AnyFI: An Anytime Frequent Itemset Mining Algorithm for Data Streams

Poonam Goyal, Jagat Sesh Challa, Shivin Shrivastava, Navneet Goyal
Advanced Data Analytics and Parallel Technologies Lab, Dept. of Computer Science & Information Systems, Pilani Campus
Birla Institute of Technology & Science, Pilani, Rajasthan, India
{poonam, jagatsesh, f2013073, goel}@pilani.bits-pilani.ac.in

Abstract— **Mining frequent itemsets from transactional data streams has been vastly studied in literature. The existing algorithms mine frequent itemsets within the stream's constrained environment of limited time and memory. However, none of them are capable of handling varying inter-arrival rates of streams. Moreover, these algorithms are not capable of giving mining results instantaneously, even with compromised accuracy if required, and improve the accuracy with increase in time allowance. These two properties characterize an anytime algorithm. In this paper, we propose AnyFI, which is the first anytime frequent itemset mining algorithm for data streams. We also propose a novel data structure, BFI-forest, which is capable of handling transactions with varying inter-arrival rate. AnyFI maintains itemsets in BFI-forest in such a way that it can give a mining result almost immediately when time allowance to mine is very less and can refine the results for better accuracy with increase in time allowance. Our experimental results show that AnyFI can handle high stream speeds upto 60,000 transactions per second (tps) with recall close to 100%.**

Keywords- Frequent Itemset Mining; Data Streams; Anytime Mining

I. INTRODUCTION

Mining for frequent itemsets (FIs) in transactional data streams is commonly used in various applications such as stock market analysis, web log analysis, retail chain analysis, etc. Researchers have proposed various algorithms to mine FIs from streams under constrained environment of limited time memory. Few of them include - Sticky Sampling and Lossy Counting [1], FP-Stream [2], CPS-Tree [3], DSM-FI [4], SWP-Tree [5], VSW [6], etc. These algorithms insert incoming transactions into a hierarchical structure in online manner, and whenever a request for mining comes from the user, they extract FIs in an offline manner, and present them as the mining result.

The existing algorithms, miss out on two important characteristics of data streams. First, very often streams do not have constant speed (or inter-arrival rate of transactions), which varies depending on application domains. For example, in retail chain analysis, the rate of arrival of transactions is higher during rush hours and lower other times. The existing algorithms are budget algorithms, i.e. they are designed for a fixed maximum stream speed and cannot process higher speeds. An ideal algorithm should be able to process any stream speed. The second is that they lack the ability to produce immediate mining results with compromised accuracy, if required. Applications such as stock market analysis sometimes require immediate results. For example, a short term stock investor, would require an immediate result. Whereas, a long term investor would wait for some additional time until more accurate results are available. So, when a request for mining result comes from the user, an ideal algorithm should give an immediate approximate result, and improve its quality with increase in time allowance. The step for mining of FIs in the existing algorithms is quite costly in terms of execution time and hence cannot give an immediate approximate result, even with compromised accuracy.

The above two properties – 1. handling varying inter-arrival rate of transactions; and 2. giving the best possible result according to the available time allowance; are the characteristics of an *anytime mining algorithm* for data streams. A few anytime algorithms are proposed for clustering [7], [8] and classification [9], [10]. There is one anytime FI mining algorithm [11] proposed for multi-user applications which possesses the second characteristic. However, the method is static and is not meant for data streams.

In this paper, we present an **Any**time **Fr**equent **I**temset mining algorithm for data streams, referred as *AnyFI*, characterized by the above two properties. To the best of our knowledge, this is first such attempt. We also propose a novel data structure known as **B**uffered **F**requent **I**temset **Forest** (*BFI-forest*), to insert the incoming transactions. BFI-forest stores *buffers* at its tree nodes, which aid AnyFI to handle variable stream speeds. AnyFI tries to insert all suffix projections (defined in section 3) of incoming transactions into its forest, depending upon the available time allowance. As a result of which, mining BFI-forest, becomes a simple traversal of its trees to accumulate FIs, without generating any candidate itemsets like in apriori like methods [12]–[14], or conditional trees like in FP-growth like methods [3], [15], thus making mining of FIs very efficient. AnyFI can also give out a mining result almost immediately with compromised accuracy, and can improve the quality of results with increase in time allowance. The experimental analysis show that AnyFI is able to handle stream speeds up to 60,000 transactions per second (tps), with recall ≈ 100%.

The remainder of the paper is organized as follows: Section II presents the related work, Section III presents the proposed algorithm and Section IV presents experimental results followed by conclusions and future work in Section V.

II. RELATED WORK

Researchers have proposed various FI mining algorithms for data streams using stream models such as landmark window (LW) and sliding window (SW). Sticky-Sampling and Lossy Counting [1] are two algorithms that are based on Apriori and following the LW model. They produce less accurate results with an error bound. FP-Stream [2] uses

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

tilted-time window (variation of LW), which takes data in batches of transactions, builds FP-tree for each batch, mines for FIs from it, and inserts them into a pattern tree, which has tilted-time windows stored at its nodes. Since, this algorithm mines every batch separately, it cannot handle variable stream speeds. DSM-FI [4] is another algorithm that uses LW. It keeps a forest of prefix trees which are similar to FP-trees and inserts each incoming transactions completely into the forest and thus cannot be interrupted anytime.

