Efficient Mining of Constrained Frequent Patterns from Streams

Carson Kai-Sang Leung* 
Quamrul I. Khan

The University of Manitoba, Canada
{kleung, qikhan}@cs.umanitoba.ca

Abstract

With advances in technology, a flood of data can be produced in many applications such as sensor networks and Web click streams. This calls for stream mining, which searches for implicit, previously unknown, and potentially useful information (such as frequent patterns) that might be embedded in continuous data streams. However, most of the existing algorithms do not allow users to express the patterns to be mined according to their intentions, via the use of constraints. Consequently, these unconstrained mining algorithms can yield numerous patterns that are not interesting to users. In this paper, we develop algorithms—which use a tree-based framework to capture the important portion of the streaming data, and allow human users to impose a certain focus on the mining process—for mining frequent patterns that satisfy user constraints from the flood of data.

1. Introduction

Data mining refers to the search for implicit, previously unknown, and potentially useful information (such as frequent patterns) that might be embedded in data (within traditional static databases or continuous data streams). Since its introduction [1], the problem of finding frequent patterns has been the subject of numerous studies. These studies can be broadly divided into two “generations”. Studies in the first generation (e.g., the Apriori framework [2] and its enhancements [3, 6, 7, 8, 15, 19, 20, 21, 23]) basically considered the data mining exercise in isolation, whereas studies in the second generation explored how data mining can best interact with other key components (e.g., human users) in the broader picture of knowledge discovery. From this standpoint, studies in the first generation rely on a computational model where the mining system does almost everything, and the user is un-engaged in the mining process. Consequently, such a model provides little or no support for user focus (e.g., limiting the computation to what interests the user). However, the support for user focus is needed in many real-life applications where the user may have certain phenomena in mind on which to focus the mining (e.g., may want to find expensive snack items). Without user focus, the user often needs to wait for a long time for numerous frequent patterns, out of which only a tiny fraction may be interesting to the user. This motivates the call for constrained mining [5, 11, 13, 24]. Several constrained mining algorithms (e.g., CAP [22], DCF [18], Dense-Miner [4]) were developed.

The above studies mainly focused on mining frequent patterns (constrained or otherwise) from traditional static databases. However, over the past decade, the automation of measurements and data collection has produced tremendous amounts of data in many application areas. The recent development and increasing use of a large number of sensors has added to this situation. Consequently, these advances in technology has led to a flood of data in various application areas. We are now drowning in streams of data but starving for knowledge. In order to be able to “drink from a fire hose” (a metaphor for making sense of the streams of data), algorithms for extracting useful information/knowledge from these streams are in demand. This calls for stream mining [10, 16, 17, 25].

When comparing with mining from traditional static databases, mining from data streams is more challenging because data streams are continuous and unbounded. Let us elaborate. To find frequent patterns from streams, we no longer have the luxury of performing multiple data scans. Once the streams flow through, we lose them. Hence, we need some techniques to capture important contents of the streams (usually, the recent data) and ensure that the captured data can be fit into memory. Moreover, due to the continuous nature of data streams, a currently infrequent pattern may be frequent in the future, and vice versa. In other words, we have to be careful not to prune infrequent patterns too early; otherwise, we may not be able to get complete information such as frequencies of frequent patterns (because it is impossible to retract the pruned patterns). Despite these challenges, several algorithms (e.g., FP-streaming [12], Moment [9]) have been proposed in recent years to find frequent patterns (or their variants like closed or maximal patterns) from the streams.

Recall that, with unconstrained mining from traditional static databases, the user may need to wait for a long time for numerous frequent patterns, out of which only a tiny fraction is interesting to the user. This problem becomes more serious when mining from data streams because of the unbounded nature of streams (i.e., huge amount of data in the streams). So, integration of constrained
mining with stream mining would be helpful. However, to our knowledge, there is no work on integrating constrained mining with stream mining. Existing algorithms—including the most relevant ones like CAP [22], DCF [18], FP-streaming [12]—fall short in different aspects. For instance, while CAP and DCF are effective in capturing user constraints, they do not handle streaming data (i.e., not stream mining). While FP-streaming effectively mines frequent patterns from data streams, it does not handle constraints (i.e., not constrained mining).

