Maximally Informative k-Itemset Mining from Massively Distributed Data Streams

Mehdi Zitouni, Reza Akbarinia, Sadok Ben Yahia, Florent Masseglia

Maximally Informative k-Itemset Mining from Massively Distributed Data Streams

Mehdi Zitouni*
Inria and LIRMM, Montpellier, France
Université de Tunis ElManar, Faculté des sciences de Tunis, LIPAH-LR11ES Tunis, Tunisia
Mehdi.Zitouni@inria.fr

Reza Akbarinia
Inria and LIRMM, Montpellier, France
reza.akbarinia@inria.fr

Sadok Ben Yahia
Université de Tunis ElManar, Faculté des sciences de Tunis, LIPAH-LR11ES Tunis, Tunisia
Sadok.Benyahia@fst.rnu.tn

Florent Masseglia
Inria and LIRMM, Montpellier, France
florent.masseglia@inria.fr

ABSTRACT

We address the problem of mining maximally informative k-itemsets (*miki*) in data streams based on joint entropy. We propose *PentroS*, a highly scalable parallel *miki* mining algorithm. *PentroS* renders the mining process of large volumes of incoming data very efficient. It is designed to take into account the continuous aspect of data streams, particularly by reducing the computations of need for updating the *miki* results after arrival/departure of transactions to/from the sliding window. *PentroS* has been extensively evaluated using massive real-world data streams. Our experimental results confirm the effectiveness of our proposal which allows excellent throughput with high itemset length.

1. INTRODUCTION

Pattern mining [1] is a core data mining operation and has been extensively studied over the last decade. Recently, mining informative patterns over big data has attracted research interests [2]. Compared with other big data queries, informative pattern mining poses great challenges due to high memory and computational costs, as well as accuracy requirement of the mining results. Such patterns can be itemsets, sequences, sub-trees, or sub-graphs, depending on the mining tasks and targeting datasets. It has important differences with frequent itemsets mining (FIM) [3]. Indeed, the latter puts focus on discovering frequently occurring patterns from different types of datasets, including unstructured ones such as transaction and text datasets, semi-structured ones such as XML datasets, and structured ones such as graph datasets. However, in data analysis [4], frequency of itemsets can not always be an effective measure to give relevant results for a various range of applications, including information retrieval [5]. Indeed, the informativeness of an itemset could result in discovering interesting new patterns that were not previously known.

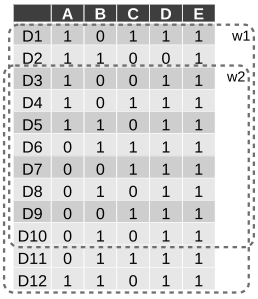


Figure 1: An example of data stream, with records comprised of features. Here, the records are documents, and their features the words they contain. We define on the stream an observation sliding window w of size 10 and a sliding interval of size 2. At the initialization of the stream, the first window ($w1$) contains the first 10 documents. After one update, the new window ($w2$) contains the most up to date 10 documents.

As stated by information theory [6], the informativeness of one pattern may be calculated through its joint entropy. Therefore, the pattern having the highest joint entropy embeds the highest information about the objects in the dataset. Such a pattern is called Maximally Informative k-itemset (*miki* in short) of length k [7]. *Miki* extraction has been shown to be of interest in many potential applications, for example it can serve as a basic tool for data mining tasks including classification, clustering, and change detection. Unfortunately, existing *miki* selection algorithms were designed towards static data. Therefore, *miki* mining over data streams becomes an important research topic with big challenges. The problem is to extract the *miki* over dynamic and very large amount of incoming data, taking into account the evo-

*The research leading to these results has received funding from the European UnionâĂŹs Horizon 2020 - The EU Framwork Program for Research and Innovation 2014-2020, under grant No. 732051.
This work has been performed in the context of the Computational Biology Institute (www.ibc-montpellier.fr)

lution of the data streams over time. Example 1 illustrates a use case of *miki* to retrieve a set of documents over a data stream.

EXAMPLE 1. *Consider similarity queries in high-dimensional datasets. In this case, we are interested in only using a small subset of the dimensions (or features) for fast record comparisons. Figure 1 represents a set of features A, B, C, D, E contained in a stream of documents* $\{d_1, \ldots, d_{12}\}$ *arriving at different time points. In this data, "1" means that the word occurs in the corresponding document, and "0" otherwise. Since a data stream cannot fit into main memory, a usual approach is to consider an observation time window that concerns the most up to date data. This is* $w1$ *and* $w2$ *in Figure 1. At each step, we focus on the data of the latest window only.* $w1$ *is the first window, at the initialization of the stream.* $w2$ *is the second one, available after to updates in the stream, and so on. Let us now consider* $w1$. $\{D, E\}$ *is a frequent itemset over* $w1$*, but this information actually provides little help for similarity queries. Given a document q in our data streams, and based on its values of D and E, we will not be able to decide which document is the closest to q. In the meanwhile, the itemset* $\{A, B, C\}$ *is infrequent on* $w1$*, but much more helpful for this task. With the values {1, 0, 0} (resp. {0, 1, 1}) we could find the corresponding document* d_3 *(resp.* d_6*).* $\{A, B, C\}$ *is called a maximally informative itemset (miki) of size* $k = 3$ *over this stream of documents. Our goal is to discover the most up to date miki, continuously and after each update in the stream. In the case of Figure 1, this is still* $\{A, B, C\}$ *in* $w2$*, but this might not be the case after a few updates.*

In the last few years, some new *miki* algorithms have been developed [8, 2]. However, to the best of our knowledge, there is no efficient solution in the literature for parallel miki discovery over data streams.

For *miki* mining in data streams, we have to address the following challenges. First, a *miki* mining algorithm needs to explore a search space with an exponential number of candidates. The length of the temporary answer set itself can be very large. Thus, in a streaming environment, even generating an approximate answer set can cost much more space than the available one. Therefore, the mining algorithm needs to be very memory efficient. Second, the computations become more challenging in presence of high speed data, since we have to quickly extract results and efficiently manage fast sliding window shifts that may affect *miki* candidates.

