Infrequent Pattern Mining on Data Streams with SRP-Tree and RP-Tree Variants

[redacted]

Brett Aseltine Eran Efron Jared Friesen Eddie Wat

Abstract—**In this document, we investigate rare pattern mining on data streams and compare the performance of RP-tree and SRP-tree algorithms with their extensions using landmark, sliding window, and time-fading models. We use datasets from the Frequent Itemset Mining Implementations (FIMI) repository containing Mushroom, Retail, T10I4D100K, and T40110D100K, to evaluate the performance and limitations of these algorithms. We describe the RP-tree and SRP-tree extensions and analyse their implementation and possible issues. In our results we show the performance of these algorithms on landmark, sliding window, and time-fading models, and examine the advantages of our RP-Tree extensions over their SRP-Tree extensions. We conclude with a discussion of possible future research directions.**

I. Introduction

Pattern mining has long been a topic of research within the field of data mining. Pattern mining is a data mining method that finds interesting patterns that exist in the data. Frequent pattern mining has received the most attention in terms of research conducted and methods produced [1]. Frequent pattern mining involves identifying patterns that frequently occur together within a database, where a level of support is used to define what is considered as frequent. Many works have been proposed regarding frequent pattern mining. The initial works revolved around the Apriori-like candidate generation and test approach [2]. A monumental step in frequent pattern mining was the development of the FP-Tree structure by Han et al [3]. As the title of Han's article [3] suggests, the FP-Tree structure removes the need for candidate generation. FP-Growth was simultaneously provided in [3] as a means to mining the FP-Tree.

In contrast to frequent pattern mining, rare pattern mining involves identity patterns that occur infrequently within a database [4]. According to [4], in many domains, events that occur frequently may be less interesting than events that occur rarely, since frequent patterns represent the known and expected while rare patterns may represent unexpected or previously unknown associations, which is useful to domain experts. For example, consider the medical domain. Frequent responses to medications are not as interesting as rare responses to medications which may indicate an adverse reaction or possible drug interaction.

The domain of rare-pattern mining has received far less academic attention than its counterpart of frequent pattern mining even though rare pattern identification has many applications in a wide range of domains. One way that rare data mining can be used in the real world is to help with credit card fraud detection. The databases that store credit card transactions contain millions of entries, requiring one to use some form of a data mining algorithm to extract useful information. Let us take the credit card transactions of an average person as an example, they will be using their credit card for everyday purchases quite often. These could be groceries, gas, electric and water bill payments, etc. These purchases will form frequent patterns that are not evident of fraudulent activity. Credit card fraud will often appear as sudden, infrequent purchases of expensive items, such as luxury clothing, fancy restaurants, high class appliances, etc. A normal, frequent pattern mining algorithm would miss these events. In order to detect these anomalies, a rare pattern approach to mining must be used.

Another use for rare data mining is network intrusion detection. The vast majority of network traffic is benign and poses no threat. Malicious network intrusions thus represent a small minority of all internet traffic. In order to find these intrusions inside the massive network packet databases, one must perform some rare pattern mining. Medical diagnostics is another place where rare pattern mining can be useful. Most diseases manifest themselves as a series of symptoms that are uncommon in the general population. Finding patients with these rare combinations of symptoms with rare pattern mining could help diagnose and treat them faster. One specific example is mammogram pictures, in which only a small fraction of the image's pixels will be cancerous cells [5].

An additional use for rare pattern mining is in insurance risk modeling. Insurance is one of the industries where the 80-20 rules applies the most. Most of the costs for an insurance company will come from a small number of their insurers. Therefore, it makes sense to use rare pattern mining to assess the risks associated with the small number of costly insurance claims. One final use for rare pattern mining is in web mining. Take an online vendor, for example. Out of all the people that visit said vendor's website, only a few of them will end up purchasing their products. In order to find out more information about their customers, it would be best for the vendor to use rare pattern mining to identify the purchasers and filter out the visitors.

A common occurrence can be identified from the provided applications. Many of these applications require the use of a data stream, that is, data that is continuous and unbounded as opposed to a static database. Upon performing research regarding the domain of mining rare patterns from data streams, one will likely encounter very few resources on the topic. This is a topic within the pattern mining domain that has not yet received a significant amount of academic attention. However, as previously exemplified, there are numerous applications for mining rare patterns from data streams.

Given that the topic of rare pattern mining of data streams has not received considerable research, there are currently a limited number of approaches to realize this functionality. Techniques exclusively developed for rare pattern mining and techniques exclusively developed for stream mining do not necessarily compose perfectly together. There needs to be dedicated research to specifically this topic. Huang [2] has proposed a novel approach crafted to mine rare patterns from a data stream using a sliding window technique. Although the implementation of a sliding window is a clever way to limit the system memory requirements, it does not preserve all data. Once a transaction has passed through the windows, it is completely discarded. The definition of a rare pattern indicates it has a low occurrence in the database. Consequently, techniques must ensure an ample amount of data is kept in the working algorithm or else few rare patterns will be found.

There are certain advantages a batch-mining approach can provide as opposed to a delayed-mining approach [6]. Firstly, Intermediate algorithms are often better at early pattern detection since data is analyzed upon arrival as opposed to delayed-mining which does not analyze until the user has specified. Second, intermediate stream algorithms can be more resource-efficient because they can mine batches as they arrive, rather than having to store all the data before analysis begins. Third, since intermediate algorithms mine batches upon arrival, mined itemsets are more accessible as opposed to being tied up in a data-structure for processing upon user request.

A. Our Contributions

Our contribution in this paper is to propose 6 intermediate algorithms for mining rare patterns from a data stream. Three of these algorithms are extensions to the RP-Tree provided by [4] in order to handle stream data. They independently include a landmark, time-fading, and sliding window approach. The 3 additional algorithms we provide are modifications to the SRP-Tree mining algorithm proposed by [2]. The SRP-Tree algorithm utilizes a sliding-window with a delayed-mining approach, we present 3 algorithms that independently implement a landmark, time-fading, and sliding-window using an intermediate mining approach. As well, each of these algorithms will use the concept of a *preMinRareSup* which will be discussed further in the reading. To our understanding a *preMinRareSup* has not been applied to mining rare patterns from data streams. In addition to proposing pseudocode representations of these algorithms, Java implementations are accessible to the reader. We empirically illustrate the performance metrics of each of these 6 new algorithms and demonstrate the effect of the *preMinRareSup* variable.

II. Related Works

The initial works concerning rare pattern mining relied on the Apriori style candidate generation and test method. With this approach, k-itemsets (itemsets of cardinality k) are used to generate $k + 1$-itemsets, which are then pruned using the downward closure property. Rarity, AfRIM, ARMIA, and Apriori-Inverse are algorithms employing an Apriori style candidate generation and test to detect rare itemsets.

Troiano et al. [7] presented the Rarity algorithm. It uses a top-down approach to find rare itemsets in a static database. It starts by finding the longest rare pattern. As it moves down the power set, it uses lattice cutting to prune frequent itemsets. Therefore, only rare itemsets are developed.

