Parallel Techniques for Big Data Analytics

Reza Akbarinia

UNIVERSITÉ DE MONTPELLIER
Faculté des sciences et techniques

H D R

Parallel Techniques for Big Data Analytics

Submitted for the degree of "Habilitation à Diriger des Recherches" of
Université de Montpellier
Speciality: Computer Science

By

Reza AKBARINIA

INRIA & LIRMM, University of Montpellier

HDR committee:

Reviewers:	João Gama, Professor, University of Porto
	Tamer Özsu, Professor, University of Waterloo
	Jean-Marc Petit, Professor, INSA Lyon
Examinators:	Frédérique Laforest, Professor, Université Jean-Monnet Saint Étienne
	Dennis Shasha, Professor, New York University
	Patrick Valduriez, Director of Research, INRIA
President:	Anne Laurent, Professor, Université de Montpellier

Abstract

Nowadays, we are witnessing the production of large volumes of data in many applications domains like social networks, medical monitoring, weather forecasting, biology, agronomy, earth monitoring, etc. Analyzing this data would help us to extract a lot of hidden knowledge about the events happened or to be happened in the future. However, traditional data analytics techniques are not efficient for analyzing such data volumes. A promising solution for improving the performance of data analytics is to take advantage of the computing power of distributed systems and parallel frameworks such as Spark.

In this HDR manuscript, I describe my research activities for developing parallel and distributed techniques to deal with two main data analytics problems: 1) similarity search over time series; 2) maximally informative k-itemsets mining.

The first problem, *i.e.*, *similarity search over time series*, is very important for many applications such as fraud detection in finance, earthquake prediction, plant monitoring, etc. In order to improve the performance of similarity queries, index construction is one of the most popular techniques, which has been successfully used in a variety of settings and applications. In our research activities, we took advantage of parallel and distributed frameworks such as Spark, and developed efficient solutions for parallel construction of tree-based and grid-based indexes over large time series datasets. We also developed efficient algorithms for parallel similarity search over distributed time series datasets using indexes.

The second problem, *i.e.*, *maximally informative k-itemsets mining* (*miki* for short), is one of the fundamental building bricks for exploring informative patterns in databases. Efficient *miki* mining has a high impact on various tasks such as supervised learning, unsupervised learning or information retrieval, to cite a few. A typical application is the discovery of discriminative sets of features, based on joint entropy, which allows distinguishing between different categories of objects. Indeed, with massive amounts of data, the miki mining is very challenging, due to high number of entropy computations. An efficient *miki* mining solution should scale up with the increase in the size of the itemsets, calling for cutting edge parallel algorithms and high performance computation of *miki*. We developed such a parallel solution that makes the discovery of *miki* from a very large database (up to Terabytes of data) simple and effective.

Contents

Chapter 1

Introduction

With the advances in the Internet and communication technology, we are witnessing the production of large volumes of data in many applications domains like the social networks, medical monitoring, weather forecasting, biology, agronomy, earth monitoring, etc. However, the traditional data management and analytics techniques, executed sequentially in a single machine, are not efficient for dealing with this data deluge. An appealing solution for improving the performance of data management and mining tasks is to take advantage of the computing power of distributed and parallel frameworks such as MapReduce[19] or Spark [78].

Since my PhD defense, I have worked on topics related to *large scale data management and analysis*. One of these topics is the distributed indexing and similarity search over large volume of time series [84, 85, 86, 87]. Parallel mining of highly informative items from big datasets [88, 89, 90] is another problem on which I worked. I have also worked on privacy preserving query processing in the cloud [91, 92, 93]. Many users and companies are interested to take advantage of the cloud computing power for managing their data. However, potentially sensitive data gets at risk of security attacks, *e.g.*, from employees of the cloud provider. This is why we need new solutions that are able to protect users privacy, particularly by data encryption, and answer the user queries over encrypted data. Another problem on which I contributed is load balancing for big data processing in parallel frameworks [94, 95, 96]. This is an important problem, because in order to efficiently answer queries over big data in frameworks such as MapReduce and Spark, we need to balance the load among the participating nodes, otherwise the performance of the parallel system may degrade significantly. I have also worked on top-k query processing in distributed systems [97, 98, 99]. The results of queries over big data may be huge, this is why top-k queries are needed to return only a small set of highly relevant results to the users. Another problem on which I contributed is replication in distributed systems [100, 101]. Replication is mandatory for the management of large volume of data in distributed systems, since without replication, we cannot guarantee the data availability.

In this HDR manuscript, I focus on my recent contributions on data analytics. Particularly, I present our parallel solutions developed to deal with the following problems:

1) similarity search over time series; 2) maximally informative k-itemsets mining. These problems are important for many applications that need to analyze large volumes of data. Below, I briefly introduce them.

Nowadays, individuals are able to monitor various indicators for their personal activities (*e.g.*, through smart-meters or smart-plugs for electricity or water consumption), or professional activities (*e.g.*, through the sensors installed on plants by farmers). This results in the production of large and complex data, usually in the form of time series [64, 60, 59], that challenge knowledge discovery. With such complex and massive sets of time series, fast and accurate *similarity search over time series* is a key to perform many data mining tasks like shapelets, motifs discovery, classification or clustering [64]. We have addressed this problem by taking advantage of parallel frameworks such as Spark.

The second problem is *maximally informative k-itemsets mining* (*miki* for short) that is one of the fundamental building bricks for exploring informative patterns in databases. Efficient *miki* mining has high impact on various tasks such as supervised learning [50], unsupervised learning [32] or information retrieval [37], to cite a few. A typical application is the discovery of discriminative sets of features, based on joint entropy [25], which allows distinguishing between different categories of objects. Indeed, with massive amounts of data, miki mining is very challenging, due to high number of entropy computations. An efficient *miki* mining solution should scale up with the increase in the size of the itemsets, calling for cutting edge parallel algorithms and high performance computation of *miki*.

In Section 1.1, I give an overview of my contributions to deal with these two problems. The details of the contributions are presented in Chapters 2 to 4. Let me point out that the research presented in this HDR manuscript was carried out jointly with colleagues, and two PhD students, which I co-supervised (see the context of each activity in its corresponding chapter).

My other contributions related to large scale data management and analysis are briefly described in Section 1.2.

1.1 Overview of Contributions Presented in Manuscript

1.1.1 Distributed iSAX index for Similarity Search over Time Series

In Chapter 2, we study the problem of similarity search over large scale time series datasets using parallel indexes. In order to improve the performance of similarity queries, index construction is one of the most popular techniques [28], which has been successfully used in a variety of settings and applications [29, 70, 9, 75, 17, 82].

Unfortunately, building an index, *e.g., iSAX* [69], over billions of time series by using traditional centralized approached is highly time consuming. A naive construction of the index in a parallel environment may lead to poor querying performances. We need to reach an ideal distribution of index construction and query processing in massively

distributed environments.

Chapter 2 describes DPiSAX [84, 87], a parallel solution, which we developed to construct iSAX-based index over billions of time series by making the most of the parallel environment by carefully distributing the work load. Our solution takes advantage of the computing power of distributed systems by using the Spark parallel framework [79]. We implemented our index construction solution, and evaluated its performance over large volumes of data (up to 4 billion time series of length 256, for a total volume of 6 Terabytes). Our experiments illustrate the performance of our index construction algorithm with an indexing time of less than 2 hours for more than 1 billion time series, while the baseline centralized algorithm needs more than 5 days.

In addition to the index construction, we developed a parallel query processing algorithm that, given a query, exploits the available nodes of the distributed system to answer the query in parallel by using the constructed parallel index. As illustrated by our experiments, and owing to our distributed querying strategy, our approach is able to process 10M queries in less than 140 seconds, while the state of the baseline algorithm needs almost 2300 seconds.

1.1.2 Parallel Discovery of Correlated Time Series Across Sliding Windows

In Chapter 3, we address the problem of finding highly correlated pairs of time series across sliding windows. This is an important problem in applications such as seismic sensor networks, financial trading, or communications network monitoring, to name a few.

To address this problem, we developed *ParCorr* [85, 86], an efficient parallel solution for continuous detection of similar time series across sliding windows. ParCorr uses the sketch principle [23] for representing the time series. It gives linear speedup over most of its steps and reduces the quadratic communication time by minimizing both the size and the number of messages. Our work includes the following contributions:

- A parallel approach for incremental computation of the sketches in sliding windows. This approach avoids the need for recomputing the sketches from scratch, after the modifications in the content of the sliding window.

- A partitioning approach that projects sketch vectors of time series into subvectors and builds a distributed grid structure for the subvectors in parallel. Each subvector projection can be processed in parallel.

- An efficient algorithm for parallel detection of correlated time series candidates from the distributed grids. In our algorithm, we minimize both the size and the number of messages needed for candidate detection.

1.1.3 Parallel Mining of Maximally Informative k-Itemsets in Big Data

In Chapter 4, we study the problem of maximally informative k-itemsets (*miki*) mining in massive datasets, where informativeness is expressed by means of joint entropy and k is the size of the itemset [36, 49, 81].

To address the *miki* mining problem, we developed a new parallel solution, namely Parallel Highly Informative K-itemSet (PHIKS in short) [88, 89, 90], that renders the discovery of *miki* from a very large database (up to Terabytes of data) simple and effective, using parallel frameworks such as MapReduce. It cleverly exploits available data at each computing node to efficiently calculate the joint entropies of *miki* candidates. For more efficiency, we provide PHIKS with optimizing techniques that allow for significant improvements of the whole process of *miki* mining. The first technique estimates the upper bound of a given set of candidates and allows for a high reduction of data communications, by filtering unpromising itemsets without having to perform any additional scan over the data. The second technique reduces significantly the number of scans over the local databases of computing nodes, *i.e.*, only one scan per step, by incrementally computing the joint entropy of candidate features.

PHIKS has been extensively evaluated using massive real-world datasets. Our experimental results show that PHIKS significantly outperforms alternative approaches, and confirm the effectiveness of our proposal over large databases containing for example one Terabyte of data.

1.2 Other Contributions

Below, I briefly present other activities, related to *large scale data management and analysis*, to which I have contributed since my PhD defense. I have co-supervised 4 PhD students in these activities.

- **Privacy preserving query processing in the cloud** [91, 92, 93]. In the context of Sakina Mahboubi's PhD thesis, we addressed the problem of privacy preserving top-k query processing in the cloud. We developed efficient solutions for processing these queries over encrypted data without decryption the data in the nodes of the cloud [92, 93]. In the context of a collaboration with the University of California at Santa Barbara and University of Rennes 1, we developed a *differential private index* for privacy preserving evaluation of range queries over encrypted data in the cloud [91].

- **Load balancing for query processing over big data** [94, 95, 96]. In the context of Miguel Liroz's PhD thesis, we dealt with the problem of *load balancing* for *big data processing* using MapReduce. We proposed efficient solutions for optimal data placement in the workers of MapReduce [96]. We also proposed a variant of MapReduce, called FP-Hadoop [95, 94], that automatically balances the load of reducers by adding a new intermediate phase to MapReduce.

- **Probabilistic data management and analysis** [102, 103, 104, 105]. In the context of Naser Ayat's PhD thesis, we dealt with the problem of *entity resolution* over probabilistic and distributed databases. In [104], we addressed the problem of *frequent itemset mining* over probabilistic data. In [105], we proposed an efficient solution for evaluating *aggregate queries*, particularly sum and count, over probabilistic data.

- **Top-k query processing in distributed systems** [97, 98, 106, 107, 108, 99, 109]. Another topic on which I have contributed is *top-k query processing*, particularly in distributed systems. For instance, in [97] we addressed the problem of profile diversification queries in large-scale recommendation systems, and proposed a *probabilistic diversification* approach that relies on top-k processing over inverted lists. In [109, 99], we dealth with the problem of top-k query processing in P2P systems, and proposed efficient solutions to return high quality top-k results to users as soon as possible. In [110, 111, 112], we addressed the problem of distributed processing of join queries in DHT (distributed hash table) networks.

  Replication in P2P systems [100, 101] In the context of Mounir Tlili's PhD thesis, we addressed the problem of optimistic replication for collaborative text editing in Peer-to-Peer (P2P) systems. We proposed a P2P logging and timestamping service, called P2P-LTR (P2P Logging and Timestamping for Reconciliation) that exploits a DHT (distributed hash table) for reconciliation. P2P-LTR is based on a service developed during my PhD thesis, namely Key-based Timestamping Service (KTS) [113], designed to generate distributed timestamps in DHTs using local counters in the nodes, and without depending on the participant clocks.

Chapter 2

Parallel Time Series Indexing and Querying using iSAX Representation

2.1 Introduction

In this chapter, we address the problem of similarity search in massive sets of time series by means of scalable index construction. Unfortunately, making an index over billions of time series by using traditional centralized approached is highly time consuming. Moreover, a naive construction of the index on the parallel environment may lead to poor querying performances. This is illustrated in Figure 2.1 where the time series dataset is naively split on the W distributed nodes (Figure 2.1). In this case, a batch of queries B has to be duplicated and sequentially processed on each node. By means of a dedicated strategy where each query in B could be oriented to the right partition (*i.e.*, the partition that must correspond to the query), the querying work load can be significantly reduced (Figure 2.2 shows an ideal case where B is split in W subsets and really processed in parallel).

Our objective is to reach such an ideal distribution of index construction and query processing in massively distributed environments. To attain this objective, we developed DPiSAX [84, 87], a parallel solution, which we developed to construct iSAX-based index over billions of time series by making the most of the parallel environment by carefully

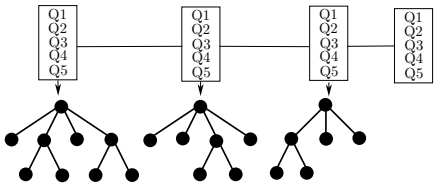


Figure 2.1 – Straightforward implementation: the batch of queries is duplicated on all the computing nodes

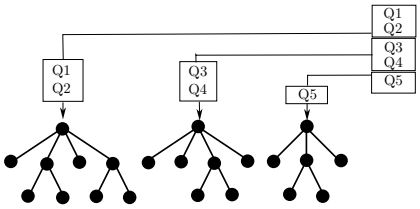


Figure 2.2 – Ideal distribution of time series in the index nodes: each query is sent only to the relevant partition

distributing the work load. Our solution takes advantage of the computing power of distributed systems by using parallel frameworks such as MapReduce or Spark [79]. Our contributions are as follows:

- A *parallel index construction algorithm* that takes advantage of distributed environments to efficiently build iSAX-based indices over very large volumes of time series. We implemented our index construction algorithm, and evaluated its performance over large volumes of data (up to 4 billion time series of length 256, for a total volume of 6 Terabytes). Our experiments illustrate the performance of our algorithm with an indexing time of less than 2 hours for more than 1 billion time series, while the state of the art centralized algorithm needs more than 5 days.
- A *parallel query processing algorithm* that, given a query, exploits the available processors of the distributed system to answer the query in parallel by using the constructed parallel index. As illustrated by our experiments, and owing to our distributed querying strategy, our approach is able to process 10M queries in less than 140 seconds, while the state of the art centralized algorithm needs almost 2300 seconds.

The rest of the chapter is organized as follows. In Section 2.2, we define the problem we address. In Section 2.3, we describe the details of our parallel index construction and query processing algorithm. In Section 2.4, we present a detailed experimental evaluation to verify the effectiveness of our approach. In Section 2.5, we discuss the related work. We conclude in 2.7.

2.2 Problem Definition and Background

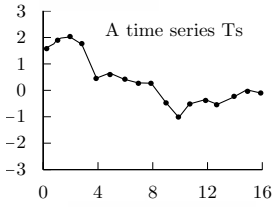
A time series X is a sequence of values $X = \{x_1, ..., x_n\}$. We assume that every time series has a value at every timestamp $t = 1, 2, ..., n$. The length of X is denoted by $|X|$. Figure 2.3 shows a time series of length 16, which will be used as running example throughout this chapter.

2.2.1 iSAX Representation

Given two time series $X = \{x_1, ..., x_n\}$ and $Y = \{y_1, ..., y_m\}$ such that $n = m$, the Euclidean distance between X and Y is defined as [29]: $ED(X, Y) = \sqrt{\sum_n^{i=1}(x_i - y_i)^2}$.

The Euclidean distance is one of the most straightforward similarity measurement methods used in time series analysis. In this work, we use it as the distance measure.

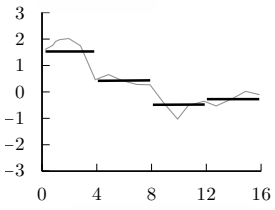
For very large time series databases, it is important to estimate the distance between two time series very quickly. There are several techniques, providing lower bounds by segmenting time series. Here, we use a popular method, called indexable Symbolic Aggregate approXimation (iSAX) representation [69, 70]. The iSAX representation will be used to represent time series in our index.



0.23

(a) b

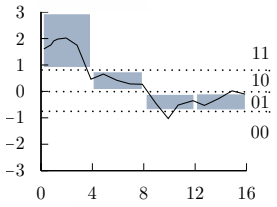
Figure 2.3 – A time series X of length 16



0.23

(a) b

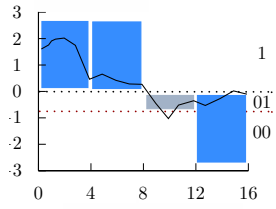
Figure 2.4 – PAA representation of X, with 4 segments



0.23

(a) b

Figure 2.5 – SAX representation of X, with 4 segments and cardinality 4, $[11, 10, 01, 01]$.



0.23

(a) b

Figure 2.6 – iSAX representation of X, with 4 segments and different cardinalities $[1_2, 1_2, 01_4, 0_2]$.

Figure 2.7 – A time series X is discretized by obtaining a PAA representation and then using predetermined break-points to map the PAA coefficients into SAX symbols

The iSAX representation extends the SAX representation [54]. This latter representation is based on the Piecewise Aggregate Approximation (PAA) representation [53] that allows for dimensionality reduction while providing the important lower bounding property as we will show later. The idea of PAA is to have a fixed segment size, and minimize dimensionality by using the mean values on each segment. Example 1 gives an illustration of PAA.

Example 1. *Figure 2.4 shows the PAA representation of X, the time series of Figure 2.3. The representation is composed of $w = |X|/l$ values, where l is the segment size. For each segment, the set of values is replaced with their mean. The length of the final representation w is the number of segments (and, usually, $w << |X|$).*

The SAX representation takes as input the reduced time series obtained using PAA. It discretizes this representation into a predefined set of symbols, with a given cardinality, where a symbol is a binary number. Example 2 gives an illustration of the SAX representation.

Example 2. *In Figure 2.5, we have converted the time series X to SAX representation with size 4, and cardinality 4 using the PAA representation shown in Figure 2.4. We denote SAX(X) = [11, 10, 01, 01].*

The iSAX representation uses a variable cardinality for each symbol of SAX representation, each symbol is accompanied by a number that denotes its cardinality. We defined the iSAX representation of time series X as $iSAX(X)$ and we call it the iSAX word of the time series X. For example, the iSAX word shown in Figure 2.6 can be written as $iSAX(X) = [1_2, 1_2, 01_4, 0_2]$.

Using a variable cardinality allows the iSAX representation to be indexable. We can build a tree index as follows. Given a cardinality b, an iSAX word length w and leaf capacity th, we produce a set of b^w children for the root node, insert the time series to their corresponding leaf, and gradually split the leaves by increasing the cardinality by one character if the number of time series in a leaf node rises above the given threshold th.

Note that previous studies have shown that the iSAX index is robust with respect to the choice of parameters (word length, cardinality, leaf threshold) [70, 17, 83]. Moreover, it can also be used to answer queries with the Dynamic Time Warping (DTW) distance, through the use of the corresponding lower bounding envelope [47].

2.2.2 Similarity Queries

The problem of similarity queries is one of the main problems in time series analysis and mining. In information retrieval, finding the k nearest neighbors (k-NN) of a query is a fundamental problem. In this section, we define k nearest neighbors based queries.

Definition 1. (APPROXIMATE k NEAREST NEIGHBORS) *Given a set of time series D, a query time series Q, and $\epsilon > 0$. We say that $R = ApkNN(Q, D)$ is the approximate k nearest neighbors of Q from D, if $ED(a, Q) \leq (1 + \epsilon)ED(b, Q)$, where a is the k^{th} nearest neighbor from R and b is the true k^{th} nearest neighbor.*

2.2.3 Spark

For implementing our parallel algorithm, we use Spark [79], which is a parallel programming framework to efficiently process large datasets. This programming model can perform analytics with in-memory techniques to overcome disk bottlenecks. Unlike traditional in-memory systems, the main feature of Spark is its distributed memory abstraction, called resilient distributed datasets (*RDD*), that is an efficient and fault-tolerant abstraction for distributing data in a cluster. With RDD, the data can be easily persisted in main memory as well as on the hard drive. Spark is designed to support the execution of iterative algorithms.

2.2.4 Problem Definition

The problem we address is as follows. Given a (potentially huge) set of time series, find the results of approximate k-NN queries as presented in definition 1, by means of an index and query processing performed in parallel.

2.3 Distributed Partitioned iSAX

In this section, we present a novel parallel partitioned index construction algorithm, called *DPiSAX*, along with very fast parallel query processing techniques.

Our approach is based on a sampling phase that allows anticipating the distribution of time series among the computing nodes. Such anticipation is mandatory for an efficient query processing, since it will allow, later on, to decide what partition contains the time series that actually correspond to the query. To do so, we first extract a sample from the time series dataset, and analyze it in order to decide how to distribute the time series in the splits, according to their iSAX representation.

2.3.1 Sampling

In Distributed Partitioned iSAX, our index construction combines two main phases which are executed one after the other. First, the algorithm starts by sampling the time series dataset and creates a partitioning table. Then, the time series are partitioned into groups using the partitioning table. Finally, each group is processed to create an iSAX index for each partition.

More formally, our sampling is done as follows. Given a number of partitions P and a time series dataset D, the algorithm takes S sample time series of size L from D using stratified sampling, and distributes them among the W available workers. Each worker takes S/W time series and emits its iSAX words $SWs = \{iSAX(ts_i), i = 1, ..., L\}$. The master collects all the workers' iSAX words and performs the partitioning algorithm accordingly. In the following, we describe the partitioning method that enables separating the dataset into non-overlapping subsets based on iSAX representations.

