers.

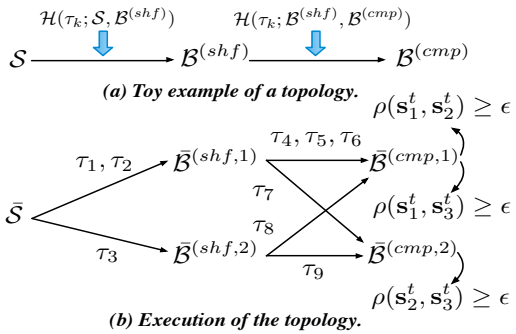
Traditional real-time data processing systems [3] designed for this task run on a standalone machine, which cannot handle the rapidly increasing amounts of time series data. This has led to the development of many distributed, fault-tolerant, and real-time computation systems [2, 4, 9, 18]. One such popular system is Apache Storm [2]. Even though these systems exist, using them for efficiently discovering correlations in time-series is still a challenging problem. Such a trend is analogous to the trend observed in map-reduce systems (e.g., Apache Hadoop); where efficiently performing database operations, like joins, using map-reduce continues to remain a challenging problem [6, 12, 15].

Distributed real-time computation.
The problem of real-time correlation discovery has been well-studied in the centralized setting [5, 7, 8, 11, 13, 16, 17, 19, 20], where the assumption is that the time series can be stored and queried on a single machine. But when this assumption does not hold, the correlation discovery methods proposed in the centralized context have to be completely re-designed for the distributed environment.

Before we deep dive into the technical details of our approaches, in the following paragraphs let us briefly understand the operation of a distributed realtime computation system. The core idea of most distributed realtime computation systems is the notion of a **topology** [2, 9]. A topology is a directed graph where the vertexes are known as **processing elements**. A processing element transforms the incoming data according to its programmed operation and transmits it to neighbouring processing element(s) as defined by the topology. In the distributed environment, one or many instances of the processing elements are executed on the nodes of a cluster, where the inter-node communication is dictated by the topology. In addition to the above processing principles, below we describe useful concepts related to time series processing:

- **Stream** is an unbounded sequence of tuples, where each tuple is a key-value(s) pair. The key or any of the values could be a number, string or a generic object. We denote a tuple as $\tau = (\tau_k, \tau_v)$ where τ_k is the key and τ_v is the value.

- **Source element** is a source of the streaming data. It can be used for reading data from a file, REST, JSON, etc., and converting it to tuples. We denote a source element by $\mathcal{S}$.
- **Action element** is a processing element that consumes tuples it receives from a source element or another action element, processes them according to the defined logic, and transmits tuples to other action elements that have subscribed to it; we denote an action element by $\mathcal{B}$.
- **Task** is an instance of either a source or action element. One or more tasks are executed in a cluster. The time series streams processed by a task are referred to as its **local streams**. A source task and action task are denoted as $\bar{\mathcal{S}}$ and $\bar{\mathcal{B}}$ respectively.
- **Parallelism** of a given action or source element is the number of its respective tasks executed in the cluster. This is a user-defined parameter. The parallelism of action element $\mathcal{B}^{(x)}$ is denoted as $\mathcal{P}^{(x)}$.
- **Shuffling function** is a function defined for each edge of the topology. It determines the task(s) of the subsequent processing elements to which a tuple emitted from a task of the preceding action element should be sent. For example, if action element $\mathcal{B}^{(shf)}$ is connected to $\mathcal{B}^{(cmp)}$ in the topology, then the shuffling function between them is denoted as $\mathcal{H}(\tau_k; \mathcal{B}^{(shf)}, \mathcal{B}^{(cmp)})$. It is popular to use a **key-based shuffling function**. It computes the hash value of a tuple key and decides to which task(s) the tuple should be sent using this hash value.



(a) Toy example of a topology.

(b) Execution of the topology.

$\tau_1 = (\langle 1 \rangle, s_{1,k})$	$\tau_4 = (\langle 1,2 \rangle, \mathbf{s}_1^t)$	$\tau_7 = (\langle 2,3 \rangle, \mathbf{s}_2^t)$
$\tau_2 = (\langle 2 \rangle, s_{2,k})$	$\tau_5 = (\langle 1,2 \rangle, \mathbf{s}_2^t)$	$\tau_8 = (\langle 1,3 \rangle, \mathbf{s}_3^t)$
$\tau_3 = (\langle 3 \rangle, s_{3,k})$	$\tau_6 = (\langle 1,3 \rangle, \mathbf{s}_1^t)$	$\tau_9 = (\langle 2,3 \rangle, \mathbf{s}_3^t)$

(c) Communicated tuples.

Figure 1: A toy example of a topology.

Distributed correlation discovery.
Given n time-series data streams, we are interested in discovering the pairs of time series that are correlated beyond a threshold ϵ over a sliding window of size h. We refer to such time-series pairs as **ϵ-correlated** pairs. Consider a toy example where we have three time-series streams $\boldsymbol{s}_1$, $\boldsymbol{s}_2$, $\boldsymbol{s}_3$ and suppose we have the topology shown in Figure 1(a). This topology performs the **naïve pair-wise correlation computation** to obtain the correlations of all the time series pairs and then filter out the unqualified pairs. Similar to performing Cartesian product using MapReduce, we use two action elements, of which the first one is responsible for annotating the tuples and the second one performs the correlation computation.

Concretely, time series are read by the source task $\bar{\mathcal{S}}$. $\bar{\mathcal{S}}$ produces tuples of the form $(i, s_{i,k})$, where $i = (1, \ldots, 3)$ and k is the timestamp of the tuple at time instant k. The tuples are then distributed to action tasks $\bar{\mathcal{B}}^{(shf,1)}$ and $\bar{\mathcal{B}}^{(shf,2)}$ using the shuffling function $\mathcal{H}(\tau_k; \mathcal{S}, \mathcal{B}^{(shf)})$, which sends tuples of time series $\boldsymbol{s}_1$, $\boldsymbol{s}_2$ are to $\bar{\mathcal{B}}^{(shf,1)}$ and time series $\boldsymbol{s}_3$ to task $\bar{\mathcal{B}}^{(shf,2)}$. Each task of $\mathcal{B}^{(shf)}$ performs two steps. First, it maintains a sliding window for each local stream. For time series i, the **sliding window** ending at time stamp t is denoted by $\boldsymbol{s}_i^t = (s_{i,t}, \ldots, s_{i,t-h+1})$ and $\boldsymbol{s}_i^t \in \mathbb{R}^h$. We use t to represent the current ending timestamp of the sliding window.

Second, tasks of $\mathcal{B}^{(shf)}$ modify the keys of the local tuples assigned to them and create output tuples of the form $(\langle i, j \rangle, \boldsymbol{s}_i^t)$ if $j > i$ and $j \neq i$, (or $(\langle j, i \rangle, \boldsymbol{s}_i^t)$ if $i > j$) for each $j = (1, \ldots, 3)$. For example, task $\bar{\mathcal{B}}^{(shf,2)}$ will emit tuples τ_8 and τ_9 as shown in Figure 1(b) and Figure 1(c). In the final phase, each task $\bar{\mathcal{B}}^{(cmp,1)}$ or $\bar{\mathcal{B}}^{(cmp,2)}$ receives two tuples for a specific time series pair, where each tuple contains the sliding window of one of the two time series that the key refers to. For example, $\bar{\mathcal{B}}^{(cmp,2)}$ will receive τ_7 and τ_9, both of them have the same key $\langle 2, 3 \rangle$. The tasks of $\mathcal{B}^{(shf)}$ modify the keys because this allows tasks of $\mathcal{B}^{(cmp)}$ to utilize the tuples having same keys to perform pair-wise correlation computation and check the threshold ϵ.

Observe that the above approach has a major shortcoming: for large number of time series (larger n), $\mathcal{B}^{(shf)}$ generates and communicates an exceptionally large number ($\mathcal{O}(n^2)$) of tuples and $\mathcal{B}^{(cmp)}$ performs quadratic correlation computation. Thus this tremendously limits its scalability. In an attempt to ameliorate this situation, we propose the **AEGIS (Affine Time-Series Grouping with Partition-Aware Shuffling)** approach. To our best knowledge, this is the first work that proposes a truly distributed and scalable solution to the continuous and distributed correlation discovery problem. Overall, this paper makes the following concrete contributions.

- We define the streaming threshold correlation (or STCOR) query, and identify the necessity of optimizing the communication and computation cost involved in processing it (cf. Section 3) .
- For reducing AEGIS's communication overhead, we propose methods for sliding window grouping and approximation, such that communicating groups becomes highly efficient than communicating individual sliding window (cf. Section 4 and 5).
- Built into AEGIS is a novel shuffling technique called **PAS (Partition-Aware Shuffling)** that has the ability to determine and shuffle sliding window groups that could contain ϵ-correlated sliding windows to the *same* action tasks. This results in localized processing within an action task and dramatic reduction in communication overhead (cf. Section 4.2).
- We propose novel pruning techniques that operate on sliding window groups. Our pruning techniques can efficiently prune the groups that do not contain ϵ-correlated sliding window pairs, and therefore can quickly eliminate the need of examining a large number of pairs (cf. Section 5).
- We implement AEGIS and competitor baselines on an open-source fully-distributed realtime computation system called Apache Storm [2], and experimentally demonstrate that AEGIS delivers orders of magnitude improved performance (cf. Section 6).