In this paper, we propose a parallel solution that deals with the above challenges. We exploit the Spark Streaming framework [9] and propose a clever combination of both information theory and massive distribution principles. We propose a new fast parallel algorithm for computing streaming entropies of *miki*, called *PentroS*, intended to discover *miki* over data streams, in massively distributed environments. In *PentroS*, we propose optimizing strategies to maximize the parallelism and to take into account the continuous aspect of the data streams. Particularly, we propose approaches that incrementally update the *miki* results after arrival and departure of transactions to/from the distributed sliding windows. Our algorithm has been extensively evaluated using large scale real-world streaming data. The experimental results show that *PentroS* allows considerable performance gains compared to baseline approaches, and confirm its high throughput on large continuous streams generated from a real world dataset of one Terabyte.

The remainder of this paper is organized as follows. Section 2 gives the formal definitions of informative itemsets and the necessary background. In Section 3, we propose our *PentroS* algorithm. Section 4 reports our experimental results over real-world transactional data streams. Section 5 discusses related work, and Section 6 concludes.

2. BACKGROUND

In this section, we formally define the problem, which we address and sketch the Spark Streaming system as the massively distributed streaming environment.

2.1 Data Streams

In a data stream, transactions arrive continuously and the volume of transactions can be potentially infinite. Formally, a data stream D can be defined as a sequence of transactions, $D = (t_1, t_2, ..., t_i, ...)$, where t_i is the i^{th} arrived transaction. To process and mine data streams, different window models are often used. A window is a sub-sequence between i^{th} and j^{th} arrived transactions, denoted as $W[i, j] = (t_i, t_{i+1}, ..., t_j)$ with $i < j$. A user can ask different types of pattern mining questions over different types of window models. The most popular type is the sliding window[10]. Given a sliding window of size w, and a current time point t, we are interested in the continuous pattern discovery in the window $W[t - w + 1, t]$. As time changes, the window keeps its size and moves along with the current time point.

2.2 Miki

Below, we define the problem of mining *mikis* (maximally informative k-itemsets) [7].

DEFINITION 1. *Let* $\mathcal{F} = \{i_1, i_2, \ldots, i_n\}$ *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 dataset* $\mathcal{T}$ *is a set of transactions.*

DEFINITION 2. *The entropy of a feature i in a dataset* $\mathcal{T}$ *measures the expected amount of information needed to specify the state of uncertainty or disorder for the feature i in* $\mathcal{T}$*. Let i be a feature in* $\mathcal{T}$*, and* $P(i = n)$ *be the probability that i has value n in* $\mathcal{T}$ *(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 [6].*

DEFINITION 3. *The binary projection, or projection of an itemset X in a transaction t is a bitmap of size* $|X|$ *where each item (i.e., feature) of X is replaced by '1' if it occurs in t and by '0' otherwise. Given a dataset* $\mathcal{T}$*, then* $D(\mathcal{T}_X)$ *is the set of all projections of X to the transactions of* $\mathcal{T}$*. The projection counting of X in a dataset* $\mathcal{T}$ *is finding the projections of X in* $\mathcal{T}$*, and associating each projection with its number of occurrences.*

EXAMPLE 2. *Let us consider Figure 1. The projection of* $\{B, C, D\}$ *in* d_1 *is* $(0, 1, 1)$*. The projections of* $\{D, E\}$ *in Figure 1 are* $(1, 1)$ *with eleven occurrences and* $(0, 1)$ *with one occurrence.*

DEFINITION 4. *Given an itemset* $X = \{x_1, x_2, \ldots, x_k\}$ *and a set of transactions* $\mathcal{T}$*, the joint entropy of* X *in* $\mathcal{T}$ *is defined as:* $H(\mathcal{T}, X) = -\sum_{J \in \{0,1\}^{|k|}} P(J) \times log(P(J))$*, where each* J *is a possible projection of* X *in the transactions, and* $P(J)$ *is the probability of this projection.*

The higher the value of $H(\mathcal{T}, X)$, the more information X provides in $\mathcal{T}$. For simplicity, we use the term entropy of an itemset X to denote its joint entropy.

EXAMPLE 3. *Let us consider the dataset of Figure 1. The joint entropy of* $X = \{D, E\}$ *on* $w1$ *is given by* $H(\mathcal{T}, X) = -\frac{9}{10}log(\frac{9}{10}) - \frac{1}{10}log(\frac{1}{10}) = 0.468$*, where the quantities* $\frac{9}{10}$ *and* $\frac{1}{10}$ *respectively represent the probabilities of the projection values* $(1, 1)$ *and* $(0, 1)$ *in the dataset. Note that there are two projections for* $\{D, E\}$ *in the dataset:* $(1, 1)$ *with nine occurrences , and* $(0, 1)$ *with one occurrence.*

DEFINITION 5. *Given a dataset* $\mathcal{T}$*, a set* $\mathcal{F} = \{i_1, i_2, \ldots, i_n\}$ *of features. An itemset* $X \subseteq \mathcal{F}$ *of length* k *is said to be maximally informative* k*-itemset, if for all itemsets* $Y \subseteq \mathcal{F}$ *of size* k*,* $H(\mathcal{T}, Y) \leq H(\mathcal{T}, X)$*. Hence, a maximally informative* k*-itemset is the itemset of size* k *with the highest joint entropy value.*

Now, we can define the problem of mining maximally informative k-itemsets, in a static dataset, as follows.

DEFINITION 6. *Given a dataset* $\mathcal{T}$ *which consists of a set of* n *features* $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$*. Given a number* k*, the problem of miki mining is to return a subset* $\mathcal{F}' \subseteq \mathcal{F}$ *with size* k*, i.e.,* $|\mathcal{F}'| = k$*, having the highest joint entropy in* $\mathcal{T}$*, i.e.,* $\forall \mathcal{F}'' \subseteq \mathcal{F} \wedge |\mathcal{F}''| = k \Rightarrow H(\mathcal{T}, \mathcal{F}'') \leq H(\mathcal{T}, \mathcal{F}')$*.*

2.3 Forward-Selection

In [7], a heuristic algorithm, called *Forward-Selection*, has been proposed to extract informative itemset of size k with the highest entropy. The *Forward-Selection* algorithm performs multiple scans over the dataset in k iterations. In the first iteration, the algorithm chooses the item with the maximum entropy. Then in each iteration j (for $2 \leq j \leq k$), the remaining n features are added to the current itemset to generate n candidates. The candidate itemset having the highest entropy is found by means of dataset scans, and is declared as the new *miki* of size j. The authors of [7] showed the advantage of *Forward-Selection* over the brute force algorithms that evaluate the entropies of all the possible subsets of size k.

