TECHNICAL UNIVERSITY OF CRETE

Scaling out streaming time series analytics on Storm

by

Nikolaos Pavlakis

A thesis submitted in partial fulfillment for the
M.Sc. degree in Computer Science

in the

Software Technology And Network Applications Laboratory — SoftNet
School of Electronic and Computer Engineering

March 2017

"Information is the oil of the 21st century, and analytics is the combustion engine."

Peter Sondergaard

TECHNICAL UNIVERSITY OF CRETE

Abstract

Software Technology And Network Applications Laboratory — SoftNet
School of Electronic and Computer Engineering

M.Sc. - Computer Science

by Nikolaos Pavlakis

Data can provide meaningful insights, *if* we are able to process it. We live in a time where the rate with which data is being generated grows exponentially, and extracting useful information from all this data, becomes harder and harder, thus mandating efficient and scalable data analytics solutions. Oftentimes, the input data to analytics applications is in the form of massive, continuous *data streams*. Consider the example of the global stock markets: An interesting piece of information for traders, portfolio managers, and so on, are the correlation/dependence patterns between different market players (e.g., equities, indexes, etc.); yet, such patterns typically change very rapidly over time, and the information is only valuable if it becomes available *in real time* (e.g., for algorithmic trading). This implies that stock market data needs to be processed in a streaming fashion, typically focusing only on a *sliding window* of recent readings (e.g., "monitor all correlations during the last hour"). In addition, data stream processing solutions need to be scalable as there are thousands of market players, implying millions of possible correlation/dependence pairs that need to be tracked in real time. This thesis introduces efficient algorithms and architectures for tackling the problem of monitoring the pairwise dependence among thousands of data streams, and introduces a generic stream processing framework, *T-Storm*, which can be used in order to easily and efficiently develop, scale-out, and deploy large-scale stream analytics applications.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Minos Garofalakis, for guiding me throughout this period, and always providing thoughtful and useful insights when obstacles were presented. Moreover, I would like to thank him for trusting me with a Research Assistant position in QualiMaster, which was a very interesting project through which I gained massive useful experience and had the chance to meet with like-minded people and collaborate in a team.

I would like to thank my dear friend A. Nydriotis for all our useful talks during this period of my life, including, but not limited to, geeky and technical discussions. I am also grateful to my best friends, E. Alibertis, D. Iliou, N. Kofinas, E. Kountouraki, S. Pavlioglou, E. Soulas, who although not being physically near me, played and important role in my life throughout all this period.

Furthermore, I would like to thank my dear Nikoleta, for supporting me and always being there for me, constantly providing inspiration and reassurance during all those time periods when I spent my whole day in my University office.

Last, but definitely not least, I am grateful to my parents, George and Natassa, for constantly caring about me in an unconditional way, and not holding grudges when I was abrupt towards them over the phone in times of stress.

This work has been partially funded from the European Union's Seventh Framework Programme under Grant Agreement 619525 (QualiMaster, http://qualimaster.eu/).

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation & Thesis Contribution

"Every two days now we create as much information as we did from the dawn of civilization up until 2003", Google ex-CEO Eric Schmidt noted in 2010.

In an era when the rate at which information is being generated is growing exponentially, there is also increasing need for efficient ways to tackle all the "Big Data" problems. Since it is often impractical, or even impossible, for modern systems to store all the incoming data that they need to process, streaming time series analytics are often employed in an effort to analyze continuous data streams and extract meaningful statistics and characteristics about the data. Such streaming approaches, try to process each data point without storing it (or at least use some kind of expiration window in order to not store it indefinitely), in order to maintain an approximate representation of the original data set with specific characteristics, depending on the problem being tackled. For example, if we want to keep an updated cumulative sum of all the points that each player has scored during the last ten basketball games, instead of storing each point being scored, along with the player who scored it, and computing the sum for each player in the end, we only need to store one counter for each player, which is incremented by the amount of points scored each time this player scores. This simplistic example illustrates that we can reduce space and time required by specific problems by maintaining a summarized version ("synopsis") of the original data, and using this summary to answer queries.

This thesis attempts to explore the domain of time series analytics and provide efficient solutions towards scaling out time series analytics for large data sets over distributed systems. More specifically, there are two main contributions; a) we introduce T-Storm, a framework that can be used in order to implement time series analytics on cluster

architectures and easily manage input and output in a streaming manner, and b) we create various components that tackle the challenging problem of monitoring the pairwise dependence among thousands of incoming data streams. These components are interchangeable, with each one of them keeping track of a different measure of dependence, with different characteristics, so that various trade-offs can be leveraged depending on user intent. Furthermore, we discuss numerous optimizations to enhance scalability, and study their performance in different scenarios.

1.2 Outline

Chapter 2 provides a detailed formulation of the problem we are addressing, along with the relevant technical challenges. Chapter 3 provides useful background related to concepts used throughout the thesis, while Chapters 4 and 5 present our main contributions in detail. Chapter 6 analyzes the performance of our T-Storm components. Finally, Chapter 7 provides information about related work, while Chapter 8 gives some closing remarks along with interesting directions for future.

Chapter 2

Problem Statement

This chapter presents the problem that addressed in this thesis, along with the related key technical challenges.

2.1 Problem Formulation

The goal of this thesis is to develop a unified framework focused on streaming time series analytics. Our framework needs to provide specific characteristics, namely:

- **Streaming model**. Input data is received or generated dynamically, and the system needs to process incoming updates as they appear.
- **Online computation**. Processing incoming data needs to be done in an online fashion (i.e. real or near-real time) so that the system does not "lag" indefinitely.
- **Distributed computation**. The system needs to efficiently distribute (i.e. "scale out") the computation among a set of machines. Each machine is responsible for computing a sub-set of the problem, yet the results being produced need to be the same as if everything was executed in a centralized manner.

We focus on the generic domain of computing pairwise statistics among several thousands of incoming data streams, since it poses a challenging problem due to the number of pairs scaling quadratically with respect to the number of input data streams. Without loss of generality, we apply our techniques on real input data gathered from the financial domain (e.g., stock price updates) and aim to monitor the pairwise dependence among several thousands of financial entities, so that interested parties (e.g., portfolio managers) can monitor highly correlated pairs. The components developed have also been incorporated and used in the infrastructure of the QualiMaster project [1].

2.2 Technical Challenges

Our problem setting poses several technical challenges, some of which are:

- **Quadratic cost**. Since we are interested in an "all-pairs" calculation, the complexity of the problem scales quadratically with respect to the input. For example, if we need to monitor 3000 incoming data streams, the load of the system amounts to roughly 4.5 million pairs to be tracked.
- **Frequent updates**. Some of the input data streams can be very "fast" (e.g., popular stocks being traded very frequently), which amounts to a total of several thousands updates per second for the system.
- **Time sensitivity**. In domains such as the finance, latency is a very important measure, and needs to be minimized in order for the results to be useful. For example, detecting a severe market risk needs to be reported in real time, so that portfolio managers can protect themselves against a catastrophic scenario.

Chapter 3

Background

This chapter summarizes required background knowledge about specific subjects that are mentioned throughout the thesis.

3.1 Distance Measures

In mathematics, a metric or distance measure [2] is a function that defines a distance between a pair of elements, in an n-dimensional space. Each distance measure satisfies the following conditions:

- $d(x, y) \geq 0$, the distance is non-negative
- $d(x, y) = 0 \Leftrightarrow x = y$, zero distance means the points are identical
- $d(x, y) = d(y, x)$, symmetry
- $d(x, z) \leq d(x, y) + d(y, z)$, triangle inequality

There are many different distance measures in literature, including Euclidean, Hamming, Cosine, Chebyshev, Manhattan, etc.

3.1.1 Euclidean Distance

The Euclidean distance between points x and y is the length of the line segment connecting them. If $x = (x_1, x_2, ..., x_n)$ and $y = (y_1, y_2, ..., y_n)$ are two points in Euclidean n-space, then the distance (d) from x to y, or from y to x is given by the Pythagorean formula:

$$d(x,y) = d(y,x) = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2 + \cdots + (y_n - x_n)^2} = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}$$

3.2 Correlation and Dependence

The correlation coefficient [3, 4] between two time series $X(x_1, ..., x_n)$ and $Y(y_1, ..., y_n)$ is an important measure that shows to what extent the two series are correlated. It is inversely proportionate to the distance between them, and ranges from -1 to 1, with a value of 0 implying that the variables are completely independent, a positive value suggesting that X and Y are correlated and a negative value suggesting they are anti-correlated (i.e. inversely correlated).

Different types of correlation measures include Pearson correlation, Kendall rank correlation, Spearman correlation, and the Point-Biserial correlation.

3.2.1 Pearson Correlation

The Pearson correlation coefficient between two time series X and Y, is defined as

$$corr(X,Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}},$$

where $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, represent the mean values of X and Y respectively.

The Pearson correlation coefficient is positive if X_i and Y_i tend to be simultaneously greater than, or simultaneously less than, their respective means, while negative (anti-correlation) if X_i and Y_i tend to lie on opposite sides of their respective means. Moreover, the stronger is either tendency, the larger is the absolute value of the correlation coefficient.

Although the Pearson correlation coefficient is a very important measure, it only captures *linear* correlation. Since we are also interested in *non-linear* dependence, we will also address more complex statistical measures of time-series dependence such as Mutual Information [5] and Transfer Entropy [6].

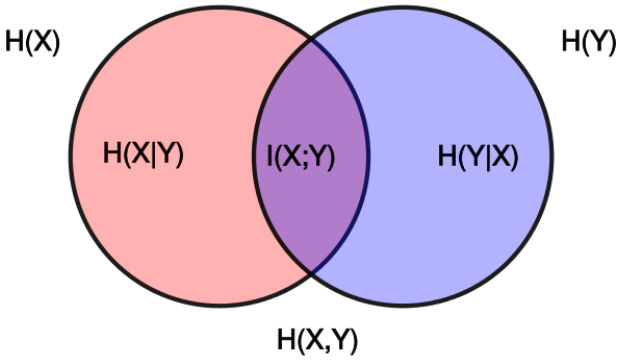

FIGURE 3.1: Venn diagram of Mutual Information

3.2.2 Mutual Information

In probability theory and information theory, the Mutual Information (MI) of two random variables is a measure of the variables' mutual dependence. Not limited to real-valued random variables and the detection of linear relationships like the (Pearson) correlation coefficient, MI is more general and determines how similar the joint distribution $p(X, Y)$ is to the products of marginal distributions $p(X)p(Y)$.

Formally, the mutual information of two discrete random variables X and Y can be defined as:

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log \left(\frac{p(x, y)}{p(x)\, p(y)} \right),$$

where $p(x, y)$ is the joint probability density function of X and Y, and $p(x)$ and $p(y)$ are the marginal probability density functions of X and Y respectively.

Intuitively, mutual information measures the information that X and Y share (as shown in Figure 3.1): it measures how much knowing one of these variables reduces uncertainty about the other. For example, if X and Y are independent, then knowing X does not give any information about Y and vice versa, so their mutual information is zero. At the other extreme, if X is a deterministic function of Y and Y is a deterministic function of X then all information conveyed by X is shared with Y: knowing X determines the value of Y and vice versa. As a result, in this case the mutual information is the same as the uncertainty contained in Y (or, X) alone, namely the entropy of Y (or, X).

Mutual information is a measure of the inherent dependence expressed in the joint distribution of X and Y relative to the joint distribution of X and Y under the assumption

of independence. Mutual information therefore measures dependence in the following sense: $I(X;Y) = 0$ if and only if X and Y are independent random variables. This is easy to see in one direction: if X and Y are independent, then $p(x,y) = p(x)p(y)$, and therefore:

$$\log\left(\frac{p(x,y)}{p(x)\,p(y)}\right) = \log 1 = 0.$$

Moreover, mutual information is non-negative (i.e. $I(X;Y) \geq 0$) and symmetric (i.e. $I(X;Y) = I(Y;X)$).

3.2.3 Transfer Entropy

Although Mutual Information provides a very useful measure of information sharing between two time series, it is a symmetric metric and does not provide any directional information. Yet, it is useful to be able to determine not only the amount of shared information between the two vectors, but also the direction of the dependence, i.e., which of the two market players "drives" the other. As an example, if there are two stocks, namely A and B, it is important to know whether A is correlated to B, but it is also (even more) important to know that, for instance, A affects B, which may mean that if A's price rises, then B's price will also rise, but the opposite might not hold (i.e., "when B's price rises, A's price will rise too").

Transfer Entropy is an asymmetric information transfer measure that can provide the required directionality information. Formally, the Transfer Entropy between two discrete random variables X and Y (from Y to X) can be defined as:

$$TE_{Y\to X} = \sum_{x_{n+1}\in X}\sum_{x_n\in X}\sum_{y_n\in Y} p(x_{n+1},x_n,y_n)\log\left(\frac{p(x_{n+1}|x_n,y_n)}{p(x_{n+1}|x_n)}\right) =$$

$$= \sum_{x_{n+1}\in X}\sum_{x_n\in X}\sum_{y_n\in Y} p(x_{n+1},x_n,y_n)\log\left(\frac{p(x_{n+1},x_n,y_n)p(x_n)}{p(x_n,y_n)\,p(x_{n+1},x_n)}\right),$$

where x_n and y_n correspond to "current" values of X and Y (i.e., at timepoint n), x_{n+1} refers to the "future" value of X (i.e., its value at timepoint $n+1$), $p(x_{n+1},x_n)$, $p(x_n,y_n)$, and $p(x_{n+1},x_n,y_n)$ are joint probability density functions, and $p(x_{n+1}|x_n,y_n)$, $p(x_{n+1}|x_n)$ are conditional probability density functions.

3.3 Data Streams

In computer science, a data stream is a sequence of data elements made available over time, being processed one at a time rather than in large batches. Streams are processed differently from batch data, in the sense that normal transformations cannot operate on streams as a whole, as they potentially have unlimited data. Transformations that operate on a stream, producing another stream, are known as filters, and can be connected in pipelines, where they can operate on one item of a stream at a time, or may base an item of output on multiple items of input, such as a moving average.

An important characteristic of a stream is that it is potentially unbounded, and continuously produces new data points. As a result, stream processing algorithms, need to take into account that storing the whole stream and performing calculations on top of it is infeasible. On the contrary, they need to process the input data in a "single pass" (i.e. each input tuple is processed once and is not revisited). In cases that the algorithm chooses to batch some data, it may re-visit specific tuples, but since the stream is potentially unbounded, this can be done only for a subset of data points at a time, typically referred to as a "window" [7].

3.4 Distributed Systems

With the increasing complexity of modern problems, even the most powerful supercomputer is often not powerful enough to solve them. As a result, distributed systems have become all the more useful. A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal.

Yet, in order to be solved by a distributed system, the problem at hand, first needs to be divided into smaller tasks, so that each component of the distributed system can be responsible for performing one or more tasks. Furthermore, apart from dividing the problem into tasks, there is also the need for all the individual results to be combined so that they can form the solution of the problem.