2. RELATED WORK

Numerous systems [2, 3, 9, 18] have been developed to process data in an high-speed environment. Storm [2] is a widely-used platform, which provides fault tolerance and tuple processing guarantees. Unlike Storm, some systems like S4 [9] cannot guarantee that each tuple will be processed. Zaharia *et al.* [18] proposed a new model using micro-batches for distributed stream processing.

Their approach abandons the classical topology structure in stream processing. However, such models have larger processing latency compared to the one-tuple-at-a-time model of [2, 3, 9]. However, they do not support operators for performing correlation queries.

Various indexing techniques for querying the correlations of static time-series data stored in a centralized system have been proposed in [7, 8, 13, 17]. Such techniques are not suitable for our dynamic and distributed environment. Computing real-time correlations using a standalone machine has been a key focus of [5, 11, 16, 19], however these techniques are ineffective in a distributed environment. The StatStream system [20] specializes in discovering correlations using a grid structure, but it incurs prohibitive communication cost in a distributed environment. Recently, partitioning-based approaches have attracted attention for batch distributed data processing [6, 12, 15]. However, such approaches are data-dependent and need an aprori data pre-processing step to estimate the data distribution. Scanning the entire data to update the data distribution is impossible in a streaming environment. In summary, this paper is the first work that efficiently discovers correlations from massive time-series data in a scalable distributed environment.

3. FOUNDATION

We concretely define the problem of discovering streaming correlations in streaming time-series data with the help of the **streaming threshold correlation (STCOR) query**. In this paper, we have used the Pearson correlation coefficient (henceforth, correlation coefficient) to illustrate our techniques, however, our methods can be used for computing many other correlation measures (refer Section 5 for details).

3.1 Problem statement

The correlation coefficient $\rho(\boldsymbol{s}_i^t, \boldsymbol{s}_j^t)$ between sliding windows $\boldsymbol{s}_i^t$ and $\boldsymbol{s}_j^t$ is defined as follows:

$$\rho(\boldsymbol{s}_i^t, \boldsymbol{s}_j^t) = \frac{(\boldsymbol{s}_i^t - \overline{s}_i^t \mathbb{1})^T (\boldsymbol{s}_j^t - \overline{s}_j^t \mathbb{1})}{(h-1)\sigma_i^t \sigma_j^t}, \tag{1}$$

where σ_i^t and $\overline{s}_i^t$ (or σ_j^t and $\overline{s}_j^t$) are the sample standard deviation and mean of sliding window $\boldsymbol{s}_i^t$ (or $\boldsymbol{s}_j^t$), respectively. Another important concept is that of a **normalized sliding window**. A normalized sliding window of $\boldsymbol{s}_i^t$ is denoted as $\hat{\boldsymbol{s}}_i^t = (\hat{s}_{i,t}, \cdots, \hat{s}_{i,t-h+1})$ and is defined as [20],

$$\hat{\boldsymbol{s}}_i^t = \frac{\boldsymbol{s}_i^t - \overline{s}_i^t \mathbb{1}_h}{\sigma_i^t \sqrt{(h-1)}}, \tag{2}$$

where $\mathbb{1}_h$ is an all ones vector of size h. The vector $\hat{\boldsymbol{s}}_i^t$ is of unit length and therefore $(\hat{\boldsymbol{s}}_i^t)^\top \hat{\boldsymbol{s}}_i^t = 1$. The correlation coefficient can also be written using the normalized sliding windows as follows: $\rho(\boldsymbol{s}_i^t, \boldsymbol{s}_j^t) = (\hat{\boldsymbol{s}}_i^t)^\top \hat{\boldsymbol{s}}_j^t$. Since $\hat{\boldsymbol{s}}_i^t$ is a unit length vector, each of its entries $\hat{s}_{i,k}$ varies between $-1 \leq \hat{s}_{i,k} \leq 1$. Thus, the range of variation of the normalized sliding window is known apriori, and is independent of the variation in the original sliding window $\boldsymbol{s}_i^t$. We shall later exploit this important observation to create partitions over the space of normalized sliding windows.

Additionally, there exists an important relationship between the correlation coefficient and the Euclidean distance between normalized sliding windows [20],

$$\mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_j^t) = \sqrt{2(1 - \rho(\boldsymbol{s}_i^t, \boldsymbol{s}_j^t))}, \tag{3}$$

where $\mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_j^t)$ is the Euclidean distance between $\hat{\boldsymbol{s}}_i^t$ and $\hat{\boldsymbol{s}}_j^t$. The correlation coefficient between two normalized sliding windows increases as the Euclidean distance between them decreases.

Next, suppose the total number of time-series pushed into the system is denoted by n and the set of all the sliding windows at time t is denoted by $\mathbf{S}^t$, then the STCOR query is defined as follows:

DEFINITION 3.1 (STCOR QUERY). *Given $\mathbf{S}^t$ continuously find the set of ϵ-correlated sliding window pairs for a given value of $0 < \epsilon \leq 1$. This query is formally written as:*

$$QC_\epsilon(\mathbf{S}^t) = \{\langle i, j\rangle | i \neq j, \rho(\boldsymbol{s}_i^t, \boldsymbol{s}_j^t) \geq \epsilon, \boldsymbol{s}_i^t \in \mathbf{S}^t, \boldsymbol{s}_j^t \in \mathbf{S}^t\}. \tag{4}$$

Alternatively, the STCOR query can be defined using the Euclidean distance between two normalized sliding windows, and is written as follows:

$$QD_\delta(\hat{\mathbf{S}}^t) = \{\langle i, j\rangle | i \neq j, \mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_j^t) \leq \delta, \hat{\boldsymbol{s}}_i^t \text{ and } \hat{\boldsymbol{s}}_j^t \in \hat{\mathbf{S}}^t\}, \tag{5}$$

where δ is related to ϵ as $\delta = \sqrt{2(1-\epsilon)}$.

As ϵ decreases, δ increases and vice versa. ϵ is always assumed to be greater than zero, however, in practice it could be negative. In such cases it can be shown that if the entries in one of the sliding windows are reversed, then the negative ϵ can be treated as positive [20]. Thus, without loss of generality, henceforth we only focus on the positive threshold ϵ.

3.2 δ–Hypercube Structure

We partition the space of normalized sliding windows $\hat{\mathbf{S}}^t$ into h-dimensional orthogonal regular hypercubes called **δ-hypercubes**, where each hypercube has edges of length δ. The hypercube in which a normalized sliding window $\hat{\boldsymbol{s}}_i^t$ is contained is identified by its **coordinate vector**, which is given as follows:

$$\boldsymbol{c}_i^t = \left(\left\lceil \frac{\hat{s}_{i,t}}{\delta} \right\rceil, ..., \left\lceil \frac{\hat{s}_{i,t-h+1}}{\delta} \right\rceil \right). \tag{6}$$

Given an hypercube $\boldsymbol{c}_i^t$, the set of its neighbouring hypercubes is denoted as $\mathcal{N}(\boldsymbol{c}_i^t)$. For instance, Figure 2(a) shows a three-dimensional δ-hypercube structure for sliding window $\boldsymbol{s}_i^t$ of length 3; thus, dimensions t, $t-1$, and $t-2$ are shown. A normalized sliding window $\hat{\boldsymbol{s}}_i^t$ is mapped to the red δ-hypercube whose coordinate vector is $\boldsymbol{c}_i^t$. Figure 2(b) shows the set of the neighbouring hypercubes $\mathcal{N}(\boldsymbol{c}_i^t)$ around $\boldsymbol{c}_i^t$. The number of neighbouring hypercubes of $\boldsymbol{c}_i^t$, $|\mathcal{N}(\boldsymbol{c}_i^t)|$ grows exponentially w.r.t. h ($|\mathcal{N}(\boldsymbol{c}_i^t)| = 3^h$).

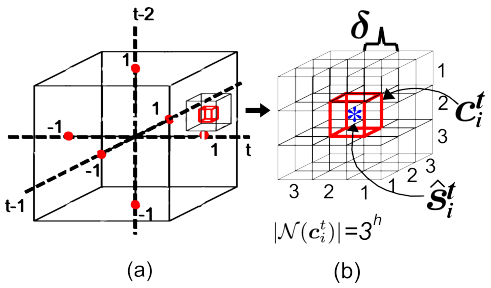


Figure 2: (a) δ–hypercube structures over the space of all normalized sliding windows. (b) $\mathcal{N}(\boldsymbol{c}_i^t)$.

