Pushing Constraints into Data Streams

Andreia Silva and Cláudia Antunes
Department of Computer Science and Engineering
Instituto Superior Técnico – Technical University of Lisbon
Lisbon, Portugal
{andreia.silva, claudia.antunes}@ist.utl.pt

ABSTRACT

One important challenge in data mining is the ability to deal with complex, voluminous and dynamic data. Indeed, due to the great advances in technology, in many real world applications data appear in the form of continuous data streams, as opposed to traditional static datasets. Several techniques have been proposed to explore data streams, in particular for the discovery of frequent co-occurrences in data. However, one of the common criticisms pointed out to frequent pattern mining is the fact that it generates a huge number of patterns, independent of user expertise, making it very hard to analyze and use the results. These bottlenecks are even more evident when dealing with data streams, since new data are continuously and endlessly arriving, and many intermediate results must be kept in memory. The use of constraints to filter the results is the most common and used approach to focus the discovery on what is really interesting. In this sense, there is a need for the integration of data stream mining with constrained mining. In this work we describe a set of strategies for pushing constraints into data stream mining, through the use of a pattern tree structure that captures a summary of the current possible patterns. We also propose an algorithm that discovers patterns in data streams that satisfy any user defined constraint.

Keywords

Data Streams, Constraints, Frequent Itemset Mining

1. INTRODUCTION

To undertake the rapid growth of data, everywhere and in a great variety of fields, the area of *Data Mining* emerged with the goal of creating methods and tools capable of analyzing these data and extracting useful information, that companies can exploit and apply to their businesses. Formally, data mining [4] is defined as the nontrivial extraction of implicit, previously unknown, and potentially useful information from data.

BigMine '13, August 11, 2013 Chicago, Illinois, USA

ACM 978-1-4503-2324-6/13/08 ...$15.00.

Frequent itemset mining [1, 15], or just *pattern mining*, plays an important role in data mining, aiming for the discovery of frequent co-occurrences in data (called *patterns*). However, one of the common criticisms pointed out to data mining, in particular to pattern mining, is the fact that it generates a huge number of patterns, independent of user expertise and expectations, making it very hard to understand and use the results [6]. The truth is that, for pattern mining, if we make the support too high, only already known patterns are found, or none at all. Otherwise, if the support is set too low, the number of patterns explode, and it is very difficult to distinguished the real useful patterns among the many uninteresting ones. A balance is required, and ways to limit the number of results, as well as to focus these results in user expectations, are needed.

Several ways have been proposed to minimize these bottlenecks (e.g. mining closed and maximal patterns), being the use of constraints to filter the results, the most common and used approach. Constraints are an efficient way to reduce the number of returned patterns and increase the efficacy of pattern mining, by returning less but more interesting results, in the user and application points of view. Some types of constraints have been proposed, as well as several algorithms that are able to push each individual type of constraints [17, 18, 14, 16, 15, 10, 3, 11]. However, these algorithms assume that all data are available from the start.

In many real world applications, data appear in the form of continuous data streams, as opposed to traditional static datasets. A *data stream* is an ordered sequence of instances that are continuously being generated and collected. Being able to deal with these unbounded quantities of data is considered one important challenge of data mining.

Since data are not all available a priori, and may be arriving at high speeds, algorithms have to sacrifice the correctness of the results by allowing some counting errors, and must maintain a memory-resident summary data structure (also called *synopsis data structure*), that stores only the information that is strictly necessary to avoid loosing patterns [12]. Despite these challenges, several algorithms have been proposed to find frequent patterns in data streams [13, 5, 12].

Nevertheless, these algorithms still suffer from the same bottlenecks as traditional pattern mining, and the consequences of the huge number of patterns is even more visible and problematic. On one side, the summary data structure needs to keep not only all current patterns, but also the minimally frequent itemsets that might become patterns later.

If the number of patterns usually returned is already huge, the number of itemsets that these structures must keep is even higher. This implies large amounts of memory and extra processing time. On another side, results are likely to be analyzed several times across the stream, which means that the more results are kept in the structure, the more has to be processed every time.

There is therefore an urgent need for the integration of stream mining with constrained mining. To the best of our knowledge, the only algorithms designed to find constrained patterns from data streams (*approxCFPS* [9] and its variations) are only able to push constraints that follow one specific property (succinctness).

In this paper we describe and discuss a set of strategies to push constraints into data stream mining, through the use of a pattern tree as a synopsis data structure. We also propose a generic algorithm, called *CoPT4Streams* (Constraint Pushing into a Pattern Tree for Streams), that combines and implements these strategies and is able to dynamically discover all patterns that satisfy any user defined constraint. *CoPT4Streams* pushes constraints into the pattern tree structure in an efficient way, by taking advantage of the properties of constraints, and filters all patterns and possible patterns in that tree, resulting in a much smaller summary, and therefore less memory and time are needed.