An easy example of a problem being deployed on a distributed system is the following. Imagine we need to add 100 numbers. Instead of doing it on one computer, going through the numbers one by one, and calculating the final sum, a distributed system could be employed in order to speed up the computation. The first computer could add the first and second numbers, the second computer could add the third and fourth numbers, and so on. Assuming the distributed system has 50 computers, it can execute 50 additions

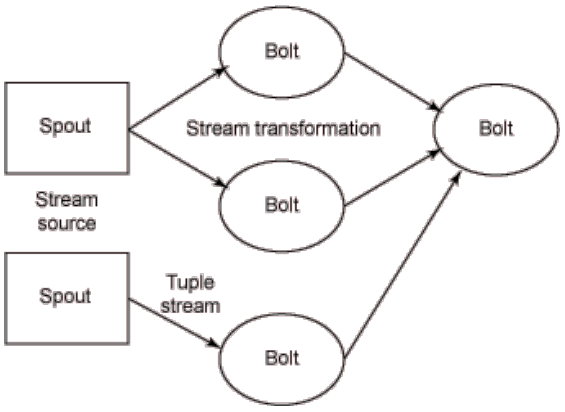

FIGURE 3.2: Conceptual architecture of a trivial Storm topology

in parallel (i.e. taking the same time as one addition would take on one computer). Yet, on the next step, there are 50 intermediate results (the sums of all those pairs), which need to be "fed" to the system again. Hence, the process can be repeated until there is only one final result, i.e. the sum of all the initial numbers. For this particular (naive) example, solving it sequentially would "cost" 100 time units (since we need to visit all the elements one by one), while solving it over a distributed system would "cost" a total time of $log_2 100 = 6.64$ time units (since at each step the problem is "halved" and each step "costs" one time unit).

3.4.1 Apache Storm

The distributed system chosen in order to tackle the problems addressed in this thesis, was Apache Storm [8]. Storm implements a data flow model in which data flows continuously through a network of transformation entities (see Figure 3.2). The abstraction for a data flow is called a stream, which is an unbounded sequence of tuples. The tuple is like a structure that can represent standard data types (such as ints, floats, and byte arrays) or user-defined types with some additional serialization code. Each stream is defined by a unique ID that can be used to build topologies of data sources and sinks. Streams originate from spouts, which flow data from external sources into the Storm topology.

The sinks (or entities that provide transformations) are called bolts. Bolts implement a single transformation on a stream and all processing within a Storm topology. Bolts can implement traditional things like MapReduce functionality or more complex actions (single-step functions) like filtering, aggregations, or communication with external entities such as a database. A typical Storm topology implements multiple transformations and therefore requires multiple bolts with independent tuple streams. Both spouts and bolts are implemented as one or more tasks within a system.

Bolts can stream data to multiple bolts as well as accept data from multiple sources. Storm has the concept of stream groupings, which implement shuffling (random but equal distribution of tuples to bolts) or field grouping (stream partitioning based upon the fields of the stream). Other stream groupings exist, including the ability for the producer to route tuples using its own internal logic.

One of the most interesting features in the Storm architecture is the concept of guaranteed message processing. Storm can guarantee that every tuple a spout emits will be processed; if it isn't processed within some timeout, Storm replays the tuple from the spout. This functionality requires some clever tricks for tracking the tuple through the topology and is one of Storm's core features.

In addition to supporting reliable messaging, Storm uses Netty to maximize messaging performance (removing intermediate queueing and implementing direct passing of messages between tasks).

Chapter 4

Scaling-out Streaming Time Series Analytics

This chapter discussed in detail how different time series analytics components were transformed and optimized in order to comply to our scenario of interest, i.e., streaming, real-time, distributed correlation computation over a cluster of machines. Section 4.1 presents a high-level view of the sliding window approach that was employed in all of our different analytics, while Sections 4.2, 4.3, and 4.4 provide more details for individual correlation measures.

4.1 Sliding Window Model

In order to provide answers in an online fashion, many of our time series analytics computations adopt the sliding window model, where calculations are performed for a maximum pre-defined timespan (e.g., one hour). This section briefly describes the mechanics of the sliding window model we used in different cases, while specific details about adapting each measure's sliding window model are presented in the respective sub-sections.

Briefly, the sliding window represents the duration of interest (e.g., the last hour), defined by the user. Following [9], each sliding window is equally sub-divided into K shorter windows called basic windows, in order to facilitate the efficient elimination of old data and the incorporation of new data. The system maintains stream digests for each basic window, as well as for the entire sliding window. When the new basic window is completed (e.g., every minute), its data is incorporated into the sliding window, and the data of the oldest basic window expires and is removed from the sliding window.

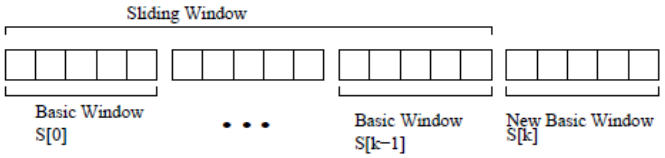

FIGURE 4.1: Sliding Window model

This way, the sliding window provides flexibility and efficiency, as it is maintained in increments of basic windows instead of being updated for every new value. Figure 4.1 presents the visual representation of the sliding window. $S[0..K-1]$ represent the digests that make up the sliding window, stored in the K basic windows, while $S[K]$ represents the new basic window being filled with streaming data.

For example, the sum over the sliding window is updated as follows:

$$\sum_{new}(s) = \sum_{old}(s) + \sum S[k] - \sum S[0]$$

4.2 Pearson Correlation

The Pearson correlation coefficient, as described in section 3.2.1, is an important measure for tracking the (linear) dependency between two time series. This section discusses gradual improvements and optimizations in order to efficiently compute this measure under distributed, streaming, real time computation requirements.

The development in Sections 4.2.2 through 4.2.6 follows along the lines of the StatStream time series indexing scheme [9]. Section 4.2.7 then discusses our distributed Storm implementations, presenting architectural trade-offs depending on the characteristics of the underlying hardware and the user's priorities.

4.2.1 The Naive Approach

Computing the Pearson correlation coefficient between two time series X and Y is fairly straightforward and follows its definition, as shown in Algorithm 1. Lines 1-6 compute the mean value for each of the two time series by adding all their values and dividing each sum with the number of samples. Lines 7-13 calculate their standard deviations, following $\sigma_x = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$, where $\bar{x}$ is the pre-computed mean value. Finally, lines 14-19 compute the Pearson correlation coefficient, following $corr(X,Y) = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sigma_x\sigma_y}$,

Algorithm 1 Pearson correlation algorithm

1: **procedure** CALCULATE_MEANS(X, Y, n)
2: $\bar{x} \leftarrow 0$, $\bar{y} \leftarrow 0$
3: **for** $i \leftarrow 1$ to n **do**
4: $\bar{x} \leftarrow \bar{x} + x_i$, $\bar{y} \leftarrow \bar{y} + y_i$
5: $\bar{x} \leftarrow \frac{\bar{x}}{n}$, $\bar{y} \leftarrow \frac{\bar{y}}{n}$
6: **return** $\bar{x}, \bar{y}$
7: **procedure** CALCULATE_MEANS_AND_DEVIATIONS(X, Y, n)
8: $\bar{x}, \bar{y} \leftarrow$ CALCULATE_MEANS(X, Y, n)
9: $\sigma_x \leftarrow 0$, $\sigma_y \leftarrow 0$
10: **for** $i \leftarrow 1$ to n **do**
11: $\sigma_x \leftarrow \sigma_x + (x_i - \bar{x})^2$, $\sigma_y \leftarrow \sigma_y + (y_i - \bar{y})^2$
12: $\sigma_x \leftarrow \sqrt{\sigma_x}$, $\sigma_y \leftarrow \sqrt{\sigma_y}$
13: **return** $\bar{x}, \sigma_x, \bar{y}, \sigma_y$
14: **procedure** CALCULATE_PEARSON_CORRELATION(X, Y, n)
15: $corr = 0$
16: $\bar{x}, \sigma_x, \bar{y}, \sigma_y \leftarrow$ CALCULATE_MEANS_AND_DEVIATIONS(X, Y, n)
17: **for** $i \leftarrow 1$ to n **do**
18: $corr \leftarrow corr + \frac{(x_i-\bar{x})(y_i-\bar{y})}{\sigma_x\sigma_y}$
19: **return** $corr$

Although simple and intuitive, this approach is not scalable, and is problematic in both directions:

- **vertical scaling** (i.e., size of the time series) : when the time series are very large, performing calculations over all their data points can be quite "expensive" computationally. Furthermore, the system needs to dedicate a lot of memory in order to store them.
- **horizontal scaling** (i.e., number of different time series) : when there is a need to track several thousands of time series in real time, monitoring all the possible pairs among them scales quadratically, since the number of possible pairs among N time series is equal to $N(N-1)/2$, i.e., $O(N^2)$.

4.2.2 Correlation from Euclidean Distance

One can show that the correlation coefficient of X and Y can be expressed in terms of the Euclidean Distance between $\hat{X}$ and $\hat{Y}$, where $\hat{X}$, $\hat{Y}$ represent the normalized time series derived from X and Y respectively. More specifically, for each element of X, represented as x_i, the respective normalized element can be calculated as

$$\hat{x_i} = \frac{x_i - \bar{x}}{\sigma_x},$$

where $\bar{x}$ is the mean value of X and σ_x is its standard deviation, defined as $\sigma_x = \sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}$.

Assuming normalized streams, the correlation between X and Y, can be expressed as

$$corr(X,Y) = 1 - \frac{1}{2}d^2(\hat{X},\hat{Y}), \tag{4.1}$$

where $d(\hat{X},\hat{Y})$ is the Euclidean Distance between $\hat{X}$ and $\hat{Y}$.

The proof for this derivation can be found in [9].

The reason behind the importance of this transformation is that it allows for efficient streaming estimation, as we demonstrate in following sections.

4.2.3 Discrete Fourier Transform

Since the correlation between two time series can be expressed in terms of Euclidean Distance, and in an effort to alleviate the first constraint of the naive approach (i.e.,

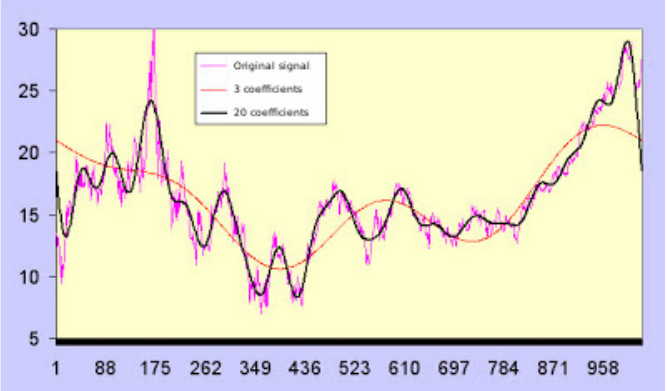

FIGURE 4.2: Approximating a signal with DFT

vertical scaling), we can employ specific properties of the Discrete Fourier Transformation (DFT) in order to tackle the problem more efficiently. This approach has been extensively used in literature for efficient similarity search [10].

More specifically, there are two important properties of DFT coefficients that we can use, namely:

- they preserve the Euclidean Distance (i.e., $d(X,Y) = d(DFT(X), DFT(Y))$) and,
- the majority of the signal's energy is concentrated in the first few coefficients.

The DFT coefficients are calculated as:

$$X_F = \frac{1}{\sqrt{n}} \sum_{i=0}^{n-1} x_i e^{\frac{-j2\pi Fi}{n}} \qquad F = 0, 1, ..., n-1$$

where $j = \sqrt{-1}$.

Approximating the time series using DFT coefficients

One can use $m = \frac{n}{2}$ (due to the symmetry property) DFT coefficients to fully represent the original time series in the frequency domain, without loss of information. Yet, the majority of a signal's energy is typically concentrated in the first few DFT coefficients, hence, we can use a small set of coefficients in order to accurately approximate the original time series and represent a time series of n data points by a set of m coefficients, where $m << \frac{n}{2}$. Typically, m lies in the range of 16...128. Figure 4.2 illustrates an example, where a signal of $n = 1000$ data points is approximated using $m = 3$ and $m = 20$ DFT coefficients.

The accuracy depends on the nature of the original time series and can vary depending on the frequencies involved in it. The less DFT coefficients used, the more "smooth" the

approximation will be, hence, a time series that is originally fairly "smooth", will only need a few coefficients in order to be accurately approximated, while a time series with frequent sudden changes (e.g., a signal consisting purely of random values) will need many more coefficients.

By approximating the original time series using DFT coefficients (which can be constructed in a single pass), the memory requirements for storing the time series are $O(m)$ instead of $O(n)$, and the correlation can be estimated over the approximation instead of the original time series ($corr(X,Y) = 1 - \frac{1}{2}d^2(\hat{X},\hat{Y}) = 1 - \frac{1}{2}d^2(DFT(\hat{X}), DFT(\hat{Y}))$). Still, the computational complexity of calculating the correlation for a specific pair of streams, still remains $O(n)$ since we have to perform a full pass over the original data in order to construct the approximation. The major contribution (in terms of computational complexity) of approximating the time series using DFT coefficients will be made clear in the next section (4.2.4).

4.2.4 Grid Structure

Since our goal is to detect highly correlated pairs of data streams, and their correlation is directly dependent to the Euclidean Distance (Eq. 4.1), then streams that have a large distance can be automatically ruled out as potential candidates.

More formally, our goal is to track pairs of streams with a correlation exceeding a user-defined threshold $t \in [0,1]$. As shown in [9], in order for the correlation to be greater than t, then $d_n(X,Y)$ needs to be lower than ϵ, with $t = 1 - \epsilon^2$, where $d_n(X,Y)$ denotes the Euclidean distance between X and Y, only taking into account the first n DFT coefficients.

By using the DFT on normalized sequences, the original sequences are also mapped into a bounded feature space. It can be shown ([9]) that each coefficient ranges from $-\frac{\sqrt{2}}{2}$ to $\frac{\sqrt{2}}{2}$, therefore the DFT feature space is a cube of diameter $\sqrt{2}$. Following [11], a Grid Structure can be used in order to report near neighbors efficiently. Using the first few $\hat{m}$ coefficients ($\hat{m} <= m <= \frac{n}{2}$), we can construct a $\hat{m}$-dimensional grid, consisting of equally-sized cells, with a cell diameter of ϵ and a total diameter of $\sqrt{2}$. There are $(2\lceil\frac{\sqrt{2}}{2\epsilon}\rceil)^{\hat{m}}$ cells in total. Typically $\hat{m} = 3$ or $\hat{m} = 4$ is sufficient.

Each stream is mapped to a specific cell inside the grid. Suppose stream X is hashed to cell $(c1, c2, ..., c_{\hat{m}})$. To detect the streams whose correlation with X is above t, only streams mapped to cells adjacent to cell $(c1, c2, ..., c_{\hat{m}})$ are possible candidates. Those streams constitute a super-set of the true set of highly-correlated streams. Since the cell diameter is ϵ and we use $\hat{m}$ coefficients for indexing, it is guaranteed that streams

mapped to non-adjacent cells, have $d_{\hat{m}}(X,Y) > \epsilon$, hence, their respective correlation is guaranteed to be lower than t. Due to this fact, there are no false negatives in the system.

Yet, this indexing can introduce false positives, due to the fact that $d(X,Y) \geq d_{\hat{m}}(X,Y)$, i.e., the distance over all the points of the time series can be greater than the distance derived by the first $\hat{m}$ coefficients, so their correlation can be lower than the threshold t. Hence, after determining candidate pairs from adjacent cells, their correlation needs to be calculated based on the stream signatures (m coefficients, instead of the $\hat{m}$ used for indexing in the Grid Structure), in order to eliminate the false positives.

4.2.5 Querying

After hashing all N streams to cells, the number of stream pairs to be examined (originally $N\frac{(N-1)}{2}$) is greatly reduced since we only need to check candidate pairs in adjacent cells. By maintaining the grid structure, we can (on-demand) query it for highly-correlated pairs of streams. The query can be of two forms:

- a) Given stream X, find all streams which are correlated with X with a correlation higher than t, and
- b) Among all streams, detect all pairs with a correlation higher than t.