An important property of such a hypercube structure is that given a $\hat{\boldsymbol{s}}_i^t$ and its $\boldsymbol{c}_i^t$, all the ϵ-correlated sliding windows are either contained in the hypercube $\boldsymbol{c}_i^t$ or in any of the hypercubes in $\mathcal{N}(\boldsymbol{c}_i^t)$. This property is due to relationship between ϵ and δ given in Definition 3.1. This property can be used for designing a shuffling strategy where an action task processing sliding window $\hat{\boldsymbol{s}}_i^t$, could emit tuples of the form $(\boldsymbol{c}_k^t, \hat{\boldsymbol{s}}_i^t)$ for all $\boldsymbol{c}_k^t \in \boldsymbol{c}_i^t \cup \mathcal{N}(\boldsymbol{c}_i^t)$. This strategy, although simplistic, does not work when $|\mathcal{N}(\boldsymbol{c}_i^t)|$ is large, which is typically the case. In such cases it leads to an exponential increase in communication cost and heavily constrained scalability [6].

4. REDUCING COMMUNICATION COST

This section introduces our core contribution – the **AEGIS approach**. As we shall experimentally demonstrate in Section 6, AEGIS exhibits orders of magnitude lower communication overhead as compared to baselines. The topology of AEGIS is depicted in Figure 3. As compared to topology of the naïve strategy from Section 1, AEGIS's topology differs in three aspects. First, for efficient shuffling and transmission of sliding windows, the first action element $\mathcal{B}_A^{(shf)}$ approximates and groups sliding windows (details will be discussed in Section 4.1). Second, action element $\mathcal{B}_A^{(cmp)}$ computes the correlation coefficients and removes both false negatives and false positives that could be introduced by $\mathcal{B}_A^{(shf)}$ (refer Section 5 for details). Third, the tuples between $\mathcal{B}_A^{(shf)}$ and $\mathcal{B}_A^{(cmp)}$ are shuffled using a novel approach refereed to as partition-aware shuffling (PAS) approach, which essentially shuffles a sliding window group from $\mathcal{B}_A^{(shf)}$ *only to those* action tasks of $\mathcal{B}_A^{(cmp)}$ containing correlated sliding windows (refer Section 4.2).

4.1 Grouping and Approximation

Each task of action element $\mathcal{B}_A^{(shf)}$ is assigned a subset of sliding windows $\mathbf{Z}^t \subset \mathbf{S}^t$ by a source task. When a new tuple is received from a source task at timestamp t, a action task of $\mathcal{B}_A^{(shf)}$ updates the normalized sliding windows that are present in its local sliding windows. This normalization is performed by incrementally updating all the quantities on the right-hand side of Eq. (2) [20]. The normalized sliding windows corresponding to $\mathbf{Z}^t$ are denoted as $\hat{\mathbf{Z}}^t$.

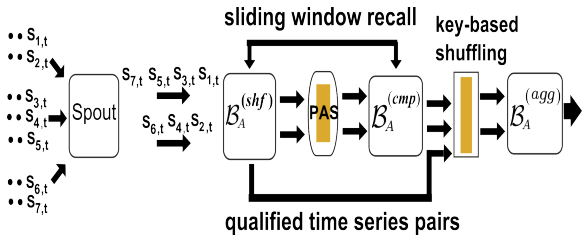


Figure 3: Topology architecture of the AEGIS approach.

Sliding window grouping: The second step that a action task of $\mathcal{B}_A^{(shf)}$ performs is to divide the sliding windows $\hat{\mathbf{Z}}^t$ into non-overlapping groups $\mathcal{G}_1^t, \ldots, \mathcal{G}_f^t$ known as **sliding window groups**. A leader is selected for each group, which is called the **pivot sliding window**. The pivot sliding window is selected such that the non-pivot sliding windows in a group can be accurately approximated using the pivot sliding window. The boundary of a sliding window group is represented by a **constant bounding hypercube (CBH)**. A CBH is a bounding hypercube constructed such that it includes the δ-hypercubes of all sliding windows in a sliding window group. For discovering the sliding window groups we propose an algorithm called **Local-Correlation Graph Based Grouping (LAP)**. LAP has two desirable properties: 1) it uses a greedy strategy to create a minimal number of groups, 2) the groups created by LAP are small, i.e., the length of each side of their CBH is at most a constant times δ. The second property is extremely important for shuffling sliding window groups only to those action tasks of $\mathcal{B}_A^{(cmp)}$ that contain ϵ-correlated pairs.

Since all the action tasks of $\mathcal{B}_A^{(shf)}$ execute the same LAP algorithm we explain this algorithm for a single task. First, correlation computation over the local sliding windows $\hat{\mathbf{Z}}^t$ is performed. This process generates an un-directed graph $L = \{V, E\}$ referred to as the **local-correlation graph**. Each vertex $v_i \in V$ ($i = 1, \ldots, |\hat{\mathbf{Z}}^t|$) represents a normalized sliding window in $\hat{\mathbf{Z}}^t$. An edge $e_{i,j} \in E$ is inserted between vertexes v_i and v_j if $\mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_j^t) \leq \delta$. In practice, the overhead of computing the local-correlation graph is controlled by making sure that only a small number of sliding windows are assigned to each task of element $\mathcal{B}_A^{(shf)}$. This can be done by increasing the parallism of element $\mathcal{B}_A^{(shf)}$ as the number of time-series n increases.

Then, LAP algorithm is executed on the local-correlation graph for finding the sliding window groups and pivot sliding windows. The objective function minimized by LAP is as follows:

$$\begin{aligned} &\underset{x_i}{\text{minimize}} \sum_{i=1}^{|\hat{\mathbf{Z}}^t|} \mathbf{I}_{\{x_i=1\}} \\ &\text{subject to} \sum_{e_{i,j} \in E} x_j \geq 1, \forall\, i = 1, \ldots, |\hat{\mathbf{Z}}^t| \\ &\qquad x_i \in \{0,1\}, \forall\, i = 1, \ldots, |\hat{\mathbf{Z}}^t| \end{aligned} \tag{7}$$

where $x_i = 1$ indicates that a vertex v_i is chosen as the pivot sliding window. The constraint ensures that each non-pivot sliding window is assigned to at least one group. The optimization problem of Eq. (7) can be reduced to a set-cover problem, which is known to be NP-complete.

Therefore, we propose a greedy version of LAP called Greedy-LAP that efficiently finds a feasible solution and has a provable near-optimality guarantee. Greedy-LAP starts by finding the set of vertexes N_{v_i} that are connected to the vertex $v_i \in L$, then it executes the following steps. STEP 1: Choose the vertex with the highest degree, let that vertex be denoted as $\bar{v}$. STEP 2: A sliding window group $\mathcal{G}_{\bar{v}}^t$ is created by making the sliding window associated with $\bar{v}$ the pivot sliding window and the sliding windows associated with the vertexes $N_{\bar{v}}$ as non-pivot sliding windows. STEP 3: This group of vertexes is removed from the graph L and STEP 1 is executed until there are no vertexes left in L. It can be shown that the above steps greedily minimize the optimization function in Eq. (7). These steps can be efficiently implemented using BFS (breadth-first-search), where we keep track of the vertexes that the algorithm is not allowed to use.

It can be proved that Greedy-LAP is a $\ln(|\hat{\mathbf{Z}}^t|)$ approximation to LAP (refer [1] for the proof). Moreover, the Greedy-LAP algorithm guarantees that the length of each side of the CBH, which contains a sliding window group, is bounded by 3δ. We call this the **3δ property** of Greedy-LAP. A proof of this property can be found in [1]. The 3δ property is heavily used in the PAS shuffling phase discussed in Section 4.2.

Sliding window approximation: This step approximates each non-pivot sliding window with its pivot. We use a particular method known as the **affine transformation** to perform this approximation [13]. Given a non-pivot sliding window $\hat{\boldsymbol{s}}_i^t$ and a pivot sliding window $\hat{\boldsymbol{s}}_m^t$, the affine transformation between $\hat{\boldsymbol{s}}_i^t$ and $\hat{\boldsymbol{s}}_m^t$ is defined as:

$$\hat{\boldsymbol{s}}_i^t = w_{i,1}\hat{\boldsymbol{s}}_m^t + w_{i,0}\mathbb{1}_h = (\hat{\boldsymbol{s}}_m^t, \mathbb{1}_h)\boldsymbol{w}_i, \tag{8}$$

where matrix $(\hat{\boldsymbol{s}}_m^t, \mathbb{1}_h)$ is of size h-by-2. Since $h \gg 2$, Eq. (8) is an over-determined system, and we use the least-squares method to obtain an approximate affine transformation $\tilde{\boldsymbol{w}}_i$ of $\boldsymbol{w}_i$ as:

$$\hat{\boldsymbol{s}}_i^t = (\hat{\boldsymbol{s}}_m^t, \mathbb{1}_h) \begin{pmatrix} \tilde{w}_{i,1} \\ \tilde{w}_{i,0} \end{pmatrix} + \boldsymbol{e}_i, \tag{9}$$

where $\boldsymbol{e}_i$ is the residual error. The advantage of using affine transformations is twofold: (a) the size of the affine transformation is not dependent on the length of the sliding window, (b) the affine transformation and the pivot sliding window can be used for estimating

certain statistical measures (including the correlation coefficient). Thus, a group can be effectively represented by the pivot sliding window plus the affine transformations for approximating the non-pivot sliding windows. In the subsequent sections, we will discuss how to shuffle a sliding window group and compute the correlations between inter-group and intra-group sliding windows.