The paper is organized as follow. Section 3 provides the background for constrained and stream mining, as well as the related work. The proposed strategies and algorithm are presented in section 4, and experimental results are shown in section 5. Finally, section 6 concludes the work.

2. PROBLEM STATEMENT

Frequent pattern mining (PM) aims for enumerating all frequent patterns that conceptually represent relations among discrete entities (or *items*). Depending on the complexity of these relations, different types of patterns arise, with the transactional patterns being the most common. A *transactional pattern* is just a set of items that co-occur frequently. A well-known example is a market-basket, the set of items that are bought frequently in the same transaction.

The oldest and more studied constraint in pattern mining is the minimum support threshold [1], which states that, to be interesting, a pattern must occur more than the given threshold. In fact, what we call traditional PM corresponds to the discovery of *frequent* itemsets from data. Hence, constrained PM is perceived as the use of constraints beyond the minimum support, i.e. the discovery of *frequent* itemsets that *satisfy some constraints*.

Formally, let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of distinct literals, called items. A subset of items is denoted as an *itemset*. A superset of an itemset X is also an itemset, that contains all items in X and more. The support of an itemset X is the number of its occurrences in the dataset, and X is frequent if its support is no less than a predefined minimum support threshold: $sup(X) \geq \sigma \in [0, 1]$.

DEFINITION 1. *A constraint C is a predicate on the powerset of I [16], i.e. $C : 2^I \mapsto \{true, false\}$. An itemset X satisfies a constraint C, if $C(X) = true$.*

In this context, a *pattern* corresponds to a frequent itemset that satisfies the constraint C, i.e. $sup(X) \geq \sigma \wedge C(X) = true$. And the problem of *constrained pattern mining* is to find all patterns in a dataset.

The previous definitions consider that the dataset is mined all together. Let us now assume that the dataset is a data stream, where new transactions arrive sequentially in the form of continuous streams. Formally, a data stream $D_s = B_1 \cup B_2 \cup ...B_k$ is a sequence of batches, where B_k is the current batch, B_1 the oldest one, and each batch is a set of transactions.

As it is unrealistic to hold all streaming data in the limited main memory, data streaming algorithms have to sacrifice the correctness of their results by allowing some counting errors. These errors are bounded by a user defined *maximum error threshold*, $\epsilon \in]0, \sigma[$, such that $\sigma \gg \epsilon$. Thus, the support calculated for each item is an approximated value: $sup'(X)$.

An *approximate pattern* is an itemset whose estimated support is no less than the minimum support threshold, minus the maximum error allowed, i.e. $sup'(X) \geq (\sigma - \epsilon)$.

In a constrained environment integrated with data streams, a pattern is an approximated frequent itemset that satisfies the constraint, i.e. $sup'(X) \geq (\sigma - \epsilon) \wedge C(X) = true$. And the problem of constrained pattern mining over data streams consists in finding all approximate patterns in D_s.

3. BACKGROUND

The integration of constrained mining and streams mining is non trivial, since both areas present a set of challenges that need to be addressed and combined. In this section we first provide some background on constrained mining and then on the characteristics of stream mining.

3.1 Constrained Mining

Constraints are filters on the data or on the results, that capture application semantics and allow the users to express their needs, and somehow control the search process and focus de algorithms on what is really interesting [2].

There are several types of constraints. According to their semantics and form, they can be divided in several categories, such as *content* constraints, filtering the content of the discovered patterns; *length* constraints, limiting the number of items in each pattern; and more complex ones, like *temporal* constraints, that take into account a temporal dimension.

How to enforce these constraints into pattern mining depends heavily on the constraints in question. Performing an extensive search is often not a viable solution, mostly due to the size of the search space. Fortunately, studies show that constraints have some properties that provide efficient strategies to prune the search space and improve the selection of patterns that satisfy them. These *"nice"* properties [15] have been proposed in the literature, and existing ones are described next.

3.1.1 Constraint Properties

In its basis, a constraint can be *anti-monotonic*, *monotonic*, or none.

Anti-monotonicity (AM) *A constraint is anti-monotonic if and only if, whenever an itemset X violates it, so does any superset of X.*

The minimum support threshold is the best known and simple example of an AM constraint [1], according to which an itemset is frequent if its support is greater or equal to a user defined threshold. It is AM in the sense that if an itemset is infrequent, so does any of its supersets.

Monotonicity (M) *A constraint is monotonic if and only if, whenever an itemset X satisfies it, so does any superset of X.*
An example is the item constraint: $C(X) = (\{i, j\} \subseteq X)$. If an itemset satisfies the constraint (i.e. contains all the desired items), all supersets will also satisfy it. However, if an itemset violates the constraint, a superset can satisfy it, by introducing the missing items.

In addition to anti-monotonic or monotonic, constraints can also be, at the same time, *succinct*.

Succinctness (S) In its essence, *a constraint is succinct if it is possible to enumerate all possible patterns, based only on items from the alphabet I* [14].

