THE HUMAN BRAIN PROJECT
T-Storm and Storm-LSH
Technical Report

Minos Garofalakis, Papapetrou Odysseas
Toutoudakis Michail, Pavlakis Nikolaos

Software Technology and Network Applications Laboratory (SoftNet)
Technical University of Crete, Greece

minos@softnet.tuc.gr
papapetrou@softnet.tuc.gr
mtoutoudakis@softnet.tuc.gr
npavlakis@softnet.tuc.gr

March 30, 2016

Contents

List of Figures

1 Introduction

In the context of SP7.4.2, we have developed two algorithms capable of detecting pairwise correlations among thousands of time series. The first one relies on Locality Sensitive Hashing (LSH)[7], [16], [5], whereas the second exploits StatStream [27]. Both algorithms are implemented over Apache Storm [1], a distributed real-time computation platform with good scalability properties that provides fault-tolerance and tuple processing guarantees.

StatStream. StatStream consists of a data stream monitoring system which computes a variety of single stream statistics (such as moving average, best fit slope, etc.) as well as multiple stream statistics (such as pairwise correlations, beta, etc.) in one pass with constant time (per input) and bounded memory. Even though StatStream is fairly efficient in terms of finding pairwise correlations, the original proposal does not consider horizontal scalability, e.g., over cloud computing resources, i.e., it is limited to the hardware resources of a single machine. In this work, we describe T-Storm, an adaptation of StatStream suitable to run over distributed resources. T-Storm is suitable for deployment over cloud computing resources, and can transparently scale with the addition of new hardware.

LSH. Locality Sensitive Hashing [9] has found extensive use in recent years from various scientists and production systems. It was originally proposed for answering nearest neighbour or proximity search queries. Its major properties are as follows: 1) It accomplishes dimensionality reduction on data since original data is summarized by lower-dimensional *signatures* that can be used for estimating similarity between points, 2) It drastically reduces the required pairwise comparisons. Through bucket hashing and amplification, we can get candidate similar points in buckets such that any two points that do not fall in at least one common bucket are dissimilar with a high probability. So we can estimate distance (equiv. similarity or correlation) only between the candidate points and not between every possible pair, 3) LSH can be implemented for various distance measures such as Jaccard, Hamming, Edit, Cosine, Euclidean e.t.c., so it can be used in a wide variety of applications. However, most of the implementations of the LSH algorithm focus on processing offline data and not real-time data. So one of the contributions of our work is the implementation of LSH on a real-time processing engine for handling streaming online data. Additionally, we use the LSH Algorithm for finding similar pairs of streams and not for answering queries of approximate nearest neighbour (ANN).

In the following, we refer to the two systems as T-Storm (for the StatStream implementation) and Storm-LSH (for the LSH implementation).

The rest of this report is organized as follows. Section 2 discusses related work in the field of distributed data processing systems. Section 3 presents the key concepts of the distributed real-time computation engine and mathematically formalizes and analyses the two algorithms implemented. Section 4 introduces the implementation and topology of the T-Storm system, while Storm-LSH is analyzed in Section 5. Finally, we provide some final remarks in Section 6. Detailed instructions as to how to set up the required environment and execute both systems are provided in Appendices A and B. In Appendix A, we have also included download instructions to a prebuilt and preconfigured cluster of machines that we created for ease of use, as well as the sources of the implemented projects.

2 Related Work

Various distributed systems have recently emerged for processing massive data in high-speed environments. Storm [1] is a widely used platform, which provides fault tolerance and tuple processing guarantees. Zaharia et al. [26] proposed another model utilizing micro-batches for distributed stream processing. However it suffers from larger processing latency compared to the one-tuple-at-a-time model of [1] and [19]. Although these systems provide an extensive set of methods for real-time processing, they do not offer operators for correlation estimation.

Computation of real-time correlations on a standalone machine has been the main interest of [6], [25], [21]. These algorithms however are inefficient for massive-data in a distributed environment. Recently, partitioning-based approaches have been proposed for distributed batch data processing [8], [22], [24]. Still, these techniques cannot be applied on single-pass scenarios since they require a data analysis step to approximate the data distribution. Clearly, scanning the entire data to update the data distribution is impossible in streaming conditions.

Focusing on data streams, Agrawal et al. in [20], proposed an interesting approximation and indexing technique using Discrete Fourier Transformation for efficient similarity search. This work leverages the properties of DFT transformation in order to significantly reduce dimensionality and to provide an efficient index for fast lookup of similar items. Building on the work of Agrawal, Zhu and Shasha introduced StatStream[27] (Statistical Monitoring of Thousands of Data Streams in Real-Time), a framework that provides a data stream monitoring system to compute a variety of single stream statistics (such as moving average, best fit slope, etc.) as well as multiple stream statistics (such as pairwise correlations, beta, etc.) in one pass with constant time (per input) and bounded memory. Even though StatStream is highly efficient, as we will show in Section 4, distributing StatStream over Storm is not trivial for two main reasons. The first one is that the system requires a lot of shared memory as it needs to calculate the signatures for all the streams and then use them later on in order to calculate the pair-wise correlation. Since the two streams may be handled by different cluster nodes, an architecture based on message-passing rather than shared memory had to be implemented. The second important reason is synchronization. Some steps of the algorithm need to be completed fully before the next steps can begin. Synchronization is trivial when StatStream is executed in a single-machine environment, yet quite complex when distributed over a cluster of machines.

In recent years, LSH has attracted attention from various communities including computer vision (CV), machine learning, statistics, natural language processing (NLP), and the industry. Research from the machine learning and statistics community utilizes LSH as a probabilistic similarity-preserving dimensionality reduction method, from which the produced signatures can provide estimations to some pairwise distance or similarity. They mostly focus on the discovery of various LSH family functions that provide an unbiased estimator for a certain distance or similarity with smaller variance [12], [10], smaller space requirements [14], [15], or faster computation of hash functions [13], [23], [17]. Interestingly, LSH is also widely used in production environments in the IT industry, for near-duplicate web page and image detection, clustering and so on. For example, it was exploited extensively for detecting near-duplicate web pages in search engines [3], [4], even by Google [18].

3 Background

In this section we begin by briefly discussing the computation model offered by Storm, which is used as the real-time computation engine. Next, we provide some background related to the two implemented systems T-Storm and Storm-LSH. T-Storm is based on StatStream [27] while Storm-LSH on [7], [5]. We introduce some theoretical background behind each algorithm in it's own section.