To answer to query a), the system employs the logic behind algorithm 2, namely, it first creates a list of "*candidate*" streams, which comprises of all the streams hashed in the same cell as X and all the streams hashed in neighboring cells, and then examines X's correlation with each one of those streams to determine whether it is higher than the threshold.

Algorithm 2 Detect all streams that are highly correlated with X

```
1: procedure CORRELATED_WITH_X(X, t)
2:     same ← {streams hashed in the same cell as X}
3:     neighbors ← {streams hashed in neighboring cells}
4:     candidates ← {same} ∪ {neighbors}
5:     result ← {}
6:     for each stream Y in {candidates} do
7:         if (correlation(X, Y) > t) then
8:             result ← {result} ∪ Y
9:     return result
```

To answer to query b), the system follows algorithm 3, instead of simply repeating algorithm 2 for all streams, in order to avoid performing duplicate work. More specifically,

we employ a "filter" (declared in line 2 ,checked in line 12 and updated in line 18), in order to avoid re-checking cells that have already been processed. This can happen due to symmetry since the system needs to examine each cell, and for each cell's computation, it needs to compare it against neighboring cells. The algorithm first loops through all the cells of the Grid Structure (line 4), and, for each cell, it performs two distinct tasks; 1) it computes pairs of streams mapped in the same cell being examined (lines 5-9) and 2) it loops through adjacent cells (line 11), and, if the pair of cells has not already been processed (line 12), it computes inter-cell pairs of streams (lines 13-17).

Algorithm 3 Detect all pairs of streams with a correlation the exceeds t

```
1:  procedure CORRELATED(t)
2:      checkedCells ← {}
3:      result ← {}
4:      for each cell c in cells do
5:          same ← {streams hashed in c}
6:          for each stream X in {same} do
7:              for each stream Y in {same} do
8:                  if (correlation(X, Y) > t) then
9:                      result ← {result}∪ < X, Y >
10:         neighborCells ← {cells adjacent to c}
11:         for each cell v in neighborCells do
12:             if < c, v > is not contained in checkedCells then
13:                 neighbors ← {streams hashed in v}
14:                 for each stream X in {same} do
15:                     for each stream Y in {neighbors} do
16:                         if (correlation(X, Y) > t) then
17:                             result ← {result}∪ < X, Y >
18:                 checkedCells ← {checkedCells}∪ < c, v >
19:     return result
```

The combined use of DFT approximation and the grid structure greatly reduces the overall computational complexity of both queries, as well as the total memory required, since it does not need to store the original time series in order to be able to calculate the correlation among all pairs.

4.2.6 Sliding Window Model

In order for the system to be able to monitor correlations among data streams in a streaming manner, over a fixed window of time (e.g., during the past hour), it employs the sliding window model described in section 4.1, so that it can efficiently eliminate old data and incorporate new time series records. Specifically, basic windows store digests in order to be able to keep an updated estimate of the DFT coefficients over the entire sliding window, with the updates happening at the end of each basic window.

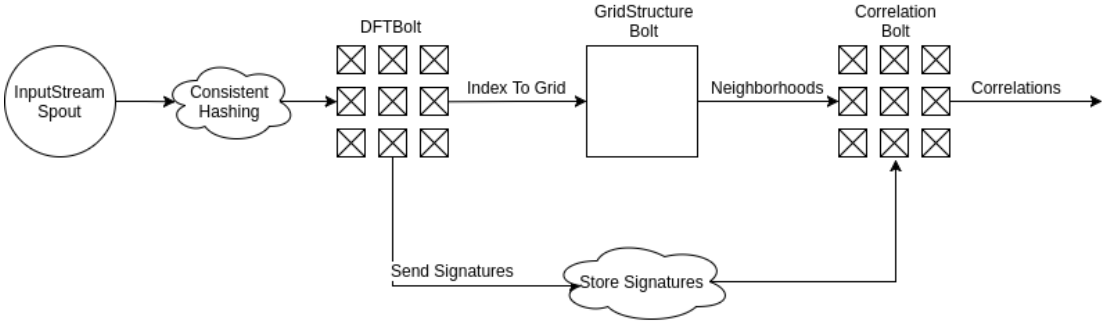

FIGURE 4.3: Topology of the distributed implementation in Storm

Let X_F^{old} be the F-th DFT coefficient of the series in sliding window starting at time $t_s = 0$ and ending at time $t_f = n - 1$ and X_F^{new} be that coefficient of the series with $t_s = b$ and $t_f = b + n - 1$, then,

$$X_F^{new} = e^{\frac{j2\pi Fb}{n}} X_F^{old} + \frac{1}{\sqrt{n}}(\sum_{i=0}^{b-1} e^{\frac{j2\pi F(b-i)}{n}} x_{n+i} - \sum_{i=0}^{b-1} e^{\frac{j2\pi F(b-i)}{n}} x_i), \quad F = 1, ..., m$$

This way, at the end of every basic window, the (first m) DFT coefficients for each stream are updated to incorporate the new values of the stream. After each stream's coefficients are updated, the stream is hashed to its corresponding cell in the grid structure based on its first $\hat{m}$ coefficients. Lastly, after all streams have been hashed to their cells, the grid is queried and all pairs for which the correlation is higher than the threshold t are reported to the output.

4.2.7 Distributed Implementation

This Section discusses the functionality of the Storm components (spouts/bolts), that were created for the distributed computation of the Pearson Correlation using the optimizations described in previous sections. The Storm topology of our implementation is shown in Figure 4.3.

InputStreamSpout

This **spout** implements the source of data for the whole system. It needs to fetch new values from an external source (e.g., a live API stream, a database, a file, etc.) and emit *tuples* to the rest of the topology. The tuple emitted consists of two fields: the *ID* of the input time series (e.g., stock name) and the *value* of the time series at that certain timepoint (e.g., stock price). It is also responsible for keeping track of the number of tuples per time series emitted and also emit a *resetWindowStream* signal when a new basic window is filled with values.

DFTBolt

This **bolt**^{0} receives tuples from *InputStreamSpout* and computes single stream statistics (such as moving average, standard deviation, DFT coefficients, etc.) using the sliding window approach described in section 4.2.6. It is responsible for keeping digests for each time series and incrementally computes the DFT coefficient digests of each time series.

The DFTBolt also receives the *resetWindowstream* from *InputStreamSpout* which indicates that the basic window is full and the digests need to be updated. At this stage, it performs all necessary actions in order to incorporate the new digests and discard the expired ones. These actions include updating the DFT coefficients for each time series based on its basic window digests, updating the standard deviation of each time series (referring to the whole sliding window), computing the **normalized DFT coefficients** using the updated standard deviation, and locating the grid cell for each stream based on its first $\hat{m}$ coefficients.

Finally, it emits the *gridHashKey* along with the time series *ID* to the *GridStructureBolt*, and the time series signature (*normalizedDFTCoefficients*), along with its *ID*, to the *CorrelationBolt* for later use, in order to compute pair-wise correlations using the DFT coefficients.

Each task of this component also emits a *resetWindowstream* signal towards the *GridStructureBolt* for synchronization purposes.

GridStructureBolt

This **bolt** is responsible for maintaining the Grid Structure. The Grid Structure holds all the stream *IDs*, each hashed to a specific cell based on its first $2\hat{m}$ dimensions (i.e., the first $\hat{m}$ DFT coefficients[1]). Each cell may contain multiple stream *IDs*. The Grid Structure receives tuples from the *DFTBolt* and moves each stream *ID* to the appropriate cell. When the basic window is full (*resetWindowStream* has been received from all the components of the *DFTBolt*), the Grid Structure scans all the cells and creates, based on neighboring cells, possibly correlated pairs of streams that need to be examined. Finally, it emits these pairs to the *CorrelationBolt* in order to calculate the required correlations.

The Grid Structure is implemented in a centralized fashion (i.e., in the memory of a specific node) in favor of latency and simplicity. Since the number of streams to be monitored (N) lies in the range of thousands or at most millions, it is not of vital importance to distribute the Grid Structure among multiple cluster nodes. In the case that this requirement arises, a distributed implementation of the Grid Structure has

[1]Each DFT coefficient introduces two dimensions: one for its real part and one for its imaginary part.

also been created. It uniformly distributes the cells to multiple cluster nodes and uses message-passing in order to perform the necessary neighborhood calculations. The space complexity of the Grid Structure is $O(N)$ as it needs to maintain all the stream IDs, each of them stored in the cell it was mapped to. It has been implemented as a HashMap, instead of a matrix, with the *key* being the cell id (e.g., $(c1, c2, ..., c_{\hat{m}})$) and the value being a list of all the streams that were mapped in this cell. There are two extreme cases; a) all the streams get mapped to the same cell, in which case there is only one entry in the HashMap, containing a list of N IDs, and b) all the streams get mapped to different cells, in which case there are N entries in the HashMap, each containing only one element. In both cases, the space complexity remains $O(N)$.

CorrelationBolt

This **bolt**[0] is responsible for receiving a set of time series IDs and computing their pair-wise correlations. The received set contains false positives that need to be filtered out, yet, as explained in Section 4.2.4, the system does not introduce any false negatives.

The correlation computation is performed by using the signatures, i.e., the m normalized coefficients for each time series (**timeseriesCoefficients**) previously received from *DFTBolt* and calculating pair-wise Euclidean distance of the coefficients. Since the coefficients are normalized, they represent the normalized time series, so that the correlation can be computed using the Euclidean distance; and, since DFT preserves Euclidean distance, the Pearson correlation of the original streams can be computed through the Euclidean distance of their normalized DFT coefficients.

4.2.7.1 Storing the Signatures

An important aspect of the system that has not yet been addressed is where and how the DFT coefficients are stored. When a task from the *CorrelationBolt* needs to calculate the correlation between two streams, it needs to access their respective DFT coefficients (i.e., signatures) so that it can calculate the Euclidean distance between them. Yet, since there are many tasks, and each stream may take part in multiple pairs (e.g., pair X, Y is calculated by task 1, while pair X, Z is calculated by task 2), choosing where each stream's coefficients are stored, can be a complicated decision.

One can easily deduce that the minimum total memory required for storing the coefficients is $O(mN)$, since there are N streams in total, and we are storing m coefficients

[0]Multiple instances (i.e., tasks) of this component can be executed in parallel. The parallelism can be controlled by the user through Storm, and it heavily depends on the size of the cluster.

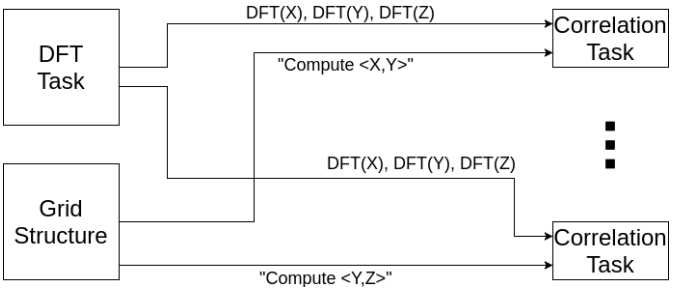

FIGURE 4.4: Replication of signatures to all *CorrelationBolt* tasks.

for each one of them. This section presents some possible design choices and briefly discusses the trade-offs involved.

Replication of Information

A fairly simple approach is to replicate the information across all tasks of the *CorrelationBolt*, and store **all** the signatures for all the streams in each task. Although this increases the memory requirements to $O(kmN)$ (with k being the number of *CorrelationBolt* tasks) and the tuples being emitted from *DFTBolt* tasks to *CorrelationBolt* from $O(N)$ (i.e., emitting one tuple for each stream) to $O(kN)$ (i.e., emitting the same tuple to all k tasks for each stream), it greatly improves the end-to-end latency of the system, as it completely eliminates the need for synchronization among *CorrelationBolt* tasks.

Since N varies in the range of thousands, and m is typically in the range of 16...128, the memory footprint for storing all the signatures is usually manageable by a single modern day machine.

Figure 4.4 shows a visual representation of this approach. *DFTBolt* computes the signatures for each stream and emits all of them to all the *CorrelationBolt* tasks. *GridStructureBolt* "requests" the computation for pairs $< X, Y >$ and $< Y, Z >$ from different tasks. The tasks can immediately begin the computation since each task already has the required signatures.

Shared Memory

Another simple approach is to introduce some form of "Shared Memory" among all *DFTBolt* and *CorrelationBolt* tasks (e.g., a distributed hash table - DHT - like *HazelCast*). This way, instead of emitting tuples from *DFTBolt* to *CorrelationBolt* tasks, and having to figure out where to emit, the *DFTBolt* tasks only need to "put" data in the DHT, while the *CorrelationBolt* tasks can simply "get" the data they need from the DHT, when they need it.

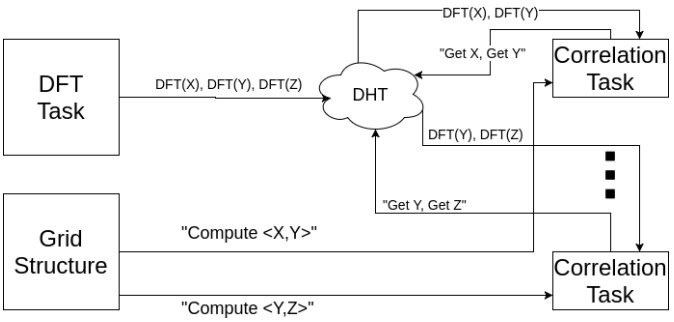

FIGURE 4.5: Storing the signatures in Shared Memory.

This approach keeps the memory complexity at $O(mN)$ since each signature is stored only once, and distributed among all tasks.

Figure 4.5 shows a visual representation of this approach. *DFTBolt* computes the signatures for each stream and stores them in a DHT. *GridStructureBolt* "requests" the computation for pairs $< X, Y >$ and $< Y, Z >$ from different tasks. The tasks need to request the required signatures from the DHT, wait until they get a reply, and then begin the computation of the correlation.

Asynchronous Communication

A more complicated approach focused towards keeping the memory complexity at $O(mN)$, is to manually distribute the signatures to specific tasks and employ communication (i.e., message-passing) among the tasks to allow them to "request" data that they do not own. This is done through the use of consistent hashing based on the ID of each stream. More specifically, *DFTBolt* tasks use a well-defined, deterministic hash function to map a specific stream's signature to a specific *CorrelationBolt* task, which is responsible for storing this stream's signature locally. Using the same hash function, when a *CorrelationBolt* task (e.g., task 1) needs to compute the correlation for a specific pair of streams (e.g., X, Y), if it does not have the signatures stored locally, it can know which other task has the data it needs (e.g., task 1 has X's signature and task 2 has Y's signature). This way, it can request the missing data from the other tasks, store the fact that it has a "pending" request for this pair, and continue performing useful work until the response is received, at which point it can calculate the correlation for that pair and delete the "pending" request.

Figure 4.6 shows a visual representation of this approach. *DFTBolt* computes the signatures for each stream and and emits each signature to a specific *CorrelationBolt* task, based on consistent hashing. *GridStructureBolt* "requests" the computation for pairs $< X, Y >$ and $< X, Z >$ from the same task. This task only contains the signatures for X and Z, not for Y. Hence, when it receives the request for $< X, Y >$, it requests the

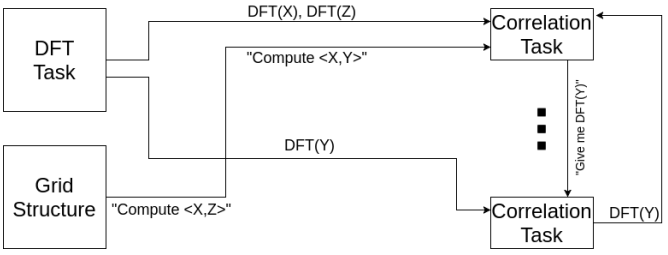

FIGURE 4.6: Storing a subset of the signatures in each task.

signature for Y from the appropriate task (using the same hash function as *DFTBolt*) and stores a "pending" request for $< X, Y >$. Yet, instead of waiting for the response, it continues its execution and processes $< X, Z >$, which it can begin computing immediately since it already has both signatures. Finally, when it receives the response for Y's signature, it computes the correlation for X, Y.