A simple example is the value constraint $C(X) = X.price \leq €100$. We can select from the alphabet all items X_1 with $price \leq €100$, and the itemsets that satisfy C are exactly only those in the strict powerset of X_1. This is a *succinct anti-monotonic* constraint (SAM), as supersets of itemsets with some item with $price > €100$ will never satisfy it.

There are constraints that are not overall anti-monotonic neither monotonic, and therefore it is not easy to push them in an efficient way. However, with some assumptions, many of them can be converted and treated as that. In this sense, constraints can also be *prefix-* or *mixed-monotone*.

Prefix-Monotonicity *A constraint is prefix-monotone*[1] *if there is an order of items that allows the algorithms to treat it as anti-monotonic or monotonic [16].* By fixing an order on items, each transaction can be seen as a sequence, and therefore we can use the notion of prefixes and suffixes, as the first or last items in the ordered transaction, respectively.

Formally, a constraint C is prefix anti-monotonic (PAM) (resp. prefix monotonic (PM)) if there is an order R over the set of items, and assuming each itemset $X = i_1 i_2 ... i_n$ is ordered accordingly to order R, such that, whenever an itemset X violates (resp. satisfies) C, so does any itemset with X as prefix ($X' = X \cup \{i_{n+1}\} = i_1 i_2 ... i_n i_{n+1}$).

For example, an aggregate constraint like $C(X) = avg(X) \geq 20$, is not monotonic neither anti-monotonic. But, if we order the items in a value-descending order, an itemset X has higher average than its supersets X', and C is prefix anti-monotonic: if X satisfies C, also all its supersets X'.

Mixed-Monotonicity *A constraint is mixed-monotone if it can be considered both anti-monotonic and monotonic, at the same time, for different groups of possible values (positive and negative)*[11].

Formally, let the set of items I be divided into two disjoint groups based on their monotonicity relating to a constraint C: let I^{AM} be the set of anti-monotonic items, and I^{M}, the set of monotonic items. Then, a constraint is mixed monotone if, for any itemset X: (a) whenever X satisfies C, all supersets of X formed by adding items from the I^{M} group, also satisfies C; and (b) whenever X violates C, all supersets of X formed by adding items from the I^{AM} group, also violates C.

[1]Prefix-monotone constraints were first proposed with the name of *convertible constraints* [16, 15].

This property was proposed in particular for *sum* constraints. $C(X) = sum(X) \geq v$, for example, is monotonic for positive values (including zero), and anti-monotonic for negative values.

3.1.2 Related Work

There are several algorithms for pushing constraints of a specific type or that have a specific property. The problem is that those algorithms are specific, and there is no general algorithm capable of incorporating any constraint, and still taking advantage of constraint properties at the same time.

Srikant et al. [17] were the first to introduce item constraints, the first different from minimum support. They proposed three apriori-based algorithms, *MultipleJoins*, *Reorder* and *Direct*, that are able to deal with boolean combinations of these constraints, i.e. of the form $i \in S$ or $i \notin S$. Succinct constraints were first proposed by Ng et al. [14], as well as an apriori-based algorithm, called *CAP* (Constrained APriori). Later on, [10] proposed *FPS* (FP-tree based mining of Succinct constraints) that uses the same techniques but in a pattern-growth approach. These algorithms are only able to push succinct constraints. Pei et al. [16] proposed prefix-monotone constraints as well as a pattern growth algorithm, *FIC* (Frequent Itemset mining with Convertible constraints), that is able to push them into the discovery process, by growing only valid prefixes. Finally, mixed monotone constraints were recently proposed by Leung et al. [11], in particular for *sum* constraints, along with a pattern-growth algorithm *FPM* (Frequent Pattern mining for Mixed monotone constraints). The problem of the above algorithms is that none of them is able to deal with data streams.

3.2 Stream Mining

A data stream is an ordered sequence of instances that are continuously being generated and collected. The nature of these streaming data make the mining process different from traditional data mining in several aspects: (1) data are not all available a priori, and therefore each element should be examined at most once; (2) new data may be arriving at high speeds, so the processing of each element should be as fast as possible; (3) memory usage should be limited, even though new data elements are continuously arriving; (4) the results generated should be always available and updated; and (5) the results should be as accurate as possible.

This implies the definition of some counting error, smaller than the minimum support, that allows the algorithms to discard part of the data that is not promising. Itemsets that occur less than the maximum error are infrequent and can be discarded, and itemsets that occur more than the error, but less than the minimum support, are possibly frequent, and should be kept in some summary data structure, since they may become frequent later.

Several algorithms have been proposed to find frequent patterns in data streams [12], being *Lossy Counting* [13] and *FP-Streaming* [5] the most known. FP-Streaming, proposed by Giannella et al., is an efficient algorithm that adapts FP-Growth [7] to find frequent unconstrained patterns over a data stream. They make use of the FP-tree structure and its compression properties to maintain time sensitive frequency information about patterns. The stream is divided into batches and a tree structure (called *FP-stream*) is updated at every batch boundary, so that it contains all the

current patterns. Each node in this tree represents a pattern (from the root to the node) and its frequency is stored in the node, in the form of a *tilted-time window table*, which keeps frequencies for several time intervals.