Adda et al. [8] presented the AfRIM algorithm, which begins with the itemset that contains all items found in the database. Candidate generation occurs by finding common k-itemset subsets between all combinations of rare $k + 1$-itemset pairs in the previous level. Note that AfRIM examines itemsets with zero support, which may be inefficient.

Szathmary et al. [9] proposed two algorithms that can be used together to mine rare itemsets: MRG-Exp and ARIMA. The first algorithm, MRG-Exp, finds all MRG by using MGs for candidate generation in each layer in a bottom-up fashion. The second algorithm, ARIMA, uses these MRGs to generate the complete set of rare itemsets.

Apriori-Inverse proposed by Koh et al. [10] is used to mine perfectly rare itemsets, which are itemsets that only consist of items below a maximum support threshold (maxSup). Apriori-Inverse is similar to Apriori, except that at initialization, only 1-itemsets that fall below maxSup are used for generating 2-itemsets. Since Apriori-Inverse in- verts the downward-closure property of Apriori, all rare itemsets generated must have a support below maxSup. In addition, itemsets must also meet an absolute minimum support, for example 5, in order for them to be used for candidate generation. Since the set of perfectly rare-rules may only be a small subset of rare itemsets, Koh et al. also proposed several modifications that allow Apriori-Inverse to find near-perfect rare itemsets.

All the above methods employ a candidate generate-and-test approach used in Apriori which can be computationally expensive. Additionally, these algorithms attempt to identify all possible rare itemsets, and as a result require a significant amount of execution time. RP- Tree algorithm was proposed by Tsang et al. [4] as a solution to these issues. RP-Tree avoids the expensive itemset generation and pruning steps by using a tree data structure, based on FP-Tree [3], to find rare patterns. However, it requires two database scans which is not suitable in a data stream environment.

There has not been a significant amount of research regarding rare pattern mining in data streams. The first proposition was from Huang et al [6] with the presentation of the Streaming Rare Pattern Tree (SRP-tree) algorithm. This algorithm uses a sliding window model to quickly and accurately detect infrequent patterns in a data stream environment.

To our understanding, there has not been any work mining rare patterns from data streams using a batch/intermediate mining technique, any works using the landmark window or a time-fading window model, or the use of a *preMinRareSup*. Our work incorporates these techniques.

III. Background and Preliminaries

In this section, we provide definitions of key terms that explain the concepts of rare pattern mining in a data stream. To begin, we shall present the concepts associated with frequent pattern mining given it has provided the foundation for rare pattern mining.

A. Frequent Pattern Mining

The problem of frequent pattern mining is that of finding relationships among items in a database [2]. Resource [2] describes the problem as follows: given a database D with transactions $T_1 \dots T_N$, determine all patterns P that are present in at least a fraction s of the transactions. The fraction s (represented as either a fraction or an absolute number) is referred to as the minimum support. Let I = $i_1, i_2, \dots, i_n$ be a set of literals, called items, that represent units of information in an application domain. A set X = $i_l, \dots, i_m \subseteq I$ and $l, m \in [1, n]$, is called an itemset, or a k-itemset if it contains k items. A transaction t = (tid, Y) is a tuple where tid is a transaction-id and Y is a pattern. If $X \subseteq Y$, it is said that t contains X or X occurs in t.

B. Rare Itemset Mining

Our work relies on the definitions of rare itemsets provided by Tsang et al [4]. An itemset is considered rare if its support falls below a threshold, called the minimum frequent support (*minFreqSup*) threshold. A consideration that must be taken when mining rare items is to account for noise. We use *minRareSup* as a noise filter, in which items with support strictly below this threshold are considered noise. Then, an itemset is considered rare if it has support strictly less than the minimum frequent support threshold but also above or equal to the minimum rare support threshold. [4] provides a further means to classifying rare items. *Rare-item-itemsets* are itemsets that consists of only rare items and itemsets that consists of both rare and frequent items. *Non-rare-item-itemsets* are itemsets which consists of only frequent items which fall below the minimum support threshold. For clarity these concepts will be demonstrated with an example, and then presented formally.

Consider a domain with the four items a, b, c, z and the itemset supports a = 0.50, b = 0.50, c = 0.50, z = 0.02, ab = 0.30, ac = 0.15, az = 0.15, and let minFreqSup = 0.2 and minRareSup = 0.01. Then, itemset ab is not a rare itemset because its support is greater than minFreqSup. Itemset ac is a non-rare-item itemset because all items are frequent and its support lies between minFreqSup and minRareSup. Itemset az is a rare-item itemset because it includes a rare item z and its support lies between minRareSup and minFreqSup.

1) Definitions: Definition 1: An itemset X is a *rare itemset* iff
$supportX < minFreqSup, supportX \geq minRareSup$

Definition 2: An itemset X is a *non-rare-item itemset* iff $\forall x \in X, supportx \geq minFreqSup, supportX < minFreqSup$

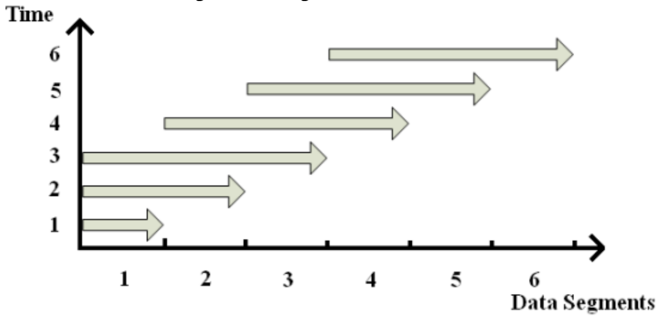
Definition 3: An itemset X is a *rare-item itemset* iff $\exists x \in X, supportx < minFreqSup, supportX < minFreqSup$

C. Stream Mining

A data stream DS can be formally defined as an infinite sequence of transactions $DS = [t_1, t_2, \dots, t_m]$ where $t_i, i \in [1, m]$ is the ith transaction in the data stream. We shall discuss three window models for working with data streams. We may interchangeably use the terms batch and data segment in the following discussion, each of which refers to a collection of transactions received in a data stream.

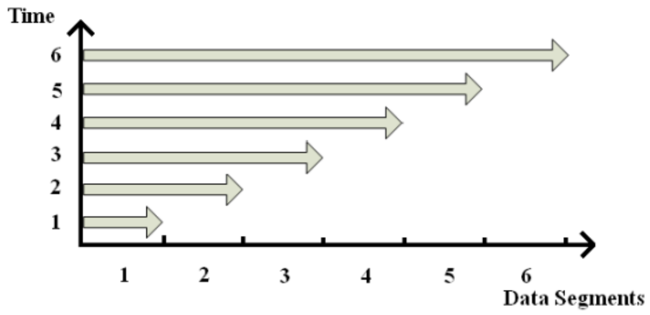
1) Sliding Window Model: A window W is a set of all transaction between the i th and j th (where $j > i$) transactions and the size of W is $|W| = j–i$. The size of the window is defined by the user, and slides over the data stream over time. A window may also be represented as a set of batches. Figure 1.1 obtained from [12] provides an illustration of the sliding window model for batch data. Initially, the window is empty. As data flows, the window begins to fill. Note that the window will never hold more data than the user defined window size. Should the window be full and slide over new data, the old data is removed from the window. As a consequence of such window behaviour, old data is completely discarded.