Comparison

The first approach is preferable when the system is executed on top of hardware with sufficient memory, where latency is the most important factor. It is also beneficial when the network links among the cluster nodes are "slow" (e.g., geographically distributed cluster instead of local network), because even though we are emitting k messages instead of 1, those are emitted in parallel, so the time cost is "hidden", while in the case of synchronization, one task would need to wait for the other task to receive its message and reply, before the final result can be calculated.

The second approach is most favorable when the system is memory-bound. The memory footprint is kept to a minimum, yet latency increases since when calling the "get" for a specific stream whose signature is not stored on the same cluster node, the DHT needs to (request and) fetch this data from the corresponding node where it is stored. Still, this approach provides simplicity and flexibility in the sense that the DHT is an external dependency and can be interchanged with a different one in the future. Furthermore, DHTs usually offer the possibility to configure some "replication factor" f, so that the data is replicated f times across the cluster. This is quite useful when the cluster nodes are unreliable and can "die" easily.

The third approach is similar to the second in the terms of memory, yet it also reduces the total latency of the system, since communication is performed in an asynchronous manner and the tasks can continue processing other pairs while waiting for other tasks to respond to their requests, instead of having to wait for the DHT to respond to the (remote) "get" command. Yet, this approach is limited in terms of flexibility, in the sense that it does not offer all the configuration that a DHT can offer, and most importantly the replication of data for resiliency. This approach should be preferred in the same

cases as the second one, except in the case of unreliable cluster nodes, when resiliency is very important.

All the approaches have been implemented and incorporated in our system, with the first approach being chosen as the "default" one, in favor of end-to-end latency. All the experiments presented in Chapter 6, involve the "**Replication of Information**" approach.

4.3 Mutual Information

This section discusses the streaming computation of Mutual Information (3.2.2). Initially, we analyze the way that the probability density functions involved in MI computation are represented. Then, we introduce a naive approach, followed by optimizations related to CPU efficiency, memory efficiency, latency minimization, and, lastly, an approach that tackles the computation in a truly streaming manner. Finally, we provide a detailed overview of our distributed implementation for streaming MI computation.

4.3.1 Representation of Probability Density Functions

The Mutual Information computation involves three probability density functions, namely, two marginal PDFs (one for each stream) and one joint PDF. PDFs are commonly estimated in literature using histograms. More specifically, a PDF P comprises of a set of equally-sized *buckets* or *bins*, spanning along the full range of the possible values of the original stream. The number of buckets B, is chosen depending on the level of accuracy required, with larger B leading to more accurate estimation of the PDF. Throughout our work, we use B to denote the number of buckets used for the estimation of one-dimensional PDFs, as well as the size of each dimension in multi-dimensional PDFs. Hence, the total size of a two-dimensional PDF is B^2.

Each bucket contains the total number of samples that were mapped to it, divided (i.e., normalized) by the total number of samples overall, so that the sum of all the buckets is one. Hence, each bucket represents the relative likelihood of the original stream having a value falling in the range that the bucket covers.

4.3.1.1 Matrix Representation

Histograms are commonly represented by an array (for a single dimension) or a matrix (for multiple dimensions) of counters. Each cell corresponds to a bucket, and contains a counter of the total number of samples that were mapped to this bucket. After updating all the cells' counters by mapping all the stream samples to the appropriate array/matrix cells, to get the estimate of the PDF, each cell is divided by the total number of samples. Figure 4.7 illustrates an example of marginal and joint PDFs being estimated by normalizing the arrays and the matrix of counters.

Although simple and intuitive, this representation is quite inefficient in terms of memory, for large B, especially when the PDFs are skewed (i.e., sparse) and not uniform. Since the focus of this work is the pair-wise computation of the Mutual Information measure

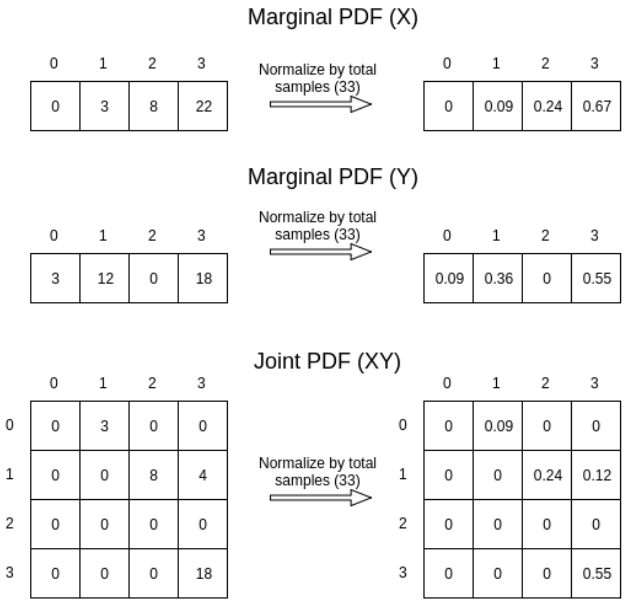

FIGURE 4.7: Example of the matrix PDF representation for $B = 4$, with 33 samples.

under the assumption of thousands of data streams, where, for each pair, there exists the need to store the joint PDF of the two streams, the total memory requirements for the system are quite high. Specifically, for N data streams, there are $N(N-1)/2$ pairs, and the joint PDF of each pair requires B^2 space. With a typical B of 512, each joint PDF requires 1MB of space ($512 * 512 * 4$(sizeof int)), and, as such, if the application is interested in monitoring 1000 data streams, the total memory required, for the joint PDFs alone, is roughly 488GB ($1000 * 999/2 * 1$MB). Section 4.3.1.2, discusses a more powerful and efficient representation.

4.3.1.2 Sparse Representation

In an effort to remove the memory bottleneck, an analysis was performed on typical input data. Both live and historical data from the financial domain were used in order to determine the percentage of actual, useful entries in the PDFs. This kind of data is, by nature, not uniformly distributed. For example, a stock's previous value greatly affects its next value. Hence, as confirmed by the analysis, the PDFs (especially the joint ones), appear to be very sparse. Joint PDFs among stocks where found to be $1 - 2\%$ full, especially in the real-time scenario, which is the focus of this work, and where the window of interest is relatively small (e.g., 1 hour).

This observation leads to the conclusion that there is a need for a sparse representation of the PDFs. The operations that the system needs to perform on top of the joint PDF are two, namely **insert** a new value at indices $[bin_x][bin_y]$ and **lookup** the value of the

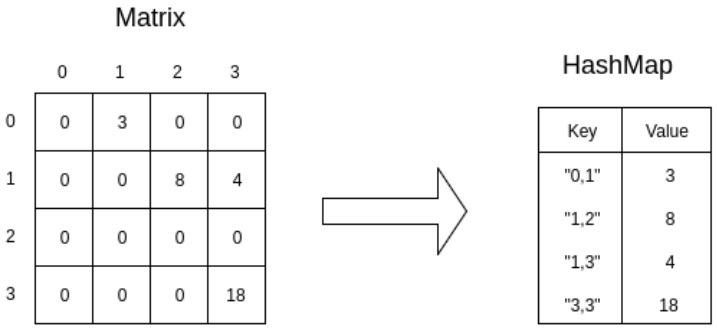

FIGURE 4.8: Example of the sparse PDF representation for $B = 4$, with 33 samples.

cell at indices $[bin_x][bin_y]$. The best data structure that behaves similar to an array (i.e., has a constant time of insertion and lookup for a specific index) is the *HashMap*. A *String* key and an *Integer* value are used to define the *HashMap*. The key corresponds to the concatenation of the two indices separated by a delimiter (e.g., cell [3][8] would produce a key equal to "3,8") and the value corresponds to the value of this specific cell. This way, the *HashMap* can replace the original matrix representation of the PDF. An example transformation is shown in figure 4.8. This simple, yet powerful, optimization leads to a great improvement both in terms of memory (since only the useful values are stored and no space is wasted on "empty" cells), but also in terms of computation performance. This fact becomes clear if one considers the computational complexity of MI, which transforms from $O(B^2)$ to $O(K)$, where $K << B^2$ is the number of non-empty cells of the joint PDF (typically $K = 0.01B^2$). In short, in order to compute MI, all the cells of the $2D$ PDF take part in a sum. Yet, cells with a counter equal to zero, do not modify the value of the sum, hence, omitting them altogether reduces the complexity without affecting the result.

4.3.2 Offline Algorithm

The naive pair-wise implementation of Mutual Information is quite simple and straightforward. It takes as input the two marginal probability density functions as well as the joint probability density function and calculates the Mutual Information between two random variables X and Y as shown in Algorithm 4.

Lines 1-8 perform the PDF estimation based on the values of X and Y. More specifically, for each value, a bucket is computed for each of the time series (lines 3 and 5), following $bin = B\frac{value-min}{max-min}$, where B is the number of buckets/bins, $value$ is the current time series value being examined, and max/min are the respective maximum and minimum values for this time series. Having determined the appropriate buckets, the marginal

(lines 4 and 6), as well as the joint (line 7), probability density functions are updated. After all the updates are performed, the PDFs are normalized using the total number of samples (n) and returned (line 8). Finally, MI is calculated using the updated PDFs, following $I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \left(\frac{p(x,y)}{p(x)\, p(y)} \right)$, in lines 9-14.

Algorithm 4 Mutual Information algorithm

1: **procedure** ESTIMATE_PDFS($values_X, values_Y, n, B$)
2: **for** $i \leftarrow 1$ to n **do**
3: $bin_X = B \frac{values_X[i] - min_X}{max_X - min_X}$
4: $pdf_X[bin_X] += 1$
5: $bin_Y = B \frac{values_Y[i] - min_Y}{max_Y - min_Y}$
6: $pdf_Y[bin_Y] += 1$
7: $pdf_{XY}[bin_X][bin_Y] += 1$
8: **return** $\frac{pdf_X}{n}, \frac{pdf_Y}{n}, \frac{pdf_{XY}}{n}$
9: **procedure** CALCULATE_MI($pdf_X, pdf_Y, pdf_{XY}, B$)
10: $mi = 0$
11: **for** $i \leftarrow 1$ to B **do**
12: **for** $j \leftarrow 1$ to B **do**
13: $mi \leftarrow mi + pdf_{XY}[i,j] \log \frac{pdf_{XY}[i,j]}{pdf_X[i] pdf_Y[j]}$
14: **return** mi

The implementation of Algorithm 4 makes the following assumptions:

- The probability density functions are pre-calculated and do not change. This requires that all the data is known beforehand.
- The calculation of MI is performed in an offline batch fashion.
- The data is assumed to be synchronized (i.e., new values of X and Y arrive at each time tick).

4.3.3 Streaming PDF Estimation

This implementation removes the assumption that the data exists somewhere stored, and provides a streaming approach where the PDF of each stream is updated online as new values are being received. Furthermore, it lifts the synchronized data constraint by making the (typical in the financial domain) assumption that the last value received for each stream is valid until receiving a new one. Thus, for every new value of X, we can update the marginal PDF for X, as well as the joint PDF for XY, by assuming the value of Y to be equal to the last received value of Y, and vice versa. The calculation of MI can now be performed repeatedly at a user-defined interval t, or on demand, while always referring to the latest estimated PDFs. The pseudo-code referring to this implementation is shown in Algorithm 5.

The procedure in lines 1-8 is executed for each new value of X or Y and updates the respective marginal PDF, as well as the joint PDF. The marginal PDF is updated in the same way as in Algorithm 4, while the joint PDF uses the new bucket for the stream that received an update and the last bucket that was computed for the other stream (when it received its last update). Furthermore, the sizes need to be incremented as well (lines 4 and 7), since the algorithm does not know the size of the data beforehand, and the PDFs need to be normalized each time, based on their respective sample sizes.

Algorithm 5 Mutual Information algorithm, Streaming PDF estimation

1: **procedure** UPDATE_PDFS($X, value, B$)
2: $bin_X = B\frac{value - min_X}{max_X - min_X}$
3: $pdf_X[bin_X] += 1$
4: $size_X += 1$
5: $last\_bin\_Y \leftarrow$ the bin where the last value of the other stream got mapped
6: $pdf_{XY}[bin_X][last\_bin\_Y] += 1$
7: $size_{XY} += 1$
8: **return** $\frac{pdf_X}{size_X}, \frac{pdf_{XY}}{size_{XY}}$
9: **procedure** CALCULATE_MI($pdf_X, pdf_Y, pdf_{XY}, B$)
10: **for** $i = 1$ to B **do**
11: **for** $j = 1$ to B **do**
12: $mi = mi + pdf_{XY}[i,j] \log \frac{pdf_{XY}[i,j]}{pdf_X[i] pdf_Y[j]}$
13: **return** mi

4.3.4 Sliding Window Model

Our system needs to monitor the MI measure in an online fashion. Hence, the sliding window model described in Section 4.1 is employed. More specifically, MI needs to keep an updated view of the various different probability density functions. In order to do so, each basic window keeps digests related to the frequency of occurrence of each PDF bin (using the technique detailed in Section 4.3.1.2). Furthermore, digests referring to the entire sliding window are also stored independently. When a new basic window (i.e., $S[K]$) is full, its PDF statistics are incorporated into those of the full sliding window. Lastly, the PDF statistics of the oldest basic window (i.e., $S[0]$) are subtracted from the sliding window and the MI computation which takes into account the updated PDFs, is triggered. This way, the sliding window always maintains PDF estimations that refer to the desired duration (i.e., the length of the sliding window), updated in basic window increments, and the system can compute the updated MI value at a fixed interval (e.g., one minute - defined by the length of the basic window).

4.3.5 Removing the Fixed Interval

Although calculating MI at a fixed interval makes sense in the case of data streams having the same "speed", it does not make much sense when the input data consists of both "fast" and "slow" data streams. In the financial domain, there is a wide variety of data streams, ranging from very "slow" to very "fast". Consider the example of a highly traded stock (e.g., GOOGL) versus a relatively new and not so popular stock. Such characteristics lead to the observation that the system needs to be adaptive and treat each pair of streams differently.

Since the MI computation is based on the histograms representing the PDFs, the MI of two "slow" streams does not change very often, as the underlying PDFs do not change often due to not receiving many samples. In order to better distribute CPU resources, and compute MI for "fast" pairs more often than for "slow" pairs instead of computing it for all pairs at a fixed interval, the condition that defines when the MI will be calculated changes to a function $f(s)$ of the number of samples (s) received since the last time that MI was calculated. For example, $f(s)$ could be defined so that the system calculates the new value of MI for a specific pair after receiving 1000 new samples for this pair. This way, pairs involving "fast" streams are being updated more often, and the (more-frequently) updated value of their MI is available to the user sooner.

4.3.6 Streaming MI Computation

The approach described so far, is not truly "streaming". The Mutual Information index is only calculated either at a fixed interval or after receiving a specific amount of samples, as it makes no sense "paying" a quadratic cost (B^2, i.e., looping through the entire $2D$ PDF) to calculate the Mutual Information for every update in the PDF. Yet, this fact implies that the estimated output value of MI is "delayed". Furthermore, since the system needs to keep statistics for every basic window, and for every pair, it requires a lot of memory (which scales as the number of basic windows increases). In addition, MI values are recomputed from scratch, which is clearly wasteful, as only a small part of MI actually changes with the window movement.