Integration with dimensionality reduction techniques: Even though dimensionality reduction methods are not the focus of this paper, we briefly discuss how our framework can incorporate such techniques. Orthonormal transformation based dimensionality reduction (e.g., discrete Fourier transformation (DFT), random projections, etc.) can be seamlessly performed in the action element $\mathcal{B}_A^{(shf)}$. It is noteworthy that the aforementioned sliding window grouping works on such dimension-reduced sliding windows. In addition, due to the distance-preserving nature of dimensionality reduction techniques, the subsequent correlation computation can be performed on lower-dimensional sliding windows [8] without any modification.

Emitting sliding window group tuples: Lastly, each sliding window group is used for forming a output tuple of $\mathcal{B}_A^{(shf)}$. This tuple is constructed as follows. For a group $\mathcal{G}_j^t$, the *key* of the output tuple is the coordinate vector $\boldsymbol{c}_j^t$ corresponding to this group's pivot sliding window $\hat{\boldsymbol{s}}_j^t$. An action task of $\mathcal{B}_A^{(cmp)}$ uses $\boldsymbol{c}_j^t$ to locate the position and distribution of the sliding windows in $\mathcal{G}_j^t$. The *value* of the output tuple contains the pivot sliding window $\hat{\boldsymbol{s}}_j^t$ and the approximate solution $\tilde{\boldsymbol{w}}_i$ and error norm $\|\boldsymbol{e}_i\|$ for each non-pivot sliding window $\hat{\boldsymbol{s}}_i^t$ in $\mathcal{G}_j^t$. It is noteworthy that this information can be used to sufficiently reconstruct the non-pivot windows, such that correlation coefficients (and other measures) can be computed with acceptable accuracy. If the LAP algorithm, in the process of constructing the local-correlation graph, finds that a sliding window pair is already ϵ-correlated, then that pair is directly sent to a task of $\mathcal{B}_A^{(agg)}$ as shown in Figure 3.

4.2 Partition-Aware Shuffling

The naïve key-based shuffling function, discussed in Section 1 is communication inefficient due to the quadratic data replication. On the other hand, the way of shuffling sliding windows among δ-hypercubes, discussed in Section 3 makes the communication inefficiency even more severe, as the sliding window size h increases. Therefore, we propose an enhanced partition-aware shuffling function that (a) restricts the number of shuffling dimensions, thereby replicating sliding window groups independent of h and n, and (b) shuffles and replicates groups only to those tasks that could contain other groups with ϵ-correlated sliding windows.

The idea of PAS is to create coarse-grained partitions as compared to the ones created by the δ-hypercube structure of Section 3.2. The δ-hypercubes were created from the query threshold ϵ, on the contrary the PAS partitions are created based on the parallelism $\mathcal{P}_A^{(cmp)}$ of the second action element $\mathcal{B}_A^{(cmp)}$ in the topology. In PAS three things are ensured: 1) groups that lie in a particular PAS partition are consistently assigned to the same partition, which cannot be guaranteed in key-based shuffling, 2) each partition is handled by an unique action task of $\mathcal{B}_A^{(cmp)}$, 3) the sliding window groups that are not fully contained in a PAS partition are replicated and shuffled only to the neighbouring partitions, such that the replication of a sliding window group is bounded by $\mathcal{P}_A^{(cmp)}$ and communication overhead is drastically minimized.

Partitioning: The partitioning function f_{PAS} of PAS maps each normalized sliding window $\hat{\boldsymbol{s}}_i^t$ into a vector called the **partition vector** $\boldsymbol{p}_j^t$. Concretely, $f_{PAS} : \hat{\boldsymbol{s}}_i^t \mapsto \boldsymbol{p}_i^t$ and is written as:

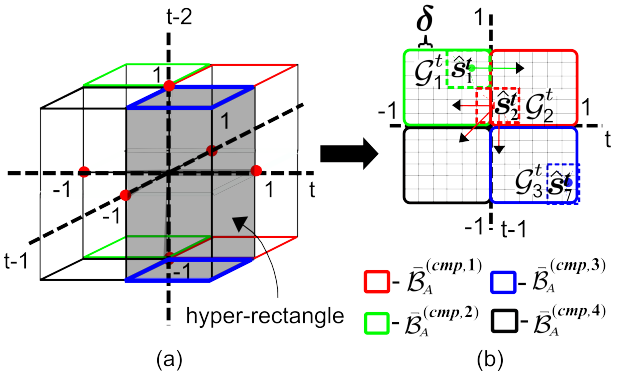


Figure 4: Illustration of PAS shuffling: (a) Parallelism $\mathcal{P}_A^{(cmp)}$ based hypercube partitioning; (b) Controlled sliding window group replication and shuffling amongst the partitions.

$$\boldsymbol{p}_i^t = f_{PAS}(\hat{\boldsymbol{s}}_i^t) = (\text{sgn}(\hat{s}_{i,t})\lceil|\hat{s}_{i,t}|\rceil, ..., \text{sgn}(\hat{s}_{i,t-h+1})\lceil|\hat{s}_{i,t-h+1}|\rceil), \tag{10}$$

where $\text{sgn}(x)$ extracts the sign of its argument. Since $-1 \leq \hat{s}_{i,t} \leq 1$, each entry of the partition vector $\boldsymbol{p}_i^t$ is either -1 or 1. Next, the number of partitions created by the partition vector is matched to the user-defined parameter $\mathcal{P}_A^{(cmp)}$ as follows: compute $\bar{h} = \log_2(\mathcal{P}_A^{(cmp)})$, and if $\bar{h} \leq h$ the dimensionality of the partition vector is reduced by retaining only the first $\bar{h}$ entries of the partition vector. In case after computing $\bar{h}$ we find that $\bar{h} > h$, we simply keep all the entries in the partition vector.

An example of this operation is demonstrated in Figure 4(a) for 3D sliding windows. Suppose, $\mathcal{P}_A^{(cmp)} = 4$ then we have $\bar{h} = 2$. Since, $\bar{h} < 3$, only dimensions t and $t-1$ are used for partitioning in Figure 4(b). Thus, normalized sliding windows lying in the blue-grey hyper-rectangle are assigned to the blue partition (and to the $\bar{\mathcal{B}}_A^{(cmp,3)}$ action task) in Figure 4(b), similarly sliding windows in the red partition are assigned to task $\bar{\mathcal{B}}_A^{(cmp,1)}$, so on and so forth.

Controlled group replication and shuffling: There could be more than one partition to which a sliding window group has to be assigned in order to find ϵ-correlated pairs. The partitions to which a group should be assigned are determined as follows. Let $\bar{h}_s \subset (1, \ldots, \bar{h})$ be a subset of dimensions referred to as the **dimension subset**. Given a dimension subset $\bar{h}_s$, the **sub-permutation set** $R_j^{\bar{h}_s}$ of a partition vector $\boldsymbol{p}_j^t$ is defined as the set of all the permutations of $\boldsymbol{p}_j^t$, such that *only* the entries corresponding to the dimensions present in the dimension subset $\bar{h}_s$ are permuted and the remaining are held constant. For example, if $\boldsymbol{p}_j^t = (-1, 1, 1)$ and $\bar{h}_s = \{2, 3\}$, then only the 2^{nd} and 3^{rd} dimension of $\boldsymbol{p}_j^t$ are permuted to form $R_j^{\bar{h}_s}$ as follows: $R_j^{\bar{h}_s} = \{(-1, -1, -1), (-1, -1, 1), (-1, 1, -1), (-1, 1, 1)\}$. If a dimension subset could be found such that its sub-permutation set contains only the partitions that could potentially have ϵ-correlated sliding windows with one or more sliding windows from group $\mathcal{G}_j^t$, then $\mathcal{G}_j^t$ should be assigned to all the partitions present in the sub-permutation set. Such a dimension subset could be found using the following lemma:

LEMMA 4.1 (DIMENSION SUBSET GENERATION). *Given a sliding window group $\mathcal{G}_j^t$ with a pivot $\hat{\boldsymbol{s}}_j^t$ and $\boldsymbol{c}_j^t$ in partition $\boldsymbol{p}_j^t$, a dimension $k = (1, \cdots, \bar{h})$ is added to the set $\bar{h}_s$ if $\delta \cdot (c_{j,k} - 2 \cdot sgn(c_{j,k})) \cdot p_{j,k}^t < 0$, otherwise it is irrelevant.*

PROOF. The proof heavily relies on the 3δ property. Refer [1] for details. □

Embedded in Lemma 4.1 is also an algorithm for group replication. We essentially scan the vectors $\boldsymbol{c}_j^t$ and $\boldsymbol{p}_j^t$ of group $\mathcal{G}_j^t$ and check the condition given by Lemma 4.1 to generate the dimension subset $\bar{h}_s$, which is used for creating the sub-permutation set $R_j^{\bar{h}_s}$.