3.1 Storm

Storm is a real time distributed platform where we implement and run topologies [1]. Topologies are computational graphs, which can be deployed over a cluster. Each node in a topology performs a well-defined data processing task. Links between nodes indicate the flow of data between the nodes. In addition to the above real-time computation principles, the following concepts are important as well:

- **Tuple** contains the actual data. A tuple is a named list of values that is emitted through the streams.
- **Stream** is a sequence of tuples. Through the streams, tuples are communicated between the nodes (bolts and spouts) of our topology.
- **Spout** is a source of streams. Streams originate from spouts, which flow data from external sources into the Storm topology. Spouts and bolts run the "logic" of the topology.
- **Bolt** consumes any number of input streams, performs some processing, and possibly emits new streams. Bolts can implement traditional functionality, like MapReduce transformations, or more complex actions (single-step functions) like filtering, aggregations, or communication with external entities such as a database.
- **Grouping** A stream grouping tells a topology how to send tuples between two components. Spouts and bolts execute many tasks across the cluster in parallel. Storm implements shuffling (random but equal distribution of tuples to bolts) or field grouping (stream partitioning based upon specific fields of the stream). Other stream groupings exist as well, including the ability for the producer to route tuples using its own internal logic.
- **hashTask function** is a function defined for each edge of the topology. It determines the task(s) of the subsequent processing element to which a tuple should be sent. The key-based shuffling function computes the hash value of a tuple key and sends it to the task to which the hash value is assigned. It's decision is based on two criteria: 1) Each key value must always be sent to the same task id of the next component and 2) The total number of streams must be load balanced to the number of tasks.

3.2 StatStream

StatStream consists of a streaming analytics framework that computes statistics over streams of data in a single pass. The statistics it computes comprise to both single stream statistics as well as multiple stream statistics. StatStream performs all calculations in a sliding-window fashion, where old data expire and new data get added in the current window. In order to compute the multiple stream statistics (such as the correlation coefficients), the system is using approximation techniques, namely the Discrete Fourier Transform, in order to produce signatures that accurately represent the original data, while introducing a small error factor. It is important to point out, that no revisiting of the expiring data is needed.

The system distinguishes three time periods (see Figure 1):

- **time-point** - the smallest time unit regarding the input data, e.g. second.
- **basic window** - a sequence of time-points over which the system maintains a digest incrementally, e.g., a few minutes.
- **sliding window** - a sequence of basic windows which defines the complete period of interest, e.g. an hour. The user might ask, "which pairs of stocks were correlated with a value of over 0.9 for the last hour?"

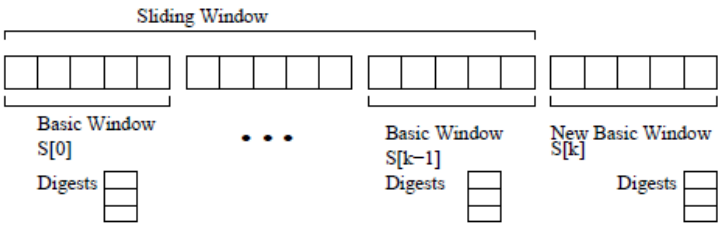

Figure 1: Sliding windows and basic windows - Figure taken from [27]

3.3 Statistics to monitor

Statistics the system can monitor consist of:

- Single stream statistics, such as moving average, standard deviation, etc.
- Multiple stream statistics such as Pearson correlation

$$corr(s,r) = \frac{\frac{1}{w}\sum_{i=1}^{w} s_i r_i - \bar{s}\bar{r}}{\sqrt{\sum_{i=1}^{w}(s_i-\bar{s})^2}\sqrt{\sum_{i=1}^{w}(r_i-\bar{r})^2}}$$

3.4 Maintaining statistics over sliding windows

To compute the statistics over a sliding window, the system maintains some digests (such as running sum and standard deviation) for each stream in order to be able to compute the statistics rapidly. The sliding window is equally divided into smaller windows called "basic windows", in order to efficiently remove old data and introduce new data to the current statistics. It keeps digests for both basic windows and sliding window. For example, the sum over the sliding window is updated as follows:

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

The size of the basic window plays an important role as it defines the maximum allowed computation time before the system starts "lagging". In order for the system to be considered "online", the results must be reported before the next basic window starts.

3.5 Computing pairwise correlations

The efficient identification of highly correlated streams potentially requires the computation of all pairwise correlations and could be proportional to the total number of time-points in each sliding window times all pairs of streams. The system makes this computation more efficient by (1) using a discrete Fourier transform of basic windows to compute the correlation of stream pairs approximately; (2) using a grid data structure to avoid the approximate computation for most pairs. The system computes the discrete Fourier transformation over each basic window for each stream and then updates the sliding window Fourier approximation by using the existing basic window digests. At this stage, every stream is described by a set of Fourier coefficients. Using DFT properties, the correlation between two streams is shown to be equal to the Euclidean distance of their normalized Fourier coefficients. Hence, the system can use a Grid structure and hash each stream s to a certain cell based on a subset of its first few Fourier coefficients. When the input query is to find all streams that are correlated with stream s with a correlation higher than a certain threshold t, then only streams hashed in neighboring cells in the grid structure need to be examined as they consist of a super-set of the true correlated streams (false positives). No false negatives are introduced.

3.6 LSH

In this section we introduce the mathematical theory behind the LSH algorithm. Initially, we define what a distance measure is and following we analyze Cosine and Euclidean distance measures that have been implemented into our platform Storm-LSH. Finally, we present the LSH mathematical theory.

3.6.1 Definition of a Distance Measure

A distance measure is a value that tells us how close or far two objects are. In real life distance usually refers to a physical length. In mathematics a distance measure is something more abstract and many types of distances exist. For example between two strings there is the notion of "Edit" distance. Theoretically, a distance measure is a function $d(x, y)$ that takes as arguments two points and produces as result a number which denotes how "far" or "close" the two points are. A distance measure must have the following properties:

1. $d(x, y) \geq 0$ (no negative distances)
2. $d(x, y) = 0$ if and only if $x = y$ (distances are all positive except the distance of a point to itself)
3. $d(x, y) = d(y, x)$ (Distances are symmetric)
4. $d(x, y) \leq d(x, z) + d(z, y)$ (Triangle Inequality)

3.6.1.1 Euclidean Distance In mathematics the Euclidean distance between two points is the length of the line that connects them. In literature someone may encounters it as *Pythagorian Metric* or $L_2 - Norm$ and is defined as:

$$d([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_n]) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{1}$$

However there are other distance measures that have been used for Euclidean Spaces. For any constant r we can define the $L_r - Norm$ to be the distance measure d defined by:

$$d([x_1, x_2, ..., x_n], [y_1, y_2, ..., y_n]) = \left(\sum_{i=1}^{n}(x_i - y_i)^r\right)^{(1/r)}$$

3.6.1.2 Cosine Distance Each point is the space is represented by a vector that originates from the origin of the axes and ends at the point in the space. The Cosine distance between two points, is the cosine of the angle that form two vectors. This angle will be in the range 0 to 180 degrees, regardless of how many dimensions the space has. We calculate the Cosine distance by computing the cosine of the angle, and then we convert it in degrees range by applying the arc-cosine function.