Table 2.1 – A sample S of 8 time series converted to iSAX representations with iSAX words of length 2

Time series	iSAX words	Time series	iSAX words
TS_1	$\{01, 00\}$	TS_5	$\{00, 10\}$
TS_2	$\{00, 01\}$	TS_6	$\{01, 11\}$
TS_3	$\{01, 01\}$	TS_7	$\{10, 00\}$
TS_4	$\{00, 00\}$	TS_8	$\{10, 01\}$

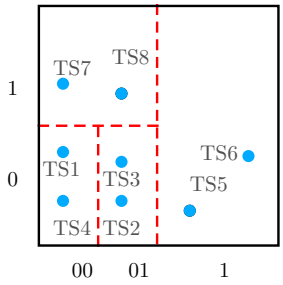


Figure 2.8 – Partitioning according to DPiSAX.

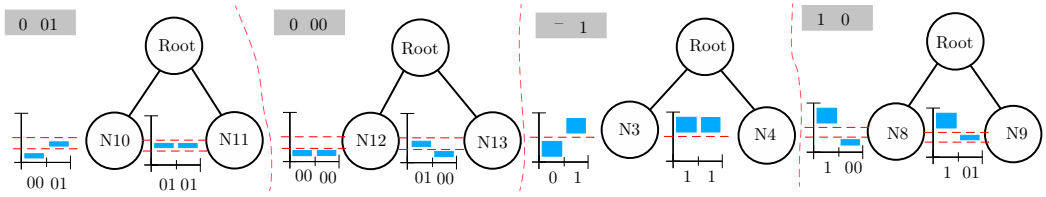


Figure 2.9 – DPiSAX indexes after partitioning and indexing. The partitioning principle of DPiSAX allows better balance.

Figure 2.10 – The result of the partitioning algorithm on sample S (from Table 2.1) into four partitions.

2.3.2 Partitioning Algorithm

Here, our partitioning paradigm considers the splitting power of each bit in the iSAX symbols, before actually splitting the partition. As in the basic approach, the biggest partition is considered for splitting at each step of the partitioning process. The main difference is that we don't use the first bit of the n^{th} symbol for splitting the partition. Instead, we look for all bits (whatever the symbol) with the highest probability to equally distribute the time series of the partition among the two new sub-partitions that will be created. To this effect, we compute for each segment the $\mu \pm \sigma$ interval, where μ is the mean and σ is the standard deviation, and we examine for each segment if the break-point of the additional bit (*i.e.*, the bit used to generate the two new partitions) lies within the interval $\mu \pm \sigma$. From the segments for which this is true, we choose the one having μ closer to the break-point.

In order to illustrate this, let us consider the blue boxes of the diagrams in Figure 2.9. We choose the biggest blue box that ensures the best splitting by considering the next break-point.

Example 3. *Let's consider Table 2.1, where we use iSAX words of length two to represent the time series of a sample S. Suppose that we need to generate four partitions. To generate four partitions, we compute the $\mu \pm \sigma$ interval for the first segment and the second segment, and choose the first bit of the second segment to define two partitions. The first partition contains all the time series having their second segment in iSAX word starting with 0, and the second partition contains the time series having their second segment in iSAX word starting with 1. We obtain two partitions: "0" and "1". The biggest partition is "0" (i.e., the one containing time series $TS1$ to $TS4$, $TS7$ and $TS8$). We compute the $\mu \pm \sigma$ interval for all segments over all the time series in this partition. Then, the partition is split again, according to the first bit of the first symbol. We now have the following partitions: from the first step, partition "1", and from the second step, partitions "00", and "10". Now, partition "00" is the biggest one. This partition is split for the third time, according to the second bit of the first symbol and we obtain four partitions. Figure 2.8 shows the obtained partitions and Figure 2.9 shows the indexes obtained with these partitions.*

2.3.3 Index Construction

DPiSAX, our parallel index construction, sequentially splits the dataset for distribution into partitions. Then each worker builds an independent iSAX index on its partition, with the iSAX representations having the highest possible cardinalities. Representing each time series with iSAX words of high cardinalities allows us to decide later what cardinality is really needed, by navigating "on the fly" between cardinalities. The word of lower cardinality being obtained by removing the trailing bits of each symbol in the word of higher cardinality. The output of this phase, with a cluster of W nodes, is a set of W iSAX indexes built on each split.

Table 2.2 – Default parameters

Parameters	Value	Parameters	Value
iSAX word length	8	Leaf capacity	1,000
Basic cardinality	2	Number of machines	32
Maximum cardinality	512	Sampling fraction	10%

2.3.4 Query Processing

Given a collection of queries Q, in the form of time series, and the index constructed in the previous section for a database D, we consider the problem of finding time series that are similar to Q in D, as presented in definition 1.

Given a batch B of queries, the master node identifies the right partition where the index is stored and sends the corresponding query by using its iSAX words. Then, each query is sent to the partition that has the same iSAX word as the query. Each worker uses its local index to retrieve time series that correspond to each query $Q \in B$, according to the approximate k-NN criteria. On each local index, the approximate search is done by traversing the local index to the terminal node that has the same iSAX representation as the query. The target terminal node contains at least one and at most th iSAX words, where th is the leaf threshold. A main memory sequential scan over these iSAX words is performed in order to obtain the k nearest neighbors using the Euclidean distance.

2.4 Performance Evaluation

In this section, we report experimental results that show the performance of DPiSAX for indexing time series.

The parallel experimental evaluation was conducted on a cluster of 32 machines, each operated by Linux, with 64 Gigabytes of main memory, Intel Xeon CPU with 8 cores and 250 Gigabytes hard disk. The iSAX2+ approach was executed on a single machine with the same characteristics.

We compare our solution to two state-of-the-art baselines: the most efficient centralized version of iSAX index (*i.e.*, iSAX2+ [17]), and Parallel Linear Search (PLS), which is a parallel version of the UCR Suite fast sequential search (with all applicable optimizations in our context: no computation of square root, and early abandoning) [64].

The presentation of our experiments is divided into two sections. In Section 2.4.2, we measure the index construction times with different parameters. In Section 2.4.3, we focus on the query performance of our approach.

Reproductibility: we implemented our approach on top of Apache-Spark [79], using the Java programming language. The iSAX2+ index is also implemented with Java [1].

[1] Our code is available at `http://djameledine-yagoubi.info/projects/DPiSAXShort/`.

2.4.1 Datasets and Settings

We carried out our experiments on two real world and synthetic datasets, up to 6 Terabytes and 4 billion series. The first real world data represents seismic time series collected from the IRIS Seismic Data Access repository [2]. After preprocessing, it contains 40 millions time series of 256 values, for a total size of 150Gb. The second real world data is the TexMex corpus [45]. It contains 1 Billion time series (SIFT feature vectors) of 128 points each (derived from 1 Billion images). Our synthetic datasets are generated using a Random Walk principle, each data series consisting of 256 points. At each time point the generator draws a random number from a Gaussian distribution N(0,1), then adds the value of the last number to the new number. This type of generator has been widely used in the past. [29, 9, 69, 16, 17, 82]. Table 2.2 shows the default parameters (unless otherwise specified in the text) used for each approach. The iSAX word length, PAA size, leaf capacity, basic cardinality, and maximum cardinality were chosen to be optimal for iSAX, which previous works [69, 70, 16, 17, 82] have shown to work well across data with very different characteristics.

2.4.2 Index Construction Time

In this section, we measure the index construction time in DPiSAX and compare it to the construction time of the iSAX2+ index.

Figure 2.11 reports the index construction times for all approaches on our Random Walk dataset. The index construction time increases with the number of time series for all approaches. This time is much lower in the case of DPiSAX, than that of the centralized iSAX2+. On 32 machines, and for a dataset of one billion time series, DPiSAX builds the index in 65 minutes , while the iSAX2+ index is built in more than 5 days on a single node.

Figure 2.12 illustrates the parallel speed-up of our approach on the Random Walk dataset. The results show a near optimal gain for DPiSAX.

Figure 2.13 reports the performance gains of our parallel approach when compared to the centralized version of iSAX2+ on our synthetic and real datasets. The results show that DPiSAX is 40-120 times faster than iSAX2+. We observe that the performance gain depends on the dataset size in relation to the number of Spark nodes used in the deployment. Note that the time Spark needs to deploy on 32 nodes is accounted for in our measurements. Thus, given the very short time needed to construct the DPiSAX index on the seismic dataset (420 seconds), the proportion of the time taken by the Spark deployment, when compared to index construction, is higher than for the much larger Random Walk dataset.

[2] `http://ds.iris.edu/data/access/`

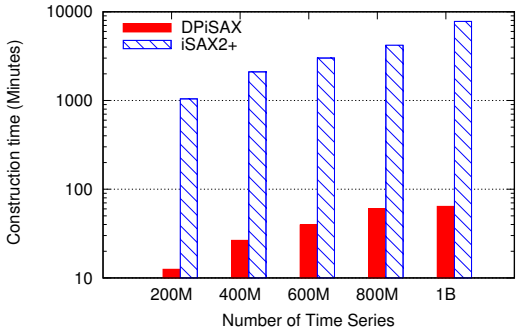


Figure 2.11 – Logarithmic scale. Construction time as a function of dataset size. DPiSAX is run on a cluster of 32 nodes. iSAX2+ is run on a single node. With 1 billion Random Walk TS, iSAX2+ needs 5 days and our distributed algorithm needs less than 2 hours.

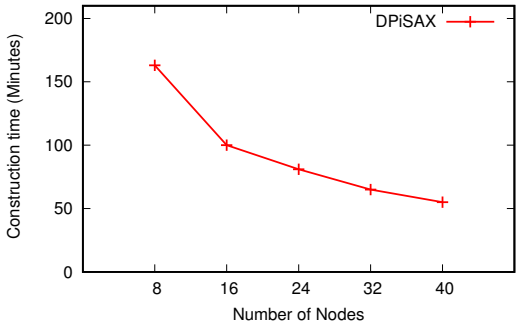


Figure 2.12 – Construction time as a function of cluster size. DPiSAX has a near optimal parallel speed-up. With 1 billion TS from the Random Walk dataset.

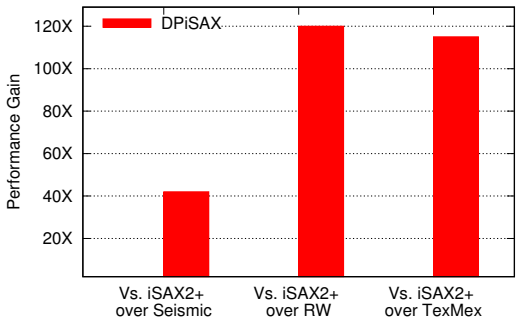


Figure 2.13 – Performance gain on iSAX2+ in construction time, over seismic (40 millions TS), Random Walk (RW, 1 billion TS) and TexMex (1 billion TS), with a cluster of 32 nodes.

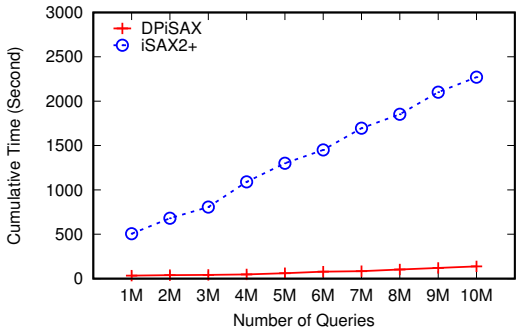


Figure 2.14 – Cumulative query answering time (Approximate 10-NN). DPiSAX on a cluster of 32 nodes, iSAX2+ on a single node.

2.4.3 Query Performance

We evaluate the querying performance of our algorithm, and compare it to that of iSAX2+. We use our synthetic data, and generate Random Walk queries with the same distribution as described in Section 2.4.1.

Figure 2.14 compares the cumulative query answering time of our parallel approach to that of iSAX2+, for answering approximate k nearest neighbors queries with a varying size of query batch. We observe that the time performance of DPiSAX is better than that of the iSAX2+ by a factor of up to 16. Note that the total time to answer 10 millions queries is 2270 sec for iSAX2+ and only 138 sec for DPiSAX.

2.5 Related Work

In the context of time series data mining, several techniques have been developed and applied to time series data, *e.g.*, clustering, classification, outlier detection, pattern identification, motif discovery, and others. The idea of indexing time series is relevant to all these techniques. Note that, even though several databases have been developed for the management of time series (such as Informix Time Series[3], InfluxDB[4], OpenTSDB[5], and DalmatinerDB[6] based on RIAK), they do not include similarity search indexes, focusing on (temporal) SQL-like query workloads. Thus, they cannot efficiently support similarity search queries, which is the focus of our study.

In order to speed up similarity search, different works have studied the problem of indexing time series datasets, such as Indexable Symbolic Aggregate approXimation (iSAX) [69, 70], iSAX 2.0 [16, 17], iSAX2+ [17], Adaptive Data Series Index (ADS Index) [82] and Dynamic Splitting Tree (DSTree) [75]. The iSAX index family (iSAX 2.0, iSAX2+, ADS Index) is based on SAX representation [54] of time series, which is a

[3]https://www.ibm.com/developerworks/topics/timeseries
[4]https://influxdata.com/
[5]http://opentsdb.net/
[6]https://dalmatiner.io/

symbolic representation for time series that segments all time series into equi-length segments and symbolizes the mean value of each segment. As an index structure specifically designed for ultra-large collections of time series, iSAX 2.0 proposes a new mechanism and also algorithms for efficient bulk loading and node splitting policy, wich is not supported by iSAX index. In [17], the authors propose two extensions of iSAX 2.0, namely iSAX 2.0 Clustered and iSAX2+. These extensions focus on the efficient handling of the raw time series data during the bulk loading process, by using a technique that uses main memory buffers to group and route similar time series together down the tree, performing the insertion in a lazy manner. In addition to that, DSTree based on extension of APCA representation, called EAPCA [75] segments time series into variable length segment. Unlike iSAX which only supports horizontal splitting, and only the mean values can be used in splitting, the DSTree uses multiple splitting strategies. All these indexes have been developed for a centralized environment, and cannot scale up to very high volumes of time series.

In this work, we proposed a parallel solution that takes advantage of distributed environments to efficiently build iSAX-based indices over billions of time series, and to query them in parallel with very small running times. To the best of our knowledge, this is the first chapter that proposes such a solution.

2.6 Context of the work

The work on DPiSAX has been done in the context of Djamel-Edine Yagoubi' PhD thesis co-supervised by me and Florent Masseglia from INRIA Zenith team and Themis Palpanas from Université Paris-Descartes.

2.7 Conclusion

In this chapter, I described DPiSAX, our efficient parallel solution to index and query billions of time series. We evaluated the performance of our solution over large volumes of real world and synthetic datasets (up to 4 billion time series, for a total volume of 6TBs). The experimental results illustrate the excellent performance of DPiSAX (*e.g.*, an indexing time of less than 2 hours for more than one billion time series, while the state of the art centralized algorithm needs several days). The results also show that the distributed querying algorithm of DPiSAX is able to process millions of similarity queries over collections of billions of time series with very fast execution times (*e.g.*, 140s for 10M queries), thanks to our load balancing mechanism. Overall, the experimental results show that by using our parallel technique, the indexing and mining of very large volumes of time series can now be done in very small execution times, which are impossible to achieve using traditional centralized approaches.

Chapter 3

Parallel Method to Identify Similar Time Series Across Sliding Windows

In this chapter, we study the problem of finding highly correlated pairs of time series across multiple sliding windows. Doing this efficiently and in parallel could help in applications such as sensor fusion, financial trading, or communications network monitoring, to name a few.

An easy-to-understand motivating use case for finding sliding windows correlation comes from finance. In that application, the time series consist of prices of trades of different stocks. The problem is to find pairs of stocks whose return profiles look similar over the most recent time period (typically, a few seconds). A pair of time series (*e.g.*, Google and Apple prices) that were similar before and have since diverged, where say Google went up more than Apple, might present a trading opportunity: sell the one that has gone up relative to the other and buy the other one. The return profile is based on the weighted average price (by volume) of the stock over time t (perhaps discretized in milliseconds), denoted wprice(t). The return at t is the fractional change, $(wprice(t) - wprice(t-1))/wprice(t-1)$.

While prices are stable over time (*e.g.*, a stock whose price is 100 will tend to stay around 100), the returns resemble white noise. We call such time series "uncooperative", because standard dimensionality reduction techniques such as Fourier or Wavelet Transforms either sacrifice too much accuracy or reduce the dimensionality too little. Random sketch-based methods and some other explicit encoding methods work well for both cooperative and uncooperative time series. Moreover, the sketch-based methods work nearly as well as Fourier/Wavelet methods for cooperative time series. So, for the sake of generality, this chapter uses the sketch method of [23], and compares the result with the state-of-the-art explicit encoding method iSax [18].

The need for speed comes from increasing scale and the advantage of reacting quickly. An irony of improving technology is that sensor speeds and numbers increase vastly faster than computational speed. For this reason, linear or near linear-time algorithms become increasingly vital to give timely responses in the face of the flood of data. In most applications, speed turns out to be of greater importance than completeness, so a minor loss

in recall is often acceptable as long as precision is high. In trading, for example, there is only a fictitious monetary loss in missing an opportunity, but the opportunities a system reports should be real and must be timely to be actionable.

Another motivating example is the sensor fusion for earth science. Correlations of distant sensors in seismic data may indicate a large scale event. Consider, for example, a set of sensors spaced over several possible earthquake zones. Temporal correlations of pairs of sensors over a time window may suggest that these pairs are responding to the same seismic cause. Missing some correlations is acceptable, because a major event will reveal many correlations so a recall of 90% or more is quite enough.

We addressed the problem of finding correlated pairs of time series across sliding windows by developing ParCorr [85, 86], an efficient parallel solution that uses the sketch principle for representing the time series. Our solution includes the following contributions:

- A parallel approach for incremental computation of the sketches in sliding windows. This approach avoids the need for recomputing the sketches from scratch, after the modifications in the content of the sliding window.

- A partitioning approach that projects sketch vectors of time series into subvectors and builds a distributed grid structure for the subvectors in parallel. Each subvector projection can be processed in parallel.

- An efficient algorithm for parallel detection of correlated time series candidates from the distributed grids. In our algorithm, we minimize both the size and the number of messages needed for candidate detection.

The rest of this chapter is organized as follows. Section 3.1 defines the problem. The details of ParCorr are presented in Section 3.2. In Section 3.3, we present a performance evaluation of ParCorr through experiments in a distributed environment using real and synthetics datasets. Section 3.5 presents the context of this work, and Section 3.6 concludes.

3.1 Problem Definition

A streaming time series is a potentially unending series of values in time order. A data stream, for our purposes, is a set of streaming time series. They are normalized to have zero mean and unit standard deviation. Correlation over windows from the same or different series has many variants. This work focuses on the synchronous variation, defined as follows:

Given a data stream of N_s streaming time series, a start time p_s, and a window size w, find, for each time window W of size w, all pairs of streaming time series ts_1 and ts_2 such that ts_1 during time window W is highly correlated (over 0.7 typically) with ts_2 during the same time window.

Euclidean distance is the target metric of the state of the art iSAX algorithm. In addition, Euclidean distance is related to Pearson correlation as follows:

$$D^2(\hat{x}, \hat{y}) \quad = \quad 2 \times m \times (1 - corr(x, y)) \tag{3.1}$$

Here $\hat{x}$ and $\hat{y}$ are obtained from the raw time series by computing $\hat{x} = \frac{x - avg(x)}{\sigma_x}$, where $\sigma_x = \sqrt{\sum_{i=1}^{m}(x_i - avg(x))^2}$. m is the length of the time series. So, we offer parallel algorithms for both sliding window Euclidean and correlation metrics in this work.

3.2 Algorithmic Approach

Following [23], our basic approach to find similar pairs of sliding windows in time series (whether Euclidean distance or Pearson correlation, for starters) is to compute the dot product of each normalized time series over a window size w with a set of random vectors. That is, for each time series t_i and window size w and time period $k..(k + w - 1)$, we compute the dot product of $t_i[k..k+w-1]$ with r random $(-1/+1)$ vectors of size w. The r dot products thus computed constitute the "sketch" of t_i at time period $k..(k + w - 1)$. Next we compare the sketches of the various time series to see which ones are close in sketch space (if $w >> r$, which is often the case, this is cheaper than working directly on the time series) and then identify those close ones.

The theoretical underpinning of the use of sketches is given by the Johnson-Lindenstrauss lemma [46].

Lemma 1. *Given a collection C of m time series with length n, for any two time series $\overrightarrow{x}, \overrightarrow{y} \in C$, if $\epsilon < 1/2$ and $n = \frac{9logm}{\epsilon^2}$, then*

$$(1 - \epsilon) \leq \frac{\| \overrightarrow{s}(\overrightarrow{x}) - \overrightarrow{s}(\overrightarrow{y}) \|^2}{\| \overrightarrow{x} - \overrightarrow{y} \|^2} \leq (1 + \epsilon)$$

holds with probability $1/2$, where $\overrightarrow{s}(\overrightarrow{x})$ is the Gaussian sketch of $\overrightarrow{x}$ of at least n dimensions.

The Johnson-Lindenstrauss lemma implies that the distance $\|\mathbf{sketch(t_i)} - \mathbf{sketch(t_j)}\|$ is a good appproximation of $\|\mathbf{t_i} - \mathbf{t_j}\|$. Specifically, if $\|\mathbf{sketch(t_i)} - \mathbf{sketch(t_j)}\| < \|\mathbf{sketch(t_k)} - \mathbf{sketch(t_m)}\|$, then it's very likely that $\|\mathbf{t_i} - \mathbf{t_j}\| < \|\mathbf{t_k} - \mathbf{t_m}\|$.

The sketch approach, as developed by Kushilevitz et al. [51], Indyk et al. [44], and Achlioptas [5] makes use of these guarantees. Note that the sketch approach is closely related to Locality Sensitive Hashing [35], by which similar items are hashed to the same buckets with high probability. In particular, the sketch approach is very similar in spirit to SimHash [22], in which the vectors of data items are hashed based on their angles with random vectors. The major contribution of our work consists of combining an incremental strategy with a parallel mixing algorithm and an efficient communication strategy.

3.2.1 The case of sliding windows

In the case of sliding windows, we want to find the most similar time series pairs at jumps of a basic window b, *e.g.*, for windows in time ranges 0 to $w-1$ seconds, b to $b+w-1$ seconds, $2b..2b+w-1$, ... where $b << w$.

There are two main challenges:

1. If we compute the sketches from scratch at each basic window, we are doing some redundant computation. We call that the naive method. Instead, we want to compute the sketches incrementally and in parallel.

2. When scaling this to a parallel system, we want to reduce communication costs as much as possible. We need to develop good strategies for this as communication is quadratic in the number of execution nodes, so constant coefficients matter.