Hence, some natural questions to ask are: Can we integrate constrained mining with stream mining in a tree-based framework? If so, how? Are there any efficient approaches? We answer these questions in this paper. Specifically, the key contribution of this work is the development of efficient algorithms, called approxCFPS and exactCFPS, for mining constrained frequent patterns (i.e., frequent patterns that satisfy user constraints) from streams of data. Note that this work is a non-trivial integration of constrained mining with stream mining in a tree-based framework. The resulting algorithms push the constraints deep inside the mining process. As a result, the computation for mining is proportional to the selectivity of constraints. Moreover, our exactCFPS algorithm generates all and only those frequent patterns (with complete frequency information) that satisfy the constraints. Thus, the number of patterns generated during the mining process is proportional to the selectivity of constraints. Furthermore, our algorithms only require a reasonable amount of memory space. As a preview, Table 1 shows the salient features of our algorithms as compared with their most relevant ones.

The paper is organized as follows. In the next section, related work is discussed. We start discussing our work in Section 3, where we push constraints into the mining process. While we propose an approximate algorithm for mining constrained frequent patterns from streams in Section 3, we propose an exact algorithm in Section 4. Section 5 shows the experimental results. Finally, conclusions are presented in Section 6.

2. Related work

In this section, we first provide some background about (i) constraints [22] and (ii) a stream mining algorithm called FP-streaming [12], which are both relevant to the remainder of this paper.

2.1. Constraints

Ng et al. [22] proposed a constrained frequent-set mining framework within which the user can use a rich set of constraints—including SQL-style constraints (e.g., $Q_1 \equiv \min(S.Price) \geq 20$, $Q_2 \equiv S.Type = \text{snack}$, $Q_3 \equiv \max(S.Qty) \geq 300$)—to guide the mining process to find only those itemsets satisfying the constraints. Here, constraint $Q_1$ says that the minimum price of all items in an itemset $S$ is at least 20; constraint $Q_2$ says that all items in an itemset $S$ are of type snack. Both constraints are anti-monotone because any supersets of an itemset violating the constraints also violate the constraints (e.g., if an itemset $X$ contains an item whose $Price < 20$, $X$ violates $Q_1$ and so do any supersets of $X$). Constraints $Q_1$, $Q_2$, and $Q_3$ are all succinct because one can directly generate precisely all and only those itemsets satisfying the constraints (e.g., by using a precise "formula", called a member generating function [22], that does not require generating and excluding itemsets not satisfying the constraints). For instance, itemsets satisfying $Q_3$ can be precisely generated by combining at least one item whose $Qty \geq 300$ with some optional items (of any $Qty$), thereby avoiding the substantial overhead of the generation and exclusion of invalid itemsets. It is important to note the following [22]: A majority of constraints are succinct. For constraints that are not succinct, many of them can be induced into weaker constraints that are succinct!

2.2. The FP-streaming algorithm

Giannella et al. [12] designed the FP-streaming algorithm to mine (unconstrained) frequent patterns from data streams. Given an incoming batch of transactions in the data stream, the first step is to call the FP-growth algorithm [14] with a threshold that is lower than the usual minimum support threshold $\text{mins}up$ to find "frequent" patterns. We call this lower threshold $\text{preMinsup}$. Here, an itemset is "frequent" if its frequency is no less than $\text{preMinsup}$. Note that, although we are interested in truly frequent patterns (i.e., patterns having frequency $\geq \text{mins}up \geq \text{preMinsup}$), FP-streaming uses $\text{preMinsup}$ to avoid pruning an itemset too early. An itemset $X$ having $\text{preMinsup} \leq \text{frequency}(X) < \text{mins}up$ is currently infrequent but may become frequent later; so, $X$ is not pruned.

Once the "frequent" patterns are found, the second step of FP-streaming is to store and maintain them in a tree structure called $\text{FP-stream}$. Key differences between an FP-tree and an FP-stream include the following. First, an FP-tree captures the transactions of a traditional database, whereas an FP-stream captures "frequent" patterns. To elaborate, each path in an FP-tree represents a transaction, and each path in an FP-stream represents an itemset. Second, each node in an FP-tree contains one frequency value, whereas each node in an FP-stream contains a natural (or logarithmic) tilted-time window table (containing multiple frequency values, one for each batch (or mega-batch) of transactions). Since users are often interested in recent data than older data, the FP-stream captures only some recent batches of transactions in the stream. As a new batch of transactions flows in, the window slides and the frequency values of each node shift as well.