Given two vectors x and y, the cosine of the angle between them is the dot product $x \cdot y$ divided by the $L_2 - norms$ of x and y (i.e., their Euclidean distances from the origin).

$$x \cdot y = x^T y = [x_1, x_2, ..., x_n][y_1, y_2, ..., y_n]^T = \sum_{i=1}^{n} x_i y_i$$

Thus

$$\theta = arccos(\frac{x^T y}{||x||\ ||y||}) = arccos(\frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}})$$

3.6.1.3 Locality-Sensitive (LS) Families of Hash Functions

A Family of hash functions is any set of Hash Functions from which we can pick one at random efficiently. Suppose we have a space S of points with a distance measure $d(x, y)$.

Let $d_1 < d_2$ be two distances according to that distance measure d. A Family H of hash functions is said to be $(d_1, d_2, p_1, p_2) - sensitive$ for any x and y in S if:

1. if $d(x, y) < d_1$ then the probability of all $h \in H$, that $h(x) = h(y)$ is at least p_1
   i.e.: $Prob[h(x) = h(y)] \geq p_1$
2. If $d(x, y) > d_2$, then the probability over all $h \in H$, that $h(x) = h(y)$ is at most p_2
   i.e.: $Prob[h(x) = h(y)] \leq p_2$

HashFunctions in LSH algorithm are used for hashing data points to values. The key idea is to hash the points using several random hash functions and create smaller *signatures* or *sketches* that represent the original data. In later steps of the algorithm those reduced *signatures/sketches* are used for estimating similarity rather than the original data.

How to pick Random Vectors

In our implementation we used two different families of functions. For the Euclidean distance, we rely on the work of [7] and [16] which was also analyzed in [11] while for Cosine distance we relied on [11], [2] and [5].

Specifically for the Cosine distance, it suffices to restrict our choice to vectors whose components are $+1$ and 1. For the Euclidean distance on the other hand, we generate a random projection vector v of dimension d whose each entry is chosen independently at random from a p-stable distribution. More precisely, for the L_2 Euclidean distance we used the *Gaussian Normal Distribution* $N(0, 1)$.

Amplifying a Locality-Sensitive Family

Given a $(d_1, d_2, p_1, p_2) - sensitive$ family $\mathcal{F}$, we can construct new families $\mathcal{G}$ by randomly picking functions of the $\mathcal{F}$ and apply *AND* and/or *OR* constructions to them [11].

To create an *AND-construction*, we define a new family $\mathcal{G}$ of hash functions g, where each function g is constructed by picking randomly k functions $h_1, ..., h_k$ from $\mathcal{F}$. Then for a hash function $g \in \mathcal{G}, g(x) = g(y)$ if and only if all $h_i(x) = h_i(y)$ for all k randomly picked functions. Because the members of $\mathcal{F}$ are independently chosen for any $g \in \mathcal{G}, \mathcal{G}$ is a (d_1, d_2, p_1^k, p_2^k)-sensitive family.

To create an *OR-construction*, we define a new family $\mathcal{G}$ of hash functions g, where each function g is constructed again by picking randomly k functions $h_1, ..., h_k$ from $\mathcal{F}$. However, for the OR-construction, a hash function $g \in \mathcal{G}, g(x) = g(y)$ if for at least one of the randomly picket functions of $\mathcal{F}$ stands $h_i(x) = h_i(y)$. Since the members of $\mathcal{F}$ are independently chosen for any g $\in \mathcal{G}, \mathcal{G}$ is a $(d_1, d_2, 1 - (1 - p_1)^k, 1 - (1 - p_2)^k)$-sensitive family.

Effect of AND and OR Constructions

The $AND - construction$ reduces all probabilities, but if we choose **F** and r wisely, we can make the small probability p_2 shift very close to 0, while the higher probability p_1 to stay significantly away from 0.

Similarly, the $OR - construction$ makes all probabilities rise, but by choosing **F** and b again wisely, we can make the larger probability shift towards 1 while the smaller probability remains away from 1.

Applying an AND-Construction on r rows followed by an OR-Construction on b bands, shifts the initial probabilities according to the equation

$$p_{new} = 1 - (1 - p_{old}^r)^b \tag{2}$$

Composing Constructions

We can cascade AND- and OR-constructions in any order to make the low probability close to 0 and the high probability close to 1. Of course the more constructions we use, and the higher the values of r and b that we pick, the larger the number of functions from the original family that we are forced to use. Thus, the better the final family of functions is, the longer it takes to apply the functions from this family. The time/space requirements of the algorithm are proportional to $r \cdot b$, so the increase in the values of r and b is subject to quality-efficiency tradeoff.

$$noOfHashFunctions = r * b$$

4 T-Storm

This section presents the Storm components (spouts/bolts) that were created for T-Storm as well as their respective functionality. The Storm topology of the implementation is shown in Figure 2

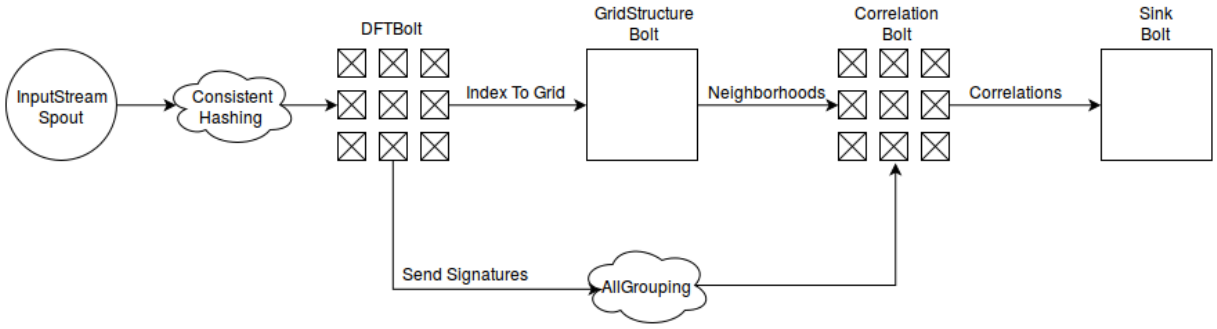

Figure 2: Topology of the distributed StatStream implementation in Storm

In summary, the input data are fetched into the system by the *InputStreamSpout*, where they are transformed into Storm streams and the necessary fields are emitted towards the *DFTBolt*. This bolt computes single-stream statistics, as well as the DFT coefficients for each stream, in sliding window fashion. When a basic window expires, the *DFTBolt* emits the DFT coefficients towards the *CorrelationBolt* and at the same time, it emits calculates the id of the Grid Structure where each stream is hashed and emits it towards the *GridStructureBolt*. The *GridStructureBolt* keeps track of the cells where streams are hashed. When the basic window expires, this bolt creates sets of neighboring cells and emits them to *CorrelationBolt*, which, having each stream's DFT coefficients, can compute the pairwise correlation among all streams that belong to the same set.