Intuitively, the replication condition given in Lemma 4.1 can be explained as follows: a sliding window group is relevant to a partition if the group's CBH is (a) fully contained in the partition, (b) overlaps with the partition, or (c) shares a boundary with the partition. For example, consider Figure 4(*b*), the group $\mathcal{G}_3^t$ (in blue) is fully contained in the partition $(1,-1)$, therefore it is only replicated and shuffled to that partition. $\mathcal{G}_1^t$ (in green) is contained in partition $(-1,1)$ and shares a boundary with $(1,1)$, therefore it is replicated to those two partitions. Likewise, $\mathcal{G}_2^t$ is replicated to partitions $(1,1)$, $(-1,1)$, $(-1,1)$, and $(-1,-1)$.

5. COMPUTING CORRELATION MEASURES

Since all the action tasks of $\mathcal{B}_A^{(cmp)}$ perform the same function, in the following paragraphs we only describe the actions performed by a single task. Recall that each action task of $\mathcal{B}_A^{(cmp)}$ is responsible for processing all the sliding window groups assigned to a single PAS partition. Suppose, if exactly one group is shuffled to a task of $\mathcal{B}_A^{(cmp)}$, then this task has no action to perform. This is because even if there were ϵ-correlated pairs in that group, the LAP algorithm would have already detected and communicated them to tasks of the final action element $\mathcal{B}_A^{(agg)}$.

If more than one groups are shuffled to an action task, then for each group pair, the task judges whether it is necessary to examine the contents of these groups in detail or the pair can be pruned using a simple criterion. It turns out that it is indeed possible to derive a pruning criterion that only uses the pivot sliding windows of the groups to decide whether further examination is required. The following lemma discusses this criteria:

LEMMA 5.1 (PRUNING CRITERION). *Given two sliding window groups $\mathcal{G}_j^t$ and $\mathcal{G}_v^t$ respectively with pivot sliding windows $\hat{\boldsymbol{s}}_j^t$ and $\hat{\boldsymbol{s}}_v^t$, if there exists a dimension k such that $(c_{j,k}+2) < (c_{v,k}-1)$ (or $(c_{j,k}-2) > (c_{v,k}+1)$), where $k \in \{t, \cdots, h-t+1\}$, then the sliding windows in $\mathcal{G}_j^t$ and $\mathcal{G}_v^t$ are not correlated above ϵ. Otherwise, we have to compute the correlation coefficient between each pair of sliding windows in $\mathcal{G}_j^t$ and $\mathcal{G}_v^t$ and verify whether they are ϵ-correlated.*

PROOF. Refer [1]. □

Intuitively, Lemma 5.1 works as follows. A sliding window group pair requires further examination if (a) the CBH of the groups in the pair overlap with each other, or (b) they share a boundary with each other. For instance, in Figure 4(*b*) the CBHs of groups $\mathcal{G}_1^t$ and $\mathcal{G}_2^t$ share a boundary with each other, therefore further examination is required for these groups. On the contrary, the CBH of $\mathcal{G}_3^t$ in the blue partition does not overlap with that of $\mathcal{G}_2^t$, and therefore this group pair can be pruned. Next, using Lemma 5.1, only the sliding window group pairs that require further examination are selected. Then, for each group pair the sliding window pairs are formed by considering all the sliding windows in both groups together. Now, for each sliding window pair, three possibilities could occur: **(1)** both the sliding windows in the pair are non-pivot windows, **(2)** the pair has one pivot and one non-pivot windows. **(3)** both the sliding windows in the pair are pivot windows.

Recall that in a group the non-pivot sliding windows can only be reconstructed approximately. Naturally, if such approximately reconstructed sliding windows are used for correlation computation, the final query answer could contain sliding window pairs that are either false positives or false negatives. In the following paragraphs, we derive both upper and lower bounds for cases **(1)** and **(2)**, such that we can detect and delete any potential false positives or negatives in the query answer. Let us begin with the first case.

(1) Given two non-pivot sliding windows $\hat{\boldsymbol{s}}_i^t$ and $\hat{\boldsymbol{s}}_u^t$ that are approximated respectively by pivot sliding windows $\hat{\boldsymbol{s}}_j^t$ and $\hat{\boldsymbol{s}}_v^t$ as:

$$\hat{\boldsymbol{s}}_i^t = (\hat{\boldsymbol{s}}_j^t, \mathbb{1}_h)\tilde{\boldsymbol{w}}_i + \boldsymbol{e}_i,\ \hat{\boldsymbol{s}}_u^t = (\hat{\boldsymbol{s}}_v^t, \mathbb{1}_h)\tilde{\boldsymbol{w}}_u + \boldsymbol{e}_u. \quad (11)$$

We have,

$$\mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_u^t) = \|\hat{\boldsymbol{s}}_i^t - \hat{\boldsymbol{s}}_u^t\|$$
$$= \|(\tilde{w}_{i,1} \cdot \hat{\boldsymbol{s}}_j^t - \tilde{w}_{u,1} \cdot \hat{\boldsymbol{s}}_v^t) + \mathbb{1}_h(\tilde{w}_{i,0} - \tilde{w}_{u,0}) + \boldsymbol{e}_i - \boldsymbol{e}_u\|.$$

Let $\pi_{jv} = (\tilde{w}_{i,1} \cdot \hat{\boldsymbol{s}}_j^t - \tilde{w}_{u,1} \cdot \hat{\boldsymbol{s}}_v^t) + \mathbb{1}_h(\tilde{w}_{i,0} - \tilde{w}_{u,0})$. Using the triangular inequality, we can derive,

$$-\mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_u^t) \leq \|\pi_{jv}\| - \|\boldsymbol{e}_i - \boldsymbol{e}_u\| \leq \mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_u^t). \quad (12)$$

Since $\|\boldsymbol{e}_i - \boldsymbol{e}_u\| \leq \|\boldsymbol{e}_i\| + \|\boldsymbol{e}_u\|$, the second inequality of Eq. (12) is transformed to,

$$\mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_u^t) \geq \|\pi_{jv}\| - (\|\boldsymbol{e}_i\| + \|\boldsymbol{e}_u\|).$$

Similarly, applying $\|\boldsymbol{e}_i - \boldsymbol{e}_u\| \geq |\ \|\boldsymbol{e}_i\| - \|\boldsymbol{e}_u\|\ |$ to the first inequality of Eq. (12), we can write the **lower bound** between two non-pivot sliding windows:

$$\mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_u^t) \geq \max\{\|\pi_{jv}\| - (\|\boldsymbol{e}_i\| + \|\boldsymbol{e}_u\|), \\ |\ \|\boldsymbol{e}_i\| - \|\boldsymbol{e}_u\|\ | - \|\pi_{jv}\|\}. \quad (13)$$

Let us denote the above lower bound as $\mathcal{L}_1$. Similarly, the **upper bound** is derived as follows:

$$\mathcal{D}(\hat{\boldsymbol{s}}_i^t, \hat{\boldsymbol{s}}_u^t) = \|\pi_{jv} + \boldsymbol{e}_i - \boldsymbol{e}_u\| \leq \|\pi_{jv}\| + \|\boldsymbol{e}_i\| + \|\boldsymbol{e}_u\|. \quad (14)$$

Let us denote the above upper bound as $\mathcal{U}_1$.

(2) Suppose the pair under consideration contains one pivot sliding window and one non-pivot sliding window. Let $\hat{\boldsymbol{s}}_j^t$ be the pivot sliding window and $\hat{\boldsymbol{s}}_u^t$ be the non-pivot sliding window, then the **lower bound** is given as follows:

$$\mathcal{D}(\hat{\boldsymbol{s}}_u^t, \hat{\boldsymbol{s}}_j^t) \geq |\ \|\pi_{uv}\| - \|\boldsymbol{e}_u\|\ |, \quad (15)$$

where $\pi_{uv} = \tilde{w}_{u,1} \cdot \hat{\boldsymbol{s}}_v^t - \hat{\boldsymbol{s}}_j^t + \tilde{w}_{u,0}\mathbb{1}_h$. Let us denote this lower bound as $\mathcal{L}_2$. Similarly, the **upper bound** is given as,

$$\mathcal{D}(\hat{\boldsymbol{s}}_u^t, \hat{\boldsymbol{s}}_j^t) \leq \|\pi_{uv}\| + \|\boldsymbol{e}_u\|. \quad (16)$$

Let us denote this upper bound as $\mathcal{U}_2$.