These observations lead to an improved, streaming version of the existing MI estimator, which updates and outputs the new value of MI for **every** input tuple, yet avoiding the quadratic cost per update, and only "paying" a linear (with respect to B) cost per update. Moreover, the memory footprint remains constant as it is no longer dependent on basic windows.

Using the assumption that the approximation technique for the probability density functions is performed through histogram binning, our streaming MI variant tries to isolate the terms from the original calculation that do not get affected by the new value, and, as such, do not need to be re-computed during the update. This way, MI is being monitored in an incremental way (incorporating changes induced by new data, and eliminating influence from old data), instead of being re-constructed from scratch each time.

The main observation resulting to this approach is that $p(i) = C(i)/N_i$, where $p(i)$ is the probability density function for vector I, $C(i)$ is the count of values that end up in a specific cell, and N_i is the total number of samples observed from vector I. Hence, $p(x) = C(x)/N_x$, $p(y) = C(y)/N_y$, and $p(x,y) = C(x,y)/N_{xy}$.

Using this observation and some calculations, the Mutual Information computation can be analyzed as follows:

$$I(X;Y) = \sum_{y\in Y}\sum_{x\in X} p(x,y)\log\left(\frac{p(x,y)}{p(x)\,p(y)}\right) =$$

$$= \sum_{y\in Y}\sum_{x\in X}\frac{C(x,y)}{N_{xy}}\log\left(\frac{\frac{C(x,y)}{N_{xy}}}{\frac{C(x)}{N_x}\,\frac{C(y)}{N_y}}\right) =$$

$$= \underbrace{\frac{1}{N_{xy}}}_{\text{O(1)}}\underbrace{\sum_{y\in Y}\sum_{x\in X} C(x,y)\log\left(\frac{C(x,y)}{C(x)\,C(y)}\right)}_{\text{O(B) (1 row or 1 column)}} - \underbrace{\frac{\log\frac{N_{xy}}{N_x N_y}}{N_{xy}}}_{\text{O(1)}}\underbrace{\sum_{y\in Y}\sum_{x\in X} C(x,y)}_{\text{O(1) (=previous value+1)}}$$

Examining each term of the computation, it is clear that terms $\frac{1}{N_{xy}}$ and $\frac{\log\frac{N_{xy}}{N_x N_y}}{N_{xy}}$ require constant time ($O(1)$), as they only depend on the number of samples of each stream. Furthermore, $\sum_{y\in Y}\sum_{x\in X} C(x,y)$ also requires constant time, since, for every update, it is incremented by 1, because each update only affects one cell of the $2D$ PDF, the counter of which is increased by 1. As for the term $\sum_{y\in Y}\sum_{x\in X} C(x,y)\log\left(\frac{C(x,y)}{C(x)\,C(y)}\right)$, since each update refers to a specific time series (either X, or Y), and we make the assumption that the value of the other time series remains constant, then, only one row or one column ($\sum_{y\in Y}$ for given x or $\sum_{x\in X}$ for given y) needs to be updated, instead of the entire $2D$ matrix.

By keeping statistics for the term $C(x,y)\log\left(\frac{C(x,y)}{C(x)\,C(y)}\right)$, for each cell, in a streaming manner, the system can efficiently eliminate, from the total sum, the "contribution" of each cell being updated, and incorporate its updated "contribution", i.e., the one that refers to the updated values of the PDFs. Hence MI can be updated for every input tuple, without performing wasteful operations, with each update costing $O(B)$ time, and, most importantly, minimizing the end-to-end latency since the updated result is

directly available, without having to wait for the basic window to expire. Furthermore, even if the theoretical cost of every update is $O(B)$, in practice, due to the sparse representation of the PDFs (Section 4.3.1.2), each update is executed in $O(K)$ time, with K consistently much smaller then B.

4.3.7 Distributed Computation

In real-world applications, there is often the need to compute the Mutual Information measure among thousands of data streams, in a pair-wise fashion. Hence, if the applications need to keep track of N data streams, MI needs to be computed for all the possible pairs, i.e., $N(N-1)$, which, in terms of complexity, is $O(N^2)$. Since MI is a bi-directional measure, the exact number of pairs that need to be tracked is equal to $N(N-1)/2$.

Our Storm topology performing this computation is shown in Figure 4.9. **InputStreamSpout** is responsible for fetching updates from the "outside world" into the system. It emits a single String value containing the information it read from the external source. **PreprocessorBolt** receives this String and performs some simple data-cleansing, like extracting specific fields from the String. It emits a tuple containing necessary fields (like stream id and the new value) to the **MutualInfoBolt**. Finally, **MutualInfoBolt** is responsible for performing the actual computation of the Mutual Information measure.

In order to distribute the work to multiple cluster nodes, each node needs to be responsible for computing the MI index for a distinct subset of the total pairs to be computed. The new values being received by the input streams are emitted to all the **MutualInfoBolt** tasks and each task is responsible of incorporating each new value into all the pairs that contain this stream and for which this task is responsible.

More specifically, each task knows the total number of tasks through Storm's configuration object, as well as its own task id. Furthermore, each task keeps an ordered list of the distinct stream ids that have been received so far. When a new id is received, all pairs among this id and all the existing ones are generated in each task, and, using the total tasks and its own task id, each task can independently decide which pairs it is responsible for. This process is shown in pseudo-code in Algorithm 6.

In lines 1-2, each task gets its own task id, as well as the total number of tasks from Storm. *idsSoFar* (line 3) holds an ordered list of all the distinct stream ids that have been received so far. For each update received, if the stream id is new (line 7), each task generates all the pairs among this id and all the already existing ones (lines 14-18). It then loops through all these new pairs (line 9) and, for each one of them, decides whether

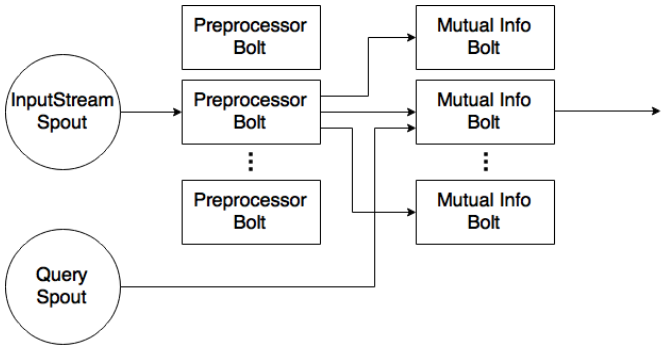

FIGURE 4.9: Storm topology of MI distributed computation

it needs to track it or not (line 10), based on the total distinct pairs so far ($pairCounter$) and its task id ($myTaskId$). Finally, the new stream id is added in $idsSoFar$. This way when there is a future update for this id, each task can update its own pairs involving this id.

Algorithm 6 Tracking pairs independently

```
1:  integer myTaskId = stormConfig.getThisTaskId()
2:  integer totalTasks = stormConfig.getTotalTasks()
3:  fifo idsSoFar
4:  list pairsToTrack
5:  integer pairCounter = 0
6:  procedure TRACKPAIRS(newStreamId)
7:      if newStreamId is not contained in idsSoFar then
8:          fifo newPairs = generateNewPairs(idsSoFar, newStreamId)
9:          for each pair p contained in newPairs do
10:             if pairCounter mod totalTasks == myTaskId then
11:                 pairsToTrack.add(p)
12:             pairCounter++
13:         idsSoFar.add(newStreamId)
14: procedure GENERATENEWPAIRS(idsSoFar, newStreamId)
15:     fifo newPairs
16:     for each id i contained in idsSoFar do
17:         newPairs.add(i, newStreamId)
18:     return newPairs
```

4.3.7.1 Explicit Distribution of Work - Mapper

Although, using the approach above, the workload is distributed equally among all cluster nodes, it requires that each new value is transmitted to all the cluster nodes responsible for MI computation. Depending on the number of nodes, this pre-requisite can introduce a big load caused by a large number of network transactions. In the effort to alleviate this issue, an intermediate component is used between the **Preprocessor-Bolt** and the **MutualInfoBolt**, namely **MapperBolt**. The updated Storm topology

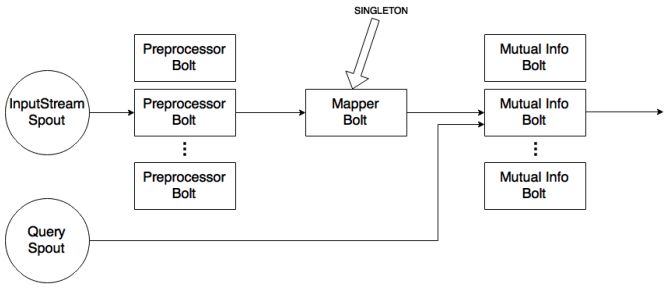

FIGURE 4.10: Storm topology with explicit workload distribution

including this component is shown in Figure 4.10. **MapperBolt** is a singleton component (i.e., only executes one task, in one cluster node) and acts as a "router". It is responsible for two tasks:

- **During startup** : Receiving the full list of possible stream ids and computing a mapping where each pair is mapped to a specific task of the **MutualInfoBolt**.
- **During execution** : Routing each incoming tuple to the necessary **MutualInfoBolt** tasks based on this mapping.

The use of this component reduces network traffic without sacrificing workload distribution, since it only emits incoming updates to the tasks that actually need them. Yet, it introduces two negative aspects, namely:

- The system has a single point of failure. Since this component is a singleton, if the node executing it crashes, the whole system will crash, while in the topology that does not contain this component, if a node crashes, the system will only "lose" the pairs that were assigned to that specific node. This is a trade-off to be considered depending on a variety of factors related to the execution environment (i.e., number of cluster nodes, network speed, hardware stability, etc.).
- It requires knowledge of the full list of stream ids on startup in order to create the mapping configuration.

4.3.7.2 Explicit Distribution of Work - Dynamic Mapper

Since many real-world applications do not possess explicit knowledge about the input data beforehand, there is a need to alleviate the second limitation introduced by the **MapperBolt**. Hence, a dynamic version of this component was created. The Storm topology remains the same (i.e., Figure 4.10), yet the implementation of **MapperBolt** changes internally. This version does not need the full list of stream ids during startup,

yet it computes the pairs in an online manner, similar to the way the pairs are assigned in Algorithm 6. More specifically, it executes the same process, yet instead of storing the pair in $pairsToTrack$, it builds a mapping filter in order to know where to "route" each incoming tuple. This way, the second limitation introduced by the **MapperBolt** is lifted, network traffic is still reduced, worload is distributed optimally, and the "pair assignment" calculations are only performed once (in **MapperBolt**) instead of multiple times (in all tasks of **MutualInfoBolt**).

4.3.7.3 Shared-Memory Model

As explained in Section 4.3, the computation of the Mutual Information index requires the estimation (and storage) of three Probability Density Functions (PDFs), namely P_x, P_y, and P_{xy} for each pair of streams. Although P_{xy} is unique for each pair, P_x and P_y are needed in multiple pairs. More specifically provided that the total number of streams is N, each one of P_x, P_y is needed $N-1$ times, as it takes part in the computation of $N-1$ pairs (i.e., all the pairs of this stream with all other streams). Hence, in an effort to avoid replicating this information multiple times and among multiple cluster nodes, a Shared-Memory model approach was introduced. Instead of storing P_x and P_y independently for each pair of streams, a Distributed Hash Table (DHT) was used in order to store the marginal distribution of each stream. This way, the data is distributed among cluster nodes, and only saved once. When a cluster node needs this information, it does a lookup in the DHT, which retrieves the data from the node where it is stored and forwards it to the node that needs it.

The usage of the Shared-Memory model introduces yet another trade-off, namely between computation latency and memory consumption. If this approach is used, then the computation latency increases, since the data is not stored locally in each node and need to be retrieved through a network call (which takes much more time than a local memory lookup). Yet, the total memory consumption is greatly reduced, since there is no data replication and the necessary information is only stored once.

4.4 Transfer Entropy

Transfer Entropy (3.2.3) is an important measure that, apart from dependence, also shows causality. Unfortunately, Transfer Entropy is quite demanding with respect to computation resources since it requires approximating 3-dimensional PDFs, which means that its complexity is cubic ($O(B^3)$), both in temrs of memory and time. As in Mutual Information, there are several challenges to be addressed, yet the main challenge lies in estimating the required probability density functions in memory, since Transfer Entropy is calculated over a 3-dimensional PDF instead of 2-dimensional one.

This section presents our optimizations for streaming Transfer Entropy computation.

4.4.1 Representation of Probability Density Functions

In the case of Transfer Entropy, the amount of memory required to store the required probability density functions for each pair, grows proportionally to the cube of the number of bins (B) used (i.e., O(B^3), since the largest PDF is 3-dimensional ($p(x_{n+1}, x_n, y_n)$). Hence, if we set B equal to 512, the 3-dimensional PDF **for each pair** requires 512MB (512∗512∗512∗4(sizeof int)) in order to be stored. Hence, if the application needs to track 1000 data streams, then a staggering 488TB of memory is needed (1000*999*512MB[2]).

By applying the same technique described in Section 4.3.1.2, we can vastly reduce the amount of actual memory needed, which is further enhanced by the fact that the 3-dimensional PDF is even sparser than the 2-dimensional one. The percentage of non-empty cells in the PDF is of course dependent on the input data, yet, frequency analysis performed on typical input data from the financial domain, shows that it is usually close to 0.01% populated.

4.4.2 Offline Algorithm

As with Mutual Information, the naive pair-wise implementation of Transfer Entropy is quite straightforward. It takes as input one marginal probability density function ($p(x_n)$), as well as three joint probability density functions ($p(x_{n+1}, x_n, y_n)$, $p(x_n, y_n)$, $p(x_{n+1}, x_n)$) and calculates the Transfer Entropy between two random variables X and Y (and specifically from Y to X)[3] as shown in Algorithm 7.

[2]Note that TE is directional, so the number of pairs is $N(N-1)$ instead of $N(N-1)/2$.

[3]The algorithm refers to the computation of $TE_{Y\to X}$. In the case of $TE_{X\to Y}$, the same procedure is executed, with the streams reversed.

The algorithm loops through all the data (line 2), computes the buckets for the current values of X (line 3) and Y (line 6) and the "future" value of X (X_{n+1} - line 5), and uses them to update the marginal PDF of X (line 4), the joint XY PDF (line 7), the joint PDF of X with its "future" occurrences (line 8), as well as the joint $3D$ PDF (line 9). The procedure that estimates the PDFs (line 1) returns the normalized PDFs (line 10), based on the size of the data (n). Finally, TE is computed in lines 11-17, by looping through the $3D$ PDF and updating the sum (line 16), based on its definition (provided in Section 3.2.3).