Since what defines what is promising or not depends only on the number of occurrences of an itemset, this tree must usually keep a lot of possibly frequent patterns, which results in poor performance, not only in terms of the memory needed, but also on the time needed to update this tree. This problem could be addressed by using constraints not only to filter what is necessary to keep in the tree, but also to define what is promising or not in a user perspective, making the results less and more interesting at the same time, and therefore increasing the performance and applicability of pattern mining results. Unfortunately, both of the above algorithms cannot handle constraints.

Leung et al. [9] was the first to propose the integration of data streams with constrained mining, with two algorithms, *ApproxCFPS* and *ExactCFPS* (Approximated and Exact Constrained Frequent Patterns for Streams), for finding all approximated or exact patterns, respectively, in data streams, that satisfy succinct constraints. Both algorithms are able to push succinct constraints deep into the algorithm FP-Streaming. The ideas are simple and consist in, for SAM constraints, remove all single items that violate the constraint before processing each transaction; and for SM constraints, for each batch, divide the items into mandatory and optional items, and order transactions so that mandatory items appear first, and therefore there is only the need to mine itemsets that first contain the mandatory items. More recently, they also proposed an adaptation (*aCoCo* [8]), that uses an optimized FP-Tree for when the memory is limited. While the algorithms efficiently push constraints into data stream mining, they are only able to handle constraints that are succinct.

4. PUSHING CONSTRAINTS INTO DATA STREAMS

In this work we describe a set of strategies for pushing constraints into stream pattern mining, through the use of a pattern tree structure, similar to the FP-Stream [5]. We also describe a generic algorithm, called CoPT4Streams (Constraint Pushing into a Pattern Tree for Streams), that combines and implements these strategies and is able to dynamically discover all patterns that satisfy any user defined constraint. CoPT4Streams pushes constraints into the pattern tree structure at each batch boundary in an efficient way, by taking advantage of the properties of constraints, and filters all patterns and possibly patterns in that tree, resulting in a much smaller summary, and therefore less memory and time needed.

Since it is an algorithm that is applied to the pattern tree, any data streams algorithm can be used along with our CoPT4Streams, provided that it uses a pattern tree as its synopsis data structure.

4.1 Pattern-Tree

A pattern-tree is a compact prefix tree structure that holds information about patterns. In the streaming environment, each node of a pattern tree contains an item, an approximate support and a maximum error, and edges link items that occur together, forming the patterns. Therefore, each node in a pattern-tree corresponds to an approximate pattern, composed of the items from the root to this node, and the estimated support and error attached to this node. As a prefix tree, patterns that share the same prefix also share the same nodes corresponding to that prefix. And therefore, the size of the tree is usually much smaller than having them in a list or a table, and the search for an itemset is usually much faster.

Note that this description corresponds to a basic pattern tree. Each node may contain more fields, if they are necessary for the streaming algorithm, and the strategies for pushing constraints into the pattern tree remain the same. An example is an FP-Stream [5], that contains in each node a table of frequencies for several time intervals, instead of just one frequency.

4.2 Naive Approaches

A naive approach, named *CoPT4Streams+*, is to perform a simple depth-first search (DFS) to traverse the tree and test all nodes for the constraint, independently of its property (note that, when we test a node for a constraint, we mean that we test the itemset corresponding to that node). However, it is not the most efficient approach, since not all nodes need to be tested. We can take advantage of constraint properties and perform a constrained DFS, stopping the search at some points and avoiding unnecessary tests.

Another possible approach is to push the constraint right before inserting each itemset in the pattern-tree. However, while this may be better in terms of memory, because the pattern-tree would never have unpromising itemsets, this means that we have to test every itemset. By scanning the tree, we may skip the constraint checking of a lot of itemsets.

4.3 Efficient Constraint Pushing Strategies

In order to push constraints into a pattern-tree, we define a set of strategies that can be used, based on constraint properties.

Since we are integrating data streams and constraints, some questions arise. Note that the pattern tree must be updated in every batch, to renew the current approximated frequent itemsets. And therefore the order in which the items in patterns are inserted in the tree must remain the same across the batches. (1) Data are not available a priori, and so we do not know all possible items in the beginning. In the cases where the order of items matter (e.g. for prefix-monotone constraints), new items that should be placed between already known items may appear. Is it possible to efficiently take advantage of constraint properties, even when the order of items changes? (2) In a static application, invalid itemsets could be removed from the tree, since they do not satisfy the constraint (for both AM and M constraints). In a data stream, these itemsets could reappear in following batches, and valid supersets of current invalid itemsets could also appear later (in the case of M constraints). Can we, at some batch, remove itemsets in the tree that violate the constraint?