To gain a better understanding of this algorithm, let us consider the following example.
3. Integrating constrained mining with stream mining: an approximate algorithm

In this section, we describe how we integrate constrained mining with stream mining in a tree-based framework. Let us start with a naïve approach, which we call FP-streaming++. It first applies FP-streaming (an unconstrained mining algorithm) to find “frequent” patterns from streams, and then applies constraint checking as a post-processing step to check if these “frequent” patterns satisfy the user constraint $C$. Recall from Section 2 that FP-streaming consists of two key steps: It first calls the FP-growth algorithm with preMinsup to find “frequent” patterns, and then stores and maintains these “frequent” patterns in a tree structure called FP-stream. While simple, FP-streaming++ suffers from at least the following problems/weaknesses:

1. As constraint checking is a post-processing step (i.e., constraint $C$ is not pushed inside the mining algorithm to effect pruning as early as possible), the number of patterns stored in the FP-stream is not proportional to the selectivity of $C$. This problem is worsened when $C$ is a highly selective one (i.e., when only a tiny fraction of “frequent” patterns satisfy $C$).

2. For the same reason, the computation for mining is also not proportional to the selectivity of $C$.

3. As the use of preMinsup is just a heuristic, there is no guarantee that all and only those frequent patterns (with complete frequency information) can be found.

An improved approach, which we call FP-streaming*, is to push the constraint $C$ inside the FP-streaming algorithm. A deep look at the FP-streaming algorithm reveals that the number of “frequent” patterns stored in the FP-stream depends on the number of patterns generated by FP-growth. So, if we apply the post-processing step of checking $C$ earlier (say, after obtaining “frequent” patterns from FP-growth but before storing them in the FP-stream), we could reduce the size of the FP-stream. By so doing, the number of “frequent” patterns to be stored in the FP-stream is proportional to the selectivity of $C$. This solves Problem 1. How about the other two problems/weaknesses?

3.1. Finding from streams the patterns that satisfy succinct anti-monotone constraints

Recall that a constraint can be succinct, and/or anti-monotone, or neither. While FP-streaming* works well for the general case (i.e., any constraint $C$—regardless of whether it is succinct or not), we could do better if $C$ is succinct. Specifically, we could push a succinct constraint $C$ deeper inside the mining process (i.e., inside the FP-growth algorithm used in the first step of FP-streaming).

In this section, we propose an approximate algorithm, called approxCFPS, for mining constrained frequent patterns (i.e., frequent patterns that satisfy succinct constraints) from streams. Our algorithm consists of the following main operations: (i) constraint checking of transactions in the current batch, (ii) construction of an FP-tree for this current batch, (iii) recursive growth of valid “frequent” patterns, and (iv) storing of these patterns in an FP-stream structure.

Note that a succinct constraint can also be anti-monotone. We call those succinct anti-monotone constraints as SAM constraints and those succinct non-anti-monotone constraints as SUC constraints. Let us first show how our approxCFPS algorithm handles a SAM constraint $C_{SAM}$. The algorithm exploits two nice proper-
ties of $C_{SAM}$: (i) anti-monotonicity (i.e., if a pattern violates $C_{SAM}$, then all its supersets also violate $C_{SAM}$) and (ii) succinctness (i.e., one can easily enumerate all and only those patterns that are guaranteed to satisfy $C_{SAM}$). Hence, any pattern $\nu$ satisfying $C_{SAM}$ must consist of only items that individually satisfy $C_{SAM}$. In other words, $\nu \subseteq \text{Item}^m$ (i.e., the set of items that individually satisfy $C_{SAM}$). Due to succinctness, items in $\text{Item}^m$ can be efficiently enumerated.