Algorithm 7 Transfer Entropy algorithm

1: **procedure** ESTIMATE_PDFS($values_X, values_Y, n, B$)
2: **for** $i \leftarrow 1$ to $n-1$ **do**
3: $bin_X = B\frac{values_X[i]-min_X}{max_X-min_X}$
4: $pdf_X[bin_X] += 1$
5: $bin_{Xn+1} = B\frac{values_X[i+1]-min_X}{max_X-min_X}$
6: $bin_Y = B\frac{values_Y[i]-min_Y}{max_Y-min_Y}$
7: $pdf_{XY}[bin_X][bin_Y] += 1$
8: $pdf_{X_{n+1}X}[bin_{Xn+1}][bin_X] += 1$
9: $pdf_{X_{n+1}XY}[bin_{Xn+1}][bin_X][bin_Y] += 1$
10: **return** $\frac{pdf_X}{n}, \frac{pdf_{X_{n+1}X}}{n}, \frac{pdf_{XY}}{n}, \frac{pdf_{X_{n+1}XY}}{n}$
11: **procedure** CALCULATE_TE($pdf_X, pdf_{XY}, pdf_{X_{n+1}X}, pdf_{X_{n+1}XY}, B$)
12: $te = 0$
13: **for** $i \leftarrow 1$ to B **do**
14: **for** $j \leftarrow 1$ to B **do**
15: **for** $k \leftarrow 1$ to B **do**
16: $te \leftarrow te + pdf_{X_{n+1}XY}[i,j,k] \log \frac{pdf_{X_{n+1}XY}[i,j,k]pdf_X[j]}{pdf_{XY}[j][k]pdf_{X_{n+1}X}[i][j]}$
17: **return** te

As with MI, the implementation of Algorithm 7 makes the following assumptions:

- The probability density functions are pre-calculated and do not change, i.e., all the data is required beforehand.
- The calculation of TE is performed in an offline batch fashion.
- The data is assumed to be synchronized (i.e., new values of X and Y arrive at each time tick).

4.4.3 Streaming PDF Estimation

Similarly to MI, the assumption that each stream maintains the same value until it receives a new update still holds, and it can be used in order to compute the required

PDFs incrementally, in a streaming manner. Hence, TE too, can be calculated at a fixed interval t, or on demand, over the always-up-to-date PDFs.

4.4.4 Sliding Window Model

The same sliding window approach described in Section 4.3.4 is also employed for the computation of Transfer Entropy, with the key difference being that different statistics are stored in each basic window in order to facilitate the incorporation of new values and expiration of old ones related to the PDFs required by TE instead of those required by MI. More specifically, while MI maintains frequency statistics for pdf_X, pdf_Y, and pdf_{XY}, TE maintains statistics for pdf_X, $pdf_{X_{n+1}X}$, pdf_{XY}, and $pdf_{X_{n+1}XY}$.

4.4.5 Removing the Fixed Interval

Even more importantly than in the case of Mutual Information, the Transfer Entropy computation needs to avoid being calculated at a fixed interval, mainly for two important reasons:

- The computation is "heavy" (even more than MI)
- The PDFs do not "change" often, especially for "slow" data streams

Hence, we can employ the same principle and calculate the TE measure based on the number of samples received rather than a fixed interval. Empirically, in order to get a good balance between resource consumption and latency minimization, the threshold of samples for TE needs to be higher than the corresponding one for MI, which can be directly justified by the two reasons mentioned above.

4.4.6 Streaming TE Computation

The approach described so far, presents similar drawbacks for TE, as with MI, namely:

- It requires a lot of memory (proportional to the number of basic windows), since it needs to maintain PDF statistics for each pair, and for each basic window
- Updates to the TE value only get calculated and reported every basic window or after receiving a specific amount of samples, which introduces a "delay".

In an effort to alleviate the aforementioned drawbacks, the mathematical model behind the algorithm re-designed to work in a streaming manner, and report updates as soon as they can be calculated, effectively alleviating any unnecessary delays, and keeping the memory footprint constant. Furthermore, as with MI, unnecessary and redundant calculations are avoided.

This approach follows the idea described for Mutual Information in Section 4.3.6, i.e., instead of iterating over the full dimensionality of the PDFs ($2D$ in MI and $3D$ in TE) in order to calculate the aggregated result, our algorithm needs to keep track of the "contribution" that each specific PDF cell has in the final result, and update the result in a streaming fashion (i.e., removing the old contribution and incorporating the new one). More specifically, every new update on stream X, only affects specific cells of the PDFs of all pairs $< X, Y >$, whereas all the remaining cells do not get modified. Hence, only the modified cells will ultimately affect the result, allowing the algorithm to leverage this observation and only perform a small subset of the operations that the batch update would perform.

Moving towards this observation, the original computation

$$TE_{Y\to X} = \sum_{x_{n+1}\in X}\sum_{x_n\in X}\sum_{y_n\in Y} p(x_{n+1},x_n,y_n)\log\left(\frac{p(x_{n+1},x_n,y_n)p(x)}{p(x_n,y_n)\,p(x_{n+1},x_n)}\right),$$

which has a complexity of $O(B^3)$, can now be expressed as

$$TE_{Y\to X} = \underbrace{\frac{1}{N_{x_{n+1}x_ny_n}}}_{\text{O(1)}}\underbrace{\sum_{x_{n+1}\in X}\sum_{x_n\in X}\sum_{y_n\in Y} C(x_{n+1},x_n,y_n)\log\left(\frac{C(x_{n+1},x_n,y_n)C(x)}{C(x_n,y_n)\,C(x_{n+1},x_n)}\right)}_{\text{O}(B^2)\text{ (for every }x_ny_n\text{)}}$$

$$-\underbrace{\frac{\log\frac{N_{xy}N_{x_{n+1}x_n}}{N_{x_{n+1}x_ny_n}N_{x_n}}}{N_{x_{n+1}x_ny_n}}}_{\text{O(1)}}\underbrace{\sum_{x_{n+1}\in X}\sum_{x_n\in X}\sum_{y_n\in Y} C(x_{n+1},x_n,y_n)}_{\text{O(1) (=previous value+1)}}$$

Although the expanded expression seems more complicated at first sight, it provides the opportunity of independently updating the result for each input update. As with MI, the terms only involving the sample sizes, as well as the term involving the sum over all PDF cells, can be computed in constant time. Moreover, by maintaining a "contribution" matrix for the first sum, and only updating $TE_{Y\to X}$ by removing old contributions of specific cells and adding new ones, the system can achieve truly streaming updates. More specifically, although the first sum initially seems to have $O(B^3)$ complexity, since

only the cells corresponding to the new value introduced by x_{n+1} actually influence the computation, the algorithm only need to iterate over all $x_n y_n$ values corresponding to this specific x_{n+1}, hence making it an $O(B^2)$ computation. Moreover, since the algorithm also exploits the sparsity of PDFs (described in previous sections) and represents them using sparse data structures, $O(B^2)$ constitutes a worst-case complexity, while, in practice, the actual operations needed, are much less.

4.4.7 Distributed Computation

Since the distribution of work happens in the level of pairs, and not in the level of each individual pair's computation, the same procedure that was followed to distribute the computation of Mutual Information, is also employed for Transfer Entropy.

Chapter 5

T-Storm Framework

5.1 Overview

Since all the systems presented in chapter 4 are related with monitoring various data streams in an online fashion and detecting their pairwise dependence, a unified framework was created in order to provide the combined functionality of all of them. T-Storm establishes a software suite that treats each one of those individual systems as "black boxes" with different characteristics. These "black boxes" are interchangeable in the sense that they provide the same interface (i.e., to which extend stream X influences stream Y), with each one of them providing different benefits that can be leveraged as trade-offs depending on the user's intent. For example if the user is interested in detecting linear relationships among many thousands of data streams, they should choose the Pearson correlation component, yet if they are interested in monitoring fewer data streams but with higher detail, like non-liner relationships and directionality, then they should choose the Transfer Entropy component.

Thus, T-Storm was created in order to offer the possibility of:

- Easily inserting input data in the system without knowing details about it
- Easily configuring everything to leverage all possible trade-offs (e.g., which "black box" to use, defining whether the hardware is memory-bounded or whether latency is more important than memory, defining parameters like sliding window size, thresholds, etc.)
- Receiving the output of the system in a completely independent environment (e.g., the user's PC) than where everything is deployed (e.g., a cluster)

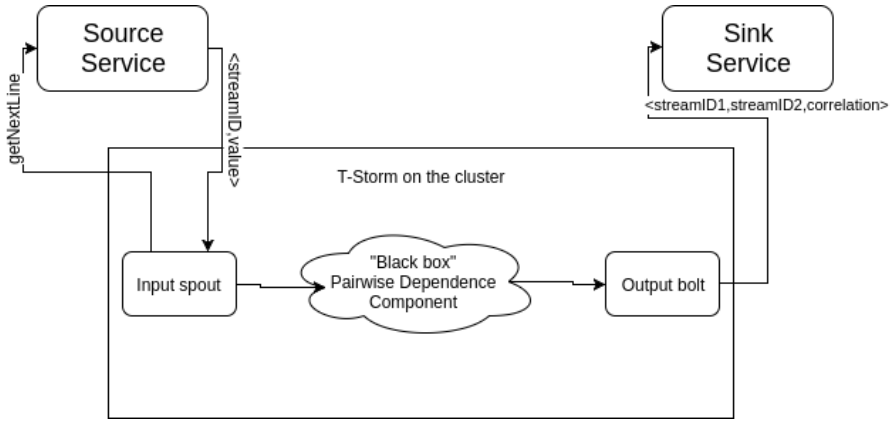

FIGURE 5.1: T-Storm Architecture

5.2 Architecture

T-Storm's architecture accounts for the fact that usually a job meant to be executed on a cluster of machines, requires a way to provide input and receive output, and the user who issues the job is usually working on a PC that is not part of the cluster. T-Storm makes it easy for the user to handle input/output data, as well as configure the entire system without having to remotely copy files in and out of the cluster. T-Storm's architecture is shown in Figure 5.1.

5.2.1 Source Service

This service is independent of the cluster and acts as an external source of data for T-Storm. It is responsible for providing data in a specific format and implements the required synchronization and communication behavior in order to provide this data to the system over the network. The source has been implemented (in standalone Java) as an external block-waiting service for the convenience of the user and the stability of the system.

The main entities involved are:

- **Source.java**: This class implements a server listening for connections initiated from T-Storm's spout. When it receives a new connection, it starts a **ConsumerHandler** thread.
- **ConsumerHandler.java**: This thread is responsible for communicating with T-Storm's spout and providing tuples one-by-one upon request. More specifically it receives a request for a new tuple from the spout, and when this happens, it needs to generate/load the next tuple from a class implementing the **InputProvider** Interface and reply to this request with the new tuple.

- **InputProvider.java**: This Interface describes the intended structure of any class that is meant for providing data to T-Storm using this Source service. Example implementations of this interface include classes that generate tuples based on some mathematical functions, and classes that read data that is stored in a file.erlinegenerates data dynamically, while **ExampleInputFileProvider.java** is an example class that loads data that is stored in a file.

The intended usage of this service provides the ability to concurrently supply multiple sources for T-Storm from multiple different locations.

5.2.2 Processing Elements

T-Storm incorporates interchangeable components which act as processing elements. In favor of simplicity, it "hides" their implementation from the user, and treats them as a "black box", for which the user only needs to know **what it does** and how to configure it externally, **not how it is implemented**.

New processing elements can be added to the system transparently, without altering any functionality related to I/O, as long as they implement the **PairwiseStreamingDependence** interface. This interface defines the input and output of the processing element. The input needs to consist of two fields, namely the stream ID and the new value of this stream, while the output consists of three fields, namely two stream IDs (the pair of streams whose dependence is being calculated) along with the updated value representing their dependence.

Although all the processing elements of T-Storm implement this interface, the meaning of the output differs depending on which component is being used. For example, when calculating how stream X is related to stream Y, if the Pearson Correlation component is chosen, then only one tuple $< X, Y, c >$ will be generated, where c represents their correlation and ranges from -1 to 1, while if the Transfer Entropy component is chosen, then two tuples ($< X, Y, c >$ and $< Y, X, c >$) will be generated (due to directionality), namely how much X influences Y and vice versa. In that case, c ranges from 0 (completely independent) to $log(B)$, where B is the number of bins configured for the PDF estimation.

5.2.3 Sink Service

This service is independent of the cluster and acts as an external sink. It is responsible for receiving results from T-Storm in a specific format, splitting them in appropriate fields and providing the structured result to the user.

The main entities involved are:

- **Sink.java**: This class implements a server listening for connections initiated from T-Storm's final bolt. When it receives a new connection, it starts a **Producer-Handler** thread.
- **ProducerHandler.java**: This thread is responsible for communicating with T-Storm's final bolt and receiving results one-by-one. Every time there is a new result produced by T-Storm, it receives a message containing this result, splits it to appropriate fields, and feeds it to a class implementing the **OutputProcessor** Interface.
- **OutputProcessor.java**: This Interface describes the intended structure of any class that is meant for processing results received from the Sink service. Example implementations of this interface can range from classes that simply print the results to screen, to classes that construct visual representations of the data (e.g., graphs, charts, etc.).

The intended usage of this service provides the ability to concurrently monitor results generated by T-Storm from multiple different locations, by executing multiple sinks.

Chapter 6

Results

This Chapter presents results related to the systems described in Chapter 4. Section 6.1 discusses our experimental setup, section 6.2 presents the data sets used for experiments, including real (6.2.1) and synthetic (6.2.2) ones, while section 6.3 shows the results of those experiments.

6.1 Experimental Setup

To evaluate T-Storm and its processing elements, various different experiments and test scenarios were considered. Yet, a similar process was followed for each one of the components, since they all aim to provide efficient approaches to the problem of monitoring the pairwise dependence among thousands of data streams in real time, each one with its own characteristics.

The respective centralized (batch) approach was implemented for each one of the components so that it can be used as "ground truth". Hence, by "feeding" the same input data sets to both the centralized version, and each one of the different implementations of each component, their correctness could be verified.

Having verified their correctness, the performance of the individual components was evaluated against different metrics, in an effort to make each component's strengths and weaknesses clear, as well as trade-offs among them. Such metrics include total execution time of a fixed-duration data set, scalability of the total number of data streams with increasing workers on the cluster, and average latency per update.

Some of the evaluations were executed on SoftNet's cluster, which consists of 21 nodes (3 nodes with 32Gb of RAM, 3 with 16Gb, and the rest with 8Gb), while others (which

were independent of scalability), were executed (in "single-threaded" mode) on a single cluster node with 32Gb of RAM and an Intel CPU clocked at 2.10GHz.

We distinguish five different components, namely:

- **Pearson Correlation**, the implementation of which includes all the optimizations mentioned in section 4.2, where the implementation of storing the coefficients follows the approach described as "Replication of Information" (4.2.7.1).
- **Mutual Information (streaming)**, the implementation of which includes all the optimizations mentioned in section 4.3, following the "per-update" approach described in section 4.3.6 (i.e., updating the output as soon as a new update is received, without waiting for a basic window to fill).
- **Mutual Information (windowed)**, the implementation of which includes all the optimizations mentioned in section 4.3, following the sliding window approach described in section 4.3.4.
- **Transfer Entropy (streaming)**, the implementation of which includes all the optimizations mentioned in section 4.4, following the "per-update" approach described in section 4.4.6 (i.e., updating the output as soon as a new update is received, without waiting for a basic window to fill).
- **Transfer Entropy (windowed)**, the implementation of which includes all the optimizations mentioned in section 4.4, following the sliding window approach described in section 4.4.4.

6.2 Data Sets

Both real and synthetic data sets were used for evaluations. This section presents the characteristics of each data set. In general, the evaluations were performed using two main data sets (one real and one synthetic), which however were sub-sampled in order to create many smaller ones, so that we can see how the components scale as the size of the data set increases, while also maintaining the same characteristics.

All of our data sets consist of sequential tuples/updates of the form $< streamID, value >$, where $streamID$ is a unique identifier for each stream (e.g., the stock name), while $value$, is the latest price update received for this specific stream.

6.2.1 Real Data

For this data set, we used financial data from a 2-week period surrounding the UK referendum, also known as BrExit. We analyzed all the data from this period, in hourly buckets, both in terms of total data streams and in terms of total updates, and picked one of the busiest hours of one of busiest days, namely the 30th of June 2016 at 16:00:00-17:00:00. The particular data set, **referred to as RD**, was divided into smaller data sets, where only a subset of the original data streams was kept for each one, in order to evaluate scalability incrementally.