4.1 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 time point (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.

4.2 DFTBolt

This **bolt**[1] receives tuples from *InputStreamSpout* and computes single stream statistics (such as moving average, standard deviation, DFT coefficients, etc.) in a sliding-window fashion. It is responsible for keeping digests for each time-series. It incrementally computes the DFT coefficient digests of each time-series.

It 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 necessary actions in order to incorporate the new digests and discard the expired ones. Such actions include updating the DFT coefficients for each time-series based on its basic window digests (as described in StatStream), 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 calculating the grid hash key for each stream based on its first $\hat{n} < 2n$ dimensions

.

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

Finally, it emits the *gridHashKey* along with the time-series *ID* towards the *GridStructureBolt* and the time-series signature (*normalizedDFTCoefficients*), along with its *ID*, towards the *CorrelationBolt* for later usage, in order to calculate pairwise correlations using the DFT coefficients.

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

4.3 GridStructureBolt

This **bolt** is responsible for maintaining the Grid Structure. The Grid Structure holds all the stream *IDs*, each one hashed to a specific cell based on its first $\hat{n} < 2n$ dimensions (i.e. the first $\hat{n}$ Fourier Coefficients). 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 *neighborhoods* of streams hashed to adjacent cells. Finally, it emits these *neighborhoods* towards the *CorrelationBolt* in order to generate the appropriate stream pairs and calculate the needed correlations.

The Grid Structure is implemented in a centralized fashion in favor of latency and simplicity over memory of a specific node. The space complexity of the Grid Structure is O(n) as it holds at maximum 1 stream in each cell, and a total of n streams. Since n lies in the ranges 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 been created. It uniformly distributes the cells to multiple cluster nodes and uses message-passing in order to perform the necessary neighborhood calculations.

4.4 CorrelationBolt

This **bolt**[1] is responsible for receiving a set of time-series IDs and computing their pairwise correlations. The received set contains false positives that need to be filtered out, but does not include any false negatives.

The correlation computation is performed by using the normalized coefficients for each time-series (**timeseriesCoefficients**) previously received from *DFTBolt* and calculating pairwise Euclidean distance of the coefficients. Since the coefficients are normalized, they represent the normalized time-series, the correlation of which can be computed using the Euclidean distance, and since DFT preserves the Euclidean distance, the Pearson correlation of the original streams is equal to the Euclidean distance of their normalized DFT coefficients.

[1] This component's implementation is scalable and its parallelism hint can be set to arbitrary values depending on the nodes available on the cluster.

5 Storm-LSH

This section introduces our second contribution, the Storm-LSH framework. The topology of our implementation is shown in Figure 3. External Source Spout connects to an external source server and requests data. The data is preprocessed by the spout and tuples are generated and sent to the LSHBolt. LSHBolt maintains the windowing functionality, updates the signatures of the streams at the end of each *basic window*, estimates the buckets with the hashed points in them, and sends them to the next node of our topology. Then, Pair computation bolt takes the buckets and estimates per bucket all the unique possible pairs among the hashed points. Then it emits the estimated pairs to similarity computation bolt which performs the final distance/similarity computation, checks the results against a threshold, connects and emits them to the sink server. Each one of the components is analyzed thoroughly in the following sections.

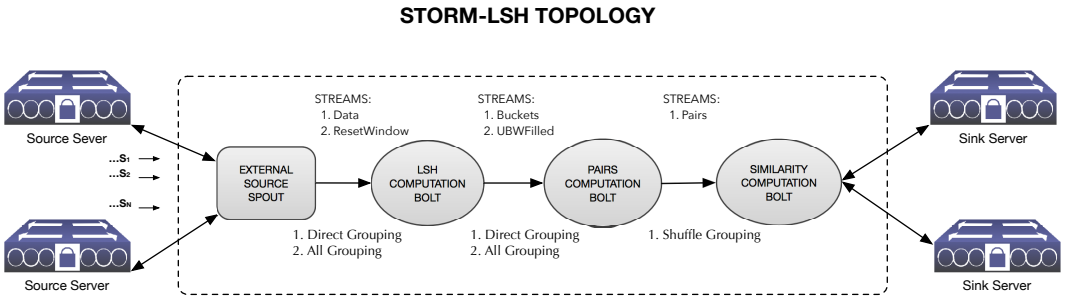

Figure 3: Topology Architecture of STORM-LSH

5.1 Windowing Scheme

The system's windowing scheme is implemented as StatStream and implements also three time objects.

- **Time Point** is a value of a stream that arrives in the topology at constant time intervals.
- **Basic Window** keeps the incoming time points. Besides the time points *basic window* keeps and the *hash functions* used for hashing the streams. Whenever a new *time point* arrives *hash functions* values for that specific position are generated and dot products of *time points* with *hash functions* values are updated. *Basic windows* are used for sums and dot products incremental estimation that are finally used in the estimation of the *signature/sketch* of the entire *sliding window*.
- **Sliding Window** A user-defined sequence of basic windows which defines the period of interest, e.g. 10 minutes. Note that similarity is estimated over the whole *sliding window*.

In Figure 4 you can see the windowing scheme of our implemented algorithm. In our implementation we assume that the data of the streams enters our topology aligned, e.g. we have one value for every stream in constant time intervals. Similar to StatStream, whenever a new *basic window* is filled up, it is pushed to the structure of the *sliding window*, and the oldest *basic window* is discarded.

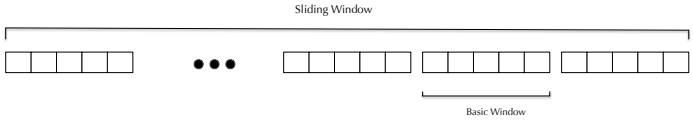

Figure 4: Sliding - Basic Window

5.2 External Source Spout

External Source Spout is responsible for the following operations. It connects to the source server or servers (each task of the spout opens a unique connection) and requests for new data. It then preprocesses new arrived data into tuples and emits them to the next element of our architecture the LSH

Bolt. It is also responsible for emitting reset window synchronization messages denoting that Basic Window is filled. Thus, External Source Spout declares two streams: 1) *data stream* and 2) *reset window stream*.