Our approxCFPS algorithm discovers frequent patterns satisfying $C_{SAM}$ as follows. We first apply constraint checking on all domain items in order to find $\text{Item}^m$. We then extract from the incoming batch of streaming transactions those items that are in $\text{Item}^m$, and capture them in an FP-tree. Items that are not in $\text{Item}^m$ can be discarded because any patterns contain any non-$\text{Item}^m$ item do not satisfy $C_{SAM}$. Once these $\text{Item}^m$ items are found, all we need to do is to recursively apply the usual FP-tree based mining process to each projected database of “frequent” patterns (i.e., apply to each $\alpha$-projected database—which is a collection of transactions having $\alpha$ as its prefix—where $\alpha \subseteq \text{Item}^m$). Like FP-streaming, we also use $\text{preMinSup}$ (instead of $\text{minSup}$) during the mining process. At the end, we find all valid “frequent” patterns and store them in the FP-stream structure. To gain a better understanding of how our approxCFPS algorithm handles $C_{SAM}$, let us consider the following example.

**Example 2** Consider the first batch of transactions shown in Example 1. Assume that the prices of items $a, b, c, d$ are 60, 10, 30, 40 respectively. Let constraint $C_{SAM}$ be the SAM constraint $Q_1 \equiv \min(S.Price) \geq 20$, $\text{minsup}$ be 3, and $\text{preMinSup}$ be 2. Our proposed approxCFPS algorithm discovers valid “frequent” itemsets as follows. It first enumerates from the domain items those valid items $a, c, d$ (i.e., items with individual $Price \geq 20$). Among them, item $c$ is infrequent (with frequency $< \text{preMinSup}$) and is thus removed. Then, approxCFPS builds an FP-tree. Afterwards, the usual FP-tree based mining process (with only frequency check) is applied recursively to subsequent projected databases. As a result, valid “frequent” itemsets $\{a\}, \{a, d\}, \{a, d\}$ are found. These itemsets are then stored in the FP-stream structure in the same fashion as in FP-streaming.

Note that approxCFPS (which computes only three valid “frequent” itemsets and stores them in the FP-stream) requires less computation and storage space than FP-streaming++ (which computes five “frequent” itemsets and stores them in the FP-stream, but only three of them are valid) and FP-streaming* (which also computes five “frequent” itemsets, but stores only the three valid ones in the FP-stream).

### 3.2. Finding from streams the patterns that satisfy succinct non-anti-monotone constraints

Next, let us turn our attention to how our approxCFPS algorithm handles a SUC constraint $C_{SUC}$. Note that for SUC constraints, they satisfy the “succinctness” property so that one can easily enumerate all and only those patterns that are guaranteed to satisfy $C_{SUC}$. However, SUC constraint do not satisfy the “anti-monotonicity” property. In other words, if a pattern violate $C_{SUC}$, there is no guarantee that all or any of its supersets would violate $C_{SUC}$. Hence, not all valid patterns are composed of only mandatory items (as for SAM constraints). Instead, any pattern $\nu$ satisfying $C_{SUC}$ is composed of mandatory items (i.e., items satisfying $C_{SUC}$) and possibly some optional items (i.e., items not satisfying $C_{SUC}$). In other words, a valid pattern $\nu$ is usually of the form $\alpha \cup \beta$, where (i) $\alpha \subseteq \text{Item}^m$ (the set of mandatory items) such that $\alpha \neq \emptyset$, and (ii) $\beta \subseteq \text{Item}^q$ (the set of optional items). Due to succinctness, items in $\text{Item}^m$ and $\text{Item}^q$ can be efficiently enumerated.

Hence, we cannot apply the same procedure as we did for $C_{SAM}$. Some modification is needed; otherwise, we may miss some valid frequent patterns. Specifically, our approxCFPS discovers frequent patterns satisfying $C_{SUC}$ as follows. We first apply constraint checking in order to divide the domain items into two sets—the set $\text{Item}^m$ consisting of all mandatory items and the set $\text{Item}^q$ consisting of all optional items. We then extract from the incoming batch of streaming transactions the items that belong to these two sets. These items are captured in an FP-tree in such a way that mandatory items appear below optional items (i.e., mandatory items are closer to the leaves, and optional items are closer to the root). With this item-ordering scheme, all we need to do is to recursively apply the usual FP-tree based mining process to each projected database of only those valid “frequent” patterns (i.e., apply to each $\alpha$-projected database—which is a collection of transactions having $\alpha$ as its prefix—where $\alpha \subseteq \text{Item}^m$). Like FP-streaming, we also use $\text{preMinSup}$ (instead of $\text{minSup}$) during the mining process. At the end, we find all valid “frequent” patterns that are guaranteed to satisfy $C_{SUC}$.
To summarize, our proposed approxCFPS algorithm exploits the properties of succinct constraints ($C_{SAM}$ or $C_{SUC}$), and pushes these constraints deeper inside the mining process (i.e., inside the FP-growth algorithm used in FP-streaming). As a result, pruning for constraint satisfaction is done once-and-for-all at the initial step when approxCFPS enumerates all the individually valid items (i.e., $\text{Item}^n$ items). At recursive steps/projected databases, all the valid “frequent” patterns can be grown from the projected databases of these $\text{Item}^n$ items, and no constraint checking is required. This solves our Problem 2; hence, the number of patterns stored in the FP-stream structure as well as the computation for mining is proportional to the selectivity of constraints.