The answer is yes to both questions, essentially because for a pattern to appear in the pattern tree (i.e. to be approximately frequent), all of its subsets must appear too. But we will delve into these questions further ahead.

We assume that constraints have fixed parameters (for example, $min(X) < v$, in which v is a fixed threshold), i.e. parameters that do not depend on the number of transactions

seen so far, and do not change across different batches (e.g. we do not consider constraints like $min(X) < min$(all items seen so far)). This makes the satisfaction of constraints permanent, meaning that, if an itemset satisfies (reps. violates) a constraint in some batch, it always satisfies (reps. violates) the same constraint, in any later batch.

4.3.1 Anti-Monotonicity:

Pushing an AM constraint (C_{AM}) is pretty straightforward. While performing a DFS, if the node:

(a) Satisfies C_{AM}: keep it in the tree;

(b) Violates C_{AM}: there is no need to search its subtree because all supersets also violate the constraint. Therefore we can prune the tree and remove this node, as well as all of its children.

Answering to question (2), for AM constraints, itemsets that violate the constraint can be removed, because they will never satisfy the constraint, and even if they reappear in later batches because they are frequent, they will be removed again, because they violate the constraint.

4.3.2 Monotonicity:

To incorporate a monotonic constraint (C_M), we cannot remove nodes that violate it, because the supersets of this node (its children) can satisfy it. So, while traversing the tree, if the node:

(a) Satisfies C_M: keep it in the tree. We do not need to scan the subtree, because all supersets satisfy the constraint, and there is nothing to remove.

(b) Violates C_M: If it is a leaf node (has no supersets), we can remove it, as well as all parents that become a leaf because of this elimination. If it is not a leaf, continue the search to its children, since they can satisfy the constraint.

Answering again to question (2), for M constraints, all itemsets with no supersets in the tree (leafs) that violate the constraint can be removed, because they will never satisfy the constraint. Note that, if some valid superset appears in later batches, it means that both that itemset and the superset are frequent, and therefore both will appear in the tree, in the same branch. However, only the superset will be returned as pattern, because it is the only one valid. Summing, there is no need to keep an invalid itemset in the tree, while it has no valid supersets.

4.3.3 Succinctness:

In the presence of a succinct constraint, we can apply the strategies for C_{AM} or C_M, whether it is succinct anti-monotonic (C_{SAM}) or succinct monotonic (C_{SM}), respectively. However, the succinctness of a constraint allow us to know, by looking for single items, which of them satisfy or not satisfy the constraint. Therefore, we can use that to obtain a more efficient search.

With this in mind, we can first divide the items into two groups: items that satisfy or are necessary to the satisfaction of the constraint, I^s; and items that violate, or are not necessary to the satisfaction of the constraint, I^v. And before inserting itemsets into the pattern-tree, we can order the itemsets according to those groups.

C_{SAM}: With a SAM constraint, single items that violate it can be discarded. If we order items in itemsets so that I^v appears before I^s (I^v closer to the root and I^s to the leafs), when applying the C_{AM} strategy, we only need to check the first level of the pattern-tree. If the node violates the constraint, remove it and its sub-tree; if the node satisfies, all of its children will also satisfy, because they belong to I^s, so we can also skip testing for the constraint.

C_{SM}: In the case of a SM constraint, I^s contains the mandatory items and I^v the optional items. If an itemset with items from I^s satisfies the constraint, all of its supersets formed by adding items from I^s or I^v also satisfy it. Itemsets with items only from I^v violate the constraint. In this sense, if we order itemsets so that items from I^s appear first than items from I^v, when applying the C_M strategy, we only need to do it until the first node from I^s that satisfies (all supersets satisfy), or until the first node from I^v, because if we arrive to a node from this group, and still need to test the constraint, it means it has not been satisfied by items from I^s, and the following items (children) will also not satisfy it because they are optional. In this case, we do not need to test this node, neither any child, and we may remove them from the tree.

Succinct constraints refer to question (1), since they need the items to be ordered according to two groups. This means that we do not know the overall order a priori, and therefore new items from the first group may appear and need to be placed before all already known items from the second group. This poses no problem, as long as the relative order of existing items does not change, because if an itemset with new items appear in the tree, all subsets will also appear, and all subsets that not include these new items remain with the same order.

4.3.4 Prefix-Monotonicity:

Since prefix-monotone constraints can only be treated as AM (C_{PAM}) or M (C_{PM}) constraints if items are ordered by a particular order, we just need to sort the itemsets according to that order before inserting them in the pattern-tree, and apply the C_{AM} or C_M strategy, respectively. Otherwise, we have to traverse the whole tree and check all nodes for the constraint.

Looking at question (1), and like for succinct constraints, this order is not a problem, as long as the relative order of existing items does not change.