In a data stream environment, a naive approach for miki mining would be a straightforward implementation of *Forward-Selection* (see Section 2.3) repeatedly on multiple sets of transactions. The idea is simple, the transaction data stream is divided into multiple batches (stored in RDDs) which arrive at different time points and on each RDD, we mine the corresponding *miki*. Clearly, the corresponding generating and pruning principles lead to very bad performances and are not suited for distributed environment like Spark [9] and MapReduce [11]. Worse still, since it is a transaction data streams, patterns could be truncated by batches and connections between batches are cut, thus the obtained *miki* result may not be correct.

In this respect, we propose an algorithm to extract *mikis* correctly from continuous data in a massively distributed environment, introducing a new incremental principles for updating the *miki* results over a data stream.

3. DISTRIBUTED MIKI MINING

Our approach starts with a complete *miki* discovery, from scratch, on the first sliding window, at the initialization of the process. This discovery from scratch might also be done at regular points in the stream whenever incremental computation is not possible (details given in Section 3.1). This initial *miki* discovery proceeds in two major rounds. In the first round, it computes the local *miki* on each split of the sliding window. Then, it considers the union of all the local *mikis* as a set of candidates to be checked over the global distributed sliding window in the second round. To perform the first round, for a given sliding window $\mathcal{SW}$ at a time point t in the stream, presented as an RDD containing all the transactions arriving between t_i and t_j with $i < j$, we apply the principles of *Forward-Selection* in parallel on each split of the RDD. The straightforward approach would be to centralize the local *mikis* obtained in the first round, and hope to find the global *miki* among this set in the second round.

However, this heuristic is optimistic since it considers that the global *miki* will appear in at least one split. Actually, it is possible that the global *miki* is never found as a local *miki* in the first round[1]. This is why, in the second round, we need a larger number of candidate itemsets, in order to maximize the chances to obtain the actual *miki*. This can be done by exploiting the set of candidates that are built, locally on each split, in the first round. The last step of *Forward-Selection* aims at calculating the projection counting of $|\mathcal{F}| - k$ candidates and then computing their local entropy. Instead of only considering the itemset having the highest entropy, we will emit to an RDD, for each candidate X, its projection counting in the split. The new RDD will include, for each local candidate X_i ($1 \leq i \leq m$, where m is the number of splits), the projection counting of X in a subset of $\mathcal{T}$.

Doing so, the chances to obtain the actual global *miki* in the second round are higher, but it is still possible that a local candidate X has not been proposed in the entire set of splits in the first round. Consider, for example, a biased data distribution, where a split contains some features with high entropies, and these features have low entropies on the other splits. Then, X is proposed in some splits and not the other ones. Therefore, for each candidate X obtained in the first round, we have two possible cases:

1. X is a candidate itemset on all the splits, so we have its projections in all the splits, and we are able to calculate

[1]The illustration supporting this claim is omitted due to lack of space, however it is easy to show a counter-example.

its exact projection counting on $\mathcal{T}$.

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 $\mathcal{T}$.

In the first case, we have collected all the necessary information for calculating the entropy of X on $\mathcal{T}$ in the second round with no further data scans. 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, in the second round, we need to check the entropy of X on $\mathcal{T}$ by means of a new distributed data scan in order to compute its exact projection counting. The goal of this second round is therefore to check whether no local candidate has been ignored at the global scale. At the end of this round, we have the respective entropies of all the promising candidate itemsets and we are able to pick the one with the highest entropy.

To compute the global projection counting of a candidate in the splits of the sliding window, we proceed as follows. Let W be a sliding window. When W is divided into multiple splits, we have to count for each projection p of an itemset X, its corresponding number of occurrences over the entire W. To do so, in a first step, we start by emitting X with its projection p from each split of W (done using the $flatMap$ transformation in Spark Streaming). Then, after the inclusion tests of the projection over transactions, we count the total number of occurrences in all splits. Afterwards, we compute the global counting of the projection (in Spark, this is done by means of the $reduceByKey$ transformation).

The above mentioned approach is the basic version of *PentroS* (without optimizations), and is referred to in the remainder as *Basic-PentroS*. As shown by our experiments reported in Section 4, *Basic-PentroS* outperforms *Forward-Selection* over sliding windows. However, by exploiting some concepts of information theory, and proposing new entropy computing principles, we significantly improve its performance. Our goal is to gain more than one order of magnitude, by further speeding up its execution at different parts of the process, as explained in the remainder. It is worth of mention that though *PentroS* is implemented here in Spark Streaming, our generic theoretical contribution can be used for fast incremental computation of itemsets'entropies in any distributed data streaming environment.

3.1 Streaming Principle

After the initialization step described above, *PentroS* continuously evaluates the entropies of *miki* candidates of size j ($1 \leq j < k$). At any point in the data stream, we have the entropies of all the *miki* candidates of size j obtained at the previous step. However, we cannot guarantee that the *miki* over the current window is actually in the set of candidates obtained at the previous step. Therefore, in order to maintain a reliable result, *PentroS* adopts the following strategy. After an update in the sliding window, if the itemsets of size j having the highest entropy do not change then the *miki* does not change between the previous window and the current one; otherwise, we will generate new candidates. In other words, if for any value of j the local *miki* of size j in a split has changed (*i.e.*, a candidate of size j has became the new *miki* of size j) then we apply *Forward-Selection* on this split, starting from size j to k. The following sections are dedicated to i) fast incremental entropy computation strategies applied to updating the entropy of the current *miki* of size 1 to k; and ii) optimized candidate enumeration in the case we need to re-compute the current *miki*, starting from any size j.