4. Integrating constrained mining with stream mining: an exact algorithm

In this section, we show how we deal with Problem 3. Like FP-streaming, our approxCFPS algorithm also uses $\text{preMinsup}$ during the mining process (FP-growth). The reason is that, when dealing with data streams, a currently infrequent pattern may become frequent in the future. So, we better not prune these patterns too early; otherwise, we lose frequency information about these patterns. However, the use of $\text{preMinsup}$ is just a heuristic. Its success strongly depends on the value of $\text{preMinsup}$. If it is too high (e.g., too close to $\text{minsup}$), we may lose frequency information of some patterns. To another extreme, if it is too low, lots of redundant patterns (e.g., those patterns with frequency lower than $\text{minsup}$ but not lower than $\text{preMinsup}$) may be generated and stored in the FP-stream structure.

In this section, we propose an exact algorithm, called exactCFPS, for mining from streams those frequent patterns that satisfy succinct constraints. Here, we combine the FP-tree with the FP-stream structure. Our proposed algorithm consists of the following main operations: (i) constraint checking of transactions in the current batch, (ii) construction of a modified FP-tree, and (iii) recursive growth of valid truly frequent patterns from the modified FP-tree. There is no need for $\text{preMinsup}$ or the FP-stream structure.

Specifically, the exactCFPS algorithm discovers constrained frequent patterns from streams as follows. When a batch of transactions flows in, we first exploit constraints ($C_{SAM}$ or $C_{SUC}$) and push constraints deep into the mining process (i.e., deep inside FP-growth) in the same fashion that we did in approxCFPS. In other words, if it is a SAM constraint, we apply constraint checking on all items in the transactions to find those belong to $\text{Item}^n$; if it is a SUC constraint, we apply constraint checking on all items in the transactions to divide the items into two sets (one for $\text{Item}^n$ and another for $\text{Item}^0$). Then, we insert all those transactions into a modified FP-tree where (i) each tree path represents a transaction and (ii) each tree node contains a list of frequency value (instead of just one frequency value). Also, we keep all (frequent and infrequent) items: all $\text{Item}^n$ items for $C_{SAM}$, all $\text{Item}^n$ and $\text{Item}^0$ items for $C_{SUC}$. This is because a currently infrequent item may become frequent in the future. Then, when the next batch of transactions flows in, we shift the frequency values in the list at each node by removing the frequency value of the oldest batch of transactions and appending the frequency value of the newest batch. At any point $T$ in time, this modified FP-tree captures the most important portion (usually the recent portion w.r.t. $T$) of the streams so that truly frequent patterns (using the regular $\text{minsup}$) can be mined from the modified FP-tree. Since constraint checking has been done at the initial step and no further constraint checking is needed at recursive steps/projected databases (due to succinctness), all we need to do is to recursively apply the usual FP-tree based mining process to each projected database of valid frequent patterns. See the following examples.

Example 4 Consider the same batch of streaming data, auxiliary item information, and $C_{SAM}$ as in Example 2. Our proposed exactCFPS algorithm discovers valid truly frequent itemsets as follows. It first enumerates from the domain items those valid items $a, c, d$. Then, exactCFPS inserts transactions (with invalid items removed from batches of transactions) into a modified FP-tree.