4.3.5 Mixed-Monotonicity:

Mixed-monotone constraints (C_{Mix}) are both AM and M, for different groups of values. In this case, we just have to divide the items, as they appear, into those groups: I^{AM} and I^M, and put I^M before I^{AM} in the tree, i.e. sort itemsets so that items from the I^M group appear above items from I^{AM}. The idea is to start with the C_M strategy, until a node that satisfies it, or a node from I^{AM} appears. From that node, we can apply the C_{AM} strategy to all of its supersets (children) and prune invalid nodes from its sub-tree.

4.4 Algorithm CoPT4Streams

Since there are a lot of similarities between the strategies presented above, they can be combined into one single generic strategy or algorithm. We propose therefore the algorithm *CoPT4Streams* (Constraint Pushing into a Pattern Tree for Streams), that is able to efficiently and effectively push any constraint into a pattern tree, when mining data streams.

The idea is to run *CoPT4Streams* over the pattern tree resulting of the mining of each batch, and using the resulting smaller tree to mine the next batches. By doing this, the algorithm is able to filter what is really interesting for the users, and keep smaller summary structures, which result in improvements on the memory and time needed, as well as on the number of the patterns returned.

Since constraint satisfaction is permanent, we can perform an extra optimization (besides using constraint properties) and only compute the satisfaction of some node once, by e.g. keeping one flag in each node indicating if it satisfies or violates the constraint. Thus, we can mitigate the constraint checking for nodes that remain in the tree from one batch to another (nodes closer to the root).

Essentially, to push a constraint, *CoPT4Streams* works as follows. For each batch, and for each approximate pattern discovered by the streaming algorithm, it is ordered according to the order of items for that constraint, if exists, and inserted in the tree (if there is no order, items are put in the pattern-tree in a support-descending order, which is known to improve the compactness of the tree [7]).

At each batch boundary, we can push the constraint C into the pattern-tree, by scanning the tree according to the constraint property. So, for each node, if the node is new in the tree (i.e. if we never checked for the constraint), we can first see, in the case of succinct or mixed constraints, if the item in the node belongs to the second group of items. If so, it means the node can be discarded (the constraint was not satisfied by the first group of items), along with its children. Then, or in the case of other type of constraints, we should check for the constraint (and store the result into the satisfaction flag in the node).

When we know the result of the constraint checking, (1) if the itemset corresponding to this node satisfies C: (a) C is mixed and we can change the strategy to AM; (b) C is monotonic and no child needs testing; or (c) C is succinct AM, and also no child needs testing (only the first level of the tree). (2) if the itemset violates C, it is not a pattern, and if C is AM (including SAM and PAM) we can prune the tree from here.

After checking the constraints, if the node was not pruned, we can test its children. Finally, after pushing C into the children, if the node is not a pattern and is a leave, we can remove it. Note that this final node pruning is made for every constraint, even if they have no "nice" properties. However, in this case all nodes need to be tested.

5. EXPERIMENTAL RESULTS

The goal of these experiments is to analyze the behavior of our algorithm in the presence of a data stream, and all types of constraints, and prove that *CoPT4Streams* is able to effectively and efficiently push them into a pattern-tree at each batch, taking advantage of their properties.

In these experiments we use a database automatically generated by the program developed at IBM Almaden Research Center [1]. The dataset has $100k$ transactions, with an average of 10 items per transaction and a domain of 1000 items (with values from zero to 1000). In addition, in order to test the mixed-monotone constraint, we consider an equivalent dataset but with negative values (by making values vary from -500 to 500).

Since the behavior of the algorithm can depend on the selectivity of the constraints, we use it in our experiments. Selectivity is defined as the ratio of frequent itemsets that violate the constraint, over the total number of frequent itemsets, i.e. how much we can filter. Therefore, we test *CoPT4Streams* with several constraints with different selectivities, varying from 10% to 90%. Note that, the more selectivity, the more we can filter, and the less patterns are returned. But on the other hand, the less selectivity, the more patterns need to be kept and returned (and we get closer to the problems of unconstrained techniques). We also tested several minimum supports and errors, and since results are consistent, we present the results for a support of 0.1% and an error of 0.01% (a common way to define the error, is $\epsilon = 0.1\sigma$), and results presented correspond to the average of several runs with different constraints with equivalent selectivity. Also, to have a term of comparison, we test our algorithm against *CoPT4Streams+*, a version that checks all nodes for the constraints (i.e. that does not take into account constraint properties).

The data streams algorithm used was a simplification of FP-Streaming [7], that keeps only one support per node (does not take into account temporal aspects). FP-Streaming was chosen because it is an efficient algorithm that does not suffer from the candidate generation problem, and keeps current patterns into a pattern tree. The size of each batch is defined by $|B| = 1/\epsilon$, which corresponds to 100 batches of 1000 transactions in each batch. By definition, data streaming techniques return more patterns than traditional algorithms for static datasets, and the higher the error allowed, the more patterns are returned and the less accuracy they obtain. By incorporating constraints into data streams, we can filter not only the patterns returned, but also the patterns that must be kept in memory, improving the performance of the algorithms, either in terms of time, memory and results. The computer used to run the experiments was an Intel Core i7 CPU at 2GHz (Quad Core), with 8GB of RAM and using Mac OS X Server 10.7.5 and the algorithm was implemented using the Java (JVM version 1.6.0_37).