Fig. 1. Sliding Windows Model



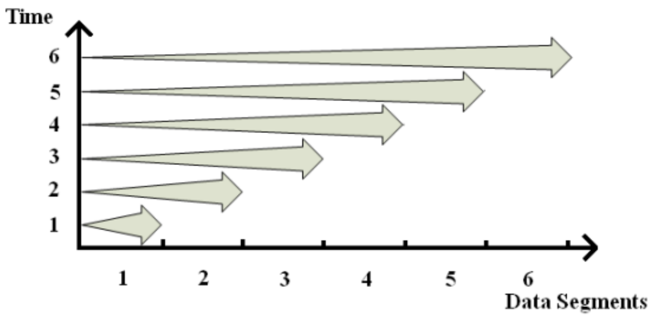
2) Landmark Window Model: The landmark model is not a fixed size windowing technique. The windows starts from a user specified time, called the *landmark time*, and continues to grow as new transactions are obtained from the stream. Figure 2 obtained from [?] provides an illustration of the landmark window technique. Notice there is no information removal after the landmark time. Consequently, should the user specify the landmark time at the first data point then all stream data will be preserved and there is no information loss.

Fig. 2. Landmark Window Model



3) Time-fading (dampened) Window Model: The time-fading window model is a technique to prioritize current data over older data. Figure 3 obtained from [12] provides an illustration of the time-fading window model. Upon being received from the stream, data items are added to the window with their current frequency. When a new batch enters the stream, all older batch are reduced by a *time-factor* (denoted α) producing a compounding effect on the weighting of data. Consequently, as the window slides along the data stream, older data becomes less significant, but data is never fully discarded. That is, once a transaction has been encountered in the stream, it remains in the window but with an ever-reducing significance.

Fig. 3. Time Fading (dampened) Window Model



To better illustrate the three window techniques we provide an example based on the simplified database in table 1.

1) Example with sliding window: Using a window of size 2 and after processing batch 3, the window returns the value 4 + 6 = 10.
2) Example with landmark model: Using a landmark of 1 and after processing batch 3, the window returns 2 + 4 + 6 = 12.
3) Example with time-fading model: Using a time factor of 0.8and after processing batch 3, the window returns 2*(0.8)2 + 4*(0.8) + 6 = 10.48.

D. Intermediate (Batch) Stream Mining

Batch-mining [13] is a stream mining technique wherein each batch of data is mined and the resultant itemsets with

TABLE I
SIMPLIFIED EXAMPLE DATABASE

Batch	First itemset support
1	2
2	4
3	6

their corresponding frequencies are stored in an intermediate structure that is continuously updated. We will interchangeably use the terms batch mining and intermediate mining. The choice of intermediate data structure can have significant effect on the run-time of the streamlining algorithm. Our implementations use a table as the intermediate data structure since it allows for very efficient insertion and access. Itemsets are represented as keys, and the corresponding values are the support of that itemset. With this implementation, an itemset can be accessed or inserted in O(1) time. This becomes very beneficial when working with multiple batches since for each newly mined itemset, we must determine if it already exists in the intermediate structure. There are two possibilities to consider when inserting a newly mined itemset into the table. One case arises if the item does not yet exist in the table. The search can be performed in O(1) time, and the insertion can be performed in O(1). The other case (when the itemset already exists in the table) requires an access of O(1) time and a support update of O(1) time. Thus, the time complexity to update the intermediate table upon mining a batch B is thus O(*m*) where m is the number of mined itemsets from batch *B*.

To illustrate the importance of an efficient intermediate structure, consider an implementation with a list. If the newly mined itemset does not exist in the list, the process takes O(n) time to perform the search and O(1) to insert the item resulting in O(n) time. If the itemset does exist in the list, the process again takes O(*n*) time since a search must be performed to find the itemset. Thus, the time complexity to update a list intermediate structure upon mining a batch *B* is O($n * m$) where n is the number of items in the list (previously mined items) and m is the number of newly mined items from batch *B*.

E. RP-Tree Algorithm

The RP-Tree algorithm proposed by Tsang et al. [4] is a novel proposal in the domain of rare-pattern mining and provides the foundation for our work. The RP-Tree algorithm is a modification of the FP-Growth algorithm proposed by [3]. FP-Growth is a frequent itemset mining algorithm which uses a frequent-pattern tree (FP-Tree) to store a of database transactions and reduces the required number of database scans to two. The first scan finds the set of items in the database with support over the minimum frequent support threshold, while the second scan is used to construct the initial FP-tree. RP-Tree algorithm follows a similar schema, using the first database scan to count item support, and a second database scan to build the initial tree. RP-Tree has the unique feature of using only the transactions which include as least one rare item when building the initial tree and pruning the others. This restriction is followed since transactions with only rare items cannot contribute to the support of a rare-item itemset. Consider a database with rare items a, b, c. A transaction must contain at least one of a, b, or c, to avoid being pruned. It should be noted that the ordering of items in a transaction during insertion into the initial tree is according to item frequent of the original database (not pruned database).

Following the construction of the initial tree, conditional pattern bases and conditional trees for each rare item are constructed. Each conditional tree and the corresponding rare item are used as the arguments for FP-Growth. Tsang et al. [4] present a simplified version of FP-Growth in their work. It should be noted that single prefix path optimization can be used to improve the performance of FP-Growth. MinRareSup is used as the threshold to prune items from the conditional trees. The union of results from each FP-Growth call is a set of itemsets that each contain a rear-item, or rare-item itemset. Tsang et al. [4] present the two conditions that demonstrate the results of RP-Tree in the complete set of rare-item itemsets. Namely,

1) Rare-items will never be the ancestor of a non-rare item in the initial tree due to the tree construction process.
2) All itemsets that involve a particular item a can be found by examining all nodes of a and the nodes of all items that have a lower support than a in the initial tree.

In algorithm 1, we present the pseudocode representation of the RP-Tree algorithm from [4]. A pseudocode representation of FP-Growth with single prefix path optimization can be found at [3].