In *data stream* it emits data tuples using a *hashTask* function which decides the task id of the next bolt the data will be send to. In *reset window* stream *external source spout* emits a "reset" synchronization message to all tasks of listening bolts, that denotes the end of basic window (i.e. basic window has been filled up and we can compute the *signatures*).

5.3 LSH Computation Bolt

LSH Bolt performs the following tasks: 1) maintain the windowing scheme, 2) fill the basic windows with newly arrived values, 3) maintain the hash functions for the signatures calculation, 4) update the *signatures* every time a window gets filled and 5) calculate the buckets that streams get hashed in by using the amplification technique.

It subscribes in two streams: 1) *data stream* and 2) *reset window stream* of external source spout. Whenever it receives a value in "data" stream it pushes it to the Basic Window. Before pushing, it checks the position of the stream (i.e. the number of values that has been pushed so far to Basic Window). It then generates (online) the values for all the Hash Functions for the current position, if it has not already been generated by another stream for that particular position.

Whenever all of LSH Bolt's tasks receives a "reset" message from the External Source Spout, they: 1) push the "finished" Basic Window into the Sliding Window, 2) update the *signatures* of the Sliding Window, 3) generate a new Basic Window to be filled with the new values that will arrive, 4) estimate the buckets and the points that fall in them, using amplification by splitting the updated signatures into bands.

LSH Bolt declares two output streams: 1) *buckets stream* and 2) *basic window filled stream* for syncing purposes. In *buckets stream* it emits the buckets id's with the hashed in points and their *signatures* for the pairs and distance estimation, while in *basic window filled stream* it emits a reset message so that the next element of the topology will be aware when *basic window* has been filled and *signatures* estimation has been completed for the similarity to estimated.

5.4 Pairs Computation Bolt

Pairs Computation Bolt is responsible for: 1) estimating the pairs that are candidates for similarity calculation and 2) maintain stream positions in buckets as values change. Position maintenance is accomplished by checking if stream changed bucketId so that it must be deleted from old bucket and added to the new one.

It subscribes in two streams: 1) *basic window filled stream* of LSH Bolt for syncing and 2) *buckets stream* of LSH Bolt. Whenever it receives a message at *basic window filled stream* from all tasks of LSH Bolt, (which denotes that all windows has been filled and processed), it computes and emits pairs of points to next Bolt so that the similarity estimation can be completed.

Whenever Pairs Computation Bolt receives tuples from the *Buckets stream* it updates a local Multi-map <bucketNo, streamID> by adding the pair to Multi-map and checks for stream movement (stream changed bucket and need to update Multi-map - delete old entries).

Finally, Pairs Computation Bolt scans the buckets that by construction contain candidate pairs of high similarity and generates all combinations by two. It declares one stream named "pairs" where it emits to subscribers the estimated pairs and their signatures for similarity-distance estimation.

5.5 Similarity Computation Bolt

Similarity Computation Bolt is responsible for: 1) estimating the distance of the received pairs and 2) connecting and sending the results to external sink server. It subscribes to one stream where it receives the data and declares no output stream since it is the final element of our topology.

Depending on the chosen distance measure Cosine or Euclidean the estimated correlation differs. For the Cosine distance we estimate the Hamming distance of the two *sketches* of the points (i.e. the number of points they differ). From Hamming Distance we can estimate their angle [11] by using the equation:

$$Angle(x, y) = \frac{hamming(x, y)}{size\_of\_signature} * 180 \tag{3}$$

For the Euclidean Distance we just estimate the true Euclidean Distance of the pairs that have been generated. Euclidean distance can be estimated on both real/original data or from the generated *signatures*. The idea is that if two points are close in the original space, the will also be close in the reduced space, and if they were far originally they will be far in the reduced space as well. However we must adjust the thresholds accordingly.

Finally, estimated distance or similarity is checked against a threshold defined by the user and only pairs with smaller distance or bigger similarity that the threshold are sent to the sink server.

6 Conclusion

This section summarizes our work in the field of detecting correlations among thousands of data streams in real-time. The key contributions of both systems developed and provided as a service (T-Storm and STORM-LSH) are listed as follows:

- Efficient use of approximation techniques (such as DFT and hash functions) in order to reduce dimensionality (hence computational and spatial complexity) and represent original streams with reduced signatures.
- Efficient use of indexing methodologies (such as Grid Structure and LSH index) built on top of the signatures that leverage their characteristics in order to provide fast lookup.
- Can easily compute pairwise ($O(n)$) correlations among several thousands of data streams.
- Design and implementation of a widely distributed and scalable system that grows depending on the user's needs.
- Provided to the user as "black box". The user does not need to possess any prior knowledge related to distributed systems and/or real-time frameworks.

Furthermore, **T-Storm** adapts the StatStream mentality in a complex distributed architecture using message-passing in an efficient way, while **Storm-LSH** tackles the problem of implementing the LSH algorithm on real-time data and answering similar-pairs queries, rather than nearest-neighbor queries that it is widely used for.

References

[1] Apache storm. http://storm-project.net/.

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[3] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC*, pages 327–336, 1998.

[4] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29(8-13):1157–1166, 1997.

[5] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388. ACM, 2002.

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

[7] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262. ACM, 2004.

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

[9] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613. ACM, 1998.

[10] J. Ji, J. Li, S. Yan, B. Zhang, and Q. Tian. Super-bit locality-sensitive hashing. In *NIPS*, pages 108–116, 2012.

[11] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, Ch. 3, 2nd Ed.* Cambridge University Press, 2014.

[12] P. Li, K. W. Church, and T. Hastie. Conditional random sampling: A sketch-based sampling technique for sparse data. In *NIPS*, pages 873–880. MIT Press, 2006.

[13] P. Li, T. Hastie, and K. W. Church. Very sparse random projections. In *KDD*, pages 287–296. ACM, 2006.

[14] P. Li and A. C. König. b-bit minwise hashing. In *WWW*, pages 671–680. ACM, 2010.

[15] P. Li, A. C. König, and W. Gui. b-bit minwise hashing for estimating three-way similarities. In *NIPS*, pages 1387–1395. Curran Associates, Inc., 2010.

[16] P. Li, M. Mitzenmacher, and A. Shrivastava. Coding for random projections. In *ICML*, volume 32 of *JMLR Proceedings*, pages 676–684. JMLR.org, 2014.