We first analyzed the average size of the pruned pattern-tree. When applying constraints, more itemsets can be discarded, and therefore the pattern tree is smaller than in an unconstrained environment. In turn, a smaller pattern tree in every batch may have an impact on the time needed to update the tree and on the number of constraint checks the algorithm needed to make. Note that the update time is perceived as the time needed to process one batch of transactions until the complete update of the pattern tree. Since the trends are the same, whether a constraint is AM or M, fig. 1 to 3 show the average results when in the presence of AM (an average of both AM, SAM and PAM) and M (both M, SM and PM) constraints. The only difference is that, in the unconstrained case, as well as for the simple AM and M constraints, there is no need to sort the items in the patterns. On the other side, succinct, prefix- and mixed-monotone constraints require that items are put in the pat-

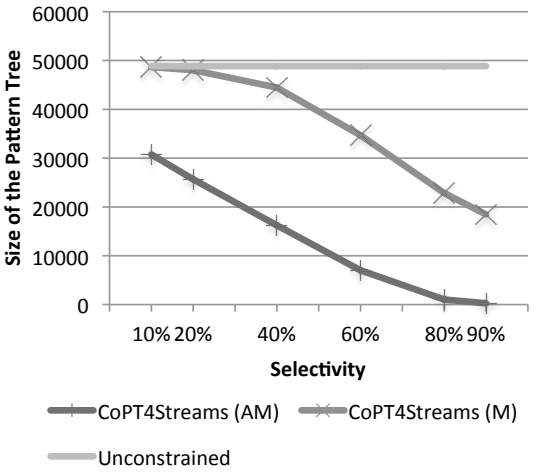

Figure 1: Average size of the pattern tree, after pruning the constraint

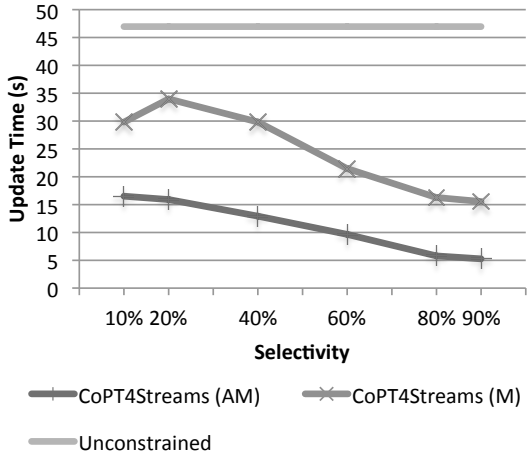

Figure 2: Average time needed to update the pattern tree

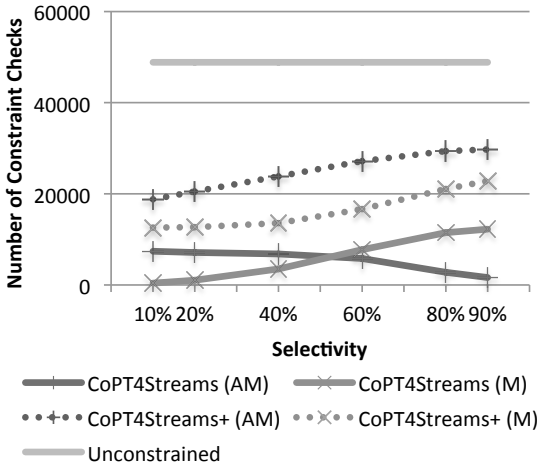

Figure 3: Average number of constraint checks

tern tree sorted according to some specific order. This means that all itemsets must be sorted before, which results in an overhead in time, that depend on that order.

As expected, as the selectivity increases, more itemsets can be removed from the tree, and therefore the size of the pattern tree is smaller, as well as the time needed to update smaller pattern trees. We can also confirm in fig. 1 that AM constraints allow us to prune much more itemsets than M constraints, leading to much smaller pattern trees. This is explained by the fact that itemsets that violate M constraints but have supersets that satisfy them, cannot be discarded from the tree. By similar reasons, AM constraints need, in average, less time to update the pattern tree than M constraints. Fig. 2 also shows that pushing AM or M constraints into the pattern tree results in a decrease of the update time, even when the selectivity os low.

In fig. 3 we analyze the average number of constraint checks. We can state that pushing constraints is always better, even with the naive approach, CoPT4Streams+, due to the resulting smaller pattern trees from one batch to another. Nevertheless, taking into account constraint properties to avoid constraint checks (CoPT4Streams) requires significantly less number of constraint checks. It is interesting to see that, as the selectivity increases, the number of constraint checks for AM constraints decreases, because the number of itemsets that can be discarded increases. But on the contrary, for M constraints, the number of tests increases along with the increase of the selectivity. This happens because the M strategy only stops checking when itemsets satisfy the constraint. And if there are more items that violate it, more itemsets need to be tested.