(3) In this case, since both the sliding windows are pivot windows, which are not approximated, we compute the Euclidean distance between them and verify whether it satisfies the threshold δ.

Observe that for computing the above upper and lower bounds only the pivot sliding windows, affine transformations, and norm of the residual error are required. Therefore, in PAS only these quantities are emitted as tuples. As is shown above, even the approximated sliding windows are sufficient to derive effective accuracy bounds. Finally, given two normalized sliding windows, if both the sliding windows are non-pivot then bounds $\mathcal{L}_1$ and $\mathcal{U}_1$ are computed, otherwise if one window is a pivot sliding window and the other is non-pivot, then $\mathcal{L}_2$ and $\mathcal{U}_2$ are computed. Then these bounds are used for eliminating false positives as follows:

- **True Positive:** If $\mathcal{U}_k < \delta$, then the sliding window pair is qualified and is reported to the last action element.

- **False Positive**: If $\mathcal{L}_k \leq \delta \leq \mathcal{U}_k$, then sliding window pair could be a false positive. In this case, $\mathcal{B}_A^{(cmp)}$ requests $\mathcal{B}_A^{(shf)}$ to send the original sliding windows for this pair and then it verifies whether the pair is qualified by computing precise correlation coefficient. $\mathcal{B}_A^{(shf)}$ retains the original sliding windows of a particular time instant, until $\mathcal{B}_A^{(cmp)}$ has finished processing them.
- **True Negative**: If $\mathcal{L}_k > \delta$, then pair is not qualified and is dropped.

The value of k can be 1 or 2 depending on whether case **(1)** or **(2)** is valid. In the final step all the sliding window pairs that are emitted by the tasks of $\mathcal{B}_A^{(cmp)}$ are aggregated by tasks of $\mathcal{B}_A^{(agg)}$, and the results are continuously returned to the user as shown in Figure 3.

Computing alternative correlation measures: The proposed AEGIS approach can seamlessly handle diverse correlation (or similarity) measures, such as cosine similarity, extended Jaccard similarity and Euclidean distance by adopting a measure-specific normalization processes for different measures. In the interest of space, we refer the interested reader to [1].

5.1 Cost Analysis

Here, we analyze the cost incurred by the AEGIS approach. The computational cost of the element $\mathcal{B}_A^{(shf)}$ includes (a) the cost of updating the normalized sliding windows, which can be performed in constant time, and (b) the cost of local-correlation graph construction and Greedy-LAP, which is significantly low as each task is assigned only a very small number of sliding windows.

The cost of communicating between $\mathcal{B}_A^{(shf)}$ and $\mathcal{B}_A^{(cmp)}$ can be decomposed as a product of the number of replicas performed by PAS and the cost of communicating each replica. The number of replicas for a sliding window group in PAS does not depend on n and h. And due to our advanced grouping and approximation methods the size (and therefore the communication cost) of each replica is typically low, and only increases extremely slowly. Next, the computation cost incurred by each task of action element $\mathcal{B}_A^{(cmp)}$ depends on the number of sliding window groups assigned to it. Again, the number of groups assigned to each task of $\mathcal{B}_A^{(cmp)}$ is typically very small. The communication between action elements $\mathcal{B}_A^{(cmp)}$ and $\mathcal{B}_A^{(agg)}$ depends on the number of qualified sliding window pairs. Since the number of such pairs is unknown apriori, we have to omit the analysis for $\mathcal{B}_A^{(agg)}$.

6. EXPERIMENTAL EVALUATION

In this section, we perform extensive experimental evaluation comparing AEGIS with baseline approaches. First, we describe the baselines in Section 6.1. Then, we introduce the parameters and metrics used for the experimental evaluation, the datasets and the cluster setup. The implementation of AEGIS and the baselines is done using Apache Storm. We choose Storm here, because Storm has lower processing latency compared to other distributed realtime computation system (e.g, S4, Spark Streaming) due to the one-at-a-time data processing model. Action and source elements are respectively implemented as bolts and spouts in Storm. Bolts and spouts in Storm have user-specified number of tasks (i.e., parallelism) that execute in parallel in the cluster. We implement the PAS shuffling using a *custom grouping function* provided by Storm.

6.1 Baselines

NAIVE: This is the naïve approach described in Section 1, but is improved to incrementally compute correlations [7].

DFTCQ: This is a DFT (discrete Fourier Transform) based approach proposed in [20], but we have adapted it to the distributed setting. It has a topology consisting of three action elements. The first action element shuffles a DFT-reduced sliding window in the way similar to NAIVE. The second action element computes the correlation measure and forwards qualified pairs to the last element, where duplicate removal is performed. For removing false-positives a sliding window recall phase, similar to AEGIS, is performed in the second action element.

LSHCQ: LSHCQ is based on locality sensitive hashing (LSH) [14]. LSH constructs multiple hash tables to find ϵ-correlated pairs. If two normalized sliding windows are correlated, they lie closer to each other in the Euclidean space. LSHCQ uses this property to replicate and shuffle potential qualified sliding window to the same task. The topology of LSHCQ consists of three action elements. The first element computes the hash value of the normalized sliding windows for each hash table. Sliding windows that are mapped to a bucket in each hash table are shuffled to the same task of the second action element, where the correlation computation is performed over the sliding windows in each bucket per hash table. LSH parameters are chosen to minimize the processing latency while ensuring the failure probability (i.e., the probability of not reporting a certain qualified pair) at 5% [14].

6.2 Parameters and Metrics

We use four evaluation parameters to establish the efficacy of AEGIS: sliding window size h, query threshold ϵ, parallelism of $\mathcal{B}_A^{(cmp)}$ denoted $\mathcal{P}_A^{(cmp)}$, and the time interval Δ between tuples in the spout $\mathcal{S}$ known as the **injection interval**. Δ is set by adjusting the data emitting frequency of the spout $\mathcal{S}$. For other parameters, we have a **basic setup** where: $\Delta = 1$sec, $h = 100$, $\epsilon = 0.95$ and $\mathcal{P}_A^{(cmp)} = 4$.

We use three performance metrics: communication cost, processing latency, and peak capacity. **Communication cost** is measured by the amount of data units communicated between the bolt tasks of $\mathcal{B}_A^{(shf)}$ and $\mathcal{B}_A^{(cmp)}$. Here, a data unit is a basic data type, which could be float, integer, etc. For AEGIS, the communication cost incurred during PAS and sliding window recalling during correlation computation is also included. **Processing latency** is the average processing time for each task of elements $\mathcal{B}_A^{(shf)}$ and $\mathcal{B}_A^{(cmp)}$ considered together. The processing time of $\mathcal{B}_A^{(agg)}$ is not included since it is insignificant, due to the use of hash-sets for duplicate removal. As before, for AEGIS, the time spent on recalling sliding windows is also included in the processing latency. The **peak capacity** is the maximum number of time series that an approach can simultaneously process without causing bottlenecks [2, 20]. A **bottleneck** is caused when sliding windows from the current time instant have to wait (in memory), for the sliding windows from a previous time instant to finish processing [2, 20]. Bottlenecks caused by any bolt tasks are detected and reported by the Storm cluster UI [2]. All the performance metrics reported in this section are computed by averaging every 20 seconds for 10 times, after the cluster reaches a stable state.

6.3 Datasets and Cluster Details

We use one synthetic and one real dataset for evaluating all the approaches. The **synthetic dataset** is generated as follows. Given the required number of time series n, we first generate $\frac{n}{\alpha}$ seed time series. Each seed time series is generated using a random walk model [20]. From each seed time series $\boldsymbol{s}_i$, we produce α dataset as follows:

$$s_{j,t} = \gamma_{j,t} + \beta_j \cdot s_{i,t},$$

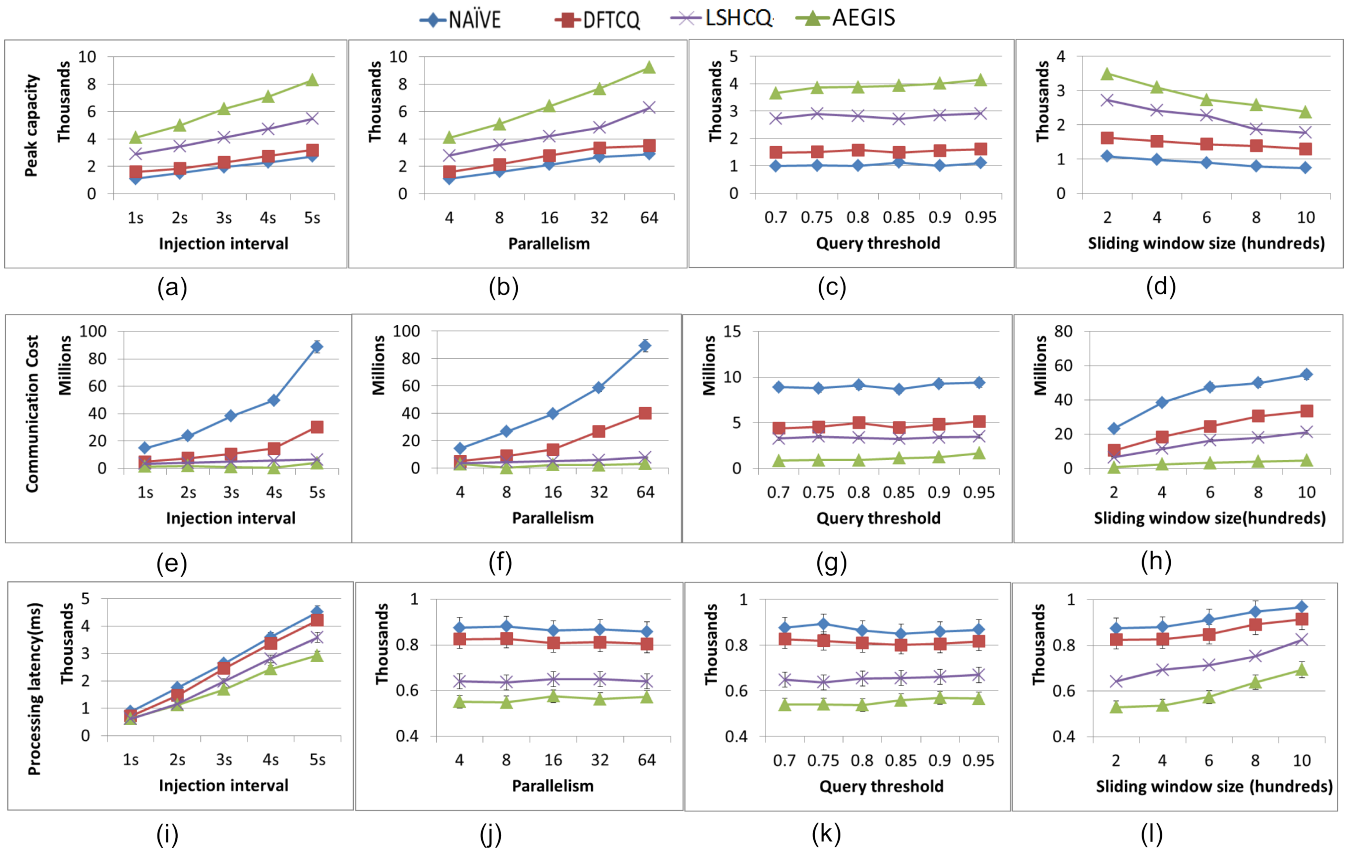


Figure 5: The performance metrics as a function of the parameters shown at peak capacity. (a)-(d) the peak capacity itself, (e)-(h) communication cost at above corresponding peak capacities and (i)-(l) processing latency at above corresponding peak capacities.

where $\gamma_{j,t}$ and β_j are real random numbers between $[0, 100]$, and β_j is sampled once for each time series $\boldsymbol{s}_j$, while $\gamma_{j,t}$ is sampled once for *each entry* in the dataset time series $\boldsymbol{s}_j$. In our experiments, we set $\alpha = 100$ and $n = 10000$.

The **real dataset** is the *Google Cluster Usage* [10] data. It records extensive activities of 12K cluster nodes from a data center over a span of 29 days. We extract three parameters: CPU usage, memory usage and disk space usage for each cluster node. The total number of time series extracted is 36K. For both datasets, the source element $\mathcal{S}$ reads the data files stored in the local file system and then continuously pushes the data to Storm at a given injection interval.

Cluster Setup: The experiments are performed using a cluster consisting of 1 master and 8 slaves. The master node has 64GB RAM, 4TB disk space (4 x 1TB disks in RAID5) and 12 x 2.30 GHz (Intel Xeon E5-2630) cores. Each slave node has 6 x 2.30 GHz (Intel Xeon E5-2630) cores, 32GB RAM and 6TB disk space (3 x 2TB disks). All the nodes are connected via 1GB Ethernet.

6.4 Measuring Peak Performance

Peak performance is the performance measured when the cluster is at peak capacity. The results shown in these set of experiments are for real data. The results for synthetic data exhibit similar trends in [1] and are omitted due to space limitations. The aim of this set of experiments is to demonstrate how each metric (communication cost, processing latency, or peak capacity) varies as a function of one parameter, while the other parameters are set to their basic setup values, *and the cluster is processing time series at peak capacity*.

For instance, assume we are interested in studying the effect of injection interval, we start by setting the other parameters to their basic setup values (i.e., $\mathcal{P}_A^{(cmp)} = 4$, $\epsilon = 0.95$, $h = 100$). Then as we start to increase the injection interval, the cluster is able to handle more and more time series, since now it has more time to process each one of them. Therefore, we start injecting more time series into the cluster until it reaches peak capacity. Then at peak capacity we note and report the values of the metrics

Peak Capacity: The peak capacity increases as a function of the injection interval and parallelism (refer Figure 5(a) and (b)). This is because more resources and processing time becomes available in the cluster thereby improving peak capacity. Therefore, at the highest level of parallelism and injection interval, AEGIS and LSHCQ exhibit 3x and 2x more peak capacity than NAIVE. In addition, the increase of query threshold has very little effect on the peak capacities of NAIVE, DFTCQ and LSHCQ approaches (refer Figure 5(c)). However, the peak capacity of AEGIS presents an increasing trend, since increasing query correlation threshold (ϵ) leads to the decrease of the distance threshold (δ), which leads to lower replication and improved communication efficiency. For maximum level of query threshold, AEGIS exhibits 3x more peak capacity than NAIVE. AEGIS achieves about 1.5x improvement over LSHCQ because LSHCQ requires a large number of hash tables to achieve high recall rate [14], and maintaining a large number of hash tables incurs high communication overhead.

On the other hand, the sliding window size affects the peak capacity adversely (refer Figure 5(d)) for all approaches. This is because when sliding window size increases approaches like DFTCQ

typically need more DFT coefficients to retain the same amount of energy, and LSHCQ takes more time for computing the hash values. And since the parallelism (or available resources) of all the action elements is fixed in this experiment the peak capacity drops to keep the system bottleneck free. However, in practice peak capacity can be maintained by increasing parallelism. Concretely, at maximum value of the sliding window size AEGIS shows a 2.5x improvement over NAIVE.

Communication Cost: As peak capacity increases with injection interval and parallelism, the communication cost increases (refer Figure 5(*e*) and *(f)*) as well. The reason is that more communication is required for maintaining that peak capacity. Even with increased communication costs, at the highest level of the injection interval, AEGIS requires 12x and 4x lower communication than NAIVE and DFTCQ respectively, at maximum level of parallelism it needs 12x less communication over NAIVE, and at most selective query threshold it needs 4x less than NAIVE. Observe that with the increase of the query threshold, the communication costs are largely stable for all approaches (refer Figure 5(*g*)). While the communication cost significantly increases with the sliding window size for NAIVE, DFTCQ, and LSHCQ (refer Figure 5(*h*)). At the highest value of sliding window size, AEGIS needs 9x, 6x, and 4x lower communication than NAIVE, DFTCQ, and LSHCQ respectively.

Processing Latency: Since peak capacity increases with increase in injection interval, the processing latency increases because more number of time series need to be processed (refer Figure 5(*i*)). In particular, for injection interval the lowest latencies shown by AEGIS are 1.5x better as compared to NAIVE. In Figure 5(*j*), the processing latency of all approaches is relatively stable when parallelism increases. This is because the amount of sliding windows distributed to each task is relatively stable due to the increase of parallelism. In addition, the processing latencies of all approaches varies little when query threshold increases (refer Figure 5(*k*)). On the other hand, the processing latency increases with increase in sliding window size (refer Figure 5(*l*)) since action elements spend more time on processing and parsing communicated long sliding windows. For sliding window size the maximum improvement in latency obtained is 1.5x as compared to NAIVE.

6.5 Analysing Sensitivity

In this set of experiments we will study how sensitive the metrics are to the changes in the parameters, while the cluster is not operating at peak capacity, but is processing a constant ($n = 8000$) number of time series. Since the cluster is only processing a constant number of time series, here we only consider two metrics: communication cost and processing latency. The **sensitivity** of a given performance metric (communication cost or processing latency) to a given parameter is measured by varying the parameter within a pre-defined range, while setting the other parameters to their basic setup values, and computing the mean and standard deviation of the variation observed in the performance metric.

For instance, say we are interested to measure the sensitivity of the communication cost to the sliding window size h. Then we vary the sliding window size in a pre-defined range and then compute the sample mean and standard deviation of the communication cost we obtained for each sliding window size. The sample mean and standard deviation are the numbers reported as sensitivity. The results for communication cost and processing latency are shown in Table 1 and Table 2 respectively. We have highlighted the last row of both tables, since communication cost and processing latency exhibit significant variation with respect to h. Due to limited space, we show the results only for the synthetic dataset, as the trends exhibited by real dataset are similar (refer [1]).