[17] P. Li, A. B. Owen, and C. Zhang. One permutation hashing. In *NIPS*, pages 3122–3130, 2012.

[18] G. S. Manku, A. Jain, and A. D. Sarma. Detecting near-duplicates for web crawling. In *WWW*, pages 141–150, 2007.

[19] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: distributed stream computing platform. In *ICDM Workshops*, pages 170–177, 2010.

[20] C. F. R. Agrawal and A. N. Swami. Efficient similarity search in sequence databases. In *4th International Conference of Foundations of Data Organization and Algo- rithms (FODO)*, pages 69–84, 1993.

[21] Y. Sakurai, S. Papadimitriou, and C. Faloutsos. BRAID: stream mining through group lag correlations. In *SIGMOD Conference*, pages 599–610. ACM, 2005.

[22] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014.

[23] J. Wang, W. Liu, A. X. Sun, and Y. Jiang. Learning hash codes with listwise supervision. In *ICCV*, pages 3032–3039. IEEE Computer Society, 2013.

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

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

[26] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*. USENIX Association, 2012.

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

Appendices

In the Appendices section we provide detailed instructions on how to set up an environment suitable for executing both systems as well as detailed instructions as to how to provide input data to the system and receive results in real-time. On Appendix A, we provide a prebuilt and preconfigured downloadable cluster of virtual machines, for ease of use. We also provide details on how to extend it by adding more machines or create a new one from scratch. Appendix B provides instructions regarding the configuration of the two algorithms and the external services for Input/Output.

Appendix A Cluster Installation Migration Guide

For ease of use, installation, migration and extend of the application, we created a virtual cluster using VirtualBox. The Virtual Cluster is uploaded for downloading on a server on tuc.gr (pan.softnet.tuc.gr). It can easily be migrated and extended to another machine. The Cluster runs Apache-Storm 0.9.5, on top of which the algorithms are implemented.

Specifically, we created three virtual machines with ubuntu server 14.04 installed. One that runs the Storm master node, one that runs the Storm supervisor (aka worker) node and another one that runs the Zookeeper server node which is the middle-man between master and worker nodes. Zookeeper keeps the state of the nodes in case of failure and is indispensable for the Storm Platform.

Each Virtual Machine Utilizes two network interfaces. One for external communication (internet access) and another one for internal (in-cluster) communication. For in-cluster communication we have chosen the NAT ip class B network of *192.168.56.x* (secondary interface), while for internet communication two different methods have been applied. The master node, gets a real ip address (it has to be set up statically) in order someone to access storm-ui for monitoring running topologies statistics. All the other nodes, zookeeper and workers get NAT ip addresses of class B Network *10.0.2.x*. (primary interface).

The primary interface has been set up with DHCP, however if you notice any ip conflicts please set the Ips statically in */etc/network/interfaces*.

This is the minimum cluster that someone can run. Of course we could have installed all of them in one node, but in that case it would have been difficult and error prone to extend the cluster to multiple VMs. Storm and our projects have been installed in the VMs in order to be run out of the box.

In this document we analyze the process and steps needed to be taken, for installing, migrating or even extending the cluster for running the software. You can download the virtual cluster from the following url:

http://pan.softnet.tuc.gr/hbp/
username: hbp
password: Hum@nBr@!n

A.1 Installation - Migration

Transferring a virtual machine from one machine to another can be done using the following steps. Note however that VirtualBox must be preinstalled in the Target Machine. If not, the first step to be taken is to download and install VirtualBox on the Target Machine.

1. Install Virtual Box on the target machine if it hans't be installed.
2. Copy the .ova file to target machine, open VirtualBox and choose
   File → Import Appliance and choose the copied *.ova* file. VirtualMachine will be imported to VirtualBox

After importing the VMs to VirtualBox the following steps must be taken for the Virtual Machines and Cluster to be fully functional.

1. Run/open all three Virtual Machines.

2. Login using the following account: username=hbp-user, password=hbp-user.
3. Check the host name of each Virtual Machine and make sure that two Virtual Machines don't have the same hostname. The file where you setup the host name is */etc/hostname*. In order for hostname changes to be applied a restart is needed. To check the hostname just type in terminal *hostname*.
4. configure file /etc/hosts. There you can assign a name to every ip of the virtual machines of the cluster. We configure */etc/hosts* using the secondary interface ip addresses (in-cluster communication). Virtual Machines are already preconfigured and probably you won't have to change anything, however these checks must be made in order to be sure that cluster will work as expected.
5. When you will setup the Cluster in your server, you will have to assign to the master node a static ip. Don't forget to update, where available, all *storm.yaml* configuration files with the new static ip of the master node. The file is accessible (except the Zookeeper VM where no storm is installed) under the path: */home/storm/apache-storm-0.9.5/conf/storm.yaml* of every master and worker node Virtual Machines.
6. After the update please delete the directories *nimbus supervisor* and *workers* under */home/storm* so that they will be recreated when you restart the VM.
7. Finally restart all the Virtual Machines for the changes (if any) to take effect.
8. Thats all, now you can check your cluster from storm-ui found at *http://master-node-ip:8888*

If you want a new install of Storm from scratch you can find information in the following places:

- http://www.michael-noll.com/tutorials/running-multi-node-storm-cluster/
- http://knowm.org/how-to-install-a-distributed-apache-storm-cluster/
- http://stackoverflow.com/questions/30525661/how-to-configure-multi-node-apache-storm-cluster

A.2 Adding more workers to the cluster

To extend Worker/Supervisor nodes just clone the HBP-worker1 Virtual Machine by following the next steps.

1. Do a full clone give it a different name i.e. HBP-worker2 and check the box that says "Reinitialize the MAC address of all network cards".
2. Start the cloned Virtual Machine
3. Fix the new host name under /etc/hostname
4. Update */etc/network/interfaces* by giving to the cloned Virtual Machine a new unique unused ip address of the B-Class Network *192.168.56.x*. In the beginning it will have the same ip address as the cloned HBP-worker1. It needs to be changed.
5. Update /etc/hosts of all Virtual Machines with new node ip address of the new worker node.
6. In the new cloned Virtual Machine go to directory */home/storm/* and delete any existing *nimbus supervisor* and *workers* directories.
7. Restart the Virtual Machine.
8. Check cluster from storm-ui *http://master-node-ip:8888*.