3.2.2 Parallel incremental computation of sketches

To explain the incremental algorithm consider the example of Figure 3.1. The sketch for random vector v_1 is the dot product of the time series with v_1, *i.e.*, $1 \times (-1) + 2 \times 1 + ... + 4 \times (-1) = 4$.

Now if the basic window is of size 2, as illustrated by the "outdated" and "incoming" boxes of Figure 3.1, then we add the next two points of the time series (in this case having values $(2, 1)$ and generate two more random +/- 1 numbers for each random vector, in this case $(-1, -1)$ for v_1 and $(-1, 1)$ for v_2. To update the dot product for v_1 we subtract the contribution of the oldest two time points, viz. $1 \times (-1) + 2 \times 1 = 1$, and add in the contribution of $2 \times (-1) + (1 \times -1) = -3$ yielding a new sketch entry of $4 - 1 + (-3) = 0$. That illustrates the idea of incremental updating.

In general, the algorithm proceeds as following:

1. Partition time series among parallel sites. Replicate r random +1/-1 vectors each of size w to all sites. These random vectors will later be updated in a replicated fashion.

2. For each site,

   (a) Initially, take the first w data points of each time series at that site and form the dot product with all r random vectors. So each time series t will be represented by r dot products. They constitute $sketch(t)$.

   (b) while data comes in, when data for the i_{th} basic window of size b appears for all time series, extend each random vector by a new random +1/-1 vector of size b. Then for each time series t and random vector v, update the dot product of t with v by subtracting $v[0..b-1] \cdot t[(i-1)b - w...ib - w - 1]$ and adding $v[w..w+b-1] \cdot t[(i-1)b..ib-1]$. Change the $sketch(t)$ with all the updated dot products.

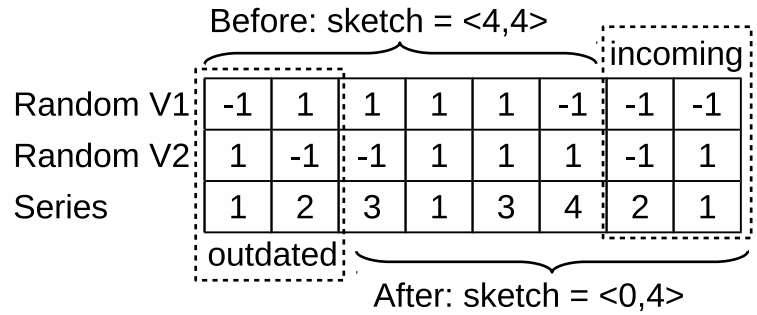


Figure 3.1 – A streaming time series, two random vectors, and the sketches that correspond to their dot product before and after the update on the data stream. The first sketch of the time series is computed on the six first values, and the second sketch is computed on the six last values. After the update, the "outdated" values are removed and the "incoming" ones are added to the streaming time series, so the work is proportional to the size of the basic window rather than the full window.

This step has time complexity proportional to the number of time series $\times$ size of basic windows $\times$ the number of random vectors. It is perfectly parallelizable.

Step 1 calls for parallel updates of the local random vectors on each site. It is mandatory that all the sites share the same random vectors. A possible approach would be for the master node, after the completion of each new sliding window, to generate new vectors of +/- 1 having the basic window size, and send them to the sites. This takes little time but is awkward to do in Spark. Our approach is therefore to generate and send oversized random vectors (say, twice the size of the sliding window) at setup time. A site then just has to loop inside the (oversized) random vector, simulating an endless source of +/- 1 values that are the same for all the sites.

3.2.3 Parallel mixing

Once the sketch vectors have been constructed incrementally, the next step is to find sketch vector pairs that are close to one another. Such pairs might then indicate that their corresponding time series are highly correlated (or similar based on some other distance metric).

Multi-dimensional search structures do not work well for more than four dimensions in practice [68]. For this reason, as indicated in the following example, we adopt a framework that partitions each sketch vector into subvectors and builds grid structures for the subvectors.

We first explain how this works by example and then show the pseudo-code.

Example 1. Suppose we have seven time series with sketch values as shown in Table

3.1.

Table 3.1 – A sample S of 7 time series with sketch values of length 6

time series	**sketch values**
sketch(ts_1)	(11, 12, 23, 24, 15, 16)
sketch(ts_2)	(11, 12, 13, 14, 15, 16)
sketch(ts_3)	(21, 22, 13, 14, 25, 26)
sketch(ts_4)	(21, 22, 13, 14, 25, 26)
sketch(ts_5)	(11, 12, 33, 34, 25, 26)
sketch(ts_6)	(31, 32, 33, 34, 15, 16)
sketch(ts_7)	(21, 22, 33, 34, 15, 16)

First, we partition these into pairs and send the values [0 1] of each sketch vector to site 1 (Table 3.2) where this will be formed into a grid (and the time series identifiers will be placed in cells (i,j), e.g., (31,32)). Analogously, we send partitions [2 3] and [4 5] of each sketch vector to sites 2 and 3 respectively, where the second and third grids will be formed. In the first grid, ts_1, ts_2, and ts_5 map to the same grid cell; ts6 is by itself; and ts_3, ts_4, and ts_7 all map to the same cell (Table 3.3). Thus, in grid 1 we have three partitions of time series identifiers. If two time series are in the same partition, then they are candidates for similarity.

Table 3.2 – Step 1 of the algorithm: sketch partitioning. Each sketch vector is partitioned into three pairs. The ith pair of the sketch vector for each time series s goes to a grid i. The values of the ith pair determine where in that grid the identifer s is placed.

	sketch subvectors		
	[0 1]	**[2 3]**	**[4 5]**
sketch(ts_1)	(11, 12)	(23, 24)	(15, 16)
sketch(ts_2)	(11, 12)	(13, 14)	(15, 16)
sketch(ts_3)	(21, 22)	(13, 14)	(25, 26)
sketch(ts_4)	(21, 22)	(13, 14)	(25, 26)
sketch(ts_5)	(11, 12)	(33, 34)	(25, 26)
sketch(ts_6)	(31, 32)	(33, 34)	(15, 16)
sketch(ts_7)	(21, 22)	(33, 34)	(15, 16)
	assigned to grid / at site		
	1	**2**	**3**

Now, we construct a mapping ts_to_node that maps time series identifiers to nodes (for now, think of each node as a single computational site, but one could imagine placing many nodes on a single site or spreading a node among many sites). For this example, let us say ts_to_node is the identity function. So we send the relevant parts of the partition ts_1, ts_2, ts_5 to nodes 1, 2, and 5. Similarly, we send the relevant parts of ts_3, ts_4, and ts_7 to nodes 3, 4, and 7. And so on (Table 3.4). Assuming the "opt" communication strategy (see

Table 3.3 – Step 2 of the algorithm: grid construction. Time series placed in the same grid cells are grouped in partitions.

grid	cell	time series IDs
1	(11, 12)	ts_1, ts_2, ts_5
	(21, 22)	ts_3, ts_4, ts_7
	(31, 32)	ts_6
2	(13, 14)	ts_2, ts_3, ts_4
	(23, 24)	ts_1
	(33, 34)	ts_5, ts_6, ts_7
3	(15, 16)	ts_1, ts_2, ts_6, ts_7
	(25, 26)	ts_3, ts_4, ts_5

Table 3.4 – Steps 4 and 5 of the algorithm: finding frequently collocated pairs (in the example, at least 2 out of 3 grids).

node	TS clusters	candidate pairs f $\geq$ 2/3
ts_to_node(ts_1)	ts_1, ts_2, ts_5 ts_1, ts_2, ts_6, ts_7	ts_1, ts_2
ts_to_node(ts_2)	ts_2, ts_5 ts_2, ts_3, ts_4 ts_2, ts_6, ts_7	
ts_to_node(ts_3)	ts_3, ts_4, ts_7 ts_3, ts_4 ts_3, ts_4, ts_5	ts_3, ts_4
ts_to_node(ts_4)	ts_4, ts_7 ts_4, ts_5	
ts_to_node(ts_5)	ts_5, ts_6, ts_7	
ts_to_node(ts_6)	ts_6, ts_7 ts_6, ts_7	ts_6, ts_7

next subsection), the relevant part of a partition with respect to a time series t consists of t itself and the time series with identifiers higher than t. We call that a "candidate cluster of time series". We ignore clusters with just one element, as pairs cannot be derived out of them.

Let us say we require that some fraction f of the grids should put two time series in the same grid cell for us to be willing to consider that pair of time series to be worth checking in detail. For this example, set f to 2/3 (Figure 3.2).

Each node takes care of those time series that map to that node. So for example, node 1 shows that ts_1 and ts_2 satisfy the requirement. Node 2 shows nothing new concerning ts_2. Node 3 shows that ts_3 and ts_4 satisfy the requirement. Node 4 and node 5 show nothing new concerning ts_4 and ts_5 respectively. Node 6 shows that ts_6 and ts_7 satisfy the

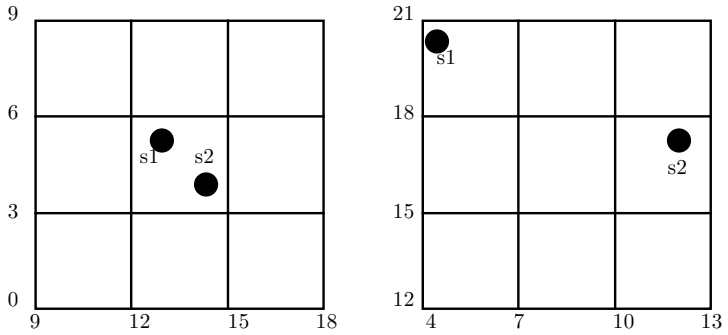


Figure 3.2 – Two series (s_1 and s_2) may be similar in some dimensions (here, illustrated by $Grid_1$) and dissimilar in other dimensions ($Grid_2$). If the series are close in a large fraction of the grids, they are likely to be similar. So, if that fraction exceeds some threshold f (2/3 in the toy example), then the algorithm checks for explicit correlation.

requirement. Node 7 shows nothing new (in fact the last node will never show anything new, so need not be considered). All those that satisfy the requirement can be tested for direct correlation. In this example, this would entail computing correlations on the last windows of length w of ts_1 and ts_2; ts_3 and ts_4; and ts_6 and ts_7 (Table 3.4).

Generalizing from this example, the algorithm proceeds as follows::

1. Partition the sketch vectors, which all have length r, into groups of size k (*e.g.*, if r is 60 and k is 2, then the partition would be 0,1, 2,3, 4,5, ..., 58,59 and we would take indexes 0 and 1 of each sketch vector and put it in the first partition. So each partition would consist of N mini-vectors of size 2 each.).

2. Each site computes a grid and puts time series identifiers in grid cell (Table 3.3). So, for each site s,

   (a) for each time series t, place the identifier of t in a grid cell corresponding to sketch(t)[I_s], where I_s are the indexes assigned to site s.

   (b) Next form a partition of the time series identifiers such that each member of the partition corresponds to a non-empty grid cell. So, two time series t_1 and t_2 will fall into the same partition if sketch(t_1)[I_s] maps to the same grid cell as sketch(t2)[I_s]. Denote the partitioning induced by this grid search on site s as partitioning(s).

   (c) Each element of the partition p in partitioning(s) represents a set of time series. If we sort them by their id, then p can represent $ts\_p\_1, ts\_p\_2, ...$

3. We estimate that two time series are close if they are in the same grid cells in a fraction f of the grids. (The parameter f is determined by a calibration step that in turn depends on the desired correlation threshold, as we will explain in the

experimental section.) We start by constructing "candidate clusters of time series" based on each grid.

4. Send each candidate cluster of time series identifier to every node corresponding to the time series in that cluster (Table 3.4). Call the mapping function between time series ids and nodes ts_to_node, to be defined as the "opt" strategy in the next subsection (For this discussion, we assume that ts_to_node is 1 to 1. If not, then if a node has say the time series groups corresponding to i_1, i_2, and i_3, then keep those groups separate.)

5. At each destination node, two time series are candidates for explicit analysis if they are in the same grid cell for some fraction f of the grids (Table 3.4). If so, compute the Pearson correlation on those two time series.

3.2.4 Communication strategies for detecting correlated candidates

Step 4 of the above algorithm requires the communication of information about each pair (t_i, tj) to one node of the system where its *grid score* (i.e., the number of grids in which the two time series are in the same cell) is computed. This communication may be done using different strategies, which in turn can have a large impact on the performance of our approach. This should come as no surprise: parallel approaches often require an optimization of communication. We compare three strategies for communicating the pairs of each grid cell:

- **All pairs communication (basic)**: In this strategy, for each cell c that contains $|count(c)|$ time series, all pairs (t_i, t_j) are generated and sent to a reducer using the pair as key (in the pair, we assume $i < j$). This ensures that all information about a pair will be sent to one reducer where its grid score can be compared with threshold f. This is the straightforward approach and will be denoted as "basic" in the rest of this chapter.

- **All time series to each responsible reducer (semi-opt)**: In this strategy, for each time series t there is a reducer r_t that is responsible for detecting the candidate time series that are correlated to t. Given a grid cell c, for each time series $t \in contents(c)$, all time series of c are sent to r_t. If, among the time series that r_t receives, the number of occurrences of a time series t' is more than the threshold f, then the pair (t, t') is considered as a candidate pair. This is the semi-optimized (semi-opt in the rest of this chapter) strategy.

- **Part of time series to each responsible reducer (opt)**: In this strategy (embodied in step 4 of the algorithm of the previous sub-section), as in the previous strategy, for each time series t there is a reducer r_t that is responsible for detecting the time series that are potentially correlated to t. But here, only some of time series of the cell are sent to r_t. Let's assume a total order on the ids of the time series, say $t_1 < t_2 < \ldots < t_n$. Given a grid cell $contents(c) = \{t_1, \ldots, t_s\}$, for each time

series $t \in contents(c)$, the time series with ids higher than that of t are sent to r_t. The idea behind this strategy is that for a potential candidate pair (t_i, t_j), we need only to count its occurrences in the one with the lower identifier (i) of the time series, not in both of them. As explained in the following analysis, this strategy requires the least amount of communication and is denoted "opt" in the rest of this chapter.

Below, we analyze the communication cost of the three strategies in terms of the size and the number of messages to be communicated for each cell. In the "basic" strategy, for the contents of each cell $contents(c) = \{t_1, \ldots, t_s\}$, all pairs ($t_i$, t_j) are generated and sent to the reducers. Thus, the number of messages for the cell c is equal to $count(c) \times (count(c) - 1)/2$. The size of each message is 2, so the size of data transferred for cell c is $count(c) \times (count(c) - 1)$. Note that in a distributed system, the number of messages is the principal factor for measuring communication cost of the algorithms. Thus, this approach does not have a good communication cost, as the number of messages for a cell c is $O(count(c)^2)$.

In the "semi-opt" strategy, for each cell c, the node containing each grid communicates $(count(c) - 1)$ time series to the node that must compute the grid score. This means that the number of sent messages is $count(c)$, and the total size of the communicated data is $count(c) \times count(c) - 1$ time series ids per grid cell. In this strategy the number of messages is $O(count(c))$ which is much better than the basic strategy.

In the last strategy, i.e., "opt", for each cell $contents(c) = \{t_1, \ldots, t_s\}$, we communicate $count(c)$ messages per grid cell, *i.e.*, one message to each node that is responsible for a time series in $contents(c)$. The size of the message depends on the id of the time series. Let $t_1, \ldots, t_s$ be the order of the time series ids. Then, we send $\{t_2, \ldots, t_s\}$ to r_{t_1}, $\{t_3, \ldots, t_s\}$ to r_{t_2}, etc. Therefore the total size of communicated data for cell c is $(count(c) - 1) + (count(c) - 2) + \ldots + 1 = count(c) \times (count(c) - 1)/2$. This strategy sends the same number of messages as "semi-opt" (*i.e.*, $O(count(c))$) for each cell, but the size of communicated data is smaller. Our experiments illustrate the benefits of this reduction in size.

3.2.5 Complexity analysis of parallel mixing

Let us analyze the time and space needed by our approach to perform parallel mixing.

The redistribution in Step 1 is proportional to the number of time series times the number of random vectors (because the number of random vectors equals the size of each sketch vector). Note that it is independent of the size of the window. This step is very well parallelizable at the level of nodes and linear in the number of time series. Step 2 (inserting into grids) is linear in the number of time series, cheaper than Step 1.

The dominant time of our approach is that of Steps 3 and 4 in which the responsible node of each grid constructs the candidate clusters of time series, and sends them to the corresponding node based on ts_to_node. If ts_to_node is many to one, then even in the worst case the number of messages is proportional to the number of destination nodes and the total message traffic from a node is proportional to the square of the number of

time series $\times$ the size of each time series identifier. That is a very pessimistic worst case because it corresponds to all time series mapping to the same grid cell in every grid. As we will see in the experimental section, the total traffic per node is linear in the number of time series in practice. Because time series ids are under 32 bits, the total traffic is light.

The last step (*i.e.*, step 5), is proportional to the size of the output, because a large fraction of pairs that pass the sketch filtering step in fact meet the correlation threshold.

The bulk of the space required for our approach is the space needed for keeping the grids for indexing the sliding windows. This space depends on the number of grids and the number of time series. The number of grids itself depends on the size of the sketches in the sliding window, and the group size (number of dimensions in each grid). Let g be the group size, s be the sketch vector size, and n the number of time series. Then, the number of grids required for indexing the sliding window data is $\frac{s}{g}$. In each grid, we need to keep the id of each time series in its corresponding cell. Thus, the total space required for storing the grids is $O(n \times idsize \times \frac{s}{g})$, where n is the number of time series, $idsize$ is the size of an id, s is the sketch vector size and g the group size. Notice that in practice, the size of our grid-based index is much less than the space required for keeping the time series in the sliding windows. For example, suppose the group size is 2, and the sketch vector size is 32 (for a sliding window of size 256). Then, the space required to store all grids is equivalent to $16 \times n$ identifiers, which is less than the space needed to store n sliding windows of size 256.

In practice, the entire procedure requires work that is the sum of i (formation of sketch vectors): number of time series $\times$ size of basic windows $\times$ number of random vectors, ii (parallel mixing, grid computation): number of time series $\times$ number of random vectors, iii (parallel mixing, candidate identification) for each grid cell, square of the number of time series $\times$ size of time series identifiers, iv (for verification of candidate pairs): number of highly correlated pairs $\times$ window size, This work, except for the communication step (which depends on the communications infrastructure), is entirely parallelizable. Which term dominates depends on how high the threshold is. For very high thresholds, part iv will be negligible, iii will be small, and so i and ii will dominate. If the threshold is low (not normally an interesting case) the algorithm could be nearly as expensive as comparing every pair of time series.

3.3 Experiments

In this section, we report experimental results that show the quality and the performance of our parallel incremental sketching approach, illustrating performance, scalability, recall, and precision. We compare our work with iSAX and show vastly improved speed at some cost in recall.

The parallel experimental evaluation was conducted on a cluster of 32 machines, with operating system Linux x86_64 kernel 3.10.0, each machine having 64 Gigabytes of main memory, an Intel Xeon CPU with 8 cores and a 256 Gigabytes hard disk.

We implemented the approaches on top of Apache-Spark 1.6.2 [79], using the Java

programming language.

Data streams are simulated by distributing the data beforehand and using synchronized sliding windows on each site. This setup allowed us to better evaluate the performance gains of our approach without depending on the specific characteristics or optimization of any dedicated streaming environment (*e.g.*, Spark streaming, Flink, Storm, etc.).

3.3.1 Comparisons

We compare ParrCorr to two baseline approaches:

- **Parallel Linear Search**. This is the straightforward comparison that compares each time series to all the other ones, computing a Pearson correlation. Correlations are sorted by decreasing order and the top-correlated ones are kept. It is implemented in parallel (each computing node compares the series it contains to all the series of the other nodes).

- **iSax** [18]. This index allows processing similarity queries using both an exact and an approximate approach. iSax shows an improvement over Parallel Linear Search: when a computing node receives a time series to be compared to its local time series, rather than applying a linear search it will use a local iSax index as a filter to identify the most similar time series.

In the data stream context, these algorithms are applied from scratch, after each update (each basic window-sized move of the sliding window). For iSax, the local indexes have to be built again after each update.

3.3.2 Datasets

We carried out our experiments on both synthetic and seismic datasets.

Synthetic dataset: Each time series in our synthetic dataset consists of 2000 values. At each time point, the generator draws a random number from a Gaussian distribution N(0,1), then adds the value of the last number to the new number. The number of time series varies from one million to 100 million depending on the experiment. This type of random walk generator has been widely used in the past. [30, 10, 71, 16, 18, 82].

Seismic dataset: The real world data represents seismic time series collected from the IRIS Seismic Data Access repository [1] at various earthquake zones. After preprocessing, the seismic dataset contains 5 million time series of 2000 values each.

To detect seismic events, there are three main types of algorithms: energy detectors, array detectors and matched filter detectors. The latter is a new kind of detector, where a representative time series is used as a template (i.e., a "matched filter") and correlated against a continuous data stream to detect new occurrences of that same signal. However, such filters require a large number of templates, making indexing an appealing approach. A time series at a given sensor functions like a geophysical fingerprint for earthquakes. A

[1] `http://ds.iris.edu/data/access/`

Table 3.5 – Default parameters

Parameters	Value
Sliding Window Size	500
Basic Window Size	20
iSAX Word Length	8
Leaf Capacity Threshold	1,000
Basic Cardinality	2
Maximum Cardinality	512
Number Of Machines	32
Correlation Threshold	0.7

seismic signal that closely matches a previous observation can be used as evidence that the newly observed event must have occurred very close to the event that generated the first observation. Moreover, if the signals are similar we can assume that the characteristics of the earthquakes are similar. There are many examples where almost identical signals produced by different earthquakes have been observed. This is typically the case during seismic crises that can last days or months, while similar signals can be recorded even if years apart. Detecting such correlations is a small variation of our problem, where all time series are compared with a few templates. Here we address the harder problem of finding all correlations among the set of time series. This might be useful in an application in which we want to detect, in a real time fashion, where similar seismic events are occurring.

3.3.3 Parameters

Table 3.5 shows the default parameters used for each experiment, unless otherwise specified. A typical application might have a large ratio between the sliding window size and the basic window size, where the basic window indicates the time interval between the recalculation of similarity. We've chosen a ratio of 50, which we have found to be reasonable for many applications. ParCorr does better relative to the other algorithms with a smaller basic window size of 10 for example, but 50 is more reasonable for high frequency measurements. The iSAX word length, leaf capacity, basic cardinality, and maximum cardinality were chosen to be optimal for iSAX (and were taken from [18]). All histograms in the figures have error bars (usually so small as to be invisible) that go from a minimum value to a maximum value (*i.e.*, 100% confidence interval) with the histogram height representing the mean.