Algorithm 1 RP-Tree

```
1: Input: $D, minRareSup, MinFreqSup$;
2: Output: $results$;                       ▷ Set of rare-item itemsets
3: Initialisation:
4: $allItems \leftarrow$ (all unique items in $D$);
5: countSupport($allItems$);                ▷ First scan of database
6: $rareItems \leftarrow (i \in allItems | i.supp \geq minRareSup \wedge i.supp < minFreqSup)$;
7: $rareItemTrans \leftarrow (t \in D | \exists r \cdot r \in rareItems \wedge r \in t)$;
8: $tree \leftarrow$ constructTree($rareItemTrans$);   ▷ Second scan of database
9: Mining:
10: $results = \emptyset$;
11: for item $a$ in $tree$ do
12:     if $a \in rareItems$ then
13:         construct $a's$ conditional pattern-base and then $a's$ conditional FP-Tree $Tree_a$;
14:         $results \leftarrow results \cup$ FP-Growth($Tree_a, a$);
15:     end if
16: end for
17: Return $results$;
```

F. SRP-Tree Algorithm

SRP-Tree is an algorithm to mine rare patterns within data streams proposed by [2]. Unlike static mining approaches which have the luxury of multiple database scans, streaming algorithms are typically restricted to only one database scan. As such, SRP-Tree uses only one database scan to mine rare patterns. Recall non-streaming techniques such as RP-Tree use the first scan to order items in a transaction by frequency before inserting them into the tree. Since SRP-Tree cannot perform multiple scans, items are read from a transaction and added to the tree in the same scan. Consequently, the SRP-Tree cannot be built based on the frequency of items within the data stream. SRP-Tree is an algorithm that mines rare patterns in a data stream using a delayed sliding window and tree-based approach.

Huang et al. [2] propose two variations of the SRP-Tree algorithm. Our extensions are based on the SRP-Tree with connection table and thus we shall present the features of this variant. In this approach, items from an incoming transaction in the stream are inserted into the tree based on a canonical ordering. The use of a canonical ordering allows for transaction content to be read from the data stream and have tree nodes organized according to a particular order. A result of using a canonical ordering to build the tree is that the ordering of items is unaffected by changes in frequency from incremental updates. According to [2], the frequency of a node in the tree is at least as high as the sum of frequencies of its children. However, this does not guarantee the downward closure property which exists in a tree ordered in frequency-descending order. The downward closure property in a traditional rare pattern tree mining algorithm (such as RP-Tree) in which rare-items will never be the ancestor of a non-rare item in the initial tree due to the construction process being violated. Thus, [2] propose an item list called a Construction Table which keeps track of each unique item in the window and the items they co-occur with along with their respective frequencies.

Only items with a lower canonical ordering in the transactions are recorded in the Connection Table. Consider a canonically ordered transaction x, y. The Connection Table stores x is connected to y with frequency of 1, but does not store y is connected to x. Huang et al. [2] validate this approach based on the properties of the canonical ordering in the construed ted tree, specifically, item x will always be the ancestor of item y. As a result of the inherent bottom-up mining approach, if item y is mined then item x is also mined, but the reverse is not guaranteed.

When mining is initiated, conditional pattern bases and conditional trees are constructed for each rare item and their connected items in the Connection Table. Thus, the addition of the Connection Table is required ensures the complete set of rare-item itemsets is captured. Like RP-Tree, each conditional tree and corresponding item are used as arguments in FP-Growth. An important distinction to note when comparing SRP-Tree to RP-Tree is that SRP-Tree is built using all transactions in the window, while RP-Tree only uses transactions with a rare item to build the tree.

We provide a pseudocode description of SRP-Tree with Connection Table as depicted in [2] in algorithm 2.

Algorithm 2 SRP-Tree (Connection Table)

```
1:  Input: $DS, W, B, minRareSup, MinFreqSup$;
2:  Output: $results$ (Set of rare item itemsets);
3:  while exist($DS$) do
4:      $t \leftarrow$ new incoming transaction from $DS$;
5:      currentBlockSize $\leftarrow$ currentBlockSize + 1;
6:      updateItemFreqList($t$);
7:      updateConnectionTable($t$);
8:      $tree \leftarrow$ updateTree($t$);
9:      Mining at the end of block
10:     if $B$ == currentBlockSize then
11:         currentBlockSize $\leftarrow$ 0;
12:         $results = \emptyset$;
13:         $I \leftarrow$ (all unique items in $W$);
14:         $R \leftarrow (i \in I | supw(i) \geq minRareSup \wedge supw(i) < minFreqSup)$;
15:         $C \leftarrow (k \in R, j \in connectionTable(k)\ |supw(k) \geq minRareSup)$;
16:         for item $a$ in $tree$ do
17:             if $a \in R$ or $a \in C$ then
18:                 construct $a's$ conditional pattern-base and then $a's$ conditional FP-Tree $Tree_a$;
19:                 $results \leftarrow results \cup$ FP-Growth($Tree_a, a$);
20:             end if
21:         end for
22:     end if
23: end while
24: Return $results$;
```

IV. Main Body

A. RP-Tree Extensions

The RP-Tree algorithm is designed to mine rare-item itemsets from a static database. In this section, we propose 3 extensions to the RP-Tree algorithm to allow the mining of rare patterns from data streams. We employ a batch-mining approach [13] that uses an intermediate structure to store mining results. We will interchangeably call batch mining intermediate mining. We make use of three commonly employed windowing techniques: landmark window, time-fading window, and sliding-window.

General Theme for RP-Tree Extensions
Each of the following 3 algorithms are an addition the RP-Tree algorithm. These algorithms employ an intermediate mining approach, wherein each batch of the data stream is mined and stored. To mine, we utilize the RP-tree algorithm. The RP-tree algorithm is an appropriate selection because it is more efficient than apriori based rare-pattern mining approaches in terms of both speed and memory requirements [4]. Speed and memory requirements are crucial to a successful intermediate stream mining approach, because each batch is mined upon its reception resulting in many instances of mining occurring. After an individual batch is mined using *preMinRareSup*, the resultant rare-item itemsets are added to a table structure storing the results. Consistent with an intermediate mining approach, upon user specification, truly rare rare-item itemsets are yielded using *minRareSup* threshold.

Since stream mining algorithms are restricted to a single scan of the database, our proposed algorithms are restricted to only performs one scan of the data stream. The traditional RP-Tree algorithm requires two scans of the database. RP-tree performs the first of two database scans to count item support. During the second scan, RP-Tree uses only the transactions which include at least one rare item to build the initial tree, and prunes the others, since transactions that only have non-rare items cannot contribute to the support of any rare-itemset. Our algorithms must account for both of these required functionalities while following the restriction of only one database scan.

During the reading of the current batch, these algorithms count item support (like RP-tree) while also storing the transactions of the current batch in memory. This allows the algorithms to then process the batch data a second time by reading from memory. It should be noted that only one batch is stored at a time. Given that one batch is typically only a small subset of the data stream, the memory requirement will likely remain practical. This memory read (second data encounter) performs the same functionality as the second database scan of RP-tree, that is, build the RP-tree.

The database scanning and data processing proposed above can be considered as the initialization stage for the current batch data. The initialization stage is concluded when each transaction of the current batch has been processed and there are no more transactions to read. Following such steps, the next course of action is to mine the tree.