A.3 Configuration Example

Zookeeper zoo.cfg

http://hadoop.apache.org/zookeeper/docs/current/zookeeperAdmin.html

# The number of milliseconds of each tick
tickTime=2000

```
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
dataDir=/var/lib/zookeeper
# Place the dataLogDir to a separate physical disc for better performance
# dataLogDir=/disk2/zookeeper

# the port at which the clients will connect
clientPort=2181

## specify all zookeeper servers
# The fist port is used by followers to connect to the leader
# The second one is used for leader election
#server.1=zookeeper1:2888:3888
#server.2=zookeeper2:2888:3888
#server.3=zookeeper3:2888:3888

# To avoid seeks ZooKeeper allocates space in the transaction log file in
# blocks of preAllocSize kilobytes. The default block size is 64M. One reason
# for changing the size of the blocks is to reduce the block size if snapshots
# are taken more often. (Also, see snapCount).
#preAllocSize=65536

# Clients can submit requests faster than ZooKeeper can process them,
# especially if there are a lot of clients. To prevent ZooKeeper from running
# out of memory due to queued requests, ZooKeeper will throttle clients so that
# there is no more than globalOutstandingLimit outstanding requests in the
# system. The default limit is 1,000.ZooKeeper logs transactions to a
# transaction log. After snapCount transactions are written to a log file a
# snapshot is started and a new transaction log file is started. The default
# snapCount is 10,000.
#snapCount=1000

# If this option is defined, requests will be will logged to a trace file named
# traceFile.year.month.day.
#traceFile=


# Leader accepts client connections. Default value is "yes". The leader machine
# coordinates updates. For higher update throughput at thes slight expense of
# read throughput the leader can be configured to not accept clients and focus
# on coordination.
# leaderServes=yes

autopurge.purgeInterval=24
autopurge.snamRetainCount=3
```

Master Node storm.yaml

```
# These MUST be filled in for a storm configuration
storm.zookeeper.servers:
- "hbp-zookeeper"
# - "server2"

nimbus.host: "hbp-nimbus"
nimbus.childopts: "-Xmx1024m -Djava.net.preferIPv4Stack=true"
ui.childopts: "-Xmx768m -Djava.net.preferIPv4Stack=true"
supervisor.childopts: "-Djava.net.preferIPv4Stack=true"
```

worker.childopts: "-Xmx768m -Djava.net.prefer.IPv4Stack=true"
storm.local.dir: "/home/storm"

ui.port: 8888

These may optionally be filled in:
List of custom serializations
topology.kryo.register:
- org.mycompany.MyType
- org.mycompany.MyType2: org.mycompany.MyType2Serializer
List of custom kryo decorators
topology.kryo.decorators:
- org.mycompany.MyDecorator
Locations of the drpc servers
drpc.servers:
- "server1"
- "server2"

Metrics Consumers
topology.metrics.consumer.register:
- class: "backtype.storm.metric.LoggingMetricsConsumer"
parallelism.hint: 1
- class: "org.mycompany.MyMetricsConsumer"
parallelism.hint: 1
argument:
- endpoint: "metrics-collector.mycompany.org"

Worker Node storm.yaml

These MUST be filled in for a storm configuration
storm.zookeeper.servers:
- "hbp-zookeeper"
- "server2"

nimbus.host: "hbp-nimbus"
nimbus.childopts: "-Xmx1024m -Djava.net.preferIPv4Stack=true"
ui.childopts: "-Xmx768m -Djava.net.preferIPv4Stack=true"
supervisor.childopts: "-Djava.net.preferIPv4Stack=true"
worker.childopts: "-Xmx768m -Djava.net.prefer.IPv4Stack=true"
storm.local.dir: "/home/storm"

supervisor.slots.ports:
- 6700
- 6701
- 6702
- 6703

These may optionally be filled in:

List of custom serializations
topology.kryo.register:
- org.mycompany.MyType
- org.mycompany.MyType2: org.mycompany.MyType2Serializer
#
List of custom kryo decorators
topology.kryo.decorators:
- org.mycompany.MyDecorator
#
Locations of the drpc servers
drpc.servers:

```
# - "server1"
# - "server2"

## Metrics Consumers
# topology.metrics.consumer.register:
# - class: "backtype.storm.metric.LoggingMetricsConsumer"
# parallelism.hint: 1
# - class: "org.mycompany.MyMetricsConsumer"
# parallelism.hint: 1
# argument:
# - endpoint: "metrics-collector.mycompany.org"
```