We calibrate the fraction f (needed for detecting candidate items in the grids) by using a small sample database. We increase f until reaching the desired recall (*e.g.*, 0.95) on the small sample, and then we use the found fraction in our experiments on big datasets.

3.3.4 Recall and Precision Measures

To understand these concepts in our applications, consider the correlation problem: we want to find all pairs of time series that have at least a correlation of some specified threshold during a given window. Call that set S_{true} In that context, the *recall* of a method that finds a set S_{method} is $|(S_{true} \cap S_{method})|/|S_{true}|$ and the *precision* is $|(S_{true} \cap S_{method})|/|S_{method}|$. This would also be true for Euclidean distances. These are completely standard uses of these terms applied to pairs and similarity metrics.

In our experiments, the default correlation threshold for Pearson is 0.7. We have also tried 0.8 and 0.9. With a Pearson threshold of 0.8, the sketch recall was over 96% and the speedup compared with iSAX was a factor of 17.56. With a Pearson threshold of 0.9, the sketch recall was over 95.7% and the speedup compared with iSAX was a factor of 18. Given any Pearson correlation, the threshold for Euclidean distance is computed by using formula 3.1.

3.3.5 Communication Strategies

Before presenting the results of our approach in detail, we evaluate here the impact of the communication strategy to detect correlated pairs. This corresponds to the discussion and analysis given in Section 3.2.4. We conducted this experiment on 5 million time series, with a basic window of 32 and a sliding window of 256. As expected and illustrated by Figure 3.3, our optimized strategy gives the best performance (response time), but the size of the gain is surprising. Therefore, in the experiments presented below, we use this optimized strategy.

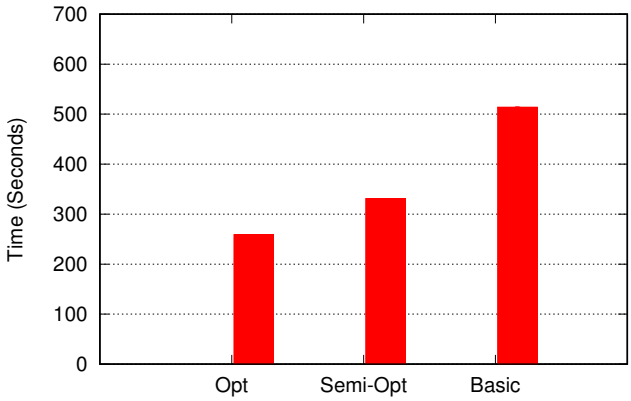


Figure 3.3 – Execution time (not including the pair checking time) for each of the communication strategies introduced in Section 3.2.4. The algorithms are run on a cluster of 32 nodes and 5 million time series (basic window of 32 and sliding window of 256). The optimized strategy gives the best response time.

3.3.6 Results

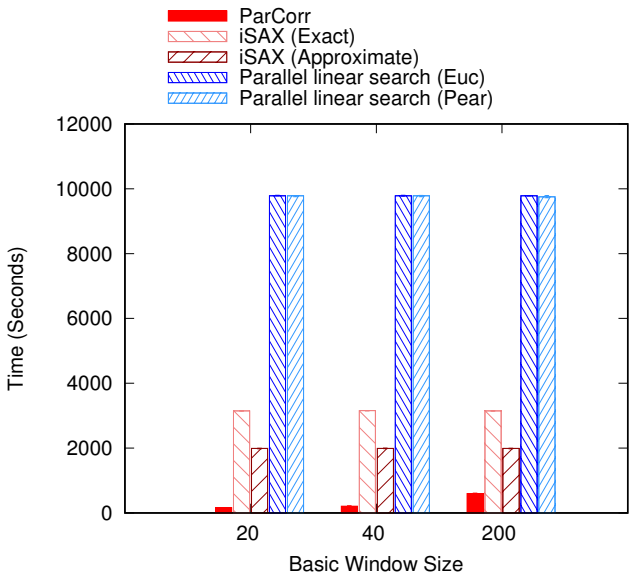


Figure 3.4 – Execution time for the calculation of the correlations for each sliding window as a function of basic window size for the random walk dataset. The algorithms are run on a cluster of 32 nodes and 5 million time series. The time for ParCorr increases as the basic window size increases, because updating the sketch vector takes slightly longer. All parameters other than basic window size are set to their values from Table 3.5.

Figure 3.4 shows that ParCorr is orders of magnitude faster for parallel correlation than the iSAX methods for the random walk dataset, though its time increases as the basic window size increases. For instance, with a basic window of 20, ParCorr takes at most 160 seconds to process a sliding window, while iSAX Approximate needs 1990 seconds. We attribute this advantage to two factors: the calculation of sketches is incremental and the parallelization of the algorithm is natural. These results also hold for the seismic data as can be seen in Figure 3.5.

Figure 3.6 shows that ParCorr scales well to large datasets containing up to 100 million time series. iSAX approximate is consistently about 50% faster than ISAX exact. Our competitors (Parallel linear search, iSAX Approximate/Exact) do not scale since they cannot handle more than 5 million time series due to the fact that both memory usage and communication costs become hard to bear.

Figure 3.7 shows that both iSAX and ParCorr enjoy a roughly linear speedup, whereas Figure 3.8 shows that ParrCorr is orders of magnitude faster in absolute time at all degrees of parallelization. ParCorr needs at most 598 seconds on 8 nodes (169 seconds on 32 nodes) while iSAX Approximate needs at most 13460 seconds (9784 seconds on 32 nodes).

Figure 3.9 shows that ParCorr's performance (using Spark) is comparable to iSAX

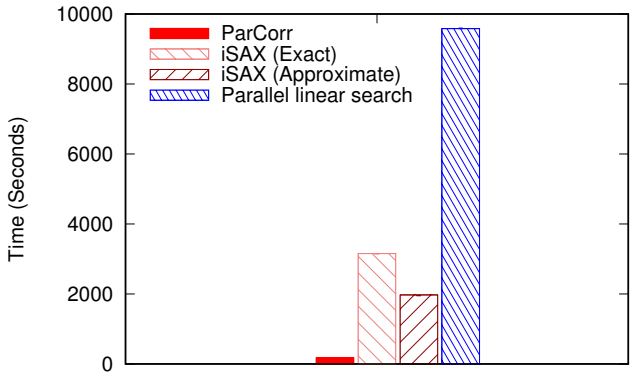


Figure 3.5 – Execution time for the calculation of the correlations for each sliding window for the seismic dataset. The algorithms are run on a cluster of 32 nodes and 5 million time series. All parameters are set to their values from Table 3.5.

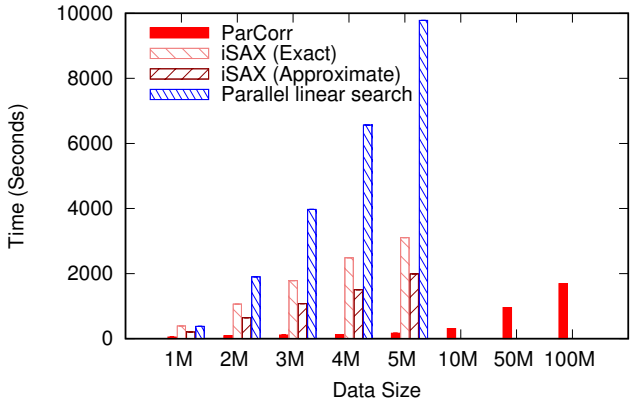


Figure 3.6 – Execution time for the calculation of the correlations for each sliding window as a function of dataset size for the random walk dataset. The algorithms are run on a cluster of 32 nodes. All parameters are set to their values from Table 3.5. Note that ParCorr scales to larger datasets nearly linearly and the times remain practical. The other methods exceeded the measurement window.

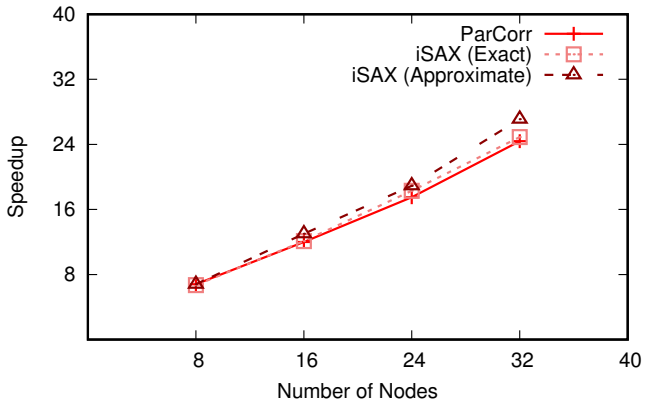


Figure 3.7 – SpeedUp: All algorithms enjoy linear speedup with roughly the same slope as the number of processing nodes increase. All parameters are set to their values from Table 3.5.

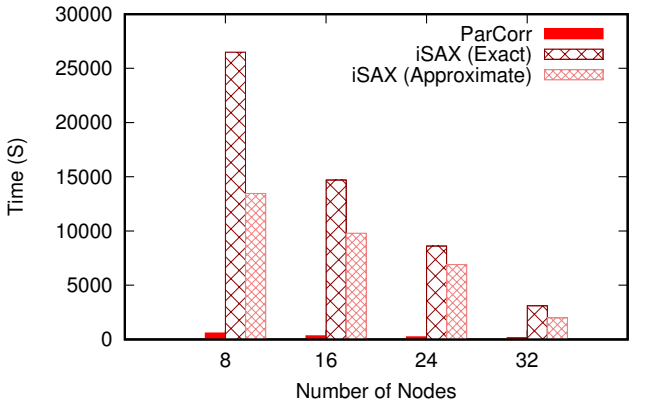


Figure 3.8 – Execution time for the calculation of the correlations for each sliding window as a function of the number of processing nodes for the random walk dataset. The algorithms are run on 5 million time series. All parameters are set to their values from Table 3.5.

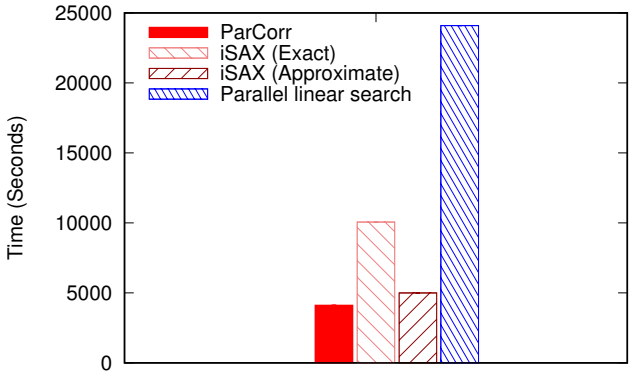


Figure 3.9 – Execution time for each sliding window on a single node for the random walk dataset. The dataset is 1 million time series. All parameters are set to their values from Table 3.5.

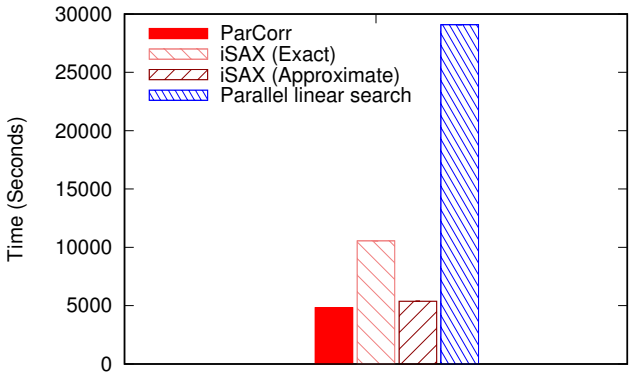


Figure 3.10 – Execution time for each sliding window on a single node for the seismic dataset. The dataset is 1 million time series. All parameters are set to their values from Table 3.5.

(running natively without Spark) on a single node. ParCorr shows a small advantage but not as much as in a parallel setting, because Spark entails some overhead. These results are consistent with the results for the seismic data as shown in Figure 3.10.

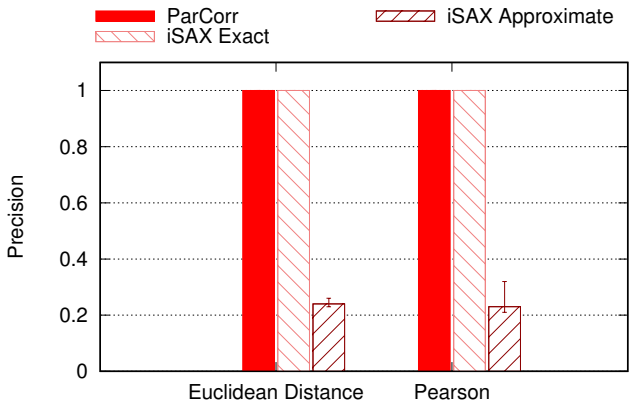


Figure 3.11 – Let Peuc be the set of pairs of time series whose final w values fall within the distance threshold in the case of Euclidean distance. And let Pcorr be the set of pairs of time series whose final w values fall above the threshold in the case of Pearson. The precision is the fraction of the set of pairs found by each algorithm that belong to Peuc or Pcorr, respectively. ParCorr has 100% precision because it checks candidate pairs that are produced by the sketch algorithm.

Figure 3.11 shows the high precision of ParCorr and iSAX. ParCorr verifies all the candidate pairs that the sketch filter produces. iSAX Exact incorporates a verification step as well. These results hold also for seismic data as seen in Figure 3.13.

Figure 3.12 shows that iSAX Exact gives perfect recall because of its bounding box guarantee. ParCorr gives no such guarantee, so for applications that require 100% recall, iSAX Exact should be used. Empirically, ParCorr yields a recall of over 90%, as shown in Figure 3.14.

The experiments on real and synthetic data show that ParCorr is fast, scales well, guarantees 100% precision, and achieves very high recall. This reduction in recall is acceptable for many applications, especially given the high gain in response time.

3.4 Related Work

The problem of correlation detection has been studied across data streams using centralized approaches [58, 55, 77, 66, 63, 62, 23, 61]. Most of them focus on reducing the computation time of the pairwise distance computation. For example in [58], Mueen et al. propose efficient algorithms based on the Discrete Fourier Transformation (DFT), to reduce the end-to-end response time of an all-pair correlation query. As Cole et al. discuss in [23], this works well when the time series are cooperative (*i.e.,* where the low frequency Fourier coefficients dominate).

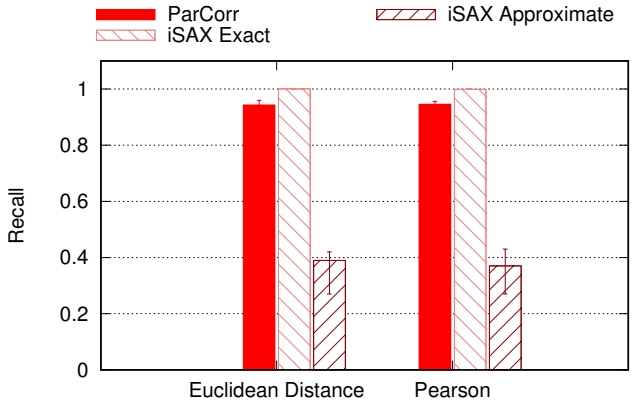


Figure 3.12 – Let Peuc and Pcorr be defined as in the caption of Figure 3.11. The recall is the fraction of Peuc or Pcorr, respectively, that is found by each algorithm. Note that iSAX exact gives higher recall than ParCorr.

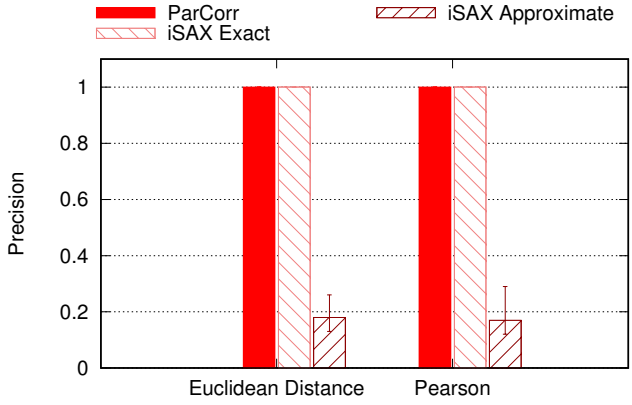


Figure 3.13 – For seismic precision, iSAX Exact and ParCorr both achieve 100% precision for Euclidean. ParCorr also achieves 100% precision for Pearson correlation.

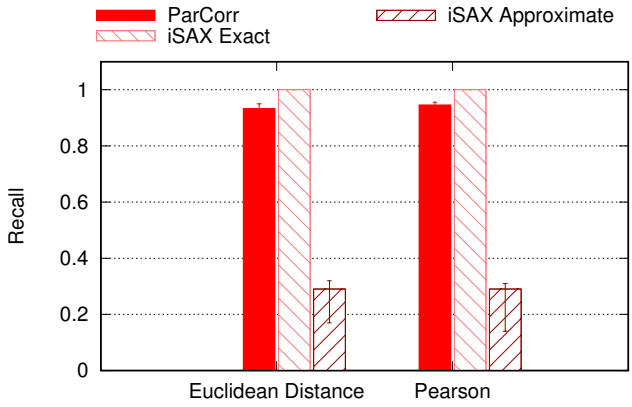


Figure 3.14 – For seismic data, iSAX Exact achieves perfect recall. ParCorr achieves over 90% for both Euclidean and Pearson correlation.

The problem of indexing and querying time series using centralized solutions has been widely studied in the literature, *e.g.*, [10, 15, 30, 71, 18]. For instance, in [10], Assent et al. propose the TS-tree, an index structure for efficient retrieval and similarity search over time series. In [71], Shieh et al. propose the multiresolution symbolic representation called indexable Symbolic Aggregate approXimation (iSAX) [18] which is based on the SAX representation. The advantage of iSAX over SAX is that it allows the comparison of words with different cardinalities, and even different cardinalities within a single word. iSAX can be used to create efficient indices over very large databases.

In our work, we take advantage of the sketch index [23] which is related to Locality Sensitive Hashing [35], in that similar items are hashed to the same bucket with high probability. The reason for using the sketch index is that it can be perfectly parallelized with a near balanced workload for the nodes participating in the index creation operation and it can be updated incrementally. The novelty of our work is to propose new methods for the incremental update and, above all, a parallelization method.

In our work, we use an approach based on incremental random sketches [23]. In the literature, several techniques have been used to perform dimensionality reduction on the size of time series. Examples of such techniques that can significantly reduce the time and space required for the index are: singular value decomposition (SVD) [30], the discrete Fourier transformation (DFT) [6], the discrete wavelets transformation (DWT) [20], the piecewise aggregate approximation (PAA) [48], and the adaptive piecewise constant approximation (APCA) [19]. We compare with iSAX [18] because it does not require time series to be cooperative (though it is more efficient when the time series is slowly changing).

Sometimes one wants to find clusters of unusual events in time series, *e.g.*, bursts of activity or bursts of unusually high values. In such settings, tiling algorithms [34, 42, 31] apply. Our problem is complementary because we are finding correlations between time series or portions of time series, but are agnostic to the level of interest of individual time points. However, our method could be used to post-process temporal regions that tiling indicates are of interest.

3.5 Context of the work

This work has been done in the context of Djamel Edine Yagoubi's PhD thesis and a collaboration with Prof. Dennis E. Shasha from the University of New York. Two engineers of the Zenith team, Oleksandra Levchenko and Boyan Kolev, have contributed on implementing a prototype of ParCorr.

3.6 Conclusion

Finding similar pairs of time series on sliding windows is useful for many applications. Methods to do so for hundreds of millions of time series in a highly efficient and scalable fashion is the contribution of this chapter. Compared with the previous state of the art

iSAX, the ParCorr solution is faster and scalable, while showing only very little loss in recall. For many applications, where scalability is mandatory, this is highly beneficial.

Chapter 4

Parallel Mining of Maximally Informative k-Itemsets in Big Data

4.1 Introduction

Featureset, or itemset, mining [40] is one of the fundamental building bricks for exploring informative patterns in databases. Features might be, for instance, the words occurring in a document, the score given by a user to a movie on a social network, or the characteristics of plants (growth, genotype, humidity, biomass, etc.) in a scientific study in agronomics. A large number of contributions in the literature has been proposed for itemset mining, exploring various measures according to the chosen relevance criteria. The most studied measure is probably the number of co-occurrences of a set of features, also known as frequent itemsets [7]. However, frequency does not give relevant results for a various range of applications, including information retrieval [37], since it does not give a complete overview of the hidden correlations between the itemsets in the database. This is particularly the case when the database is sparse [41]. Using other criteria to assess the informativeness of an itemset could result in discovering interesting new patterns that were not previously known. To this end, information theory [25] gives us strong supports for measuring the informativeness of itemsets. One of the most popular measures is the joint entropy [25] of an itemset. An itemset X that has higher joint entropy brings up more information about the objects in the database.

In this chapter, we study the problem of Maximally Informative k-Itemsets (*miki* for short) discovery in massive datasets, where informativeness is expressed by means of joint entropy and k is the size of the itemset [36, 49, 81]. *Miki* are itemsets of interest that better explain the correlations and relationships in the data. Example 4 gives an illustration of *miki* and its potential for real world applications such as information retrieval.

Example 4. *In this application, we would like to retrieve documents from Table 4.1, in which the columns $d_1, \ldots, d_{10}$ are documents, and the attributes A, B, C, D, E are some features (items, keywords) in the documents. The value "1" means that the feature occurs in the document, and "0" not. It is easy to observe that the itemset (D, E) is frequent,*

Features	Documents									
	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}
A	1	1	1	1	1	0	0	0	0	0
B	0	1	0	0	1	1	0	1	0	1
C	1	0	0	1	0	1	1	0	1	0
D	1	0	1	1	1	1	1	1	1	1
E	1	1	1	1	1	1	1	1	1	1

Table 4.1 – Features in The Documents

because features D and E occur together in almost every document. However, it provides little help for document retrieval. In other words, given a document d_x in our dataset, one might look for the occurrence of the itemset (D, E) and, depending on whether it occurs or not, she will not be able to decide which document it is. By contrast, the itemset (A, B, C) is infrequent, as its member features rarely or never appear together in the data. And it is troublesome to summarize the value patterns of the itemset (A, B, C). Providing it with the values $< 1, 0, 0 >$ we could find the corresponding document O_3; similarly, given the values $< 0, 1, 1 >$ we will have the corresponding document O_6. Although (A, B, C) is infrequent, it contains lots of useful information which is hard to summarize. By looking at the values of each feature in the itemset (A, B, C), it is much easier to decide exactly which document they belong to. (A, B, C) is a maximally informative itemset of size $k = 3$.