Since the construction of the tree is consistent with an RP-Tree, we can take a similar approach to mine the tree for rare patterns. There is one significant difference our algorithms must account being designed to manage stream data. It is possible the case arises where an item is currently too rare (below *minRareSup* and thus considered noise) but may become a rare pattern in the future (as the stream continues to provide data). As such, itemsets with support less than *minRareSup* cannot be confidently pruned. Our solution takes inspiration from the work of Leung et al. [14] in mining frequent itemsets from streams of uncertain data. While [14] proposes a *preMinSup* (where $preMinSup < minsup$) to avoid pruning items that may become frequent upon reading future batches, we introduce the concept of a *preMinRareSup* (where $preMinRareSup \leq minRareSup$). When mining a batch of data, we use the *preMinRareSup* as the pruning threshold. This new parameter provides buffer room for an itemset that may currently be noise but may become a rare pattern in the future.

As presented in [4], the result of RP-Tree mining is the complete set of rare-item itemsets with their corresponding frequency. Since we apply *preMinRareSup* when mining each batch, RP-Tree will return the complete set of rare-item itemsets based on *preMinRareSup*. Initially, these itemsets are stored in an intermediate table data structure. Upon user specification, truly rare itemsets can be obtained based on the *minRareSup*.

1) RPStreamLandmark: The RPStreamLandmark algorithm employs the general theme as described above while additionally incorporating a landmark variable. There are two possibilities that can occur during a single batch scan. If the user specified *landmarkTime* is greater than the current read time, then the transaction is ignored. The other case, when the *landmarkTime* is less than or equal to the current read time, then the current transaction (and all those going forward) will be processed. Algorithm 3 depicts a pseudocode representation of RPStreamLandmark.

Provided is an example of the RPStreamLandmark algorithm using the dataset from table 2. Batch 1 is read, and the support ordered list of all items is $<< a : 4 >, < b : 4 >, < c : 4 >, < e : 3 >, < d : 1 >>$. Using *minFreqSup* = 3, *preMinRareSup* = 0, and *minRareSup* = 0, it can be deduced item d is the only rare item and included in the rare item set.

TABLE II
TRANSACTION DATABASE

Batch ID	TID	Transactions
1	1	a,b,d
	2	a,c
	3	a,b,c,e
	4	b,c,e
	5	a,b,c,e

Construction of the initial RP-Tree only uses transactions containing a rare item. Thus, only transaction 1 is used to build the initial RP-Tree as illustrated in figure 4(a). In contrast, an initial FP-Tree constructed would use the entire database, as shown in figure 4(b). Rare items are found by building conditional pattern bases and conditional RP-Trees for each rare item, in this case d. Then, each conditional RP-Tree and conditional item are used as parameters for FP-Growth. The resultant itemsets are added to the intermediate table as shown in table 3. This process continues on further batches until the user mines for truly rare itemsets. The set of truly rare itemsets is all itemsets in the intermediate table with support $\geq$ *minRareSup*.

TABLE III
INTERMEDIATE DATA STRUCTURE AFTER PROCESSING BATCH 1

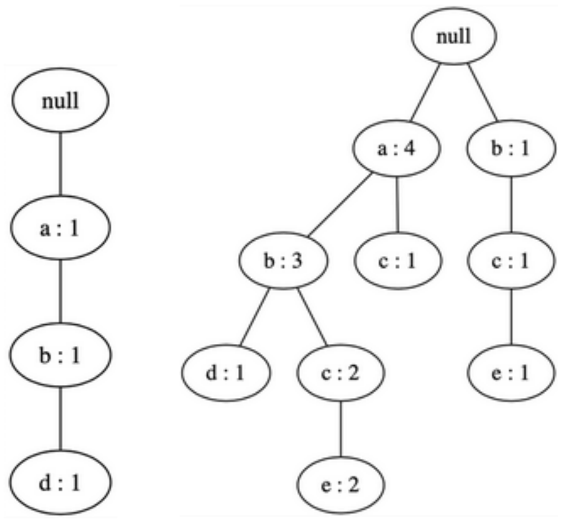
Itemset	Support
d	1
ad	1
bd	1
abd	1

2) RPStreamTimeFading: The RPStreamTimeFading algorithm employs the general theme as described above while additionally incorporating a *timeFactor* variable. Once a batch is mined it is added to the intermediate data structure with the corresponding mined support. Alongside the support all other data in the intermediate data structure is multiplied by the *timeFactor* variable. Therefore, this processing results in newer data exerting a greater impact compared to older data. Algorithm 4 depicts a pseudocode representation of RPStreamTimeFading.

3) RPStreamSlidingWindow: The RPStreamSlidingWindow algorithm employs the general theme as described above while additionally utilizing a sliding window. The size of the window is specified by the user. Once a batch is processed the mined itemsets are added to the window with their corresponding frequencies. There are two possibilities that can occur for

Algorithm 3 RPStreamLandmark

```
1: Input: $DS, preMinRareSup, minRareSup, landmark, minFreqSup$
2: Output: $result$ (set of rare-item itemsets)
3:
4: while exist($B \in DS$) do
5:     $t \leftarrow$ new incoming transaction from $DS$
6:     if $t.time < landmark$ then
7:         continue
8:     end if
9:     updateItemFreqList($t$)
10:    $batchTransactions$.add($t$)
11:
12:    Mine at the end of batch
13:    if $endOfBatch$ then
14:        $batchResult \leftarrow \emptyset$
15:        $I \leftarrow$ all unique items in batch
16:        $rareItems \leftarrow \{i \in I | sup_B(i) \geq preMinRareSup \wedge sup_B(i) < minFreqSup\}$
17:        $rareItemTrans \leftarrow \{t \in batchTransactions | \exists r \cdot r \in rareItems \wedge r \in t\}$
18:        $tree \leftarrow$ constructTree($rareItemTrans$)
19:        for item $a$ in $tree$ do
20:            if item $a \in rareItems$ then
21:                construct $a$'s conditional pattern-base and then $a$'s conditional FP-Tree $Tree_a$
22:                $batchResult \leftarrow batchResult \cup$ FP-Growth($Tree_a, a$)
23:            end if
24:        end for
25:        $potentialResult \leftarrow potentialResult \cup batchResult$
26:        $batchTransactions \leftarrow \emptyset$
27:    end if
28:
29:    Get truly rare upon user request
30:    if $getTrulyRare$ then
31:        $result \leftarrow$ getTrulyRare($potentialResult, minRareSup$)
32:    end if
33: end while
34: return $result$
```



(a) Initial Pattern Tree A

(b) Initial Pattern Tree B

Fig. 4. Pattern Trees

Algorithm 4 RPStreamTimeFading

```
Input: $DS, preMinRareSup, minRareSup, timeFactor, minFreqSup$
Output: $result$ (set of rare-item itemsets)

while exist($B \in DS$) do
    $t \leftarrow$ new incoming transaction from $DS$
    updateItemFreqList($t$)
    $batchTransactions$.add($t$)

    Mine at the end of batch
    if $endOfBatch$ then
        $batchResult \leftarrow \emptyset$
        $I \leftarrow$ all unique items in batch
        $rareItems \leftarrow \{i \in I | sup_B(i) \geq preMinRareSup \wedge sup_B(i) < minFreqSup\}$
        $rareItemTrans \leftarrow \{t \in batchTransactions | \exists r \cdot r \in rareItems \wedge r \in t\}$
        $tree \leftarrow$ constructTree($rareItemTrans$)
        for item $a$ in $tree$ do
            if item $a \in rareItems$ then
                construct $a$'s conditional pattern-base and then $a$'s conditional FP-Tree $Tree_a$
                $batchResult \leftarrow batchResult \cup$ FP-Growth($Tree_a, a$)
            end if
        end for
        $potentialResult \leftarrow (potentialResult * timeFactor) \cup batchResult$
        $batchTransactions \leftarrow \emptyset$
    end if

    Get truly rare upon user request
    if $getTrulyRare$ then
        $result \leftarrow$ getTrulyRare($potentialResult, minRareSup$)
    end if
end while
return $result$
```

older data in the window. If the window is not full, then that data remains in the window. The other case, when the window is full, then the oldest data is completely removed from the window. Algorithm 5 depicts a pseudocode representation of RPStreamSlidingWindow.