The table below summarizes the characteristics of each sub-sampled data set created. Furthermore, Figure 6.1 shows the frequency analysis for the two week period (with the selected hour being highlighted in a red rectangle), while Figure 6.2 shows how the total number of updates and the average update rate per second scale versus increasing data streams for the selected data set.

Data streams	Total updates	Avg. updates / sec
100	60960	17
200	166364	46
300	236157	66
400	295261	82
500	345032	96
600	399328	111
700	439524	122
800	468077	130
900	493256	137
1000	512078	142
1100	520395	145
1200	521763	145
1265	521864	145

Since RD refers to real data, a very important trait related to it, is that some of the data streams are quite "fast" (i.e., they have updates very frequently), while others are very "slow". For example, well known stocks such as Google's (GOOGL) and Apple's (AAPL) receive much more frequent updates than less known (maybe newly introduced) stocks. This fact can play an important role about the non-windowed components, since they perform calculations for each update, while the windowed ones aggregate information and perform a "batch" update at the end of each basic window.

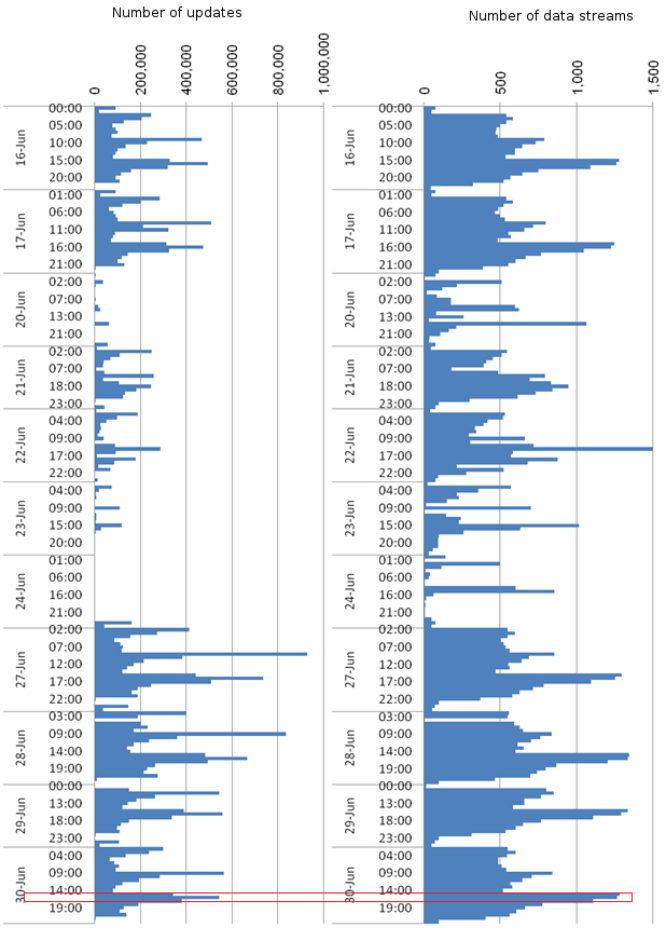

FIGURE 6.1: Frequency analysis of BrExit 2-week period

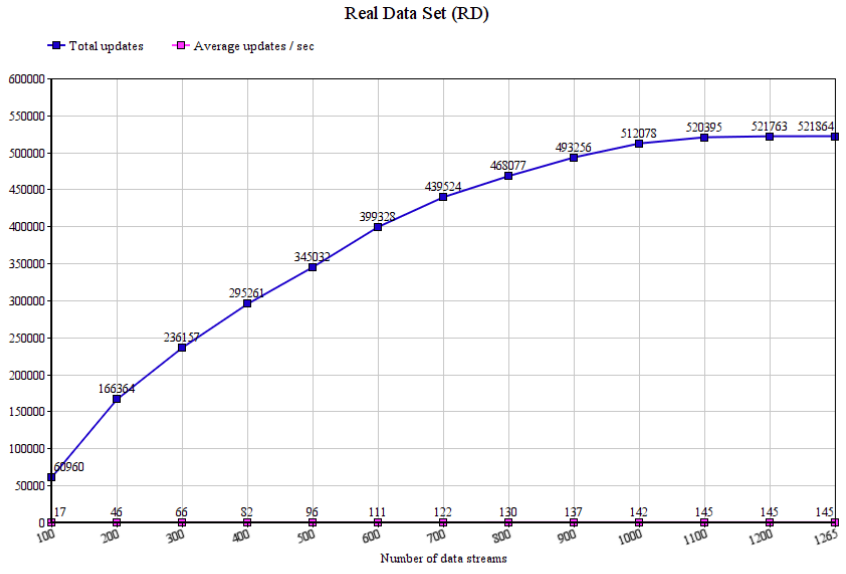

FIGURE 6.2: Real data set (RD)

6.2.2 Synthetic Data

Since the real data set offers a bounded number of data streams and total updates, a synthetic data set, **referred to as SD**, was created for the purposes of "stress-testing" the systems and also measuring their scalability under heavier load. As with the real data set, there are multiple smaller data sets under this "family". More specifically, each data set consists of a specific number of data streams, N, with $\frac{N}{4}$ total updates per second ($\frac{1}{4}$ of the data streams at random receive a new update every second) and $\frac{N}{4}3600$ updates in total (1 hour total duration).

The table below summarizes the characteristics of the synthetic data sets, while Figure 6.3 shows a graphic representation of how the total updates and update frequency change with respect to the number of data streams.

Data streams	Total updates	Avg. updates / sec
500	450000	125
1000	900000	250
1500	1350000	375
2000	1800000	500
2500	2250000	625
3000	2700000	750
4000	3600000	1000
5000	4500000	1250
6000	5400000	1500
7000	6300000	1750

The synthetic data set is more "dense" than the real data set (in the sense that there are more average updates per second), and much more uniform (in the sense that during each second, the $\frac{N}{4}$ updates are unique and refer to distinct data streams, while in the real data set, a specific data stream might receive (much) more than one update per second). The uniformity introduces an extra challenge for our components, due to the fact that multiple updates referring to the same data stream can be batched together (if they occur "close" in time), while updates being uniformly spread across different streams, need to be processed independently.

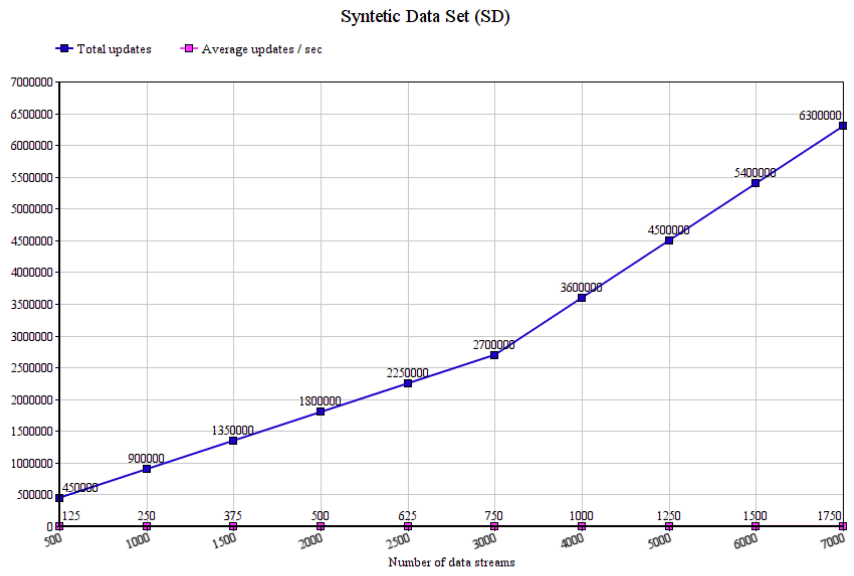

FIGURE 6.3: Synthetic data set (SD)

6.3 Performance

This section presents the performance results of the individual components evaluated against the metrics described in section 6.1 and the data sets described in section 6.2. The following sections provide numeric results related to each metric being evaluated, as well as a graphical representation of the numeric values.

6.3.1 Total execution time

This experiment measures the total time required by each component in order to fully process the real data set (RD), versus increasing number of data streams. It is executed on a single cluster node with 32Gb of memory. The intuition behind it, is to get an idea of how much the total execution time is affected by adding more data streams.

The table below shows the number of data streams of each experiment, and the total execution time of each component until the respective data set was fully processed. The empty cells on the table correspond to executions that took longer than 60 minutes in order to complete, so they violate the "real time" requirement. Figure 6.4 shows the data in a chart.

Data streams	Transfer Entropy streaming	Mutual Information streaming	Pearson Correlation	Transfer Entropy windowed	Mutual Information windowed
100	0.2	0.2	0.01	0.2	0.2
200	1.4	0.9	0.015	1.2	0.8
300	3.2	2.0	0.031	2.8	1.7
400	5.7	3.5	0.07	5.1	3.2
500	8.8	5.4	0.11	7.6	4.0
600	11.5	7.7	0.28	9.8	6.1
700	15.3	10.7	0.62	13.6	8.5
800	19.9	13.5	0.97	17.8	12.9
900	24.6	20.7	1.93	23.7	19.0
1000	31.8	23.1	2.6	46.8	41.9
1100	43.7	26.8	3.98		
1200		28.9	5.1		
1265		29	5.2		

As is made clear by the chart, the component requiring the less time in order to complete the experiment is the Pearson Correlation. Its great advantage lies in the fact that the complexity for it does not scale quadratically with respect to the number of streams (due to the pairs), because of the use of the grid structure (section 4.2.4) which greatly reduces the candidate pairs. Also, one must not directly compare the components solely

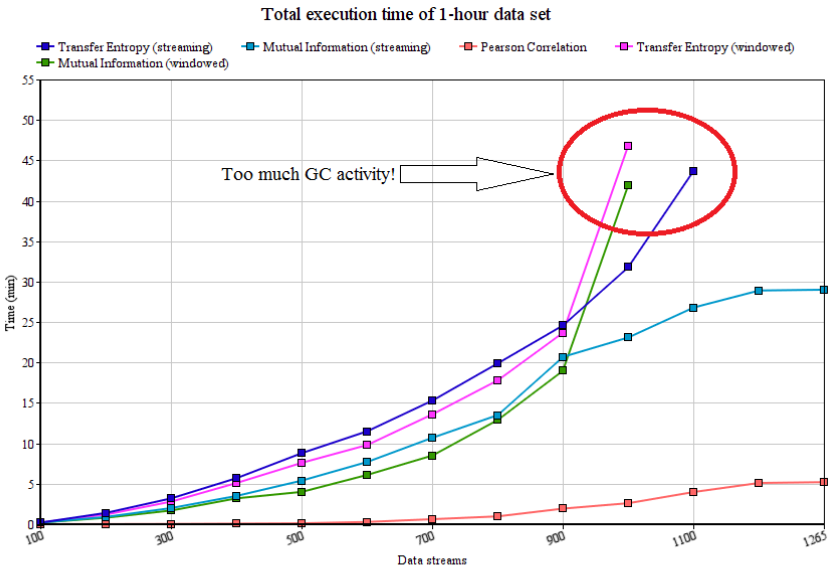

FIGURE 6.4: Total execution time

based on the execution time, since each of them provides a different quality of results. For example, since Transfer Entropy also offers directionality, one may choose to happily "pay" the extra computational cost versus the other two components in order to receive results involving this feature.

An interesting point in the chart is the range between 1000 and 1100 data streams, where the two windowed components, as well as the Transfer Entropy streaming component, start requiring (disproportionately) longer amounts of time in order to fully process the data set. This is a direct result of the fact that those components require more memory, and, at that point, they were consuming a total amount of memory which was close to the heap size of the JVM, hence the Garbage Collector was triggered quite often and required a lot of time to finish collection.

6.3.2 Scalability on the Cluster

This set of experiments aims to detect the scalability of the components as more and more cluster nodes are being used for the computation. It uses the synthetic data sets (SD) because it needs to be able to scale to larger number of data streams. Furthermore, since the synthetic data set is (intentionally) "heavier" than real data sets, these experiments also (should) constitute a lower bound to the case where real data with the same amounts of data streams could be used in the future. Each execution was also performed over a data set of one hour, with the basic window parameter of the windowed components being set to 30 seconds, and the components were evaluated against how many data streams they could support (i.e., adding more data streams would result to the data set not being able to complete within one hour).

The table below shows the corresponding numeric results, while Figures in group 6.5 show the same results in graphs. Pearson Correlation has been presented in a different graph than the rest of the components for better visibility due to different ranges.

Workers	**Transfer Entropy streaming**	**Transfer Entropy windowed**	**Mutual Information streaming**	**Mutual Information windowed**	**Pearson Correlation**
1	800	600	1100	850	23000
2	1200	950	1600	1350	21500
4	2150	1700	2900	2550	26500
8	3500	2600	4200	3800	31000
16	5500	3900	5400	4600	32500
32	6000	4300	6300	5400	33000

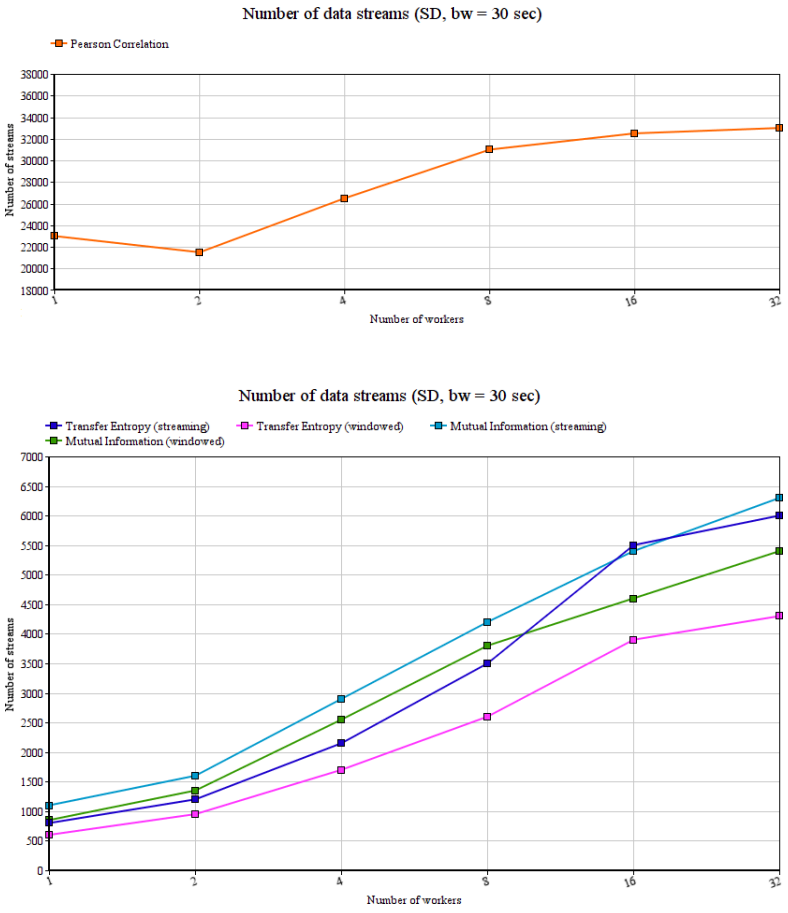

FIGURE 6.5: Scalability

As expected, since the Pearson Correlation is the most "lightweight" of the components, it can achieve processing a much higher number of data streams. This is mainly due to the fact that it does not have to track **all** possible pairs, only the candidate ones, due to the grid structure and the Euclidean distance. Yet, one can notice that although this component can process many more data streams, it can scale with a lower (relative) rate than the other components. This behavior stems from the fact that the Pearson Correlation component requires more synchronization than the others (due to indexing streams in the grid structure and creating neighborhoods), and this leads to the workload not being equally distributed among the cluster nodes. Impressive is also the fact that performance for Pearson Correlation actually drops instead of rising when going from one cluster node to two. In this case, the synchronization overhead exceeds the benefit of having one extra computation node.