/etc/hosts file

```
127.0.0.1 localhost
#127.0.1.1 hpb-1
192.168.56.101 hbp-master
192.168.56.102 hbp-zookeeper
192.168.56.101 hbp-nimbus
192.168.56.101 hbp-stormui
192.168.56.103 hpb-worker1
192.168.56.104 hpb-worker2
192.168.56.105 hpb-worker3

# The following lines are desirable for IPv6 capable hosts
::1 localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Appendix B User Manual

This appendix provides a user-guide for executing T-Storm and Storm-LSH on the provided virtual machine cluster. Furthermore, it provides instructions regarding the external services that are responsible for IO, namely, the *source* and the *sink*.

B.1 Prerequisites

The following components are required for a full execution of the T-Storm and the STORM-LSH platforms:

- At least one source of data (external service provided)
- The pre-built and set-up deployment cluster
- Optionally one or more sink(s) for receiving results (external service provided)

B.2 The Source service

This service is independent of the cluster and acts as an external source of data for the implemented platforms. 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 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 the spout. When it receives a new connection, it starts a **ConsumerHandler** thread.
- **ConsumerHandler.java**: This thread is responsible for communicating with the 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 response with the new tuple.
- **InputProvider.java**: This Interface describes the intended structure of any class that is meant for providing data to the implemented topologies using this Source service. **ExampleInputProvider.java** is an example class that generates data dynamically, while **ExampleInputFileProvider.java** is an example class that loads data that is stored in a file.

The easiest way to provide data for the system is by executing the source passing two arguments as input;the port where the source starts listening and a filename of file containing input data. This service does not need to be executed inside any specific cluster node/virtual machine. It can be executed on the user's workstation.

- java -jar source 8000 data.txt

The expected data format is as follows:
$streamID(String), timestamp(Long), value(Double)$
$e.g.$ "$s0, 1456512536, 98.44$"

In order to implement a custom class that provides data to the system, a user needs to create a class that implements the **InputProvider** interface and then replace the *provider* assignment in **ConsumerHandler.java** Constructor.

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

Important note : In order to speed up the system and not create a bottleneck in the input stage of the system, it is highly recommended to split the input data among multiple sources rather than have a single source providing all the streams. For example, if the user wants to execute the system with 10000

streams as input, it is much more efficient to run 5 sources, each supplying 2000 streams, rather than one single source supplying all 10000 streams.

B.3 The Sink service

This service is independent of the cluster and acts as an external sink. It is responsible for receiving results from the running topologies 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 **ProviderHandler** thread.
- **ProviderHandler.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 results 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.
- **InputProvider.java**: This Interface describes the intended structure of any class that is meant for processing results received from the Sink service. **ExampleOutputProcessor.java** is an example class that receives a new result and "processes" it (simply prints it to standard output).

In order to implement a custom class that processes results from the system, a user needs to create a class that implements the **OutputProcessor** interface and then replace the *processor* assignment in **ProducerHandler.java** Constructor.

The intended usage of this service provides the ability to concurrently monitor results generated by a running topology from multiple different locations, by executing multiple sinks.

B.4 Executing T-Storm and Storm-LSH on the cluster

The pre-built virtual machine cluster contains all the necessary dependencies required in order to execute the implemented platforms. Furthermore, the home directory of the "hbp-user" contains the packaged jar files of both topologies, two configuration files (called "tstorm.properties" and "storm-lsh.properties" respectively one for each platform), as well as deployment scripts (called "deploy-tstorm.sh" and "deploy-storm-lsh.sh") created for easy deployment of the Storm topologies.

B.4.1 Configuring the algorithms parameters

The user has the ability to configure the algorithms parameters by editing the configuration files provided. The configuration fields include all the user-defined algorithm parameters (e.g. basic window size, sliding window size, number of DFT coefficients used, etc.) as well as the connection configuration for sources and sinks.

B.4.1.1 Common Parameters For Both Topologies T-Storm and Storm-LSH

- **BASIC_WINDOW_SIZE**: any positive integer. Defines the number of timepoints in a window (commonly 1 timepoint per second). Common values: 60, 120 (i.e. 1 minute, 2 minutes, etc.)
- **NO_OF_BASIC_WINDOWS_IN_SLIDING_WINDOW**: any positive integer. Defines the number of basic windows in a sliding window (which is the period of interest). Common values: 5, 10 (e.g. If sliding window equals 5 and basic window equals 60 and there is 1 timepoint per second, the period of interest is 5 minutes)
- **NO_OF_WORKERS**: should equal the number of available cluster machines.
- **THREADS_PER_WORKER**: any positive integer. Common values: 1, 2, 4. Depends on number of cores per machine.

- **SOURCES_SOCKETS**: $IP1 : port1, IP2 : port2, ..., IPX : portX$. Each $IP : port$ pair corresponds to a different source.
  (e.g. 147.27.14.144:8000,147.27.14.144:8001,147.27.14.200:8000)
- **SINKS_SOCKETS**: $IP1 : port1, IP2 : port2, ..., IPX : portX$. Each $IP : port$ pair corresponds to a different sink.
  (e.g. 147.27.14.144:9000,147.27.14.144:9001,147.27.14.200:9000)

B.4.1.2 T-Storm Specific Parameter valid range/format and short explanation

- **CORRELATION_THRESHOLD**: any double value in range (0...1). Common values: 0.9, 0.95
- **COEFFICIENTS_TO_USE**: any positive integer smaller than **BASIC_WINDOW_SIZE**. Defines the number of DFT coefficients to use for generating the stream signatures. Common values: 8, 16
- **INDEX_COEFFICIENTS_TO_USE**: any positive integer smaller than **COEFFICIENTS_TO_USE**. Defines the number of coefficients used for indexing in the Grid Structure. Common values: 1, 2

B.4.1.3 Storm-LSH Specific Parameter valid range/format and short explanation

- **DISTANCE_MEASURE** is the distance measure you want the algorithm to run on incoming data points. Accepted values are "euclidean" or "cosine" without the quotes.
- **EUCLIDEAN_DISTANCE_BUCKET_SIZE**: Any real value. It declares the size of consecutive intervals you split the line you project points onto. These intervals are the "buckets" where data points are hashed to. This value conditions the initial probabilities of the LSH algorithm.

  If $EUCLIDEAN\_DISTANCE\_BUCKET\_SIZE \leq d/2$, where d is the distance of the two points, then the initial probability that they will be hashed to the same bucket is $p \leq 1/3$ while in case that $EUCLIDEAN\_DISTANCE\_BUCKET\_SIZE \geq 2d$ the probability is $p \geq 1/2$. So we have an $(a/2, 2a, 1/2, 1/3)$-sensitive family of hash functions, where $a = EUCLIDEAN\_DISTANCE\_BUCKET\_SIZE$.

  Note: This variable has meaning only when topology is run under Euclidean distance measure.
- **ROWS_NO_PER_BAND**: Any positive integer. This variable declares the number of bands the signature of each streams is split. Typical values in range $[1 - 20]$. For more analytic information read section 3.6.1.3.
- **NO_OF_BANDS**: Any positive integer. This parameter declares the number of elements each band of the signature contains. Typical values in range $[5 - 20]$. For more analytic information read section 3.6.1.3.
- **EUCL_THRESSHOLD**: Any positive real value of distance. Pairs of streams whose euclidean distance is smaller than this threshold are reported to the sink server.
- **COS_THRESSHOLD**: Any positive integer value of an angle in range [0-180]. Pairs of streams with angles smaller than this threshold are reported to the sink server.

Summary

- Pick any two distances $d_1 < d_2$
- Start with a $(d_1, d_2, (1 - d_1), (1 - d_2)) - sensitive$ family
- Apply AND-OR constructions to amplify $(d_1, d_2, p_1, p_2) - sensitive$ family, where p_1 is almost 1 and p_2 is almost 0. Just split the generated per stream signatures of size h in b bands of r elements per band so that $h = r * b$ and initial probabilities will be shifted according to eq 2.
- The closer to 0 and 1 we get, the more hash functions must be used!

B.5 Deploying The Topologies and Monitoring Execution

After configuring the parameters, the user can use the respective deployment script in order to deploy a specific topology for execution on the cluster. Note that since *Sources* and *Sinks* act as "servers", they need to be executed **before** deploying any topology on the cluster. After executing sources and sinks on the desired location/workstation, a topology can be deployed by simply executing the respective deployment script (./deploy-tstorm.sh or ./deploy-storm-lsh.sh).

The execution of the system can be monitored through Storm UI, which provides statistics like tuples sent, execution time, etc. The user can monitor Storm UI through a web browser by visiting the cluster master node's IP followed by port 8888 (e.g. http://147.27.14.208:8888/).

B.6 Summary

In summary, the steps required for a full execution are shown below:

- Set up the desired sources and execute them
- Set up the desired sinks and execute them
- Configure the algorithm parameters (including source and sink IP:port pairs) through the configuration file
- Deploy topology using the respective deployment script
- Monitor execution through Storm UI
- Receive results in sink as they are being generated in real-time