B. SRP-Tree Extensions

The SRP-Tree algorithm is designed to mine rare-item itemsets from data streams. In this section, we propose 3 extensions to the SRP-Tree algorithm. Unlike the original SRP-Tree proposal in [2], we implement a batch mining approach wherein mining occurs upon each batch read and the result is stored in an intermediate data structure (a table in our case). Additionally, we introduce the idea of *preMinRareSup* to the SRP-Tree algorithm. Then, instead of mining the SRP-Tree based on the *minRareSup* value, we mine based on a *preMinRareSup* value to allow almost rare items to remain in consideration and not be pruned too soon. Upon user request, mining for truly rare items is performed with the usual *minRareSup* value. Our final extension is the application of new windowing techniques to the SRP-Tree algorithm, namely, the landmark and time-fading windowing techniques.

1) SRPLandmark: The SRPLandmark algorithm employs the original methods proposed by [2], with the exception of intermediate mining of batches using *preMinRareSup* and using a landmark windowing technique opposed to a sliding-window technique. The *landmarkTime* is predefined by the user, and determines which transactions are added to the SRP-Tree and which are ignored. Transactions prior to the *landmarkTime* are ignored, while transactions at or after the *landmarkTime* are processed. Algorithm 6 depicts a pseudocode representation of Intermediate SRPStreamLandmark.

Provided is an example of the SRPLandmark algorithm using the dataset from table 2. Once the SRP-Tree has been applied to all the transactions in batch 1, the support ordered list is $<< a : 4 >, < b : 4 >, < c : 4 >, < e : 3 >, < d : 1 >>$. Using *preMinRareSup* = 0, *minRareSup* = 0, and *minFreqSup* = 4, only item d is rare and thus added to the set of rare items. Figure 4(b) depicts the initial SRP-Tree. Note that the SRP-Tree is built using all the transactions in the batch, whereas in the RP-Tree approaches, only transactions with rare items are used to build the tree. To find rare-item itemsets, the initial SRP-Tree is used to build conditional pattern bases and conditional SRP-Trees for each rare item, in this case d, and any additional items in the Connection Table that are connected with a rare item that has a frequency greater than *preMinRareSup*, which there does not exist in this example. Each of the conditional SRP-Trees and the conditional items are then used as parameters for the FP-Growth algorithm. The

Algorithm 5 RPStreamSlidingWindow

```
1: Input: $DS$, $preMinRareSup$, $minRareSup$, $minFreqSup$, $windowSize$
2: Output: $result$ (set of rare-item itemsets)
3:
4: while exist($B \in DS$) do
5:     $t \leftarrow$ new incoming transaction from $DS$
6:     updateItemFreqList($t$)
7:     $batchTransactions$.add($t$)
8:
9:     Mine at the end of batch
10:    if $endOfBatch$ then
11:        $batchResult \leftarrow \emptyset$
12:        $I \leftarrow$ all unique items in batch
13:        $rareItems \leftarrow \{i \in I | sup_B(i) \geq preMinRareSup \wedge sup_B(i) < minFreqSup\}$
14:        $rareItemTrans \leftarrow \{t \in batchTransactions | \exists r \cdot r \in rareItems \wedge r \in t\}$
15:        $tree \leftarrow$ constructTree($rareItemTrans$)
16:        for item $a$ in $tree$ do
17:            if item $a \in rareItems$ then
18:                construct $a$'s conditional pattern-base and then $a$'s conditional FP-Tree $Tree_a$
19:                $batchResult \leftarrow batchResult \cup$ FP-Growth($Tree_a, a$)
20:            end if
21:        end for
22:        $potentialResult \leftarrow$ updateWindow($potentialResult, batchResult$)
23:        $batchTransactions \leftarrow \emptyset$
24:    end if
25:
26:    Get truly rare upon user request
27:    if $getTrulyRare$ then
28:        $result \leftarrow$ getTrulyRare($potentialResult$, $minRareSup$)
29:    end if
30: end while
31: return $result$
```

Algorithm 6 Intermediate SRPLandmark

```
1: Input: $DS, preMinRareSup, minRareSup, minFreqSup, landmark$
2: Output: $result$ (set of rare-item itemsets)
3:
4: while exist($B \in DS$) do
5:     $t \leftarrow$ new incoming transaction from $B$
6:     if $t.time < landmark$ then
7:         continue
8:     end if
9:     updateItemFreqList($t$)
10:    updateConnectionTable($t$)
11:    $tree \leftarrow$ updateTree($t$)
12:
13:    Mine at the end of batch
14:    if $endOfBatch$ then
15:        $batchResult = \emptyset$
16:        $I \leftarrow$ all unique items in batch
17:        $R \leftarrow \{i \in I \mid sup_B(i) \geq preMinRareSup \wedge sup_B(i) < minFreqSup\}$
18:        $C \leftarrow \{k \in R, j \in connectionTable(k) \mid sup_B(k) \geq preMinRareSup\}$
19:        for item $a$ in $tree$ do
20:            if item $a \in R$ or $a \in C$ then
21:                construct $a$'s conditional pattern-base and then $a$'s conditional FP-Tree $Tree_a$
22:                $batchResult \leftarrow batchResult \cup$ FP-Growth($Tree_a, a$)
23:            end if
24:        end for
25:        $potentialResult \leftarrow potentialResult \cup batchResult$
26:    end if
27:
28:    Get truly rare upon user request
29:    if $getTrulyRare$ then
30:        $result \leftarrow$ getTrulyRare($potentialResult$, $minRareSup$)
31:    end if
32: end while
33: return $result$
```

resultant itemsets are added to the intermediate table as shown in table 3. This process continues on further batches until the user mines for truly rare itemsets. The set of truly rare itemsets is all sets in the intermediate table with support $\geq$ *minRareSup*.

2) SRPTimeFading: The SRPTimeFading algorithm employs the original methods proposed by [2], with the exception of intermediate mining of batches using *preMinRareSup* and using a time-fading windowing technique opposed to a sliding-window technique. The time-fading windows makes use of a user-defined *timeFactor*. Upon mining a batch, the resultant itemsets are saved in the intermediate table and all other (previously) mined itemsets in the table are multiplied by the *timeFactor* variable. Algorithm 7 depicts a pseudocode representation of SRPStreamTimeFading.