Miki mining is a key problem in data analytics with high potential impact on various tasks such as supervised learning [50], unsupervised learning [32] or information retrieval [37], to cite a few. A typical application is the discovery of discriminative sets of features, based on joint entropy [25], which allows distinguishing between different categories of objects. Unfortunately, it is very difficult to maintain good results, in terms of both response time and quality, when the number of objects becomes very large. Indeed, with massive amounts of data, computing the joint entropies of all itemsets in parallel is a very challenging task for many reasons. First, the data is no longer located in one computer, instead, it is distributed over several machines. Second, the number of iterations of parallel jobs would be linear to k (i.e., the number of features in the itemset to be extracted [49]), which needs multiple database scans and in turn violates the parallel execution of the mining process. We believe that an efficient *miki* mining solution should scale up with the increase in the size of the itemsets, calling for cutting edge parallel algorithms and high performance evaluation of an itemset's joint entropy in massively distributed environments.

We designed and developed an efficient parallel algorithm, namely Parallel Highly Informative K-itemSet (PHIKS in short), that renders the discovery of *miki* from a very large database (up to Terabytes of data) simple and effective. PHIKS combines both information theory and massive distribution, and takes advantage of parallel programming

frameworks such as MapReduce [26] or Spark [80]. It cleverly exploits available data at each mapper to efficiently calculate the joint entropies of *miki* candidates. For more efficiency, we provide PHIKS with optimizations that allow for very significant improvements of the whole process of *miki* mining. The first technique estimates the upper bound of a given set of candidates and allows for a dramatic reduction of data communications, by filtering unpromising itemsets without having to perform any additional scan over the data. The second technique reduces significantly the number of scans over the input database of each mapper, i.e., only one scan per step, by incrementally computing the joint entropy of candidate features. This reduces drastically the work that should be done by the mappers, and thereby the total execution time.

PHIKS has been extensively evaluated using massive real-world datasets. Our experimental results show that PHIKS significantly outperforms alternative approaches, and confirm the effectiveness of our proposal over large databases containing for example one Terabyte of data.

The rest of the chapter is structured as follows. Section 4.2 gives a formal definition of the addressed problem. The necessary background is given in Section 4.3. In Section 4.4, we present our PHIKS solution, and depict its whole core mining process. Section 4.5 reports on an experimental validation over real-world datasets. Section 4.6 discusses related work. Section 4.7 presents the context of this work, and Section 4.8 concludes.

4.2 Problem Definition

The following definitions introduce the basic requirements for mining maximally informative k-itemsets [49].

Definition 2. *Let $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be a set of literals called $features$. An $itemset$ X is a set of $features$ from $\mathcal{F}$, i.e., $X \subseteq \mathcal{F}$. The $size$ or $length$ of the $itemset$ X is the number of $features$ in it. A $transaction$ T is a set of elements such that $T \subseteq \mathcal{F}$ and $T \neq \emptyset$. A $database$ $\mathcal{D}$ is a set of $transactions$.*

Definition 3. *The entropy [25] of a feature i in a database $\mathcal{D}$ measures the expected amount of information needed to specify the state of uncertainty or disorder for the feature i in $\mathcal{D}$. Let i be a feature in $\mathcal{D}$, and $P(i = n)$ be the probability that i has value n in $\mathcal{D}$ (we consider categorical data, where the value will be '1' if the object has the feature and '0' otherwise). The entropy of the feature i is given by*

$$H(i) = -(P(i = 0)log(P(i = 0)) + P(i = 1)log(P(i = 1)))$$

where the logarithm base is 2.

Definition 4. *The binary projection, or projection of an itemset X in a transaction T ($proj(X, T)$) is the set of size $|X|$ where each item (i.e., feature) of X is replaced by '1' if it occurs in T and by '0' otherwise. The projection counting of X in a database $\mathcal{D}$ is the set of projections of X in each transaction of $\mathcal{D}$, where each projection is associated with its number of occurrences in $\mathcal{D}$.*

Example 5. *Let us consider Table 4.1. The projection of* (B, C, D) *in* d_1 *is* $(0, 1, 1)$*. The projections of* (D, E) *on the database of Table 4.1 are* $(1, 1)$ *with nine occurrences and* $(0, 1)$ *with one occurrence.*

Definition 5. *Given an itemset* $X = \{x_1, x_2, \ldots, x_k\}$ *and a tuple of binary values* $\mathcal{B} = \{b_1, b_2, \ldots, b_k\} \in \{0\,1\}^k$*. The joint entropy of* X *is defined as:*

$$H(X) = -\sum_{\mathcal{B} \in \{0,1\}^{|k|}} J \times log(J)$$

Where J= $P(x_1 = b_1, \ldots, x_k = b_k)$ *is the joint probability of* $X = \{x_1, x_2, \ldots, x_k\}$.

Given a database $\mathcal{D}$, the joint entropy $H(X)$ of an itemset X in $\mathcal{D}$ is proportional to its size k i.e., the increase in the size of X implies an increase in its joint entropy $H(X)$. The higher the value of $H(X)$, the more information the itemset X provides in $\mathcal{D}$. For simplicity, we use the term entropy of an itemset X to denote its joint entropy.

Example 6. *Let us consider the database of Table 4.1. The joint entropy of* (D, E) *is given by* $H(D, E) = -\frac{9}{10}log(\frac{9}{10}) - \frac{1}{10}log(\frac{1}{10}) = 0.468$*. Where the quantities* $\frac{9}{10}$ *and* $\frac{1}{10}$ *respectively represent the joint probabilities of the projection values* $(1, 1)$ *and* $(0, 1)$ *in the database.*

Definition 6. *Given a set* $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ *of features, an itemset* $X \subseteq \mathcal{F}$ *of length* k *is a maximally informative* k*-itemset, if for all itemsets* $Y \subseteq \mathcal{F}$ *of size* k*,* $H(Y) \leq H(X)$*. Hence, a maximally informative* k*-itemset is the itemset of size* k *with the highest joint entropy value.*

The problem of mining maximally informative k-itemsets presents a variant of itemset mining, it relies on the joint entropy measure for assessing the informativeness brought by an itemset.

Definition 7. *Given a database* $\mathcal{D}$ *which consists of a set of* n *attributes (features)* $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$*. Given a number* k*, the problem of miki mining is to return a subset* $F' \subseteq F$ *with size* k*, i.e.,* $|F'| = k$*, having the highest joint entropy in* $\mathcal{D}$*, i.e.,* $\forall F'' \subseteq F \wedge |F''| = k, H(F'') \leq H(F')$.

4.3 Background

In this section, first we detail the *miki* discovery in a centralized environment. Second, we detail the working principle of MapReduce, in particular, we depict the execution process of a MapReduce job.

4.3.1 Miki Discovery in a Centralized Environment

In [49], an effective approach is proposed for *miki* discovery in a centralized environment. Their *ForwardSelection* heuristic uses a "generating-pruning" approach, which is similar to the principle of *Apriori* [7]. i_1, the feature having the highest entropy is selected as a seed. Then, i_1 is combined with all the remaining features, in order to build candidates. In other words, there will be $|\mathcal{F}-1|$ candidates (*i.e.*, $(i_1, i_2), (i_1, i_3), \ldots, (i_1, i_{|\mathcal{F}-1|})$). The entropy of each candidate is given by a scan over the database, and the candidate having the highest entropy, say (i_1, i_2), is kept. A set of $|\mathcal{F}-2|$ candidates of size 3 is generated (*i.e.*, $(i_1, i_2, i_3), (i_1, i_2, i_4), \ldots, (i_1, i_2, i_{|\mathcal{F}-2|})$) and their entropy is given by a new scan over the database. This process is repeated until the size of the extracted itemset is k.

4.3.2 MapReduce and Job Execution

MapReduce has gained increasing popularity, as shown by the tremendous success of Hadoop [76], an open-source implementation. It is one of the most popular solutions for big data processing [13], in particular due to its automatic management of parallel execution in clusters of machines. Initially proposed in [26], it was popularized by Hadoop [76], an open-source implementation. MapReduce divides the computation in two phases, namely map and reduce, which in turn are carried out by several tasks that process the data in parallel. The idea behind MapReduce is simple and elegant. Given an input file, and two functions map and reduce, each MapReduce job is executed in two main phases: map and reduce. In the first phase, called map, the input data is divided into a set of splits, and each split is processed by a map task in a given worker node. These tasks apply the map function on every key-value pair of their split and generate a set of intermediate pairs. In the second phase, called reduce, all the values of each intermediate key are grouped and assigned to a reduce task. Reduce tasks are also assigned to worker machines and apply the reduce function on the created groups to produce the final results. Each MapReduce job includes two functions: map and reduce. For executing the job, we need a master node for coordinating the job execution, and some worker nodes for executing the map and reduce tasks. When a MapReduce job is submitted by a user to the cluster, after checking the input parameters, e.g., input and output directories, the input splits (blocks) are computed. The number of input splits can be personalized, but typically there is one split for each 64MB of data. The location of these splits and some information about the job are submitted to the master. The master creates a job object with all the necessary information, including the map and reduce tasks to be executed. One map task is created per input split. When a worker node, say w, becomes idle, the master tries to assign a task to it. The map tasks are scheduled using a locality-aware strategy. Thus, if there is a map task whose input data is kept on w, then the scheduler assigns that task to w. If there is no such task, the scheduler tries to assign a task whose data is in the same rack as w (if any). Otherwise, it chooses any task. Each map task reads its corresponding input split, applies the map function on each input pair and generates intermediate key-value pairs. , which are firstly maintained in a buffer in main memory. When the content of the

buffer reaches a threshold (by default 80% of its size), the buffered data is stored on the disk in a file called spill. Once the map task is completed, the master is notified about the location of the generated intermediate key-values. In the reduce phase, each intermediate key is assigned to one of the reduce workers. Each reduce worker retrieves the values corresponding to its assigned keys from all the map workers, and merges them using an external merge-sort. Then, it groups pairs with the same key and calls the reduce function on the corresponding values. This function will generate the final output results. When, all tasks of a job are completed successfully, the client is notified by the master.

4.4 PHIKS Algorithm

In a massively distributed environment, a possible naive approach for *miki* mining would be a straightforward implementation of *ForwardSelection* [49] (see Section 4.3.1). However, given the "generating-pruning" principle of this heuristic, it is not suited for environments like Spark [80] or MapReduce [26] and would lead to very bad performances. The main reason is that each scan over the dataset is done through a distributed job (i.e., there will be k jobs, one for each generation of candidates that must be tested over the database). Our experiments, in Section 4.5, give an illustration of the catastrophic response times of *ForwardSelection* in a straightforward implementation on MapReduce (the worst, for all of our settings). This is not surprising since most algorithms designed for a centralized itemset mining do not perform well in massively distributed environments in a direct implementation [57], [11], [56], and *miki* don't escape that rule.

Such an inadequacy calls for new distributed algorithmic principles. To the best of our knowledge, there is no previous work on distributed mining of *miki*. However, we may build on top of cutting edge studies in frequent itemset mining, while considering the very demanding characteristics of *miki*.

Interestingly, in the case of frequent itemsets in MapReduce, a mere algorithm consisting of two jobs outperforms most existing solutions [8] by using the principle of SON [67], a divide and conquer algorithm. Unfortunately, despite its similarities with frequent itemset mining, the discovery of *miki* is much more challenging. Indeed, the number of occurrences of an itemset X in a database $\mathcal{D}$ is additive and can be easily distributed (the global number of occurrences of X is simply the sum of its local numbers of occurrences on subsets of $\mathcal{D}$). Entropy is much more combinatorial since it is based on the the projection counting of X in $\mathcal{D}$ and calls for efficient algorithmic advances, deeply combined with the principles of distributed environments.

4.4.1 Distributed Projection Counting

Before presenting the details of our contribution, we need to provide tools for computing the projection of an itemset X on a database $\mathcal{D}$, when $\mathcal{D}$ is divided into subsets on different splits, in a distributed environment, and entropy has to be encoded in the key-value format. We have to count, for each projection p of X, its number of occurrences on $\mathcal{D}$. This can

be solved with an association of the itemset as a key and the projection as a value. On a split, for each projection of an itemset X, X is sent to the reducer as the key coupled with its projection. The reducer then counts the number of occurrences, on all the splits, of each (key:value) couple and is therefore able to calculate the entropy of each itemset. Communications may be optimized by avoiding to emit a $key : val$ couple when the projection does not appear in the transaction and is only made of '0' (on the reducer, the number of times that a projection p of X does not appear in $\mathcal{D}$ is determined by subtracting the number projections of X in D from $|\mathcal{D}|$).

Example 7. *Let us consider $\mathcal{D}$, the database of Table 4.1, and the itemset $X = (D, E)$. Let us consider that $\mathcal{D}$ is divided into two splits $S_1 = \{d_1..d5\}$ and $S_2 = \{d_6..d_{10}\}$. With one simple MapReduce job, it is possible to calculate the entropy of X. The algorithm of a mapper would be the following: for each document d, emit a couple $(key : val)$ where $key = X$ and $val = proj(X, d)$. The first mapper (corresponding to S_1) will emit the following couples: $((D, E) : (1, 1))$ 4 times and $((D, E) : (0, 1))$ once. The second mapper will emit $((D, E) : (1, 1))$ 5 times. The reducers will do the sum and the final result will be $((D, E) : (1, 1))$ occurs 9 times and $(((D, E) : (0, 1))$ once.*

4.4.2 Discovering *miki* in Two Rounds

Our heuristic will use at most two MapReduce jobs in order to discover the k-itemset having the highest entropy. The goal of the first job is to extract locally, on the distributed subsets of $\mathcal{D}$, a set of candidate itemsets that are likely to have a high global entropy. To that end, we apply the principle of *ForwardSelection* locally, on each mapper, and grow an itemset by adding a new feature at each step. After the last scan, for each candidate itemset X of size k we have the projection counting of X on the local dataset. A straightforward approach would be to emit the candidate itemset having the highest local entropy. We denote by *local entropy*, the entropy of an itemset in a subset of the database that is read by a mapper (i.e., by considering only the projections of X in the mapper). Then the reducers would collect the local *miki* and we would check their *global entropy* (i.e., the entropy of the itemset X in the entire database $\mathcal{D}$) by means of a second MapReduce job. Unfortunately, this approach would not be correct, since an itemset might have the highest global entropy, while actually not having the highest entropy in each subset. Example 8 gives a possible case where a global *miki* does not appear as a local *miki* on any subset of the database.

Example 8. *Let us consider $\mathcal{D}$, the database given by Table 4.2, which is divided into two splits of six transactions. The global miki of size 3 in this database is (A, B, E). More precisely, the entropy of (A, B, E) on $\mathcal{D}$ is given by $-\frac{1}{12} \times log(\frac{1}{12}) \times 4 - \frac{2}{12} \times log(\frac{2}{12}) \times 4 = 2.92$. However, if we consider each split individually, (A, B, E) always has a lower entropy compared to at least one different itemset. For instance, on the split S_1, the projections of (A, B, E) are $(0, 0, 0)$, $(0, 1, 0)$, $(1, 1, 0)$ and $(0, 1, 1)$ with one occurrence each, and $(1, 0, 0)$ with two occurrences. Therefore the entropy of (A, B, E) on S_1 is 2.25 (i.e., $-\frac{1}{6} \times log(\frac{1}{6}) \times 4 - \frac{2}{6} \times log(\frac{2}{6}) = 2.25$). On the other hand, the projections of (A, B, C)*

Split	A	B	C	D	E
S_1	0	0	1	0	0
	0	1	0	0	0
	1	0	1	0	0
	1	1	0	0	0
	0	1	1	0	1
	1	0	0	0	0
S_2	0	0	0	0	1
	0	1	0	1	1
	1	0	0	0	1
	1	1	0	1	1
	0	1	0	0	0
	1	0	0	1	1

Table 4.2 – Local Vs. Global Entropy

on S_1 *are* $(0,0,1)$*,* $(0,1,0)$*,* $(1,0,1)$*,* $(1,1,0)$*,* $(0,1,1)$ *and* $(1,0,0)$ *with one occurrence each, and the entropy of* (A,B,C) *on* S_1 *is* 2.58 *(i.e.,* $-\frac{1}{6} \times log(\frac{1}{6}) \times 6 = 2.58$*). This is similar on* S_2 *where the entropy of* (A,B,E) *is* 2.25 *and the entropy of* (A,B,D) *is* 2.58*. However,* (A,B,C) *and* (A,B,D) *both have a global entropy of* 2.62 *on* $\mathcal{D}$*, which is lower than* 2.92*, the global entropy of* (A,B,E) *on* $\mathcal{D}$*.*

Since it is possible that a global *miki* is never found as a local *miki*, we need to consider a larger number of candidate itemsets. This can be done by exploiting the set of candidates that are built in the very last step of *ForwardSelection*. This step aims to calculate the projection counting of $\mathcal{F} - k$ candidates and then compute their local entropy. Instead of only emitting the itemset having the larger entropy, we will emit, for each candidate X, the projection counting of X on the split, as explained in Section 4.4.1. The reducers will then be provided with, for each local candidate X_i ($1 \leq i \leq m$, where m is the number of mappers, or splits), the projection counting of X on a subset of $\mathcal{D}$. The main idea is that the itemset having the highest entropy is highly likely to be in that set of candidates. For instance, in the database given by Table 4.2 and $k = 3$, the global *miki* is (A,B,E), while the local *miki* are (A,B,C) on S_1 and (A,B,D) on S_2. However, with the technique described above, the itemset (A,B,E) will be a local candidate, and will be sent to the reducers with the whole set of projections encountered so far in the splits. In addition, we emit the projection counting of the selected itemsets in all steps of *ForwardSelection*. These projections will be used for optimizing our PHIKS algorithm. The reducer will then calculate its global entropy, compare it to the entropy of the other itemsets, and (A,B,E) will eventually be selected as the *miki* on this database.

Unfortunately, it is possible that X has not been generated as a candidate itemset on the entire set of splits (consider a biased data distribution, where a split contains some

features with high entropies, and these features have low entropies on the other splits). Therefore, we have two possible cases at this step:

1. X is a candidate itemset on all the splits and we are able to calculate its exact projection counting on $\mathcal{D}$, by means of the technique given in Section 4.4.1.

2. There is (at least) one split where X has not been generated as a candidate and we are not able to calculate its exact projection counting on $\mathcal{D}$.

The first case does not need more discussion, since we have collected all the necessary information for calculating the entropy of X on $\mathcal{D}$. The second case is more difficult since X might be the *miki* but we cannot be sure, due to lack of information about its local entropy on (at least) one split. Therefore, we need to check the entropy of X on $\mathcal{D}$ with a second MapReduce job intended to calculate its exact projection counting. The goal of this second round is to check that no local candidate has been ignored at the global scale. At the end of this round, we have the entropy of all the promising candidate itemsets and we are able to pick the one with the highest entropy. This is the architecture of our approach, the raw version of which (without optimization) is called *Simple-PHIKS*. So far, we have designed a distributed architecture and a *miki* extraction algorithm that, in our experiments reported in Section 4.5 outperforms *ForwardSelection* by several orders of magnitude. However, by exploiting and improving some concepts of information theory, we may significantly optimize this algorithm and further accelerate its execution at different parts of the architecture, as explained in the following sections.

4.4.3 Candidate Reduction Using Entropy Upper Bound

One of the shortcomings of the basic version of our two rounds approach is that the number of candidate itemsets, which should be processed in the second job, may be high for large databases as it will be illustrated by our experiments in Section 4.5. This is particularly the case when the features are not uniformly distributed in the splits of mappers. These candidate itemsets are sent partially by the mappers (i.e., not by all of them), thus we cannot compute their total entropy in the corresponding reducer. This is why, in the basic version of our approach, we compute their entropy in the second job by reading again the database.

Here, we propose an efficient technique for significantly reducing the number of candidates. The main idea is to compute an upper bound for the entropy of the partially sent itemsets, and discard them if they have no chance to be a global *miki*. For this, we exploit the available information about the *miki* candidates sent by the mappers to the corresponding reducer.

Let us describe formally our approach. Let X be a partially sent itemset, and m be a mapper that has not sent X and its projection frequencies to the reducer R that is responsible for computing the entropy of X. In the reducer R, the frequency of X projections for a part of the database is missing. We call these frequencies as *missing* frequencies. We compute an upper bound for the entropy of X by estimating its missing

frequencies. This is done in two steps. Firstly, finding the biggest subset of X, say Y, for which all frequencies are available and secondly, distributing the frequencies of Y among the projections of X in such a way that the entropy of X be the maximum.

4.4.3.1 Step 1

The idea behind the first step is that the frequencies of the projections of an itemset X can be derived from the projections of its subsets. For example, suppose two itemsets $X = \{A, B, C, D\}$ and $Y = \{A, B\}$, then the frequency of the projection $p = (1, 1)$ of Y is equal to the sum of the following projections in X: $p_1 = (1, 1, 0, 0)$, $p_2 = (1, 1, 0, 1)$, $p_3 = (1, 1, 1, 0)$ and $p_4 = (1, 1, 1, 1)$. The reason is that in all these four projections, the features A and B exist, thus the number of times that p occurs in the database is equal to the total number of times that the four projections p_1 to p_4 occur. This is stated by the following lemma.

Lemma 2. *Let the itemset Y be a subset of the itemset X, i.e., $Y \subseteq X$. Then, the frequency of any projection p of Y is equal to the sum of the frequencies of all projections of X which involve p.*

Proof. The proof can be easily done as in the above discussion.

In Step 1, among the *available subsets* of itemset X, i.e., those for which we have all projection frequencies, we choose the one that has the highest size. The reason is that its intersection with X is the highest, thus our estimated upper bound about the entropy of X will be closer to the real one.

4.4.3.2 Step 2

let Y be the biggest available subset of X in reducer R. After choosing Y, we distribute the frequency of each projection p of Y among the projections of X that are derived from p. There may be many ways to distribute the frequencies. For instance, in the example of Step 1, if the frequency of p is 6, then the number of combinations for distributing 6 among the four projections p_1 to p_4 is equal to the solutions which can be found for the following equation: $x_1 + x_2 + x_3 + x_4 = 6$ when $x_i \geq 0$. In general, the number of ways for distributing a frequency f among n projections is equal to the number of solutions for the following equation:

$$x_1 + x_2 + ... + x_n = f \; for \; x_i \geq 0$$

Obviously, when f is higher than n, there is a lot of solutions for this equation. Among all these solutions, we choose a solution that maximizes the entropy of X. The following lemma shows how to choose such a solution.