3.2 Incremental Entropy Computation

If we locally applied *Forward-Selection* after each update in the data stream, the algorithm would perform many scans over the sliding windows to compute the entropy of candidates and to find the local *mikis*. Actually, let k be the size of the requested itemset, and $|F|$ be the number of features in the dataset, the local cost of applying *Forward-Selection* on a split is given by the product of the number of scans and the number of candidates at each scan, *i.e.*, $O(k \times |F|)$.

This high complexity degrades the performance of local *miki* generation. Therefore, in this section, we propose an efficient approach to significantly reduce the cost of updating candidate entropies. Actually, our approach only needs a unique operation (thus $O(1)$) to update the entropy of an itemset whether a transaction is added to (or removed from) the data stream. It relies on new principles for incremental entropy computation that will allow extremely efficient updates on the entropy of an itemset.

These principles are detailed with Theorems 3.2 and 3.3 that facilitate the joint entropy computations. First, we need to reformulate the computation of the entropy of an itemset through Lemma 3.1.

LEMMA 3.1. *Given an itemset X and a dataset $\mathcal{T}$ with size n. Let $D(\mathcal{T}_X)$ be the set of projections of X in the dataset $\mathcal{T}$. For each projection $t \in D(\mathcal{T}_X)$, let f_t be the frequency of t in the dataset. Then, the joint entropy of X in $\mathcal{T}$ can be computed as:*

$$H(\mathcal{T}, X) = log(n) - \frac{1}{n} \sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t) \qquad (1)$$

Proof. The total number of projections of X in the dataset $\mathcal{T}$ is n. For each projection $t \in D(\mathcal{T}_X)$, let f_t be the frequency of t in the dataset. Thus, the probability of t is $p(t) = \frac{f_t}{n}$. Therefore, we have:

$$H(\mathcal{T}, X) = -\sum_{t \in D(\mathcal{T}_X)} \frac{f_t}{n} \times log(\frac{f_t}{n}) = -\frac{1}{n} \sum_{t \in D(\mathcal{T}_X)} f_t \times log(\frac{f_t}{n})$$

We know that $log\frac{a}{b} = log(a) - log(b)$. Thus, we have:

$$\begin{aligned} H(\mathcal{T}, X) &= -\frac{1}{n}\Big(\sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t) - \sum_{t \in D(\mathcal{T}_X)} f_t \times log(n)\Big) \\ &= -\frac{1}{n}\Big(\sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t) - log(n) \times \sum_{t \in D(\mathcal{T}_X)} (f_t)\Big) \end{aligned}$$

The sum of the frequencies of the projections in $D(\mathcal{T}_X)$ is equal to the total number of transactions in $\mathcal{T}$, i.e. n. In

other words, we have $\sum_{t \in D(\mathcal{T}_X)} f_t = n$. Thus, $H(\mathcal{T}, X)$ can be simplified as:

$$H(\mathcal{T}, X) = -\frac{1}{n}(\sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t) - log(n) \times n)$$
$$= log(n) - \frac{1}{n}(\sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t)) \square$$

3.2.1 Entropy Computation After the Arrival of a new Transaction

By introducing Theorem 3.2, based on our previous Lemma, we propose a very fast computation of the entropy of an intemset after the arrival of a new transaction to the sliding window.

THEOREM 3.2. *Given an itemset X and two datasets $\mathcal{T}$ and $\mathcal{T}' = \mathcal{T} \cup \{t'\}$. Then, the joint entropy of X in $\mathcal{T}'$ can be computed by using the joint entropy of X in $\mathcal{T}$, and the frequency of $t' \cap X$ in $\mathcal{T}'$, denoted as $f_{t'}$, as follows:*

1. *if $f_{t'} = 1$, then*
   $H(\mathcal{T} \cup \{t'\}, X) = log(n+1) - \frac{n}{n+1}(log(n) - H(\mathcal{T}, X))$
2. *if $f_{t'} > 1$, then*
   $H(\mathcal{T} \cup \{t'\}, X) = log(n+1) - \frac{1}{n+1}(n(log(n) - H(\mathcal{T}, X)) + f_{t'} \times log(f_{t'}) - (f_{t'} - 1) \times log(f_{t'} - 1)$

Proof. Using Lemma 3.1, the joint entropy of X in $\mathcal{T}'$ can be written as:

$$H(\mathcal{T}', X) = log(n+1) - \frac{1}{n+1} \sum_{t \in D(\mathcal{T}'_X)} f_t \times log(f_t) \quad (2)$$

Let $t' \cap X$ be the intersection of the new transaction t' and X, and $f_{t'}$ the frequency of $t' \cap X$ in the new dataset $\mathcal{T}'$. Let $\mathcal{T}$ be the old dataset, i.e. the dataset before arrival of t'. Thus, we have $\mathcal{T}' = \mathcal{T} \cup \{t'\}$. In our proof, we consider two cases : 1) $t' \cap X$ exists in $\mathcal{T}$, thus $f_{t'} > 1$; 2) $t' \cap X$ doesn't exist in $\mathcal{T}$, thus $f_{t'} = 1$. In the case where $t' \cap X \in \mathcal{T}$, for updating the entropy for the new dataset $\mathcal{T}'$, we have to remove the old frequency of $t' \cap X$ (i.e., $f_{t'} - 1$) from the entropy formula, and replace it by the new frequency (i.e., $f_{t'}$). Thus, Equation 2 can be written as:

$$H(\mathcal{T}', X) = log(n+1) - \frac{1}{n+1}(\sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t) + f_{t'} \times log(f_{t'}) - (f_{t'} - 1) \times log(f_{t'} - 1))$$

From Lemma 3.1, we have :

$$\sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t) = n \times (log(n) - H(\mathcal{T}, X)) \quad (3)$$

Thus, in this case, the joint entropy of X in $\mathcal{T}'$ can be written as:

$$H(\mathcal{T}', X) = log(n+1) - \frac{1}{n+1}(n \times (log(n) - H(\mathcal{T}, X)) + f_{t'} \times log(f_{t'}) - (f_{t'} - 1) \times log(f_{t'} - 1))$$

In the second case, where $t' \cap X \notin \mathcal{T}$, we can write Equation 2 as follows:

$$H(\mathcal{T}', X) = log(n+1) - \frac{1}{n+1}(\sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t) + f_{t'} \times log(f_{t'}))$$

Since $f_{t'} = 1$ and $log(1) = 0$, we can simplify the above equation as:

$$H(\mathcal{T}', X) = log(n+1) - \frac{1}{n+1}(\sum_{t \in D(\mathcal{T}_X)} f_t \times log(f_t))$$

Therefore, by using Equation 3, the above equation can be written as:

$$H(\mathcal{T}', X) = log(n+1) - \frac{1}{n+1}(n \times (log(n) - H(\mathcal{T}, X))) \square$$

By using Theorem 3.2, after the arrival of a new transaction t' to the sliding window, the entropy of a candidate X can be computed simply by using its last entropy (before the arrival of t') and the frequency of $t' \cap X$ (i.e., the intersection of the new transaction and X) in the sliding window. This significantly reduces the cost of updating the candidate entropies in the sliding windows.