When users want to find constrained frequent patterns, the usual FP-tree based mining process (with only frequency check) can then be applied recursively to subsequent projected databases of this modified FP-tree. As a result, a valid truly frequent itemset $(a, c, d)$ can be found from this first batch of transactions in the data stream.

Example 5 Consider the streaming data in Example 1, the same auxiliary item information, and $C_{SAM}$ as in Example 4. Then, during the insertion of transactions from the data stream, only $n_0 = 2$ frequency values are kept for each node in the modified FP-tree because the window size $n_0$ is 2. See the content of the modified FP-trees in Figure 2.

At time $T$ when the first two batches are read, the usual FP-tree based mining process (with only frequency check) can be applied recursively to subsequent projected databases to find valid truly frequent patterns (using $\text{minsup}$) from the modified FP-tree. This is because the tree captures the two most recent batches of transactions w.r.t. $T$.

Then, at time $T'$ when the third batch is read, the modified FP-tree needs to be updated to capture the two most recent batches w.r.t. $T'$ (the second and third batches) and discard the oldest batch in the window (e.g., the first batch). Specifically, when the window slides, the list of frequency values of each tree node is also shifted. See Figure 2.

So far, we have proposed and discussed several algorithms. Let us compare them in Table 1. These algorithms...
Table 1. Our proposed algorithms vs. the most relevant algorithms.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Constrained mining: Find constrained frequent patterns</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mining is proportional to constraint selectivity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tree-based mining: Use an FP-tree</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mine only valid, truly frequent patterns from the FP-tree</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mine only valid “frequent” patterns from the FP-tree</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mine all “frequent” patterns from the FP-tree</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Stream mining</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Do not need the FP-stream structure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Use an FP-stream structure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>The FP-stream stores only valid “frequent” patterns</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>The FP-stream stores all “frequent” patterns</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 3. Modified FP-tree vs. the FP-stream.

5. Experimental results

The experimental results cited below are based on streaming data generated by the program developed at IBM Almaden Research Center [2]. The data contain 1M records with an average transaction length of 10 items, and a domain of 1,000 items. Unless otherwise specified, we used \( \text{minsup} = 0.01\% \) and \( \text{preMinsup} = 0.0025\% \). We set each batch be of size 50,000 transactions and the window size be of two batches. All experiments were run in a time-sharing environment in a 700 MHz machine. The reported figures are based on the average of multiple runs. Runtime includes CPU and I/Os; it includes the time for FP-tree construction, frequent-pattern mining, and FP-stream construction steps (if appropriate). In the experiments, we compared the following algorithms that were implemented in C: (i) the naïve approach (FP-streaming++), (ii) the improved approach (FP-streaming*), (iii) our approximate algorithm (approxCFPS), and (iv) our exact algorithm (exactCFPS).

In the first set of experiments, we evaluated the effectiveness of constrained mining. We compared the runtimes of various algorithms. The \( y \)-axis of Figure 4 shows the runtime, and the \( x \)-axis shows the selectivity of the succinct constraint. A constraint with \( \text{pct} \% \) selectivity means \( \text{pct} \% \) of items is selected. The higher the \( \text{pct} \) value, the more is the number of selected items.

It is observed from Figure 4(a) that as the selectivity of the SAM constraint \( C_{SAM} \) decreased (i.e., fewer items are selected), the runtime remained unchanged for FP-streaming++. It is because FP-streaming++ ignores \( C_{SAM} \) at the early stage of the mining process, mines from the FP-tree that captures each incoming batch of streaming transactions to find all (valid and invalid) “frequent” patterns, stores them in the FP-stream (regardless of selectivity), and conducts constraint checking as a post-processing step. For FP-streaming*, as the selectivity of \( C_{SAM} \) decreased, the runtime decreased slightly. It is because FP-streaming* also ignores \( C_{SAM} \) at the early stage of the mining process and finds all “frequent” patterns (regardless of selectivity), but it stores only valid ones in the FP-stream. On the other hands, the runtimes for approxCFPS and exactCFPS decreased gradually as the selectivity decreased. This shows that the runtimes required by approxCFPS and exactCFPS depend on selectivity. Specifically, approxCFPS pushes
$C_{SAM}$ inside the mining process and finds only valid “frequent” patterns (i.e., ignores invalid ones) from the FP-tree that captures each incoming batch of streaming transactions, stores them in the FP-stream. For exactCFPS, it also pushes $C_{SAM}$ inside the mining process, but it mines valid truly frequent patterns from a modified FP-tree that captures the $n_b$ recent batches of streaming transactions. No FP-stream structure is needed. So, for both algorithms, the number of valid patterns depends on the selectivity of $C_{SAM}$. The computation for mining is proportional to the selectivity of the SAM constraint.