Lemma 3. *Let $\mathcal{D}$ be a database, and X be an itemset. Then, the entropy of X over $\mathcal{D}$ is the maximum if the possible projections of X over $\mathcal{D}$ have the same frequency.*

Proof. The proof is done by implying the fact that in the entropy definition (see Definition 3), the maximum entropy is for the case where all possible combinations have the same probability. Since, the probability is proportional to the frequency, then the maximum entropy is obtained in the case where the frequencies are the same. □

The above lemma proposes that for finding an upper bound for the entropy of X (i.e., finding its maximal possible entropy), we should distribute equally (or almost equally) the frequency of each projection in Y among the derived projections in X. Let f be the frequency of a projection in Y and n be the number of its derived projections, if (f modulo n) = 0 then we distribute equally the frequency, otherwise we first distribute the quotient among the projections, and then the rest randomly.

After computing the upper bound for entropy of X, we compare it with the maximum entropy of the itemsets for which we have received all projections (so we know their real entropy), and discard X if its upper bound is less than the maximum found entropy until now.

4.4.4 Prefix/Suffix

When calculating the local *miki* on a mapper, at each step we consider a set of candidates having size j that share a prefix of size $j-1$. For instance, with the database of Table 4.2 and the subset of split S_1, the corresponding mapper will extract (A, B) as the *miki* of size 2. Then, it will build 3 candidates: (A, B, C), (A, B, D) and (A, B, E). A straightforward approach for calculating the joint entropy of these candidates would be to calculate their projection counting by means of an exhaustive scan over the data of S_1 (*i.e.*, read the first transaction of S_1, compare it to each candidate in order to find their projections, and move to the next transaction). However, these candidates share a prefix of size 2: (A, B). Therefore, we store the candidates in a structure that contains the prefix itemset, of size $j-1$, and the set of $|\mathcal{F} - j|$ suffix features. Then, for a transaction T, we only need to i) calculate $proj(p, T)$ where p is the prefix and ii) for each suffix feature f, find the projection of f on T, append $proj(f, T)$ to $proj(p, T)$ and emit the result. Let us illustrate this principle with the example above (*i.e.*, first transaction of S_1 in Table 4.2). The structure is as follows: {prefix=(A, B):suffixes=C, D, E}. With this structure, instead of comparing (A, B, C), (A, B, D) and (A, B, E) to the transaction and find their respective projections, we calculate the projection of (A, B), their prefix, *i.e.*, $(0, 0)$, and the projection of each suffix, *i.e.*, (1), (0) and (0) for C, D, and E respectively. Each suffix projection is then added to the prefix projection and emitted. In our case, we build three projections: $(0, 0, 1)$, $(0, 0, 0)$ and $(0, 0, 0)$, and the mapper will emit $((A, B, C) : (0, 0, 1))$, $((A, B, D) : (0, 0, 0))$ and $((A, B, E) : (0, 0, 0))$.

4.4.5 Incremental Entropy Computation in Mappers

In the basic version of our two rounds approach, each mapper performs many scans over its split to compute the entropy of candidates and finally find the local miki. Given k as the size of the requested itemset, in each step j of the k steps in the local miki algorithm,

the mapper uses the itemset of size $j-1$ discovered so far, and builds $|F|-j$ candidate itemsets before selecting the one having the highest entropy. For calculating each joint entropy, a scan of the input split is needed in order to compute the frequency (and thus the probability) of projections. Let $|F|$ be the number of features in the database, then the number of scans done by each mapper is $O(k * |F|)$. Although the input split is kept in memory, this high number of scans over the split is responsible for the main part of the time taken by the mappers.

In this Section, we propose an efficient approach to significantly reduce the number of scans. Our approach that incrementally computes the joint entropies, needs to do in each step just one scan of the input split. Thus, the number of scans done by this approach is $O(k)$.

To incrementally compute the entropy, our approach takes advantage of the following lemma.

Lemma 4. *Let X be an itemset, and suppose we make an itemset Y by adding a new feature i to X, i.e., $Y = X + \{i\}$. Then, for each projection p in X two projections $p_1 = p.0$, and $p_2 = p.1$ are generated in Y, and the sum of the frequency of p_1 and p_2 is equal to that of p, i.e., $f(p) = f(p_1) + f(p_2)$.*

proof. The projections of Y can be divided into two groups: 1) those that represent transactions containing i; 2) those representing the transactions that do not involve i. For each projection p_1 in the first group, there is a projection p_2 in the second group, such that p_1 and p_2 differ only in one bit, i.e., the bit that represents the feature i. If we remove this bit from p_1 or p_2, then we obtain a projection in X, say p, that represents all transactions that are represented by p_1 or p_2. Thus, for each project p in X, there are two projections p_1 and p_2 in Y generated from p by adding one additional bit, and the frequency of p is equal to the sum of the frequencies of p_1 and p_2. □

Our incremental approach for *miki* computing proceeds as follows. Let X be the miki in step j . Initially, we set $X = \{\}$, with a null projection whose frequency is equal to n, i.e., the size of the database. Then, in each step j ($1 \leq j \leq k$), we do as follows. For each remaining feature $i \in F - X$, we create a hash map $h_{i,j}$ containing all projections of the itemset $X + \{i\}$, and we initiate the frequency of each projection to zero. Then, we scan the set of transactions in the input split of the mapper. For each transaction t, we obtain a set S that is the intersection of t and $F-X$, i.e., $S = t \cap (F-X)$. For each feature $i \in S$, we obtain the projection of t over $X + \{i\}$, say p_2, and increment by one the frequency of the projection p_2 in the hash map $h_{i,j}$. After scanning all transactions of the split, we obtain the frequency of all projections ending with 1. For computing the projections ending with 0, we use Lemma 4 as follows. Let $p.0$ be a projection ending with 0, we find the projection $p.1$ (i.e., the projection that differs only in the last bit), and set the frequency of $p.0$ equal to the frequency of p minus that of $p.1$, i.e., $f(p.0) = f(p) - f(p.1)$. By this way, we compute the frequency of projections ending with 0.

After computing the frequencies, we can compute the entropy of itemset $X + \{i\}$, for each feature $i \in F - X$. At the end of each step, we add to X the feature i whose joint entropy with X is the highest. We keep the hash map of the selected itemset, and remove

all other hash maps including that of the previous step. Then, we go to the next step until finishing step k. Notice that to obtain the frequency of p in step j, we use the hash map of the previous step, i.e., $H_{i,j-1}$, this is why, at each step we keep the hash map of the selected miki.

Let us now prove the correctness of our approach using the following Theorem.

Theorem 5. *Given a database $\mathcal{D}$, and a value k as the size of requested miki. Then, our incremental approach computes correctly the entropy of the candidate itemsets in all steps.*

proof. To prove the correctness of our approach, it is sufficient to show that in each step the projection frequencies of $X + \{i\}$ are computed correctly. We show this by induction on the number of steps, i.e., j for $1 \leq j \leq k$.

Base. In the first step, the itemset $X + \{i\} = \{i\}$ because initially $X = \{\}$. There are two projections for $\{i\} : p_1 = (0)$ and $p_2 = (1)$. The frequency of p_2 is equal to the number of transactions containing i. Thus during the scan of the split, we correctly set the frequency of p_2. Since there is no other projection for i, the frequency of p_1 is equal to $n - f(p_2)$, where n is the size of the database. This frequency is found correctly by our approach. Thus, for step $j = 1$ our approach finds correctly the projection frequencies of $X + \{i\}$.

Induction. we assume that our approach works correctly in step $j - 1$, then we prove that it will work correctly in step j. The proof can be done easily by using Lemma 4. According this lemma, for each projection p in step $j - 1$ there are two projections $p_1 = (p.0)$, and $p_2 = (p.1)$ in step j. The frequency of p_2 is computed correctly during the scan of the split. We assume that the frequency of p has been correctly computed in step $j - 1$. Then, Lemma 4 implies that the frequency of p_1 has been also well computed since we have $f(p) = f(p_1) + f(p_2)$. □

4.5 Experiments

To evaluate the performance of PHIKS, we have carried out extensive experimental tests. In Section 4.5.1, we depict our experimental setup and its main configurations. In Section 4.5.2, we depict the different used data sets in our various experiments. Lastly, in Section 4.5.3, we thoroughly analyze and investigate our different experimental results.

4.5.1 Experimental Setup

We implemented PHIKS algorithm on top of Hadoop-MapReduce using Java programming language version 1.7 and Hadoop [76] version 1.0.3. For comparison, we implemented a parallel version of Forward Selection [49] algorithm. To specify each presented algorithm, we adopt the notations as follow. We denote by 'PFWS' a parallel implementation of Forward Selection algorithm, by 'Simple-PHIKS' an implementation of our basic

Data Set	# of Transactions	# of Items	Size
Amazon Reviews	34 millions	31721	34 Gigabyte
English Wikipedia	5 millions	23805	49 Gigabytes
ClueWeb	632 millions	141826	1 Terabyte

Table 4.3 – Data Sets Description

two rounds algorithm without any optimization, and by 'Prefix' an extended version of Simple-PHIKS algorithm that uses the Prefix/Suffix method for accelerating the computations of the projection values. We denote by 'Upper-B' a version of our algorithm that reduces the number of candidates by estimating the joint entropies of *miki* based on an upper bound joint entropy. We denote by 'Upper-B-Prefix' an extended version of Upper-B algorithm that employs the technique of prefix/suffix. Lastly, we denote by 'PHIKS' an improved version of Upper-B-Prefix algorithm that uses the method of incremental entropy for reducing the number of data split scans at each mapper.

We carried out all our experiments on the Grid5000 [2] platform, which is a platform for large-scale data processing. In our experiments, we have used clusters of 16 and 48 nodes respectively for Amazon Reviews, Wikipedia data sets and ClueWeb data set. Each machine is equipped with Linux operating system, 64 Gigabytes of main memory, Intel Xeon X3440 4 core CPUs and 320 Gigabytes SATA hard disk.

In our experiments, we measured three metrics: 1) the response time of the compared algorithms, which is the time difference between the beginning and the end of a maximally informative k-itemsets mining process; 2) the quantity of transferred data (i.e., between the mappers and the reducers) of each maximally informative k-itemsets mining process; 3) the energy consumption for each maximally informative k-itemsets mining process. To this end, we used the metrology API and Ganglia infrastructure of the Grid5000 platform that allow to measure the energy consumption of the nodes during an experiment.

Basically, in our experiments, we consider the different performance measurements when the size k of the itemset (*miki* to be discovered) is high.

4.5.2 Data Sets

To better evaluate the performance of PHIKS algorithm, we used three real-world data sets as described in Table 4.3. The first one is the whole 2013 Amazon Reviews data set [1], having a total size of 34 Gigabytes and composed of 35 million reviews. The second data set is the 2014 English Wikipedia data set [4], having a total size of 49 Gigabytes and composed of 5 million articles. The third data set is a sample of ClueWeb English data set [3] with size of around one Terabyte and having 632 million articles. For English Wikipedia and ClueWeb data sets, we performed a data cleaning task; we removed all English stop words from all articles, and obtained data sets where each article represents a transaction (features, items, or attributes are the corresponding words in the article). Likewise, for Amazon Reviews data set, we removed all English stop words from all

reviews. Each review represents a transaction in our experiments on Amazon Reviews data set.

4.5.3 Results

In this Section, we report the results of our experimental evaluation.

4.5.3.1 Runtime and Scalability

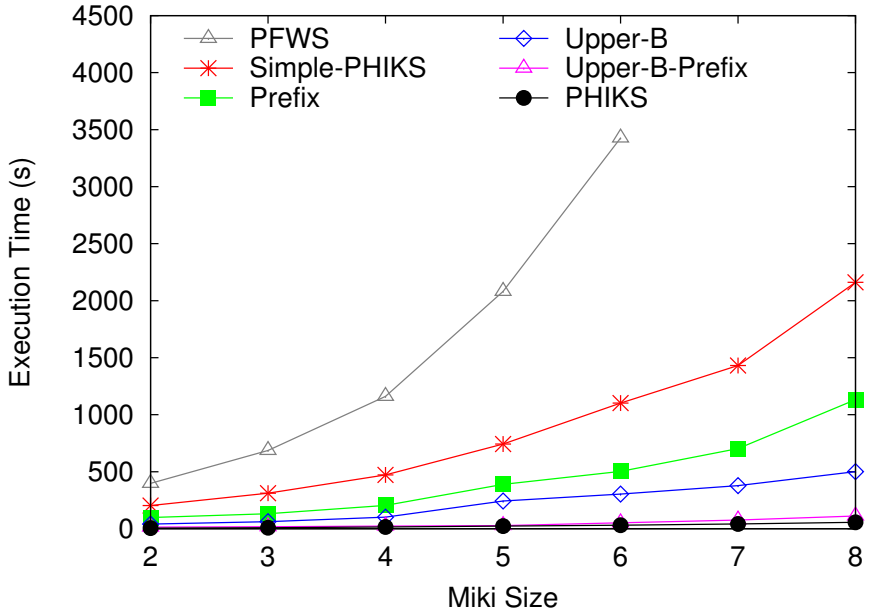
4.5.3.2 Data Communication and Energy Consumption

Figures 4.1, 4.2, and 4.3 show the results of our experiments on Amazon Reviews, English Wikipedia and ClueWeb data sets. Figures 4.1(a) and 4.1(b) give an overview on our experiments on the Amazon Reviews data set. Figure 4.1(a) illustrates the the performance of different algorithms when varying the itemset sizes from 2 to 8. We see that the response time of Forward Selection algorithm (PFWS) grows exponentially and gets quickly very high compared to other algorithms. Above a size $k = 6$ of itemsets, PFWS cannot continue scaling. This is due to the multiple database scans that it performs to determine an itemset of size k (i.e, PFWS needs to perform k MapReduce jobs). In the other hand, the performance of Simple-PHIKS algorithm is better than PFWS; it continues scaling with higher k values. This difference in the performance between the two algorithms illustrates the significant impact of mining itemsets in the two rounds architecture.

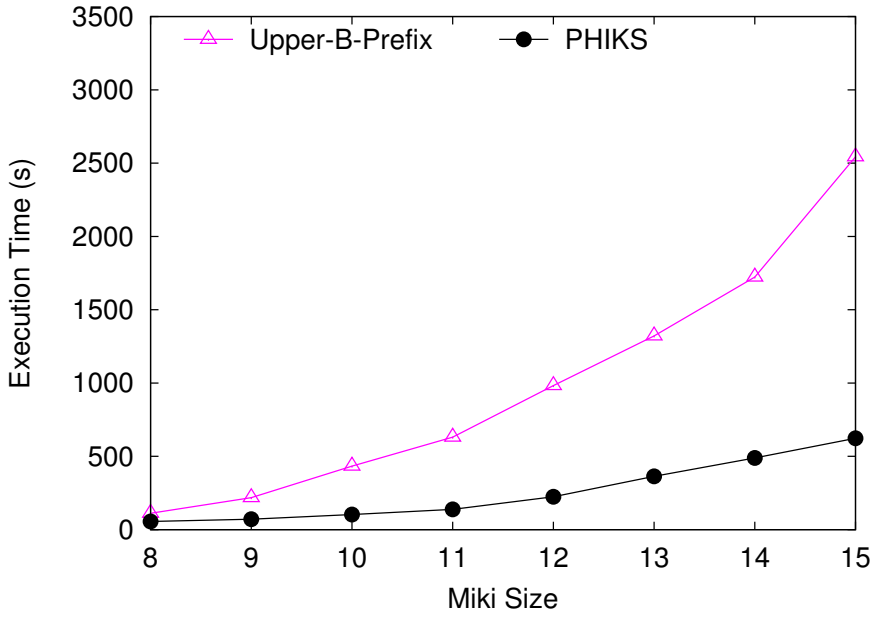
Moreover, by using further optimizing techniques, we clearly see the improvements in the performance. In particular, with an itemset having size $k = 8$, we observe a good performance behavior of Prefix comparing to Simple-PHIKS. This performance gain in the runtime reflects the efficient usage of Prefix/Suffix technique for speeding up *miki* parallel extraction. Interestingly, by estimating *miki* at the first MapReduce job, we record a very good response time as shown by Upper-B algorithm. In particular, with $k = 8$ we see that Upper-B algorithm roughly outperforms Simple-PHIKS by a factor of 3. By coupling the Prefix/Suffix technique with Upper-B algorithm, we see very good improvements in the response time, which is achieved by Upper-B-Prefix. Finally, by taking advantage of our incremental entropy technique for reducing the number of data split scans, we record an outstanding improvement in the response time, as shown by PHIKS algorithm.

Figure 4.1(b) highlights the difference between the algorithms that scale in Figure 4.1(a). Although Upper-B-Prefix continues to scale with $k = 8$, it is outperformed by PHIKS algorithm. With itemsets of size $k = 15$, we clearly observe a big difference in the response time between Upper-B-Prefix and PHIKS. The significant performance of PHIKS algorithm illustrates its robust and efficient core mining process.

Figures 4.2(a) and 4.2(b) report our experiments on the English Wikipedia data set. Figure 4.2(a) gives a complete view on the the performance of different presented algorithms when varying the itemset sizes from 2 to 8. Similarly as in Figure 4.1(a), in Figure 4.2(a) we clearly see that the execution time of Forward Selection algorithm (PFWS) is very high compared to other presented alternatives. When the itemsets size reach values