3.2.2 Entropy Computation After Departure of a Transaction

The second challenge in entropy computation appears when removing transactions from the data stream. Indeed, the entropy of an itemset candidate may change when a transaction gets out of the current sliding window. To maintain the correctness of our results with the transactions that leave the sliding window, we propose the following theorem.

THEOREM 3.3. *Given an itemset X, a dataset $\mathcal{T}$, and a transaction $t' \in \mathcal{T}$. Then, the joint entropy of X in $\mathcal{T}' = \mathcal{T} - \{t'\}$ can be computed by using the joint entropy of X in $\mathcal{T}$, and the frequency of $t' \cap X$ in $\mathcal{T}'$, denoted as $f_{t'}$, as follows:*

1. *if $f_{t'} = 0$, then*
   $H(\mathcal{T} - \{t'\}, X) = log(n-1) - \frac{n}{n-1}(log(n) - H(\mathcal{T}, X))$
2. *if $f_{t'} > 0$, then*
   $H(\mathcal{T} - \{t'\}, X) = log(n-1) - \frac{1}{n-1}(n(log(n) - H(\mathcal{T}, X)) + f_{t'} \times log(f_{t'}) - (f_{t'} + 1) \times log(f_{t'} + 1))$

The proof can be done in a similar way as that of Theorem 3.2.

By using the above theorems, we can update the candidate entropies just by taking into account their intersection with the added/removed transactions.

3.3 Reducing the Number of Candidates

The theoretical framework, proposed in the previous subsection, allows us to update an itemset entropy with very high efficiency. However, there are still cases where we need to send candidates to a global entropy counting in the second round of PentroS. Unfortunately, computing the entropies of all *miki* candidates might result in low response time. This is particularly the case i) for large sliding windows, as it will be illustrated by our experiments in Section 4 and ; ii) when the features are not uniformly distributed in the splits of RDDs.

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*. To do so, we exploit the available information about the *miki* candidates flat-mapped to the second form of the RDD.

Let us describe formally our approach. Let X be a partially sent itemset, and P be a partition that has not sent X and its projection frequencies to the transformed partition $P^{'}$ that is responsible for computing the entropy of X. In $P^{'}$, the frequency of X projections for a part of the dataset is missing, i.e., in the split of P. We call them *missing* frequencies. We compute an upper bound for the entropy of X by estimating its missing frequencies. This is done in two steps: i) finding the biggest subset of X, say Y, for which all frequencies are available; ii) distributing the frequencies of Y among the projections of X in such a way that the entropy of X is maximized.

To do so, 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 dataset is equal to the total number of times that the four projections p_1 to p_4 occur. In the second step, let Y be the largest available subset of X in the new partition $P^{'}$. 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 the first step, 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 solutions for distributing a frequency f among n projections may be huge.

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 3.4. *Let $\mathcal{T}$ be a dataset, and X be an itemset. Then, the entropy of X in $\mathcal{T}$ is the maximum if the possible projections of X in $\mathcal{T}$ have the same frequency.*

Proof. The proof is done by implying the fact that in the entropy definition (see Definition 2), the maximum entropy is obtained 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 remainder randomly.

After computing the upper bound for entropy of X in each sliding window that we handle, we compare it with the maximum entropy of the itemsets for which we have received all projections (so we know their actual entropy value), and discard X if its upper bound is lower than the current maximum found entropy. This strategy allows PentroS to significantly reduce the number of candidates sent for entropy counting in the second round.

3.4 Complete Approach

Algorithms 1 and 2 summarize the main steps of *PentroS* algorithm for *miki* discovery over a data stream in Spark Streaming. Algorithm 1 depicts the first job of *PentroS* over a sliding window $\mathcal{SW}$ in a data stream $\mathcal{DS}$. The transactions of a $\mathcal{SW}$ are partitioned and distributed across multiple nodes (multiple splits S_n). Each node emits its local candidates and their appropriate projections. In case of missing information from one node or more, an upper bound function is executed to estimate the frequency of candidate projections in $\mathcal{SW}$ and a second job is performed to check the accuracy of the obtained results. Algorithm 2 illustrates steps of our second job in *PentroS*.

4. EXPERIMENTS

In this section, we evaluate the performance of *PentroS* algorithm for *miki* mining through experiments over real-world datasets. In the remainder of the section, we first describe the experimental setup, and then report the obtained results.

4.1 Experimental Setup

To evaluate the performance of *PentroS*, we used a built-in streaming source of Spark Streaming, fed from two real-life datasets. The first one, called "English Wikipedia", represents a transformed set of Wikipedia articles into a transactional dataset, such that each line mimics an article. It contains 8 millions transactions with 7 millions distinct items and the size of the whole dataset is 4.7 Gigabytes. The second dataset[2], called "ClueWeb", consists of Web pages that were collected in January and February 2009 and is used by several tracks of the TREC conference. During our experiments, we used a part of this dataset including 632 million transactions. The size of the considered "ClueWeb" dataset