The rest of the components seem to behave in a similar manner, scaling roughly linearly with the number of pairs among data streams, with the windowed ones performing consistently worse than their streaming counterparts, since they require more memory, and as such, the nodes reach their memory limit faster, and slow down. Yet, the value of

the windowed components would be great in a (more rare) scenario, where the interest lies in monitoring much "faster" data streams (e.g., 20 updates per second instead of 1[1]), and the problem becomes CPU-bound instead of memory-bound. In such a scenario, the fact that the windowed versions aggregate incoming updates before incorporating them in the results, would greatly reduce CPU usage versus the streaming versions where calculations are performed for each and every update.

6.3.3 Latency

The aim of this evaluation is to focus on the average latency per update of the systems. As in section 6.3.1, the experiments were performed on a single cluster node, since, in a distributed computation, each update could potentially trigger the computation on many different pairs of data streams, handled by many different cluster nodes, so measuring the overall latency would require synchronization among the cluster nodes, which would inherently slow down the system itself. The total latency over the distributed execution, would also include communication costs (e.g., tuples being emitted from the source towards the processing nodes, etc.), yet measuring it on a single node, still provides an indication of how latency changes as more workload is introduced into each component. One should keep in mind that windowed components (MI, TE, as well as Pearson), are omitted from these experiments since the latency per update when using them is constant and equal to the length of the basic window (since the results are updated at the end of each basic window). Hence, only the streaming versions of MI and TE where evaluated in terms of latency per update. The evaluation includes executions against both *RD* and *SD*.

The table below shows the corresponding numeric results, while Figure 6.6 visualizes the same results in a graph.

[1]The real data set contains data streams with even 30 updates per second at specific times, yet these data streams are very few, and also this rate is not constant and only appears on "peak" times. The scenario described, refers on consistently "faster" and also numerous data streams.

Data streams	Transfer Entropy streaming - RD	Transfer Entropy streaming - SD	Mutual Information streaming - RD	Mutual Information streaming - SD
100	0.23	0.93	0.19	0.71
200	0.64	1.99	0.51	1.62
300	0.87	2.70	0.82	2.37
400	1.48	3.87	1.16	3.24
500	1.83	5.31	1.53	4.91
600	2.31	7.94	1.73	6.80
700	2.68	10.07	2.01	7.96
800	3.20		2.55	9.44
900	3.73		2.52	
1000	4.91		2.71	
1100	6.80		3.09	
1200			3.32	
1265			3.33	

Since all the experiments were executed over 1-hour data sets, any specific experiment that did not finish in time, was considered as "failed" and its latency was not reported (empty cells in the table).

The chart directly shows that these components are greatly influenced by the rate of updates, as well as the number of data streams. More data streams, means more pairs,

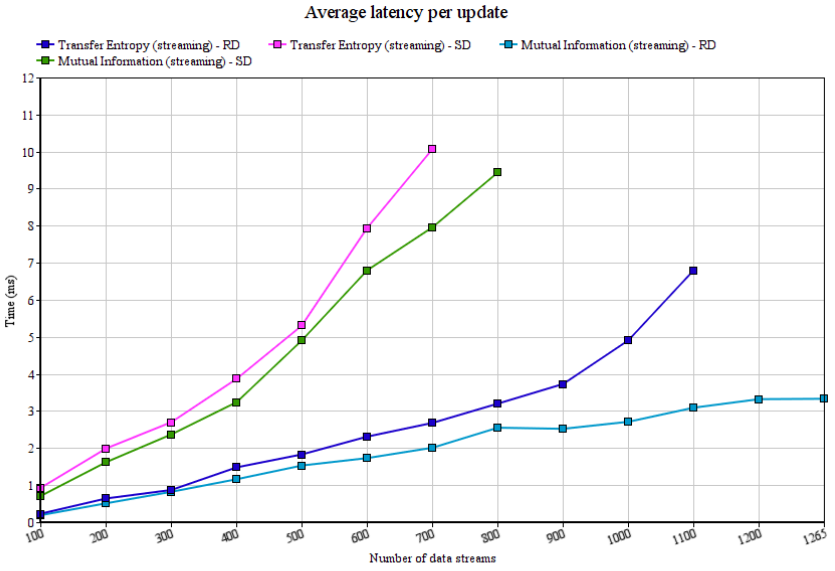

FIGURE 6.6: Average latency per update

and more updates means more calculations triggered. Yet, the great benefit of those components is instantly visible when one considers that the updated results are reported back to the user in under 10ms, while the windowed counterparts report results every basic window (e.g., 30 seconds).

Chapter 7

Related Work

The efficient management of large data series has been studied in various forms. A common approach is to first perform some kind of dimensionality reduction using various techniques [12–18], so that the original (large) data series can be represented using some (smaller) approximation/synopsis. Then, the synopses are usually used in order to build and maintain indexes [18–23] with different characteristics, depending on the original approximation and the problem at hand. Lastly, the indexes can be used for efficient retrieval of the data series and pruning purposes (i.e., reducing computational cost). Similar to our work, some of those approaches can be applied to the streaming scenario. T-Storm (Chapter 5) aims at providing a surrounding framework, using which, such techniques can be easily implemented and unified, enabling the developer to focus on the actual components, instead of the whole deployment-input-output streaming process. Our components (discussed in Chapter 4), follow similar techniques, and optimize for scalability and efficient computation of pairwise metrics in the streaming scenario.

Our Pearson Correlation Estimator discussed in Section 4.2 follows the approach presented by Shasha et al. in StatStream [9]. Yet, StatStream was designed for centralized execution, while our main contribution lies in the fact that our estimator is designed to be deployed over scalable, distributed systems, while also providing alternative implementations 4.2.7.1 in order to be highly adaptable in different scenarios (e.g., hardware with low memory or slow communication links), depending on the user's main requirement (e.g., low latency). Guo et al. [24] follow a similar approach, yet they perform their indexing using the normalized time series, instead of first computing their DFT coefficients for dimensionality reduction. Although this approach provides more accuracy (since they perform their calculations on the original time series and not on synopses), it is much less scalable, in the sense that the indexing dimensionality is equal to the size

of the sliding window and can only be tuned by reducing its size, which is impractical in many real world applications, and thus, large time series can lead to serious bottlenecks.

Nguyen et al. [25, 26] focus on generalizing the problem of mutual dependence for higher dimensions instead of focusing on pairs. Specifically, they use the cumulative entropy as their base measure, since they use some of its properties in order to achieve higher dimensions. Yet, their approach needs to compute an optimal partitioning of conditional cumulative density functions, which can be quite "expensive" in terms of computational cost, and does not fit in the streaming scenario over thousands of incoming time series. Similarly, Guha et al. [27] focus on improving correlation accuracy, by using singular value decomposition (SVD). Yet SVD re-computation is quite expensive too. Zhang et al. [28] try to represent time-series as "Macro-Boolean series" consisting of 0 and 1, where there is a 0 if the original value of the time series lies below its respective mean, and a 1 if it lies above. Although this approach is very suitable for the streaming scenario in terms of computational cost, it does not take into account the magnitude of each time point with respect to the time series mean. Hence, two time series could show similar behavior based on this metric (e.g., both lie below its mean value), while in practice, they could be behaving different (e.g., one increasing and the other decreasing - while both still being under their means). MUSCLES [29] and SPIRIT [30] mostly focus on discovering missing values (interpolation) and predicting future ones (forecasting).

In a more specific setting, Mutual Information [5] and Transfer Entropy [6] have been discussed in literature, yet, to the best of our knowledge, Transfer Entropy has not been examined in the streaming scenario. Keller et al. [31] monitor the Mutual Information between two time series in a streaming manner using nearest neighbor techniques which have been proposed by Kraskov and have been adopted as the most accurate Mutual Information estimator [32–35]. An interesting (and not trivial) problem is to examine whether similar techniques can be leveraged in order to develop a nearest neighbor estimator for Transfer Entropy, and adapt it in the streaming scenario.

Lastly, Palpanas [36] discusses the need for data Series Management Systems (SMSs) as an analogy to Database Management Systems (DBMSs), and presents characteristics that such systems should possess. The main vision is that SMSs are created as general purpose systems that can be used "out of the box" for complex time series analytics, covering a wide range of sequence queries and data mining operations. In a sense, T-Storm is an approach aiming to move towards this goal.

Chapter 8

Conclusions

8.1 Closing Remarks

This thesis focused on two main issues.

Introducing T-Storm
Chapter 5 describes T-Storm, a purpose-agnostic framework for implementing analytics over streaming data. T-Storm aims to provide a framework that a developer can use in order to easily implement time series analytics over streaming data, without having to worry about input and output. It efficiently handles providing input to the rest of the topology in a streaming manner, and receiving output from the cluster directly to the user's machine. T-Storm is independent of the problem being tackled, and can be used in any number of cases related to streaming time series analytics.

Scaling-out Streaming Time Series Analytics
Chapter 4 tackles the problem of efficiently scaling-out the problem of monitoring the pairwise dependence among thousands of incoming data streams in real time. Several components with inherently different characteristics where developed in order to provide a wide range of trade-offs between real or near-real time computation, accuracy, latency, (bi-)directionality, etc. Each of the processing elements provides its own strengths and weaknesses to the system as a whole, while all of them have been created and optimized with the streaming and distributed scenario in mind. Chapter 6 provides insightful information and reveals potential trade-offs by evaluating the components against different metrics.

8.2 Future Work

All of our processing elements exploit some kind of signature being generated using the original data. Pearson Correlation uses the DFT approximation, while Mutual Information and Transfer Entropy use the PDF estimation through histogram binning.

8.2.1 Pearson Correlation

An interesting improvement for Pearson Correlation as future work, would be to detach the distance measure (i.e., Euclidean distance on top of DFT) from the index (i.e., the grid structure) and be able to use different kinds of distance measures (e.g., Cosine or Hamming distance) depending on the user's needs. Furthermore, a generic index (e.g., LSH index) would be needed in order to be able to index similar streams irrespective of the distance measure selected, since the grid structure only works for Euclidean distance.

8.2.2 Mutual Information & Transfer Entropy

Although histogram binning is an efficient way of representing the original distribution of the time series, it is not as accurate as other ways explored in the literature. **Nearest-Neighbor** techniques and **Kernel Functions** have been used instead of histogram binning for a more accurate PDF approximation in the offline setting of these algorithms. Future work for T-Storm includes detecting efficient ways to implement such estimations with the streaming and distributed scenario in mind.

Bibliography

[1] QualiMaster Project. http://qualimaster.eu/.

[2] Michel Marie Deza and Elena Deza. Encyclopedia of distances. In *Encyclopedia of Distances*, pages 1–583. Springer, 2009.

[3] Frederick E Croxton and Dudley J Cowden. Applied general statistics. 1939.

[4] Alexander Craig Aitken. *Statistical mathematics*. Oliver And Boyd; London, 1942.

[5] Claude E Shannon and Warren Weaver. The mathematical theory of information. 1949.

[6] Thomas Schreiber. Measuring information transfer. *Physical review letters*, 85(2): 461, 2000.

[7] Charu C Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

[8] Apache Storm. http://storm.apache.org/.

[9] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.

[10] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *International Conference on Foundations of Data Organization and Algorithms*, pages 69–84. Springer, 1993.

[11] Jon Louis Bentley, Bruce W Weide, and Andrew C Yao. Optimal expected-time algorithms for closest point problems. *ACM Transactions on Mathematical Software (TOMS)*, 6(4):563–580, 1980.

[12] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[13] Kin-Pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 126–133. IEEE, 1999.

[14] Shrikant Kashyap and Panagiotis Karras. Scalable knn search on vertically stored time series. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1334–1342. ACM, 2011.

[15] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems*, 3(3):263–286, 2001.

[16] Chung-Sheng Li, Philip S. Yu, and Vittorio Castelli. Hierarchyscan: A hierarchical similarity search algorithm for databases of long sequences. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 546–553. IEEE, 1996.

[17] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11. ACM, 2003.

[18] Davood Rafiei and Alberto Mendelzon. Similarity-based queries for time series data. In *ACM SIGMOD Record*, volume 26, pages 13–25. ACM, 1997.

[19] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The ts-tree: efficient time series search and retrieval. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pages 252–263. ACM, 2008.

[20] Patrick Schäfer and Mikael Högqvist. Sfa: a symbolic fourier approximation and index for similarity search in high dimensional datasets. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 516–527. ACM, 2012.

[21] Jin Shieh and Eamonn Keogh. i sax: indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 623–631. ACM, 2008.

[22] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *Proceedings of the VLDB Endowment*, 6(10):793–804, 2013.

[23] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. Ads: the adaptive data series index. *The VLDB Journal*, 25(6):843–866, 2016.

[24] Tian Guo, Saket Sathe, and Karl Aberer. Fast correlation discovery for large-scale streaming time-series data. Technical report, 2014.

[25] Hoang-Vu Nguyen, Panagiotis Mandros, and Jilles Vreeken. Universal dependency analysis. In *Proceedings of the 2016 SIAM International Conference on Data Mining*, pages 792–800. SIAM, 2016.

[26] Hoang Vu Nguyen, Emmanuel Müller, Jilles Vreeken, Pavel Efros, and Klemens Böhm. Multivariate maximal correlation analysis. In *ICML*, pages 775–783, 2014.

[27] Sudipto Guha, Dimitrios Gunopulos, and Nick Koudas. Correlating synchronous and asynchronous data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 529–534. ACM, 2003.

[28] Tiancheng Zhang, Dejun Yue, Yu Gu, Yi Wang, and Ge Yu. Adaptive correlation analysis in stream time series with sliding windows. *Computers & Mathematics with Applications*, 57(6):937–948, 2009.

[29] B-K Yi, Nikolaos D Sidiropoulos, Theodore Johnson, HV Jagadish, Christos Faloutsos, and Alexandros Biliris. Online data mining for co-evolving time sequences. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 13–22. IEEE, 2000.

[30] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. Streaming pattern discovery in multiple time-series. In *Proceedings of the 31st international conference on Very large data bases*, pages 697–708. VLDB Endowment, 2005.

[31] Fabian Keller, Emmanuel Müller, and Klemens Böhm. Estimating mutual information on data streams. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, page 3. ACM, 2015.

[32] Shiraj Khan, Sharba Bandyopadhyay, Auroop R Ganguly, Sunil Saigal, David J Erickson III, Vladimir Protopopescu, and George Ostrouchov. Relative performance of mutual information estimation methods for quantifying the dependence among short and noisy data. *Physical Review E*, 76(2):026209, 2007.

[33] Justin B Kinney and Gurinder S Atwal. Equitability, mutual information, and the maximal information coefficient. *Proceedings of the National Academy of Sciences*, 111(9):3354–3359, 2014.

[34] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Physical review E*, 69(6):066138, 2004.

[35] Janett Walters-Williams and Yan Li. Estimation of mutual information: A survey. In *International Conference on Rough Sets and Knowledge Technology*, pages 389–396. Springer, 2009.

[36] Themis Palpanas. Data series management: the road to big sequence analytics. *ACM SIGMOD Record*, 44(2):47–52, 2015.