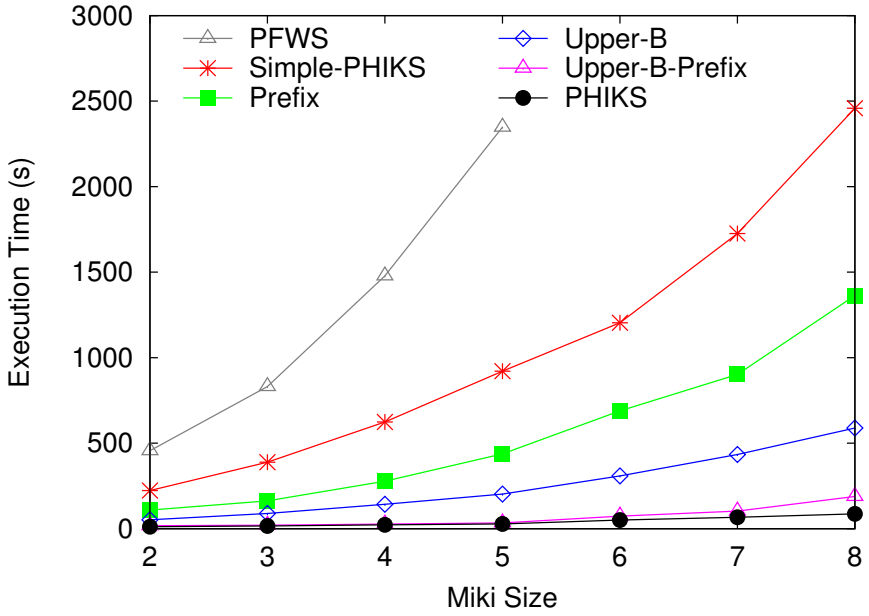


(a) All algorithms

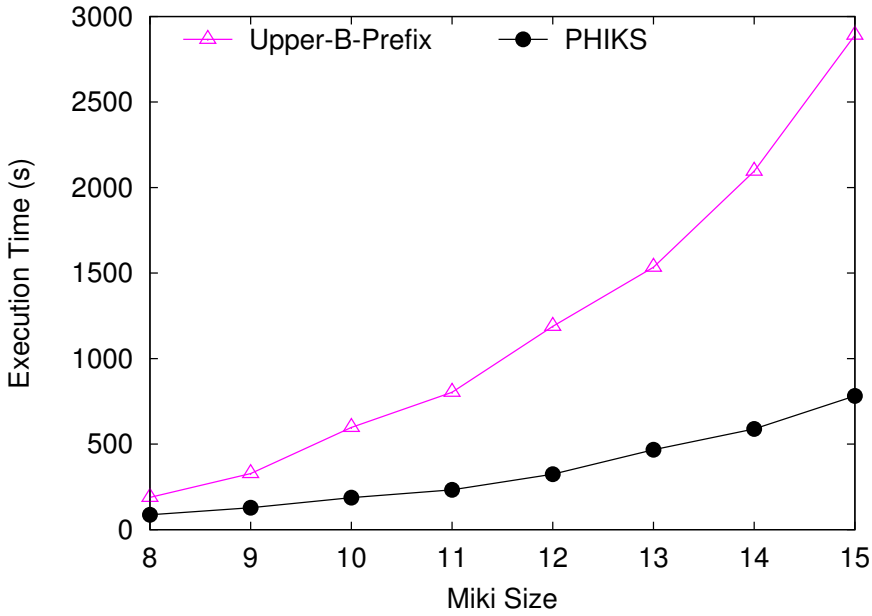


(b) Focus on scalable algorithms

Figure 4.1 – Runtime and scalability on Amazon Reviews Data Set

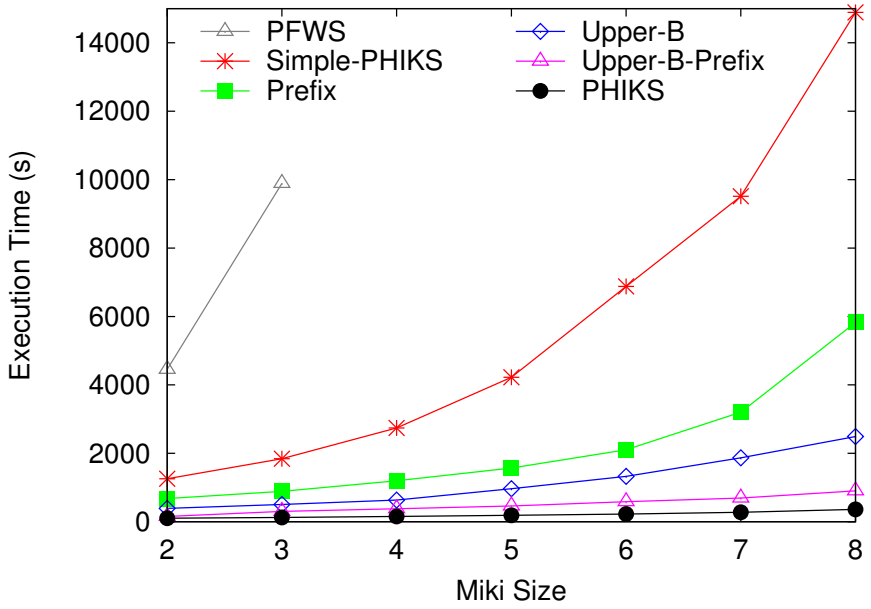


(a) All algorithms

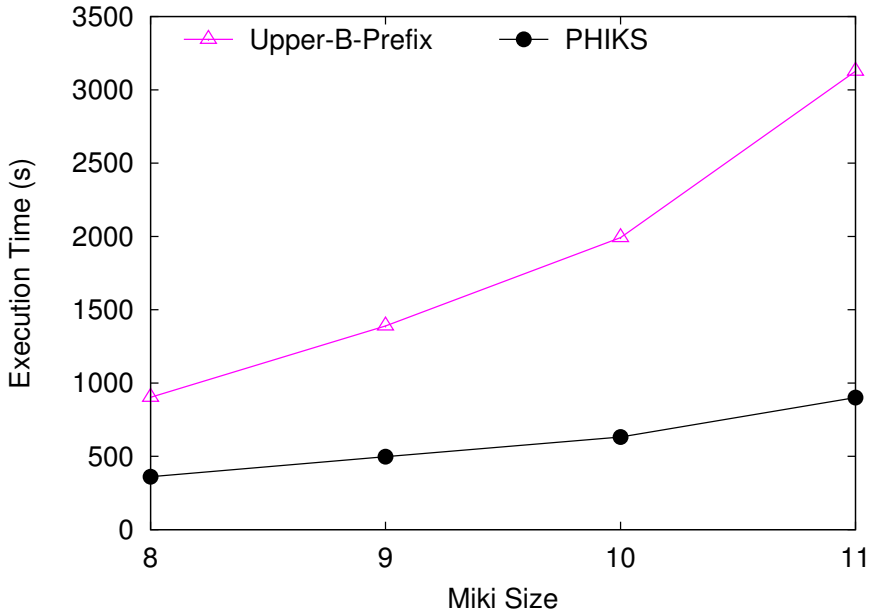


(b) Focus on scalable algorithms

Figure 4.2 – Runtime and scalability on English Wikipedia Data Set

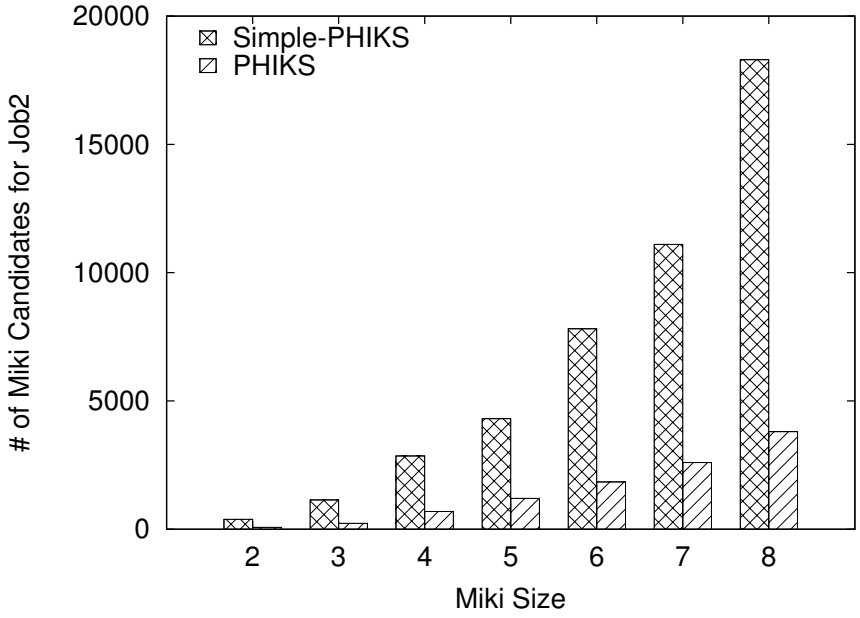


(a) All algorithms

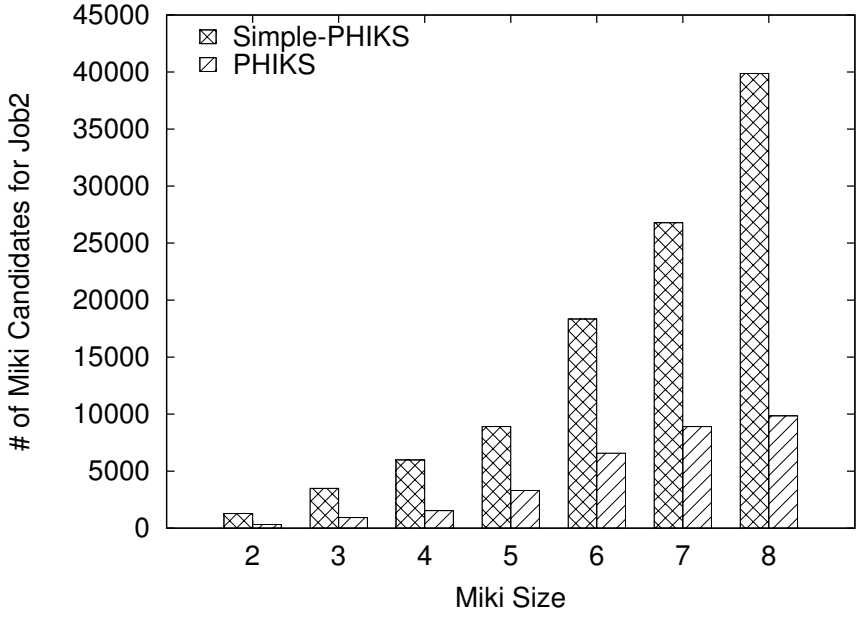


(b) Focus on scalable algorithms

Figure 4.3 – Runtime and scalability on ClueWeb Data Set

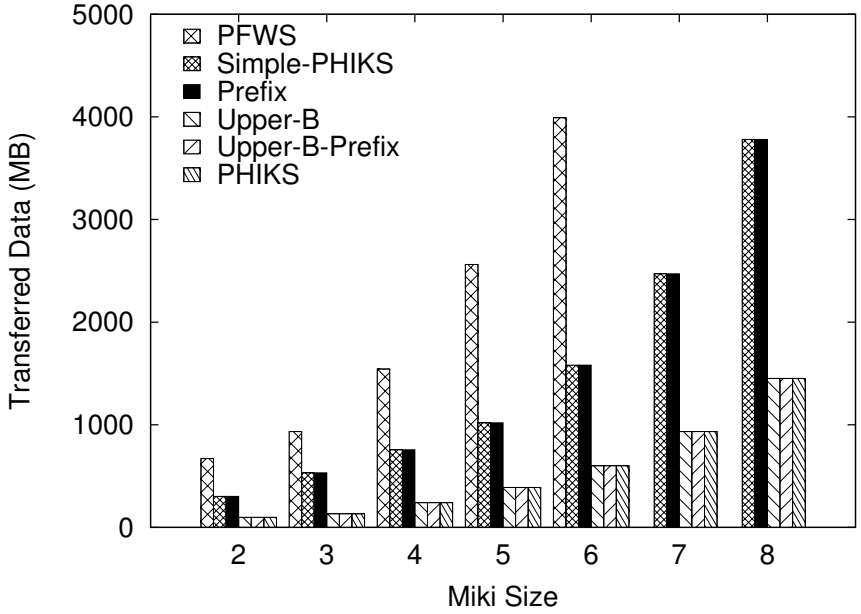


(a) Wikipedia data set

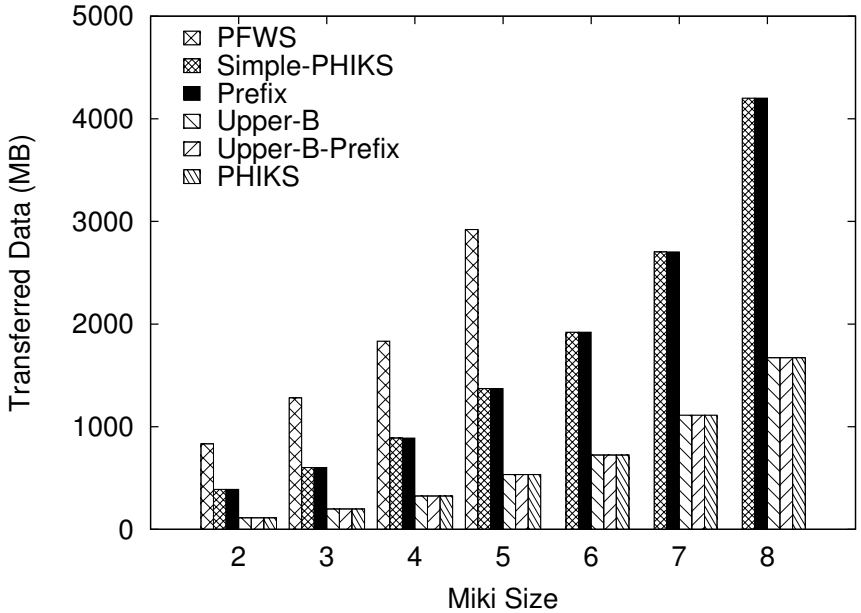


(b) ClueWeb data set

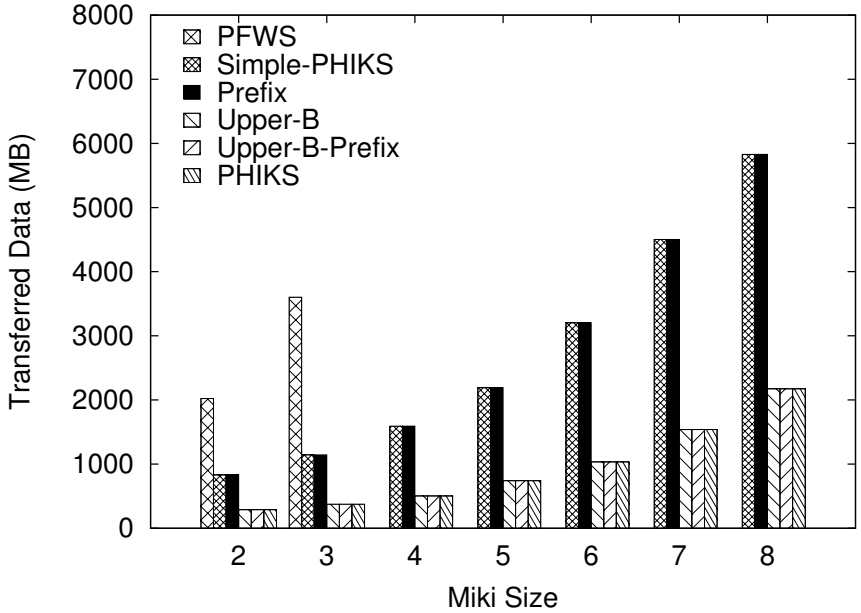
Figure 4.4 – Candidate Pruning



(a) Amazon Reviews data set



(b) Wikipedia data set



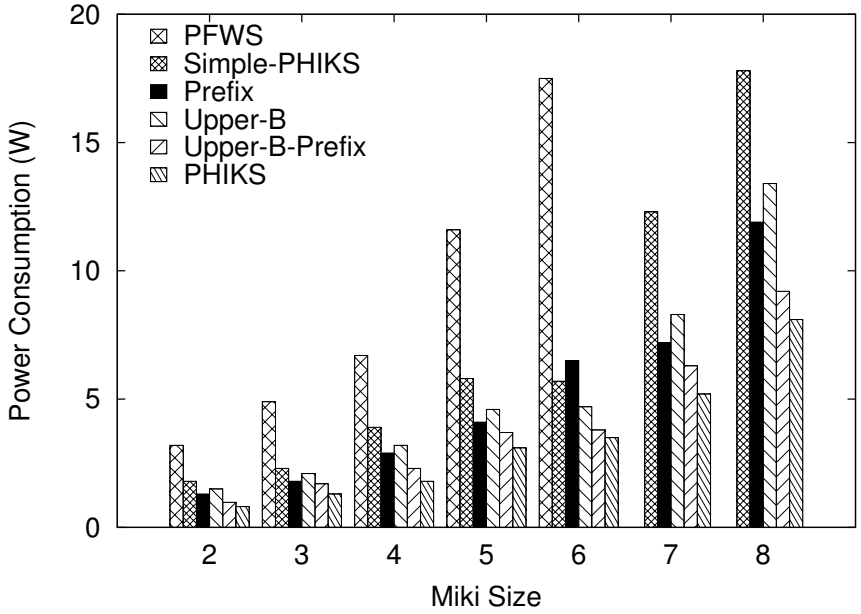
greater than $k = 5$, PFWS stops scaling. In the other side, we observe that Simple-PHIKS algorithm continues scaling and gives better performance than PFWS.

Performing more optimization, we significantly speed up the *miki* extraction. Specifically, with itemsets size $k = 8$, we see that the performance of Prefix is better than Simple-PHIKS. This difference in the performance behavior between the two algorithms explains the high impact of using Prefix/Suffix technique to speed up the whole mining process of the parallel *miki* extraction. By going on for further optimization using our efficient heuristic technique for estimating *miki* at the first MapReduce job, we get a significant improvement in the execution time as shown by Upper-B algorithm. Particularly, with itemsets size $k = 8$ we clearly see that Upper-B algorithm performance is better than Simple-PHIKS. By using Prefix/Suffix technique with Upper-B algorithm, we record a significant improvement in the performance as shown by Upper-B-Prefix. Eventually, based on our efficient technique of incremental entropy, we record a very significant performance improvement as shown by PHIKS algorithm.

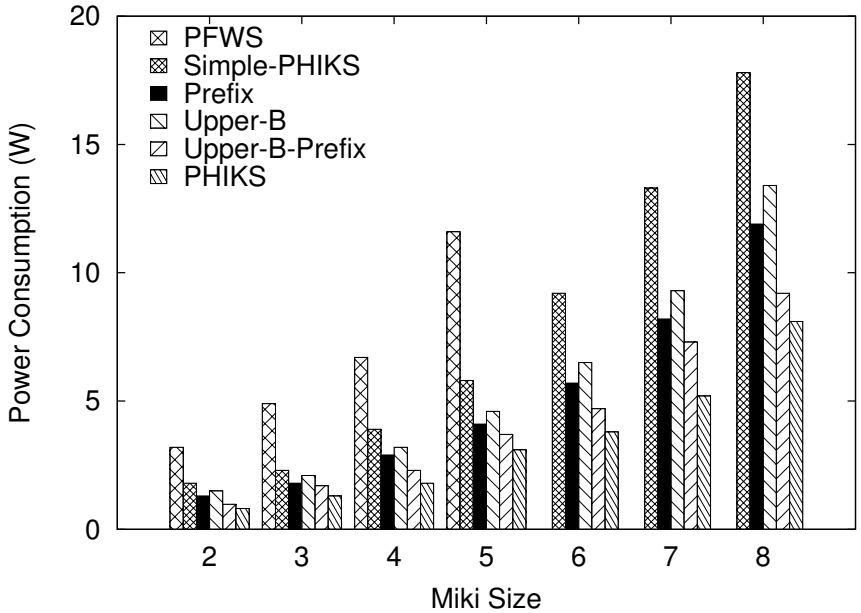
Figure 4.2(b) illustrates the difference between the algorithms that scale in Figure 4.2(a). Despite the scalability recorded by Upper-B-Prefix when $k = 8$, Upper-B-Prefix gives very less performance compared to PHIKS algorithm. In particular, with higher itemsets size (e.g., $k = 15$), we record a large difference in the execution time between Upper-B-Prefix and PHIKS algorithms. This difference in the performance between the two algorithms reflects the efficient and robust core mining process of PHIKS algorithm. In Figures 4.3(a) and 4.3(b), similar experiments have been conducted on the ClueWeb data set. We observe that the same order between all algorithms is kept compared to Figures 4.1(a), 4.1(b), 4.2(a) and 4.2(b). In particular, we see that PFWS algorithm suffers from the same limitations as could be observed on the Amazon Reviews and Wikipedia data sets in Figure 4.1(a) and Figure 4.2(a) . With an itemset size of $k = 8$, we clearly observe a significant difference between PHIKS algorithm performance and all other presented alternatives. This difference in the performance is better illustrated in Figure 4.3(b). By increasing the size k of *miki* from 8 to 11, we observe a very good performance of PHIKS algorithm. Although, Upper-B-Prefix algorithm scales with $k = 11$, it is outperformed by PHIKS.

4.5.3.3 *miki* Candidates Pruning

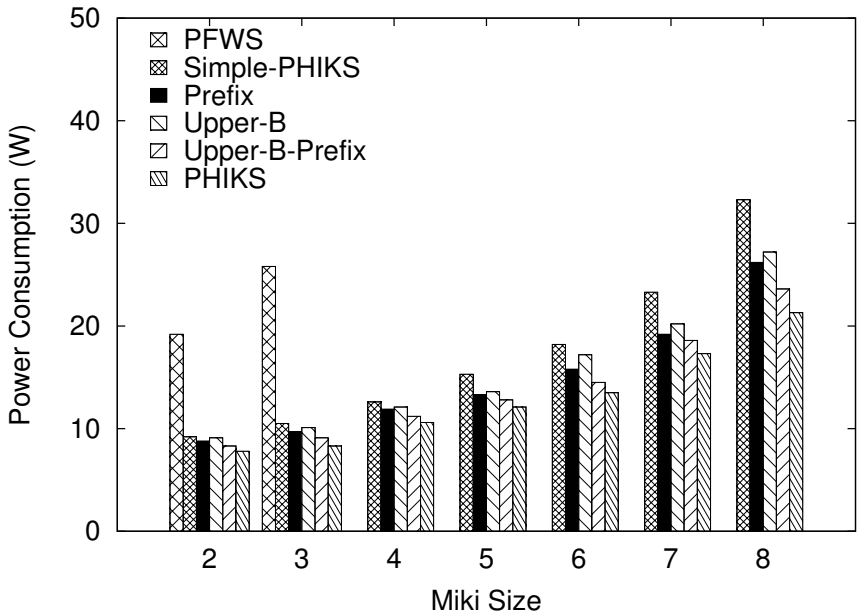
Figure 4.4 gives a complete overview on the total number of *miki* candidates being tested at the second MapReduce job for both Simple-PHIKS and PHIKS algorithms. Figure 4.4(a) illustrates the number of *miki* candidates being validated at the first MapReduce job on the Wikipedia data set. By varying the parameter size k of itemsets from 2 to 8, we observe a significant difference in the number of *miki* candidates being sent by each algorithm to its second MapReduce job. With $k = 8$, Simple-PHIKS algorithm sends to its second job roughly 6 times more candidates than PHIKS. This important reduction in the number of candidates to be tested in the second job is achieved due to our efficient technique for estimating the joint entropies of *miki* with very low upper bounds. Likewise, in Figure 4.4(b), we record a very good performance of PHIKS comparing



(a) Amazon Reviews data set



(b) Wikipedia data set



to Simple-PHIKS. This outstanding performance of Simple-PHIKS algorithm reflects its high capability and its effectiveness for a very fast and successful *miki* extraction.

Figure 4.5 gives an entire view of the quantity of transferred data (in megabyte) over the network by each presented algorithm on the three data sets. Respectively Figures 4.5(a), 4.5(b) and 4.5(c) show the performance of each presented maximally informative k-itemsets mining process on Amazon Reviews, English Wikipedia and ClueWeb data sets. In all figures, we observe that PFWS algorithm has the highest peak. This is due to its multiple MapReduce jobs executions. In the other hand, we see that Simple-PHIKS and Prefix algorithms have smaller peaks. This is because Simple-PHIKS and its optimized Prefix version algorithm (for fast computation of the local entropies at the mappers) rely on two MapReduce jobs whatever the *miki* size to be discovered. We see that Upper-B, Upper-B-Prefix and PHIKS outperform all other presented algorithms in terms of transferred data. This is due to the impact of estimating the joint entropies at their first MapReduce job which reduces the number of *miki* candidates (i.e., data) being tested at their second MapReduce job.

We also measured the energy consumption (in Watt) of the compared algorithms during their execution. We used the Grid5000 [2] tools that measure the power consumption of the nodes during a job execution. Figure 4.6 shows the total amount of the power consumption of each presented maximally informative k-itemsets mining process on Amazon Reviews, English Wikipedia and ClueWeb data sets. In Figures 4.6(a), 4.6(b) and 4.6(c) we observe that the energy consumption increases when increasing the size k of the *miki* to be discovered for each algorithm. We see that PHIKS still gives a lower consumption comparing to other presented algorithms. This is simply due to the higher optimizations in its core mining process. Actuelly the smaller number of candidates being tested during the second MapReduce job of PHIKS calls for a lower number of I/O access when computing the entropies. All of these different factors make PHIKS consumes less energy compared to other presented algorithms.

4.6 Related Work

In data mining literature, several endeavors have been made to explore informative itemsets (or featuresets, or set of attributes) in databases [7] [39] [41] [49]. Different measures of itemset informativeness (e.g., frequency of itemset co-occurrence in the database etc.) have been used to identify and distinguish informative itemsets from non-informative ones. For instance, by considering the itemsets co-occurrence, several conclusions can be drawn to explain interesting, hidden relationships between different itemsets in the data.

Mining itemsets based on the co-occurrence frequency (e.g., frequent itemset mining) measure does not capture all dependencies and hidden relationships in the database, especially when the data is sparse [41]. Therefore, other measures must be taken into account. Low and high entropy measures of itemsets informativeness were proposed [41]. The authors of [41] have proposed the use of a tree based structure without specifying a length k of the informative itemsets to be discovered. However, as the authors of [41] mentioned,

such an approach results in a very large output.

Beyond using a regular co-occurrence frequency measure to identify the itemsets informativeness, the authors of [14] have proposed an efficient technique that is more general. The main motivation is to get better insight and understanding of the data by figuring out other hidden relationships between the itemsets (i.e., the inner correlation between the itemsets themselves), in particular when determining the itemsets' rules. To this end, the authors of [14] did not focus only on the analysis of the positive implications between the itemsets in the data (i.e., implications between supported itemsets), but also they take into account the negative implications. To determine the significance of such itemsets implications, the authors of [14] have used a classic statistical chi-squared measure to efficiently figure out the interestingness of such itemsets rules.

Generally, in the itemset mining problem there is a trade-off between the itemset informativeness and the pattern explosion (i.e., number of itemsets to be computed). Thus, some itemset informativeness measures (e.g., the co-occurrence frequency measure with very low minimum support) would allow for a potential high number of useless patterns (i.e., itemsets), and others would highly limit the number of patterns. The authors of [73] proposed an efficient approach that goes over regular used itemset informativeness measures, by developing a general framework of statistical models allowing the scoring of the itemsets in order to determine their informativeness. In particular, in [73], the initial focus is on the exponential models to score the itemsets. However these models are inefficient in terms of execution time, thus, the authors propose to use decomposable models. On the whole, the techniques proposed in [73] and [14] are mainly dedicated to mining in centralized environments, while our techniques are dedicated to parallel data mining in distributed environments.