3) SRPSlidingWindow: The SRPSlidingWindow algorithm employs the original methods proposed by [2], with the exception of intermediate mining of batches using *preMinRareSup* and a minor modification to the sliding-window functionality. Upon mining a batch, the resultant itemsets are saved in the intermediate table and the window in the table is shifted. If the window is full, then the oldest transaction is removed. Since we employ an intermediate mining approach the window holds batch mining results. Algorithm 8 depicts a pseudocode representation of SRPStreamTimeFading.

Algorithm 7 Intermediate SRPTimeFading

```
1: Input: $DS, preMinRareSup, minRareSup, minFreqSup, timeFactor$
2: Output: $result$ (set of rare-item itemsets)
3:
4: while exist($B \in DS$) do
5:     $t \leftarrow$ new incoming transaction from $B$
6:     updateItemFreqList($t$)
7:     updateConnectionTable($t$)
8:     $tree \leftarrow$ updateTree($t$)
9:
10:    Mine at the end of batch
11:    if $endOfBatch$ then
12:        $batchResult = \emptyset$
13:        $I \leftarrow$ all unique items in batch
14:        $R \leftarrow \{i \in I \mid sup_B(i) \geq preMinRareSup \wedge sup_B(i) < minFreqSup\}$
15:        $C \leftarrow \{k \in R, j \in connectionTable(k) \mid sup_B(k) \geq preMinRareSup\}$
16:        for item $a$ in $tree$ do
17:            if item $a \in R$ or $a \in C$ then
18:                construct $a$'s conditional pattern-base and and then $a$'s conditional FP-Tree $Tree_a$
19:                $batchResult \leftarrow batchResult \cup$ FP-Growth($Tree_a, a$)
20:            end if
21:        end for
22:        $potentialResult \leftarrow (potentialResult * timeFactor) \cup batchResult$
23:    end if
24:
25:    Get truly rare upon user request
26:    if $getTrulyRare$ then
27:        $result \leftarrow$ getTrulyRare($potentialResult, minRareSup$)
28:    end if
29: end while
30: return $result$
```

Algorithm 8 Intermediate SRPSlidingWindow

```
1: Input: $DS, preMinRareSup, minRareSup, minFreqSup, windowSize$
2: Output: $result$ (set of rare-item itemsets)
3:
4: while exist($B \in DS$) do
5:     $t \leftarrow$ new incoming transaction from $B$
6:     updateItemFreqList($t$)
7:     updateConnectionTable($t$)
8:     $tree \leftarrow$ updateTree($t$)
9:
10:    Mine at the end of batch
11:    if $endOfBatch$ then
12:        $batchResult = \emptyset$
13:        $I \leftarrow$ all unique items in batch
14:        $R \leftarrow \{i \in I \mid sup_B(i) \geq preMinRareSup \wedge sup_B(i) < minFreqSup\}$
15:        $C \leftarrow \{k \in R, j \in connectionTable(k) \mid sup_B(k) \geq preMinRareSup\}$
16:        for item $a$ in $tree$ do
17:            if item $a \in R$ or $a \in C$ then
18:                construct $a$'s conditional pattern-base and then $a$'s conditional FP-Tree $Tree_a$
19:                $batchResult \leftarrow batchResult \cup$ FP-Growth($Tree_a, a$)
20:            end if
21:        end for
22:        $potentialResult \leftarrow$updateWindow($potentialResult, batchResult$)
23:    end if
24:
25:    Get truly rare upon user request
26:    if $getTrulyRare$ then
27:        $result \leftarrow$ getTrulyRare($potentialResult, minRareSup$)
28:    end if
29: end while
30: return $result$
```

V. Analytical and Empirical Results

REAL WORLD DATASET

In this section we present the processing time for our 6 algorithms when running on real world datasets. Testing is performed on the Mushroom, Retail, T10I4D100K and T40I10D100K datasets from the Frequent Itemset Mining Dataset Repository. Note that the datasets have been artificially segregated into batches to simulate a stream. All algorithms were implemented in Java and executed on a machine equipped with an Intel Core m3 CPU @ 1.2GHz with 8 GB of RAM running macOS Ventura 13.1.

1) The Mushroom dataset is a dense dataset with a relatively larger number of possible rules and patterns. It contains a total of 8124 instances and 22 attributes that includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota family. Batch divisions have been created every 2000 transactions.
2) The Retail dataset, compared to Mushroom, is a much sparser dataset that contains anonymized retail market basket data from a Belgian retail store over approximately 5 months. The total number of instances is 88163. Batch divisions have been created every 2000 transactions.
3) The T10I4D100K and T40I10D100K originates from the same source, the IBM Almaden Quest market basket data generator. They both contain a total of 100000 instances.

The objective of this comparison is to look at the efficiency of each algorithm under the same conditions. For each of the landmark algorithms, we use a landmark value of 1. For

TABLE IV
LANDMARK ALGORITHMS USING LANDMARK VALUE OF 1

Dataset	**# rare items**	**PreMinRareSup**	**MinRareSup**	**MinFreqSup**	**RP**		**SRP**	
					Time (s)	**Memory (mb)**	**Time (s)**	**Memory (mb)**
mushrooms	618	0.19	0.2	0.5	7.3	789.2	12.5	802.6
retail	1684	0.00019	0.0002	0.0005	2.2	75.2	–	–
T1014D100K	650	0.00019	0.0002	0.0005	0.4	99.4	91.2	145.2
T40110D100K	9800	0.00019	0.0002	0.0005	0.8	86.5	–	–

TABLE V
TIME-FADING ALGORITHMS USING TIME-FACTOR OF 0.8

Dataset	**# rare items**	**PreMinRareSup**	**MinRareSup**	**MinFreqSup**	**RP**		**SRP**	
					Time (s)	**Memory (mb)**	**Time (s)**	**Memory (mb)**
mushrooms	78	0.19	0.2	0.5	8.5	875.5	11.8	882
retail	831	0.00019	0.0002	0.0005	2.2	67.4	–	–
T1014D100K	650	0.00019	0.0002	0.0005	0.4	99.4	91.05	145.2
T40110D100K	9800	0.00019	0.0002	0.0005	0.7	86.4	–	–

TABLE VI
SLIDING-WINDOW ALGORITHMS USING WINDOW SIZE OF 3

Dataset	**# rare items**	**PreMinRareSup**	**MinRareSup**	**MinFreqSup**	**RP**		**SRP**	
					Time (s)	**Memory (mb)**	**Time (s)**	**Memory (mb)**
mushrooms	602	0.19	0.2	0.5	8.2	1212.4	10.5	1161
retail	1615	0.00019	0.0002	0.0005	1.9	82.2	–	–
T1014D100K	650	0.00019	0.0002	0.0005	0.4	99.4	98.9	145.1
T40110D100K	9800	0.00019	0.0002	0.0005	0.7	86.5	–	–

each of the time-fading algorithms, we use a time-factor of 0.8. For each of the sliding window algorithms, we use a window size of 20. Notice the different types of windowing techniques result in a different number of rare items mined. This is because the different techniques maintain older data in different ways resulting in a different mining result.