[2]http://www.lemurproject.org/clueweb09/

Algorithm 1: *PentroS*: First Job

Input: $\mathcal{DS}$ a transaction Data Stream, $\mathcal{SW}$ the length of the sliding windows, k the size of *miki*
Output: A *miki* of Size k

flatMap($S_i \in \mathcal{SW}$ **)**
- $\mathcal{F}_i \leftarrow$ the set of features in S_i
- $\forall f \in \mathcal{F}_i$ compute $H(f)$ on S_i //entropy of items
- $TopF \leftarrow max(H(f)), \forall f \in \mathcal{F}_i$
- **while** *not end of the stream and* $i \neq k$ **do**
  - $\mathcal{C}_n \leftarrow$ BuildCandidates($TopF$, $\mathcal{F}_i \setminus TopF$)
  - $\forall c \in \mathcal{C}_p, H(c_i) \leftarrow$ ComputeJointEntropy(c, S_i)
  - $TopF \leftarrow max(H(c)), \forall c \in \mathcal{C}_n$
- // $\mathcal{C}_k$ contains all the candidate itemsets of size k
- // and $\forall c \in \mathcal{C}_k$, the joint entropy of c is $H(c_i)$
- **for** $c \in \mathcal{C}_k$ **do**
  - $\mathcal{P}_c \leftarrow$ projections(c, S_i)
  - **for** $p \in \mathcal{P}_c$ **do**
    - **emit**($key = c$: $value = p$)

reduceByKey(*key: itemset c, list(values): projections(c)* **)**
- **if** *c has been emitted by all the workers* **then**
- // we have all the projections of c on $\mathcal{SW}$
- $H(c) \leftarrow$ IncrJointEntropy(c,projections(c))
- **emit**(c,$H(c)$) in a file *Complete*
- **else**
  - // the upper bound of c's joint entropy
  - $Est \leftarrow$ UpperBound(c,projections(c))
  - **emit**(c, Est) in a file "Incomplete"

close( )
- $C_{max} \leftarrow$ CandidateWithMaxEntropy("Complete")
- **emit**(C_{max}, $H(C_{max})$) in a file "CompleteMax"
- **for** $c \in$ *"Incomplete"* **do**
  - **if** $Est(c) > H(C_{max})$ **then**
    - // c is potentially a *miki*, it has to be checked -
    - **emit**(c,Null) in a file "ToBeTested"

Algorithm 2: *PentroS*: Second Job

Input: $\mathcal{SW}$ The sliding windows, *miki* of size k
Output: A *miki* of Size k

flatMap(*Whole* $\mathcal{SW}$ **)**
- Read file 'ToBeTested' from Job1 (once)
- $\mathcal{F} \leftarrow$ set of itemsets in 'ToBeTested'
- **for** $f \in \mathcal{F}$ **do**
  - $p \leftarrow$ projections(f, V_1)
  - **emit** (*key: f, value: p*)

reduceByKey(*key: itemset f, list(values): projections(f)* **)**
- $H(f) \leftarrow$ IncrJointEntropy(f,projections(f))
- write(f, $H(f)$) to a file "CompleteFromJob2"
- **emit** (*key: f, value:* $H(f)$)

close( )
- // emit miki having highest joint entropy
- Max $\leftarrow$ max("CompleteMaxFromJob1", "CompleteFromJob2")
- **emit**(*miki*,Max)

is around one Terabyte. For each dataset, we performed a data cleaning task, removed all English stop words from all articles, and obtained datasets where each article represents a transaction and the features are the corresponding words in the article.

In our streaming process we have set our sliding windows parameters to five batches as a window length and the sliding interval at which the window operation is performed to four batches. Each batch have been set to 5 seconds of incoming data.

For comparison, we implemented a parallel version of *Forward-Selection* in Spark Streaming. By taking the same default values of experiments, it launches a simple *Forward-Selection* process over each sliding window of the streaming. With new incoming and outgoing data, *Forward-Selection* is performed over RDDs with a default distribution, set by Spark Streaming, over multiple splits. In our results, we denote this parallel implementation of *Forward-Selection* as *"ParaForward-Selection"*.

Our experiments were carried out on a cluster with 32 nodes (384 cores in total), equipped with Spark 1.6.1. Each machine is equipped with a linux operating system, 96 Gigabytes of main memory, dual-Xeon X5670 with 2.93GHz 12 core CPUs and 320 Gigabytes SATA hard disk.

Reproductibility: Our code is available at https://infoproj.github.io/PentroS/, with details and instructions.

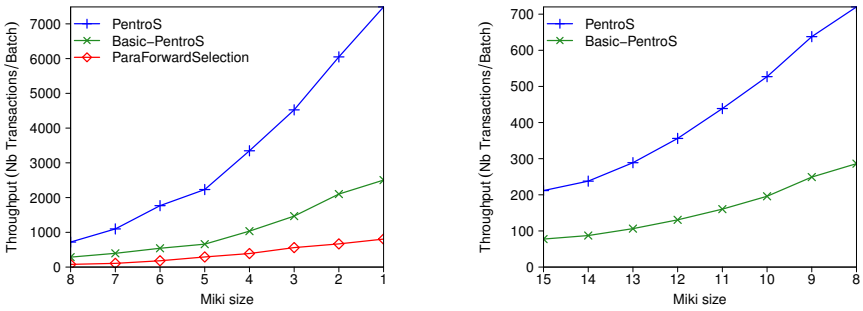
4.2 Effectiveness

The first observation is that *the mikis extracted by PentroS are the same as the ones extracted by Forward-Selection*, for any sliding window on any dataset of our experiments. We always obtain the best quality of *miki* with our approach. So, we dedicate the rest of this section to performances in terms of scalability.

4.3 Results

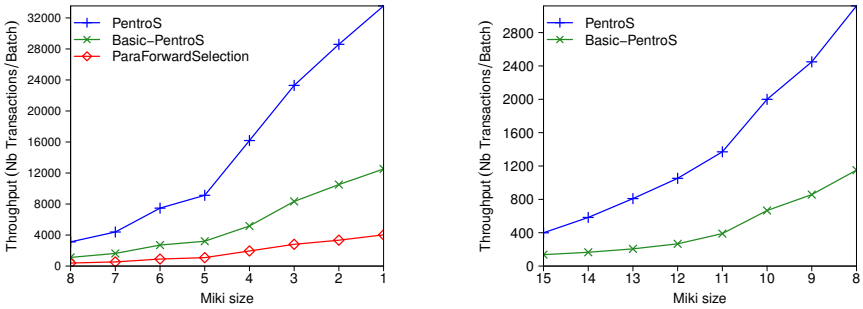
The performances of our algorithms are measured in terms of "throughput". This is the number of transactions that an algorithm is able to process in a batch. Since we have batches of five seconds, throughput in our case is the number of transactions processed within five seconds.