The authors of [49] suggest to use a heuristic approach to extract informative itemsets of length k based on maximum joint entropy. Such maximally informative itemsets of size k is called *miki*. This approach captures the itemsets that have high joint entropies. An itemset is a *miki* if all of its constructing items shatter the data maximally. The items within a *miki* are not excluding, and do not depend on each other. [49] proposes a bunch of algorithms to extract *miki*. A brute force approach consists of performing an exhaustive search over the database to determine all *miki* of different sizes. However, this approach is not feasible due to the large number of itemsets to be determined, which results in multiple database scans. Another algorithm proposed in [49] consists of fixing a parameter k that denotes the size of the *miki* to be discovered. This algorithm proceeds by determining a top n *miki* of size 1 having highest joint entropies, then, the algorithm determines the combinations of *1-miki* of size 2 and returns the top n most informative itemsets. The process continues until it returns the top n *miki* of size k.

The problem of extracting informative itemsets was not only proposed for mining static databases. There have been also interesting works in extracting informative itemsets in data streams [33] [74]. The authors of [81] proposed an efficient method for discovering maximally informative itemsets (i.e., highly informative itemsets) from data streams based on sliding window.

Extracting informative itemsets has a prominent role in feature selection

[21]. Various techniques and methods have been proposed in the literature to solve the problem of selecting relevant features to be used in classification tasks. These methods fall into two different categories, namely *Filter* and *Wrapper* methods [38]. Filter methods serve to pre-process the data before being used for a learning purpose. They aim to determine a small set of relevant features. However, these methods capture only the correlations between each feature (i.e., independent variable, attribute or item) and the target class (i.e., predictor). They do not take into account the inter correlation between the selected features (i.e., if the features are inter correlated then they are redundant). In the other hand, to determine an optimal set of relevant features, wrapper methods perform a feature's set search that maximizes an objective function (i.e., classifier performance). However, these methods yield in heavy computations (i.e., selecting each time a set of features and evaluate an objective function). To solve this problem, *Embedded* [21] methods have been proposed. The main goal is to incorporate the wrapper methods in the learning process.

Processing very large amount of data with high number of features (i.e., billions of items) to select most relevant once's is not trivial. *Miki* are very good challenging candidates to select relevant features in large-scale databases. The items (i.e., features) that a set of *miki* contains have very low inter correlations, thus they highly discriminate the whole database together. This property of *miki* make them of a highly potential success in improving different data mining tasks such as subgroup discovery [43] and classification [12] problems etc.

Parallel mining of informative itemsets from large databases based on frequency informativeness measure has received much attention recently [52] [65] [72]. For instance, In [52], the authors have proposed an efficient parallel solution to extract frequent itemsets based on FP-Growth algorithm [39]. Their algorithm PFP-Growth has gained a popular success in mining large databases. In [65], the authors have proposed an improvement of PFP-Growth algorithm by performing approximations of the candidate frequent itemsets.

To the best of our knowledge, there has been no prior work on parallel discovery of maximally informative k-itemsets in large databases.

4.7 Context of the work

This work has been done in the context of Saber Salah's PhD thesis, co-supervised by me and Florent Masseglia in INRIA Zenith team. In addition to the miki problem, Saber worked on parallel frequent itemset mining using MapReduce [90].

4.8 Conclusion

In this chapter, we presented an efficient parallel maximally informative k-itemset mining algorithm namely PHIKS that has shown significant efficiency, in terms of runtime, communication cost and energy consumption. PHIKS elegantly determines the *miki* in very large databases with at most two rounds. PHIKS comes with a bunch of optimizing

techniques that renders the *miki* mining process very fast. These techniques concern the architecture at a global scale, but also the computation of entropy on distributed nodes, at a local scale. The result is a fast and efficient discovery of *miki* with high itemset size. Such ability to use high itemset size is mandatory when dealing with Big Data and particularly one Terabyte like what we have done in our experiments. Our results show that PHIKS algorithm outperforms other alternatives by several orders of magnitude, and makes the difference between an inoperative and a successful *miki* extraction.

Chapter 5

Perspectives

Up to now, I contributed significantly in the development of scalable data analytics techniques by taking advantage of the computing power of distributed and parallel systems and frameworks such as Spark. Our techniques can analyze very large datasets, *e.g.*, terabytes of data [87, 86, 84, 89].

I plan to continue my research activities in the direction of *large scale data analytics*. Below, I present some research topics, which I am interested to explore in the next few years.

5.1 Parallel Time Series Indexing using GPUs

In the past, we developed parallel techniques for indexing time series datasets using parallel frameworks such as Spark. These frameworks allow us to index very big datasets, e.g., billions of time series. However, our solutions are not easy to use for people who are not familiar with distributed systems. For example, deploying Spark in a large-scale cluster needs highly-qualified technical staff.

Therefore, we plan to develop parallel indexing solutions for GPUs, which can be easily installed and used. For example, in the context of a collaboration with scientists from IRSTEA, we plan to develop parallel time series indexes for the Chemo-metrics field by means of GPUs. The Chemo-metrics scientists are interested in knowledge extraction from spectral data, which can be represented as time series. Partial Least Squares (PLS) regression is a technique widely used in Chemo-metrics to transform a spectra data into useful information. However, the current PLS algorithms are centralized, and their response time becomes very high when the size of the spectra database gets higher than 10,000 tuples. We plan to adapt our DPiSAX and ParCorr solutions for being used in spectra analysis. The parallel solutions should be implemented using GPUs. An important step is to find the optimized parameters for DPiSAX and ParCorr techniques in order to obtain high precision PLS results, while reducing the response time significantly.

5.2 Parallel All-Pairs Similarity Search over Time Series

One of the important types of similarity search in time series is the all-pairs-similarity-search (or similarity join). The problem is the following: given a collection of time series slices, return the nearest neighbor for each slice. To efficiently answer similarity join queries, Yeh et al. [78] proposed the *matrix profile index*. Intuitively, given a time series A and a subsequence size m, the matrix profile $P[i] = j$ if the jth subsequence of time series A is the most similar to the ith subsequence. Such an index is very useful for similarity detection in very large time series.

In the context of a collaboration with Safran (a high-technology group and supplier of systems and equipment in the Aircraft, Aerospace and Defense markets.), we plan to develop parallel matrix profile solutions for analyzing their data collected during the test procedure of their produced engines. For example, during the tests of a helicopter engine, engineers typically capture more than 400 channels of data from temperature, pressure, velocity, etc., generating a total of 5,000 different files. Analyzing this data will help the engineers to detect anomalies and optimize the engines.

We plan to build matrix profile indexes over their multi-dimensional time series. This type of index can be very useful for Safran's underlying applications. For example, the anomalies can be detected in the matrix profile as high value points (because of their high distance to other slices). Matrix profile is also efficient for detecting motifs in time series (the motifs are shown as low value points in the matrix profile). The motif detection can help the engineers to optimize the engines.

5.3 Privacy Preserving Data Analytics in Distributed Systems

There are many scenarios in which the users or organizations could get major benefits by performing analytics on their data shared in a distributed system. However, a lot of users hesitate to share their sensitive data because of privacy attacks risks. According to a recent report published by the Cloud Security Alliance [24], privacy attacks are one of the main concerns for cloud users. Thus, it is important to develop distributed data analysis solutions that preserve users privacy.

To provide such solutions, we plan to capitalize on our recent work on privacy-preserving query processing [91, 92, 93]. One solution is to take advantage of the differential privacy (DP) technique [27] that guarantees the privacy of individual data by perturbing the published data with a controlled amount of noise, usually generated by using the Laplace distribution. The added noise, if well chosen, will disappear in the results of the aggregate query. DP is the strongest privacy technique that makes no assumption about the adversary background knowledge. However, a naive utilization of this technique may lead to important loss of precision in the result. This is why well choosing the privacy and accuracy metrics is fundamental.

Another solution is to encrypt the user data, and develop distributed data analytics algorithms working on encrypted data. There are different techniques that allow to design such algorithms. For example, in a recent work [92], we took advantage of the bucketization technique for elaborating efficient algorithms for processing top-k queries over encrypted data in distributed systems.

Chapter 6

Bibliography - Part 1

[1] Amazon. http://snap.stanford.edu/data/web-Amazon-links.html.

[2] Grid5000. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home.

[3] The clueweb09 dataset. http://www.lemurproject.org/clueweb09.php/, 2009.

[4] English wikipedia articles. http://dumps.wikimedia.org/enwiki/latest, 2014.

[5] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, 2003.

[6] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. Efficient similarity search in sequence databases. In *Proceedings of the International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 69–84. Springer-Verlag, 1993.

[7] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[8] Rajaraman Anand. *Mining of massive datasets*. Cambridge University Press, New York, N.Y. Cambridge, 2012.

[9] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The ts-tree: Efficient time series search and retrieval. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 252–263, 2008.

[10] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The ts-tree: efficient time series search and retrieval. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 252–263, 2008.

[11] Klaus Berberich and Srikanta Bedathur. Computing n-gram statistics in mapreduce. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 101–112, 2013.

[12] Michael Berry. *Survey of text mining II clustering, classification, and retrieval*. Springer, New York London, 2008.

[13] Christian Bizer, Peter A. Boncz, Michael L. Brodie, and Orri Erling. The meaningful use of big data: four perspectives - four challenges. *SIGMOD Rec.*, 40(4):56–60, 2011.

[14] Sergey Brin, Rajeev Motwani, and Craig Silverstein. Beyond market baskets: Generalizing association rules to correlations. *SIGMOD Rec.*, 26(2):265–276, June 1997.

[15] Yuhan Cai and Raymond Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 599–610. ACM, 2004.

[16] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. iSAX 2.0: Indexing and mining one billion time series. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 58–67, 2010.

[17] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. *Knowledge and Information Systems (KAIS)*, 39:123–151, 2014.

[18] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. *Knowledge and Information Systems (KAIS)*, 39(1):123–151, 2014.

[19] Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Transactions on Database Systems (TODS)*, 27(2):188–228, June 2002.

[20] Kin-pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 126–133. IEEE Computer Society, 1999.

[21] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Computers and Electrical Engineering*, 40(1):16 – 28, 2014.

[22] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 380–388, 2002.

[23] Richard Cole, Dennis Shasha, and Xiaojian Zhao. Fast window correlations over uncooperative time series. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 743–749, 2005.

[24] Cameron Coles and John Yeoh. Cloud adoption practices and priorities survey report. Technical report, Cloud Security Alliance report, January 2015.

[25] T. M. Cover. *Elements of information theory*. Wiley-Interscience, Hoboken, N.J, 2006.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[27] Cynthia Dwork. Differential privacy. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1–12, 2006.

[28] Philippe Esling and Carlos Agon. Time-series data mining. *ACM Comput. Surv.*, 45(1):12:1–12:34, December 2012.

[29] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23(2):419–429, 1994.

[30] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 419–429, 1994.

[31] F. Geerts, B. Goethals, and T. Mielikäinen. Tiling databases. In *International Conference on Discovery Science*, pages 278–289, 2004.

[32] Zoubin Ghahramani. Unsupervised learning. In *Advanced Lectures on Machine Learning*, pages 72–112, 2004.

[33] Chris Giannella, Jiawei Han, Jian Pei, Xifeng Yan, and Philip S. Yu. Mining frequent patterns in data streams at multiple time granularities, 2002.

[34] A. Gionis, H. Mannila, and J.K. Seppänen. Geometric and combinatorial tiles in 0–1 data. In *Knowledge Discovery in Databases (PKDD)*, pages 173–184, 2004.

[35] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 518–529, 1999.

[36] Robert Gray. *Entropy and information theory*. Springer, New York, 2011.

[37] Ed Greengrass. Information retrieval: A survey, 2000.

[38] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, mar 2003.

[39] Han, Pei, and Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29, 2000.

[40] Jiawei Han. *Data mining : concepts and techniques*. Elsevier/Morgan Kaufmann, 2012.

[41] Hannes Heikinheimo, Eino Hinkkanen, Heikki Mannila, Taneli Mielikäinen, and Jouni K. Seppänen. Finding low-entropy sets and trees from binary data. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 350–359, 2007.

[42] A. Henelius, I. Karlsson, P. Papapetrou, A. Ukkonen, and K. Puolamäki. Semigeometric tiling of event sequences. In *Machine Learning and Knowledge Discovery in Databases. ECML PKDD*, pages 329–344, 2016.

[43] Franciso Herrera, CristóbalJosé Carmona, Pedro González, and MaríaJosé del Jesus. An overview on subgroup discovery: foundations and applications. *Knowledge and Information Systems*, 29(3):495–525, 2011.

[44] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 189–197, 2000.

[45] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *ICASSP* , 2011.

[46] W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in Modern Analysis and Probability*, volume 26 of *Contemporary Mathematics*, pages 189–206, 1984.

[47] Eamonn J. Keogh. Exact indexing of dynamic time warping. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.

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

[49] Arno J. Knobbe and Eric K. Y. Ho. Maximally informative k-itemsets and their efficient discovery. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 237–244, 2006.

[50] S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. In *Proceedings of International Conference on Emerging Artificial Intelligence Applications in Computer Engineering*, pages 3–24, 2007.

[51] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC)*, pages 614–623, 1998.

[52] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proceedings of the ACM Conf. on Recommender Systems (RecSys)*, pages 107–114, 2008.

[53] J. Lin, E. Keogh, S. Lonardi, and B. Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2003.

[54] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing sax: A novel symbolic representation of time series. *Data Min. Knowl. Discov.*, 15(2):107–144, 2007.

[55] Yasuko Matsubara and Yasushi Sakurai. Regime shifts in streams: Real-time forecasting of co-evolving time sequences. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1045–1054, 2016.

[56] Iris Miliaraki, Klaus Berberich, Rainer Gemulla, and Spyros Zoupanos. Mind the gap: Large-scale frequent sequence mining. In *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 797–808, 2013.

[57] S. Moens, E. Aksehirli, and B. Goethals. Frequent itemset mining for big data. In *IEEE International Conference on Big Data*, pages 111–118, 2013.

[58] Abdullah Mueen, Suman Nath, and Jie Liu. Fast approximate correlation for massive time-series data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 171–182, 2010.

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

[60] Themis Palpanas. Big sequence management: A glimpse of the past, the present, and the future. In *SOFSEM*, 2016.

[61] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. Streaming pattern discovery in multiple time-series. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 697–708, 2005.

[62] Spiros Papadimitriou and Philip S. Yu. Optimal multi-scale patterns in time series streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 647–658, 2006.

[63] Chang-Shing Perng, Haixun Wang, and Sheng Ma. Fast relevance discovery in time series. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 1016–1020, 2006.

[64] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2012.

[65] Matteo Riondato, Justin A. DeBrabant, Rodrigo Fonseca, and Eli Upfal. Parma: a parallel randomized algorithm for approximate association rules mining in mapreduce. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 85–94, 2012.

[66] Yasushi Sakurai, Christos Faloutsos, and Masashi Yamamuro. Stream monitoring under the time warping distance. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1046–1055, 2007.

[67] Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 432–444, 1995.

[68] D. Shasha and Y. Zhu. *High Performance Discovery in Time series, Techniques and Case Studies*. Springer, 2004.

[69] J. Shieh and E. Keogh. isax: Indexing and mining terabyte sized time series. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 623–631, 2008.

[70] J. Shieh and E. Keogh. isax: Disk-aware mining and indexing of massive time series datasets. *Data Min. Knowl. Discov.*, 19(1):24–57, 2009.

[71] Jin Shieh and Eamonn Keogh. iSAX: Indexing and mining terabyte sized time series. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 623–631, 2008.

[72] S.K. Tanbeer, C.F. Ahmed, and Byeong-Soo Jeong. Parallel and distributed frequent pattern mining in large databases. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 407–414, 2009.

[73] Nikolaj Tatti. Probably the best itemsets. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 293–302, 2010.

[74] Wei-Guang Teng, Ming-Syan Chen, and Philip S. Yu. A regression-based temporal pattern mining scheme for data streams. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 93–104, 2003.

[75] Yang W., Peng W., Jian P., Wei W., and Sheng H. A data-adaptive and dynamic segmentation index for whole matching on time series. *PVLDB*, 6(10):793–804, 2013.

[76] Tom White. *Hadoop : the definitive guide*. O'Reilly, 2012.

[77] Qing Xie, Shuo Shang, Bo Yuan, Chaoyi Pang, and Xiangliang Zhang. Local correlation detection with linearity enhancement in streaming data. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 309–318, 2013.

[78] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn J. Keogh. Matrix profile I: all pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 1317–1322, 2016.

[79] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[80] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conf. on Hot Topics in Cloud Computing*, pages 10–10, 2010.

[81] Chongsheng Zhang and Florent Masseglia. Discovering highly informative feature sets from data streams. In *Database and Expert Systems Applications*, pages 91–104. 2010.

[82] K Zoumpatianos, S Idreos, and T Palpanas. Indexing for interactive exploration of big data series. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 1555–1566, 2014.

[83] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. ADS: the adaptive data series index. *VLDB J.*, 25(6):843–866, 2016.

Chapter 7

Bibliography - Part 2 (author's references)

[84] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. Massively distributed time series indexing and querying. In *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2019.

[85] Oleksandra Levchenko, Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Boyan Kolev, and Dennis E. Shasha. Spark-parsketch: A massively distributed indexing of time series datasets. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1951–1954, 2018.

[86] Djamel Edine Yagoubi, Reza Akbarinia, Boyan Kolev, Oleksandra Levchenko, Florent Masseglia, Patrick Valduriez, and Dennis E. Shasha. Parcorr: efficient parallel methods to identify similar time series pairs across sliding windows. *Data Mining and Knowledge Discovery (DMKD)*, 32(5):1481–1507, 2018.

[87] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. Dpisax: Massively distributed partitioned isax. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 1135–1140, 2017.

[88] Saber Salah, Reza Akbarinia, and Florent Masseglia. A highly scalable parallel algorithm for maximally informative k-itemset mining. *Knowledge and Information Systems (KAIS)*, 50(1):1–26, 2017.

[89] Saber Salah, Reza Akbarinia, and Florent Masseglia. Fast parallel mining of maximally informative k-itemsets in big data. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 359–368, 2015.

[90] Saber Salah, Reza Akbarinia, and Florent Masseglia. Data placement in massively distributed environments for fast parallel mining of frequent itemsets. *Knowledge and Information Systems (KAIS)*, 53(1):207–237, 2017.

[91] Cetin Sahin, Tristan Allard, Reza Akbarinia, Amr El Abbadi, and Esther Pacitti. A differentially private index for range query processing in clouds. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, pages 208–216, 2018.

[92] Sakina Mahboubi, Reza Akbarinia, and Patrick Valduriez. Privacy-preserving top-k query processing in distributed systems. In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 281–292, 2018.

[93] Sakina Mahboubi, Reza Akbarinia, and Patrick Valduriez. Answering top-k queries over outsourced sensitive data in the cloud. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, pages 218–231, 2018.

[94] Miguel Liroz-Gistau, Reza Akbarinia, Divyakant Agrawal, and Patrick Valduriez. Fp-hadoop: Efficient processing of skewed mapreduce jobs. *Information Systems*, 60:69–84, 2016.

[95] Miguel Liroz-Gistau, Reza Akbarinia, and Patrick Valduriez. Fp-hadoop: Efficient execution of parallel jobs over skewed data. *Proceedings of the VLDB Endowment (PVLDB)*, 8(12):1856–1859, 2015.

[96] Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fábio Porto, and Patrick Valduriez. Dynamic workload-based partitioning algorithms for continuously growing databases. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 12:105–128, 2013.

[97] Maximilien Servajean, Reza Akbarinia, Esther Pacitti, and Sihem Amer-Yahia. Profile diversity for query processing using user recommendations. *Information Systems*, 48:44–63, 2015.

[98] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for efficient top-k query processing. *Information Systems*, 36(6):973–989, 2011.

[99] William Kokou Dedzoe, Philippe Lamarre, Reza Akbarinia, and Patrick Valduriez. ASAP top-k query processing in unstructured P2P systems. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 1–10, 2010.

[100] Mounir Tlili, William Kokou Dedzoe, Esther Pacitti, Patrick Valduriez, Reza Akbarinia, Pascal Molli, Gérôme Canals, and Stéphane Laurière. P2P logging and timestamping for reconciliation. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1420–1423, 2008.

[101] Reza Akbarinia, Mounir Tlili, Esther Pacitti, Patrick Valduriez, and Alexandre A. B. Lima. Replication in dhts using dynamic groups. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 3:1–19, 2011.

[102] Naser Ayat, Reza Akbarinia, Hamideh Afsarmanesh, and Patrick Valduriez. Entity resolution for probabilistic data. *Information Sciences*, 277:492–511, 2014.

[103] Naser Ayat, Reza Akbarinia, Hamideh Afsarmanesh, and Patrick Valduriez. Entity resolution for distributed probabilistic data. *Distributed and Parallel Databases (DAPD)*, 31(4):509–542, 2013.

[104] Reza Akbarinia and Florent Masseglia. Fast and exact mining of probabilistic data streams. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 493–508, 2013.

[105] Reza Akbarinia, Patrick Valduriez, and Guillaume Verger. Efficient evaluation of SUM queries over probabilistic data. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 25(4):764–775, 2013.

[106] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for top-k queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 495–506, 2007.

[107] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Processing top-k queries in distributed hash tables. In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 489–502, 2007.

[108] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Reducing network traffic in unstructured P2P systems using top-*k* queries. *Distributed and Parallel Databases (DAPD)*, 19(2-3):67–86, 2006.

[109] William Kokou Dedzoe, Philippe Lamarre, Reza Akbarinia, and Patrick Valduriez. As-soon-as-possible top-k query processing in P2P systems. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 9:1–27, 2013.

[110] Wenceslao Palma, Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Dhtjoin: processing continuous join queries using DHT networks. *Distributed and Parallel Databases (DAPD)*, 26(2-3):291–317, 2009.

[111] Wenceslao Palma, Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Distributed processing of continuous join queries using DHT networks. In *Proceedings of the EDBT/ICDT Workshops*, pages 34–41, 2009.

[112] Wenceslao Palma, Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Efficient processing of continuous join queries using distributed hash tables. In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 632–641, 2008.

[113] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Data currency in replicated dhts. In *Proceedings of International Conference on Management of Data (SIGMOD)*, pages 211–222, 2007.