When considering the mushroom dataset, all three RP-extension algorithms outperformed all SPR-extension algorithms. Within the RP-extension algorithms, the best time performance resulted from RPStreamLandmark. We hypothesize the landmark algorithm performed best because there is the least overhead to merge batches (it is simple set union).

When considering the retail dataset, again, all three RP-extension algorithms outperformed all SRP-extension algorithms. Within RP-extension algorithms, the best time performance resulted from RPStreamSlidingWindow. We hypothesize this fast execution time results because the sliding window omits older data, meaning the sliding window algorithm does not need to manage as much data as the landmark or time-fading techniques.

Of course there is more to consider than just execution time. With the landmark windowing technique, there is no loss of older data. Therefore, this approach is likely optimal when the application requires mining absolutely all rare-item itemsets. Should the application not require all rare itemsets be mined, then the sliding window model may be more suitable since it seems to outperform landmark and time-fading techniques as the dataset grows.

From the analysis, it became apparent the SRP extension algorithms were not good at handling large datasets. As such, we were not able to obtain corresponding values for the SRP algorithms with the retail dataset since the run times were not practical (at least > 2 min). We hypothesize this could be due to the fact that the design of the RP-Tree algorithms allow for the pruning of transactions with no rare itemsets, resulting in a smaller tree size which ultimately requires less work to mine.

A. Case Study: Modifying preMinRareSup

In this section, we investigate the effects of varying the preMinRareSup variable. To perform this case study, we will use the RPStreamLandmark algorithms on the mushroom dataset with minRareSup = 0.2 and minFreqSup = 0.5.

The data illustrates the effects of the preMinRareSup variable. As expected, with a lower preMinRareSup, more rare items are mined because items that are almost rare remain in consideration. Thus, if the user wants to obtain more of the rare items from the stream, there should be a significantly lower preMinRareSup than the minRareSup. As well, the data

TABLE VII
VARYING PREMINRARESUP FOR THE RPSTREAMLANDMARK ALGORITHM ON MUSHROOM DATASET

PreMinRareSup	# Rare Items	Time(s)
0.19	618	7.3
0.17	634	7.7
0.15	634	10.2
0.13	736	16.1

shows the trade-off between number of rare items mined and the execution time. This is expected since mining for more items can occur with a lower preMinRareSup.

VI. CONCLUSION

We present 6 new algorithms for mining rare-item itemsets from a data stream. Our algorithms are extensions to the RP-Tree algorithm and SRP-Tree algorithm. It is our understanding there have not been any publications to use a preMinRareSup variable to capture rare itemsets in a steaming environment. It is also our understanding there have not been any publications that have implemented an intermediate mining approach when mining rare items from data streams. As well, we had not encountered the landmark or time-fading windowing techniques for mining rare items.

Our empirical analysis illustrated the outperformance form the RP-extension algorithms. We were also able to note performance variance between the RP-extension algorithms. As well, the case study illustrated the effect of using a preMinRareSup variable to obtain more rare items from the stream.

The RP-Tree extension algorithms require the current batch to be stored in memory. Although unlikely, this is a limitation if the size of the current batch exceeds the system memory. If this case where to arise, a potential solution is to add memory to the system running the algorithms. Another potential limitation that may arise for all 6 of the discussed algorithms is the potentially rare resultant itemsets exceeds the system memory. This can arise if the number of mined rare patterns is very large. A potential solution is mine for truly rare items more often and store those resultant rare itemsets to a file to reduce the memory load.

In the future, we seek to investigate the RP-Tree extension algorithms using the RP-Tree with information gain as presented by [4]. Additionally, we are interested in analyzing the performance of SRP-extension algorithms using the second SRP variant of tree restructuring (as opposed to using a connection table). We are also interested in studying the performance of these intermediate algorithms against delayed streaming approaches such as the original SRP-Tree algorithm. As well, there are many applications that work with uncertain databases which invites us to apply these extensions to uncertain data.

REFERENCES

[1] R. Lakshmi, C. Hemalatha and V. Vaidehi, "Mining infrequent patterns in data stream," 2014 International Conference on Recent Trends in Information Technology, pp. 1-5, 2014.

[2] D. T. J. Huang, Y. S. Koh and G. Dobbie, "Rare Pattern Mining from Data Streams Using SRP-Tree and Its Variants," in Transactions on Large-Scale Data- and Knowledge-Centered Systems XXI, Berlin, Springer, 2015, pp. 140-160.

[3] J. Han, J. Pei and Y. Yin, "Mining frequent patterns without candidate generation," Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 1-12, 2000.

[4] S. Tsang, Y. S. Koh and G. Dobbie, "RP-Tree: Rare Pattern Tree Mining," in Data Warehousing and Knowledge Discovery, Berlin, Springer, 2011, pp. 277-288.

[5] S. Darrab, D. Broneske and G. Saake, "Modern Applications and Challenges for Rare Itemset Mining," International Journal of Machine Learning and Computing, vol. 11, no. 3, pp. 208-218, 2021.

[6] M. M. Gaber, A. Zaslavsky and S. Krishnaswamy, "Mining Data Streams: A Review," Centre for Distributed Systems and Software Engineering, Monash University, Melbourne.

[7] L. Troiano and G. Scibelli, "A time-efficient breadth-first level-wise lattice-traversal algorithm to discover rare itemsets," Data mining and knowledge discovery, vol. 28, no. 3, pp. 773-807, 2014.

[8] M. Adda, L. Wu and Y. Feng, "Rare Itemset Mining," in Sixth International Conference on Machine Learning and Applications (ICMLA 2007), Cincinnati, 2007.

[9] . Szathmary, P. Valtchev and A. Napoli, "Finding Minimal Rare Itemsets and Rare Association Rules," in Knowledge Science, Engineering and Management, Berlin, Springer Berlin, 2010, pp. 16-27.

[10] Y. S. Koh and N. Rountree, "Finding Sporadic Rules Using Apriori-Inverse," in Advances in Knowledge Discovery and Data Mining, Berlin, Springer, 2005, pp. 97-106.

[11] C. C. Aggarwal, "An Introduction to Frequent Pattern Mining," in Frequent Pattern Mining, Cham, Springer, 2014, pp. 1-17.

[12] F. Jiang, Frequent Pattern Mining of Uncertain Data Streams, Springer-Verlag, 2011.

[13] T.-T. Nguyen, Q. Nguyen and N. T. Hung, "Mining Incrementally Closed Itemsets over Data Stream with the Technique of Batch-Update," in Future Data and Security Engineering, Switzerland, Springer International Pubilshing, 2019, pp. 68-84.

[14] C. K.-S. Leung and B. Hao, "Mining of Frequent Itemsets from Streams of Uncertain Data," in 2009 IEEE 25th International Conference on Data Engineering, 2009.