Figures 2a and 2b report our experimental results on the whole English Wikipedia dataset. Figure 2a reports the performance results for an itemset of size k varying from 1 to 8. We see that the throughput of the *Parallel Forward-Selection* algorithm is very low compared to other algorithms. Above a size of $k = 5$ for the *miki*s, the quantity of transactions treated by *Parallel Forward-Selection* converges to 0. This is due to the multiple dataset scans performed in each sliding window to determine an itemset of size k (i.e., *Forward-Selection* needs to perform k rounds for each $\mathcal{SW}$). On the other hand, the performance of *Basic-PentroS* algorithm is much better than *Parallel Forward-Selection* and its throughput grows by a factor of 3, and it continues scaling with higher k values. This difference in the performance



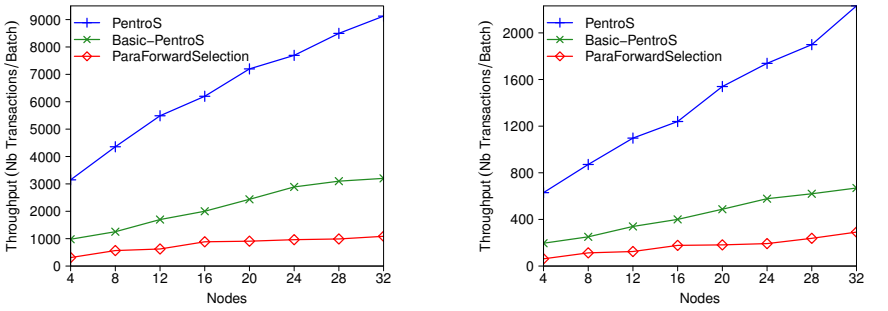
(a) All algorithms (b) Focus on scalable algorithms

Figure 2: Throughput multiplied by 10^{-3} in "Wikipedia Articles" dataset with different values of k (*miki* size). All algorithms were run with 32 nodes. The throughput of *PentroS* is around 10 times better than that of *Parallel Forward-Selection*



(a) All algorithms (b) Focus on scalable algorithms

Figure 3: Throughput multiplied by 10^{-3} in "Clue Web" dataset with different values of k (*miki* size). Like in the Wikipedia dataset, the throughput of *PentroS* over the Clue Web dataset is almost 10 times better than that of *Parallel Forward-Selection*



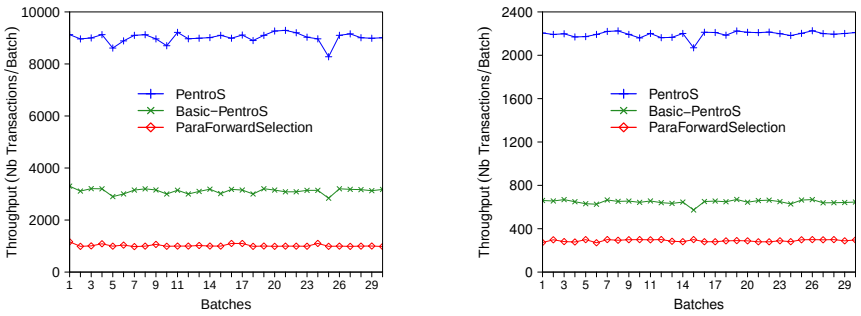
(a) On "Clue Web" Data-set (b) On "Wikipedia Articles" Data-set

Figure 4: Throughput of the algorithms by varying the number of nodes and $k = 5$ (the size of *miki*). *PentroS* outperforms *Basic-PentroS* and *Parallel Forward-Selection* for all values. Over ClueWeb dataset, the throughput of *PentroS* is almost 10 times larger than that of *Parallel Forward-Selection*. We note a similar conclusion in Figure 4b

between the two algorithms illustrates the significant impact of itemset mining in the two round architecture of *PentroS*.

Moreover, by using further optimizing techniques, we clearly see the improvements in the performance. In particular, starting from an itemset having size $k = 8$, we observe a good performance behavior of *PentroS* compared to *Basic-PentroS*. By taking advantage of our optimizing techniques, particularly by incremental entropy computations and reducing the number of data split scans, we record an improvement in the throughput of an order of magnitude between *PentroS* and *Forward-Selection*.

Figure 2b highlights the difference between the algorithms that scale in Figure 2a. Although *Basic-PentroS* continues to scale with $k = 8$, it is outperformed by *PentroS* algorithm.



(a) Over "Clue Web" Data-set (b) Over "Wikipedia Articles" Data-set

Figure 5: Behavior of the algorithms over multiple batches with 32 nodes, and size $k = 5$. The sliding window length is 5 batches, and the sliding interval is one batch. *PentroS* is almost 10 times more powerful than *Parallel Forward-Selection*

With itemsets of size $k = 15$, we clearly observe a significant difference in the response time between *Basic-PentroS* and *PentroS*.

In Figures 3a and 3b, similar experiments have been conducted on the ClueWeb dataset. We observe that the same order between all algorithms is kept compared to Figures 2a and 2b. In particular, we see that *Parallel Forward-Selection* algorithm suffers from the same limitations as could be observed on the Wikipedia dataset in Figure 2a. For instance, with k=3, in Figure 3a, the throughput of *PentroS* is 22 millions while that of *Parallel Forward-Selection* is only 2 millions.

Figure 4 illustrates the results obtained from running the algorithms using different number of nodes. Figure 4a shows the throughput over the "Clue Web" dataset. The difference in throughput between all algorithms is maintained while we observe that *Parallel Forward-Selection* does not scale well (it does not benefit from the addition of computing nodes). In Figure 4b, similar experiments have been conducted on the "Wikipedia Articles" dataset and the same tendency is maintained for all algorithms, showing the clear advantage of *PentroS*.