The behavior of Mixed constraints is consistent with the trends presented above: pushing them into the pattern trees results in much smaller trees, and therefore less constraint checks and update time, when comparing with both the unconstrained and the *CoPT4Streams+* algorithms. As the selectivity increases, the number of patterns in the trees decreases, as well as the time needed to process them. The number of constraint checks tends to be constant, independently of the selectivity of the constraints.

6. CONCLUSIONS

In this work, we describe a new set of strategies for pushing constraints into stream pattern mining, through the use of the efficient pattern-tree structure. These strategies take advantage of constraint properties, so that we can filter earlier the frequent itemsets that satisfy each constraint, and avoid unnecessary tests. By doing this for each batch of transactions, greatly decreases the size of the pattern trees that need to be maintained for this streaming environment, and therefore helps focusing the pattern mining task and returning much less but more interesting results. We also propose a general algorithm, named *CoPT4Streams*, that combines the defined strategies and is able to dynamically push any constraint into a pattern-tree, and still taking advantage of their properties (if some).

Experimental results show that the algorithm is effective and efficient. The pattern trees maintained are much smaller, which generally results in less time needed. It also checks much less nodes and needs less time than an approach that does not take into account constraint properties.

Despite the benefits of *CoPT4Streams*, it can be seen as

a post-processing approach (applied after the processing of each batch), which needs that an unconstrained algorithm run to first discover all possible frequent patterns. This usually takes much time, and results in a large quantity of frequent itemsets that need to be put in the pattern tree, and to be again evaluated later on. As future work, we intend to create a more balanced approach and use the strategies proposed here to filter itemsets during the actual discovery process.

7. ACKNOWLEDGMENTS

This work is partially supported by Fundação para a Ciência e a Tecnologia, under research project D2PM (PTDC/EIA-EIA/110074/2009) and PhD grant SFRH/BD/64108/2009.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB 94: Proc. of the 20th Intern. Conf. on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann.

[2] R. J. Bayardo. The hows, whys, and whens of constraints in itemset and rule discovery. In *Proc. of the 2004 European Conf. on Constraint-Based Mining and Inductive Databases*, pages 1–13. Springer, 2005.

[3] J.-F. Boulicaut and B. Jeudy. Constraint-based data mining. In *The Data Mining and Knowledge Discovery Handbook*, pages 399–416. Springer, 2005.

[4] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases: an overview. *AI Mag.*, 13(3):57–70, 1992.

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

[6] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15(1):55–86, Aug. 2007.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of the 2000 ACM SIGMOD*, pages 1–12, New York, NY, USA, 2000. ACM.

[8] C. K.-S. Leung, D. A. Brajczuk, and J. Yu. Efficient algorithms for stream mining of constrained frequent patterns in a limited memory environment. In *Proc. of the 2008 International Database Engineering and Applications Symposium (IDEAS 08)*, pages 189–198. ACM, 2008.

[9] C. K.-S. Leung and Q. I. Khan. Efficient mining of constrained frequent patterns from streams. In *Proc. of the 10th International Database Engineering and Applications Symposium (IDEAS 06)*, volume 0, pages 61–68. IEEE Computer Society, 2006.

[10] C. K.-S. Leung, L. V. S. Lakshmanan, and R. T. Ng. Exploiting succinct constraints using fp-trees. *SIGKDD Explor. Newsl.*, 4(1):40–49, 2002.

[11] C. K.-S. Leung and L. Sun. A new class of constraints for constrained frequent pattern mining. In *Proc. of the 27th Annual ACM Symposium on Applied Computing (SAC 12)*, pages 199–204. ACM, 2012.

[12] H. Liu, Y. Lin, and J. Han. Methods for mining frequent items in data streams: an overview. *Knowl. Inf. Syst.*, 26(1):1–30, 2011.

[13] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB 02: Proc. of the 28th Intern. Conf. on Very Large Data Bases*, pages 346–357, Hong Kong, China, 2002. Morgan Kaufman.

[14] R. Ng, L. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of data*, pages 13–24. ACM, 1998.

[15] J. Pei and J. Han. Constrained frequent pattern mining: a pattern-growth view. *SIGKDD Explor. Newsl.*, 4(1):31–39, 2002.

[16] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. of the 17th Int. Conf. on Data Engineering (ICDE 01)*, pages 433–442, Washington, USA, 2001. IEEE.

[17] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of the 21th Int. Conf. on Very Large Data Bases (VLDB 95)*, pages 407–419, San Francisco, USA, 1995. Morgan Kaufmann Publishers Inc.

[18] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. of the 3rd ACM SIGKDD Int. Conf. on Knowledge discovery and data mining (KDD 97)*, pages 67–73, 1997.