Figure 4(b) shows the results for the SUC constraints. Again, the computation for mining using the approxCFPS and the exactCFPS algorithms is proportional to the selectivity of the SUC constraint. It is important to note that, for succinct constraints ($C_{SAM}$ or $C_{SUC}$), constraint checking is performed early at the initial step instead of as a post-processing step.

In the second set of experiments, we compared the size of the FP-stream structure (i.e., space required by the FP-stream). The $y$-axis of Figure 5 shows the sizes of FP-stream, and the $x$-axis shows the selectivity of the succinct constraint. Again, Figure 5(a) shows the results for $C_{SAM}$ while Figure 5(b) shows the results for $C_{SUC}$. It is observed that as the selectivity of the constraints decreased (i.e., fewer items were selected), the size of the FP-stream remained unchanged for FP-streaming++. This is because FP-streaming++ finds all “frequent” patterns and stores them in the FP-stream (regardless of constraint selectivity). On the other hands, the sizes of the FP-stream for both FP-streaming* and approxCFPS decreased as the selectivity decreased because only valid “frequent” patterns were stored in the FP-stream structures. The experimental results show that the sizes of the FP-stream required by these two algorithms depends on constraint selectivity and were the same (due to the same collection of valid “frequent” patterns). The number of patterns stored in the FP-stream in proportional to the selectivity of constraints.

In the third set of experiments, we compared the tree size. We observed that as the selectivity of the constraints
decreased (i.e., fewer items were selected), the sizes of the FP-trees remained unchanged for both FP-streaming++ and FP-streaming* because they keep all “frequent” items in the trees. On the other hands, the size of the FP-tree for approxCFPS and that of the modified exactCFPS decreased as the selectivity decreased because they keep only valid items in the trees.

Regarding the size of the modified FP-tree for exactCFP, it was just $1.39 \times$ the size of the FP-tree and the FP-stream used in approxCFPS when selectivity was 10%; this ratio decreased to $1.09 \times$ when selectivity was 90%.

In addition to the above three sets of experiments, we have also evaluated the effects of various minsup on runtime. As expected, when the minsup increased, the runtimes of all algorithms decreased. Note that, when minsup increased, the size of FP-trees decreased for all algorithms. As a result, smaller FP-tree was needed to be traversed during the mining process, and thereby reducing the runtimes. Moreover, fewer patterns were considered frequent. This implied fewer patterns were needed to be stored in the FP-stream (except for the exactCFPS algorithm, which do not use the FP-stream at all).

Furthermore, we run scalability test. The results showed linear scalability (w.r.t. the number of transactions in a batch as well as the number of batches) for these algorithms.

To summarize, the above results show the importance and the benefits of using our developed algorithms for efficient mining of the frequent patterns that satisfy the user constraints from the flood of data (i.e., efficient constrained frequent-pattern mining from streams).

6. Conclusions

A key contribution of this paper is to develop efficient approximate and exact algorithms—called approxCFPS and exactCFPS, respectively—for mining constrained frequent patterns from streams. These algorithms are a non-trivial integration of constrained mining, tree-based mining, and stream mining. Consequently, they (i) enable human users to impose a certain focus on the mining process, (ii) capture the important portion of streaming data, and (iii) find constrained frequent patterns with complete frequency information. Among the two algorithms, approxCFPS finds constrained approximately “frequent” patterns (i.e., all frequent patterns together with some sub-frequent patterns) and stores them in an FP-stream structure, whereas exactCFPS modifies the FP-tree for mining and combines it with the FP-stream into a data structure for exact mining for constrained frequent patterns. By pushing constraints deep inside the mining process, the amount of mining computation and item storage in the tree is proportional to the selectivity of constraints.

Acknowledgement. This project is partially sponsored by Natural Sciences and Engineering Research Council of Canada (NSERC) and The University of Manitoba in the form of research grants.

References