In Figure 5, we show the behavior of the three algorithms over batches of data through the streams. We settled the length of a sliding window to 5 batches of 5 seconds, and the sliding interval is four batches. For *Basic-PentroS* and *PentroS*, we observe slight decreases in the throughput for some windows. This is due to a scan over the splits in the RDD by launching *Forward-Selection* in the case that an itemset $F-k$ has changed from one window to another. Even though, *PentroS* keeps the same performance improvements, compared to its competitors.

5. RELATED WORK

In the literature, several endeavors have been made to explore informative itemsets (or feature sets) in datasets [7, 12, 13, 14]. Different measures of itemset informativeness (e.g., frequency of itemset co-occurrence) have been used to identify and distinguish informative itemsets from non-informative ones. Mining itemsets based on the co-occurrence frequency (e.g., frequent itemset mining) measure does not capture all dependencies and hidden relationships in the dataset, especially when the data is sparse [14]. Therefore, other measures must be taken into account. In [14], low and high entropy measures of itemsets informativeness were proposed. Heikinheimo et al. [14] propose the use of a tree-based structure without specifying a length k of the informative itemsets to be discovered. However, as mentioned in [14], such an approach results in a very large output. Knobbe et al. [7] suggest to use the heuristic approach described in Section 2.3.

Another algorithm proposed in [7] 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 the 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. In [2], Salah et al. propose PHIKS, an algorithm designed to extract the sets of high entropy itemsets with multiple values k from static datasets using the MapReduce framework. However, the proposed algorithm is not appropriate for *miki* mining over data streams, in which we need to compute the *miki* dataset by taking into account the fast arrival (or departure) of data into (from) the sliding windows. There have been interesting works for extracting frequent itemsets from data streams, e.g. [15, 16]. In [17], Charikar et al. propose a new sketching technique for finding frequent items in data streams allowing to estimate the frequencies of all the items in the stream. In [18], Gilbert et al. propose an itemset mining algorithm based on Wavelet transformation for computing small space representations of massive data streams. In [19], Lam et al. propose an algorithm that based on the characteristics of an item, counts its max-frequency over a sliding window (while the length of the itemset dynamically changes). However, these solutions are not adapted to *miki* mining over missive data streams. In [8], Zhang et al. propose a centralized approach for discovering maximally informative itemsets from data streams based on a sliding window. However, the proposed approach does not scale to very fast and large scale data streams, simply because the computing resources of one machine are not sufficient to process such volumes of data.

Indeed for *miki* processing over very large and high speed

data streams, we need parallel solutions taking into account the nature of streams, in which the content of large sliding windows change continuously and very quickly. In this paper, we propose such a parallel solution. In our work, we took advantage of Spark Streaming for managing the data streams, and developed a parallel algorithm with efficient optimizing strategies that reduce significantly the time needed for updating the query results, particularly by incremental entropy computation and minimizing the number of *miki* candidates.

6. CONCLUSION

In this paper, we proposed *PentroS* a reliable parallel streaming algorithm for maximally informative k-itemsets mining using the Spark Streaming framework. *PentroS* has shown a significant efficiency in terms of throughput and scalability. It elegantly determines *miki* over a large number of sliding windows. With *PentroS*, we propose multiple optimizing techniques that render the *miki* mining process very fast. They include the incremental updating of the *miki* results after arrival and departure of transactions to/from the distributed sliding windows.

We extensively evaluated the performance of *PentroS* using large scale real-world data streams. On the whole, the results show the strength and robustness of *PentroS* in the discovery of *miki*s with high itemset size over sliding windows with a high rate of incoming and outgoing data.

7. REFERENCES

[1] F. Gorunescu, *Data Mining - Concepts, Models and Techniques*. Springer, 2011.

[2] S. Salah, R. Akbarinia, and F. Masseglia, "Fast parallel mining of maximally informative k-itemsets in big data," in *ICDM 2015*, (Atlantic City, NJ, USA), 2015.

[3] S. Moens, E. Aksehirli, and B. Goethals, "Frequent itemset mining for big data," in *IEEE BigData 2013*, (Santa Clara, CA, USA), 2013.

[4] T. A. Runkler, *Data Analytics - Models and Algorithms for Intelligent Data Analysis*. Springer, 2016.

[5] Y. Bassil, "A survey on information retrieval, text categorization, and web crawling," *CoRR*, 2012.

[6] T. M. Cover and J. A. Thomas, *Elements of information theory (2. ed.)*. Wiley, 2006.

[7] A. J. Knobbe and E. K. Y. Ho, "Maximally informative k-itemsets and their efficient discovery," in *ACM SIGKDD 2006*, (Philadelphia, PA, USA), 2006.

[8] C. Zhang and F. Masseglia, "Discovering highly informative feature sets from data streams," in *DEXA 2010*, (Bilbao, Spain), 2010.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud 2010*, (Boston, USA), 2010.

[10] O. Papapetrou, M. N. Garofalakis, and A. Deligiannakis, "Sketching distributed sliding-window data streams," *The VLDB Journal*, 2015.

[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI 2004*, (San Francisco, California, USA), 2004.

[12] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB 1994*, (Santiago de Chile, Chile), 1994.

[13] C. Ji and Z. Deng, "Mining frequent ordered patterns without candidate generation," in *FSKD 2007*, (Haikou, Hainan, China), 2007.

[14] H. Heikinheimo, J. K. Seppänen, E. Hinkkanen, H. Mannila, and T. Mielikäinen, "Finding low-entropy sets and trees from binary data," in *ACM SIGKDD 2007*, (San Jose, California, USA), 2007.

[15] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining frequent patterns in data streams at multiple time granularities," *Journal of Next generation data mining*, 2003.

[16] W. Teng, M. Chen, and P. S. Yu, "A regression-based temporal pattern mining scheme for data streams," in *VLDB 2003*, 2003.

[17] M. Charikar, K. C. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theor. Comput. Sci.*, 2004.

[18] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surfing wavelets on streams: One-pass summaries for approximate aggregate queries," in *VLDB 2001*, (Roma, Italy), 2001.

[19] H. T. Lam and T. Calders, "Mining top-k frequent items in a data stream with flexible sliding windows," in *ACM SIGKDD 2010*, (Washington, DC, USA), 2010.