Table 1: Mean communication cost (in millions of data units) as a function of the parameters for synthetic data. Numbers in parenthesis are standard deviations. Significant variation of the communication cost is shown in boldface.

	NAIVE	DFTCQ	LSHCQ	AEGIS
Δ (10-15 secs)	640 ($\pm$12)	123 ($\pm$8)	9.7 ($\pm$0.5)	2.8 ($\pm$0.7)
$\mathcal{P}_A^{(cmp)}$ (4-64 tasks)	632 ($\pm$ 16)	134 ($\pm$11)	11.1 ($\pm$0.9)	3.5 ($\pm$1.8)
ϵ (0.7-0.95)	624 ($\pm$23)	117 ($\pm$7)	10 ($\pm$1.2)	3.2 ($\pm$1.1)
h (100-1000)	**840 ($\pm$128)**	**213 ($\pm$67)**	**43.3 ($\pm$8.9)**	**29 ($\pm$6.3)**

Table 2: Mean processing latency (in seconds) as a function of the parameters for synthetic data. Numbers in parenthesis are standard deviations. Significant variation of the processing latency is shown in boldface.

	NAIVE	DFTCQ	LSHCQ	AEGIS
Δ (10-15 secs)	8.6 ($\pm$1.1)	8.3 ($\pm$1.1)	6.7 ($\pm$1.2)	4.5 ($\pm$0.8)
$\mathcal{P}_A^{(cmp)}$ (4-64 tasks)	7.5 ($\pm$1.3)	6.8 ($\pm$1.2)	5.3 ($\pm$1.3)	4.0 ($\pm$0.6)
ϵ (0.7-0.95)	8.2 ($\pm$1.5)	8.7 ($\pm$1.5)	7.1 ($\pm$1.7)	4.4 ($\pm$1.1)
h (100-1000)	**11.6 ($\pm$ 6.4)**	**9.1 ($\pm$ 5.8)**	**8.2 ($\pm$4.9)**	**7.3 ($\pm$4.2)**

Communication Cost: In Table 1, it can be observed that for all the parameters the mean communication cost of AEGIS is relatively stable and orders of magnitude lower as compared to the others. For the sliding window size h, AEGIS has nearly 20x, 10x and 2x lower cost as compared to NAIVE, DFTCQ and LSHCQ. Similarly, for the query threshold ϵ, AEGIS is nearly 60x, 10x and 3x more efficient than NAIVE, DFTCQ and LSHCQ. The injection interval Δ and the parallelism of $\mathcal{P}_A^{(cmp)}$ show similar trends. Thus, AEGIS is highly effective in managing the communication cost when dealing with significantly large variations of the parameters. This is because PAS in AEGIS robustly mitigates changes to the communication cost arising from different values of the parameters.

Processing Latency: The latency of AEGIS is lower and more robust as compared to NAIVE, DFTCQ and LSHCQ. Concretely, for the injection interval Δ, AEGIS approach has nearly 2x lower latency as compared to NAIVE. For the query threshold ϵ, average improvement in the latency of AEGIS *w.r.t.* to NAIVE and DFTCQ is approximately 2x. When sliding window length increases, the processing latencies of all the approaches increase and thus present larger mean and standard errors. Specifically, the latency of AEGIS is about 50% lower as compared to NAIVE. Overall, AEGIS exhibits orders of magnitude improvement in communication cost and processing latency, and is significantly robust to changes in the parameters.

6.6 Analysing Pruning Power

This set of experiments evaluate the pruning power of AEGIS against LSHCQ. **Pruning power** is defined as the ratio of the number of pairs that are pruned (without having to compute the correlation measure) to the total number of time-series pairs. Higher values of pruning power are considered better. Since the pruning

Table 3: Pruning power of AEGIS and LSHCQ as a function of query threshold ϵ and sliding-window length h for real data. The upper value in each cell corresponds to AEGIS, while the lower value corresponds to LSHCQ.

ϵ \ h	200	400	600	800	1000
0.7	**0.858** **0.552**	0.836 0.574	0.819 0.599	0.764 0.542	**0.717** **0.583**
0.75	0.861 0.582	0.862 0.584	0.811 0.579	0.778 0.532	0.757 0.563
0.8	0.864 0.612	0.870 0.604	0.832 0.588	0.761 0.599	0.777 0.602
0.85	0.881 0.653	0.862 0.638	0.831 0.669	0.837 0.644	0.776 0.631
0.9	0.884 0.663	0.872 0.628	0.832 0.667	0.812 0.658	0.701 0.641
0.95	**0.907** **0.713**	0.865 0.735	0.836 0.714	0.798 0.705	**0.736** **0.716**

power is directly affected by the query threshold ϵ and sliding-window length h, in Table 3 we present pruning power as a function of these two parameters. The results in Table 3 are on the real dataset (refer [1] for results on synthetic data). In this experiment Δ and $\mathcal{P}_A^{(shf)}$ are set to their basic set-up values.

Here we only compare AEGIS with LSHCQ, because NAIVE and DFTCQ approaches do not perform pruning and always evaluate all sliding window pairs. In each cell of Table 3, the upper value is for AEGIS, while the lower value is for LSHCQ. In AEGIS, PAS only shuffles together the sliding window groups that could potentially contain ϵ-correlated pairs, therefore many pairs are pruned during the data shuffling phase. Furthermore, higher values of ϵ and shorter sliding windows (lower h) lead to insignificant residual errors while approximating the non-pivot sliding windows in the sliding window groups (refer Section 4), and lead to tighter bounds on the derived correlation (refer Section 5); thereby significantly increasing the pruning power.

It is observed that pruning power of AEGIS decreases as h increases and increases as ϵ increases. AEGIS achieves the maximum pruning power 0.907 at the maximum $\epsilon = 0.95$ and minimum $h = 200$. The pruning power of LSHCQ is largely affected only by ϵ, while varying little under different sliding window sizes. Overall, AEGIS achieves a maximum of 50% improvement over LSHCQ when $\epsilon = 0.7$ and $h = 200$.

7. CONCLUSION

In this paper, we proposed approaches for real-time correlation discovering in large time-series data. Our main proposal the AEGIS approach, uses intelligent time series grouping, approximation, and partition-aware data shuffling methods to dramatically improve performance. We demonstrated that AEGIS significantly reduces the communication cost and can easily operate on a large number of time series through extensive experiments.

8. ACKNOWLEDGMENTS

This work was supported by Nano-Tera.ch through the OpenSenseII project.

9. REFERENCES

[1] Tech. report – `http://infoscience.epfl.ch/record/210363`.

[2] Storm. `http://storm-project.net/`.

[3] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora : A new model and architecture for data stream management. *VLDB Journal*, pages 120–139, 2003.

[4] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, et al. Scuba: diving into data at facebook. *PVLDB*, 6(11):1057–1067, 2013.

[5] R. Cole, D. Shasha, and X. Zhao. Fast window correlations over uncooperative time series. In *ACM SIGKDD*, pages 743–749. ACM, 2005.

[6] S. Fries, B. Boden, G. Stepien, and T. Seidl. Phidj: Parallel similarity self-join for high-dimensional vector data with mapreduce. In *ICDE*, pages 796–807. IEEE, 2014.

[7] Y. Li, M. L. Yiu, Z. Gong, et al. Discovering longest-lasting correlation in sequence databases. *PVLDB Endowment*, 6(14):1666–1677, 2013.

[8] A. Mueen, S. Nath, and J. Liu. Fast approximate correlation for massive time-series data. In *SIGMOD*, pages 171–182, 2010.

[9] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177. IEEE, 2010.

[10] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, 2011.

[11] Y. Sakurai, S. Papadimitriou, and C. Faloutsos. Braid: Stream mining through group lag correlations. In *ACM SIGMOD*, pages 599–610. ACM, 2005.

[12] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. In *PVLDB*, 2014.

[13] S. Sathe and K. Aberer. AFFINITY: Efficiently querying statistical measures on time- series data. In *ICDE*, pages 841–852, 2013.

[14] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *PVLDB*, 6:1930–1941, 2013.

[15] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. In *19th ACM SIGKDD*, pages 829–837. ACM, 2013.

[16] D. Wu, Y. Ke, J. X. Yu, S. Y. Philip, and L. Chen. Leadership discovery when data correlatively evolve. *World Wide Web*, 14(1):1–25, 2011.

[17] Q. Xie, S. Shang, B. Yuan, C. Pang, and X. Zhang. Local correlation detection with linearity enhancement in streaming data. In *ACM CIKM*, pages 309–318. ACM, 2013.

[18] 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. In *4th USENIX HotCloud*, pages 10–10. USENIX, 2012.

[19] T. Zhang, D. Yue, Y. Gu, and G. Yu. Boolean representation based data-adaptive correlation analysis over time series streams. In *CIKM*, pages 203–212. ACM, 2007.

[20] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, pages 358–369, 2002.