CPS-Tree [3] is FP-tree based algorithm that uses SW. It takes in a pane of transactions at a time and inserts them into the CPS-tree. After insertion of certain fixed number of panes, CPS-tree undergoes re-structuring and pruning. It uses FP-growth for mining FIs, whenever the request comes. FP-growth is not capable of delivering an immediate result with compromised accuracy. This is because, it enumerates many conditional trees which takes large time for larger batch sizes. MFI-TransSW [16] is another SW based approach that represents the items in the form of a bit vector and uses apriori for mining, which is very slow for lower support thresholds and is also not capable of delivering an anytime result. SWP-Tree [5] is another SW based algorithm that uses a prefix tree similar to that of FP-tree. SWP-tree additionally uses decay on its support count to give higher weightage to recent transactions. It uses FP-growth for mining FIs, which is again not capable of giving anytime result. VSW [6] is another algorithm which adjusts the size of SW on demand. It uses ECLAT [14] which is a variant of Apriori and is slow in its computations, and thus not capable of handling high speed streams.

A few algorithms were proposed in literature for anytime clustering [7], [8] and classification [9], [10] in data streams. There is one anytime FI mining algorithm for multi-user applications over large static databases [11]. It uses sampling and addresses the second aspect of an anytime algorithm, i.e., it gives an immediate approximate result and improves it with increase in time allowance. However, it is for static data and not fit for streams.

Based on above discussion, it is clear that none of the existing algorithms fit the bill for an anytime FI mining algorithm for data streams.

III. The Proposed AnyFI Algorithm

A. Formal Definitions

Following the definitions & notations given in Table 1, we define the problem statement - given *DS*, *sDS* and σ, the problem of FI mining from a data stream is to find those itemsets in *sDS* which have their frequency counts greater than the threshold - $\sigma|sDS|$. The set *sDS* is modelled as a window of different forms – *landmark window* [4], *sliding window* [3] or *damped window* [5] for different algorithms. In order to save space and execution time of algorithms, we use an additional support threshold, ϵ, to prune infrequent transactions [1], [4]. An itemset S is said to be ϵ-frequent, if $\epsilon|sDS| \leq Sup_{sDS}(S) < \sigma|sDS|$, where ϵ is a lower bound on frequency count such that if $Sup_{sDS}(S) < \epsilon|sDS|$, we say that S is *infrequent*.

TABLE I. DEFINITIONS & NOTATIONS

Notation	Definition
$I = \{i_1, i_2, i_3 \dots i_n\}$	I is set of literals denoting unique items of a domain. Literals are typically integers. $\|I\| = n$.
$S = \{i_x .. i_y\} \subseteq I\ ;\ x \leq y\ \&\ x, y \in [1 \dots n]$	S is an itemset of size k if it contains k items.
$DS = [t_1, t_2 \dots t_m)\ ;\ t_j$ is j^{th} arrived trans.	*DS* is a data stream which is continuous unbounded sequence of *transactions*.
$t_j = (tid, arrTime, S')$	t_j is a transaction which is a tuple - transaction ID, arrival time and an itemset S' ($S' \subset I$). If for an itemset S, $S \subseteq S'$, we say that t_j contains S or S occurs in t_j.
$\sigma;\ 0 < \sigma < 1$	Minimum support threshold or *min_sup*.
sDS	A finite contiguous subset of *DS*.
$Supp_{sDS}(S)$	Frequency count (or support count) of an itemset S in *sDS*, i.e. # of transactions in *sDS* in which S has occurred.
Frequent Itemset	An itemset S is said to be frequent in *sDS* if $Supp_{sDS}(S) \geq \sigma|sDS|$ (also referred as σ - frequent).
Suffix Projections	For transaction t_j = <*abcd*>, its suffix projections are <*bcd*>, <*cd*> and <*d*>. # of suffix projections= $\|t_j\|$ -1.
Head	Head is the first item of a transaction.

B. Stream Model

We adopt the *damped window* to model our streams. Damped window helps in lowering the effect of older transactions with time and thus lets us control the contribution of a transaction with respect to its arrival time. Recent transactions get greater weightage and hence recent trends can be captured more effectively. To achieve this, we use a decay factor f, $0 < f < 1$, to decay the frequency counts of itemsets with time. We denote frequency count of an itemset S at time T_1 as $freq(S, T_1)$. At time T_2, the decayed frequency count of S will be –

$$freq(S, T_2) = freq(S, T_1) . f^{T_2 - T_1} \quad (1)$$

In AnyFI, whenever we increment the frequency count of an itemset, we first decay the existing count using (1) and then increment it. Also, whenever the frequency count of an itemset S with respect to any support threshold (say ϵ or σ) is being computed, $|sDS|$ is computed as the difference between current *tid* ($curr\_tid$) and *tid* of the first transaction ($ftid$) where S has occurred ($|sDS| = curr\_tid - S.ftid$). $ftid$ is stored in tree nodes of BFI-forest (explained next).

C. BFI-forest

BFI-forest is a summary data structure that stores the enumerated *suffix projections* of arriving transactions. The itemsets are so arranged in the forest that mining FIs becomes very efficient. Fig. 1 illustrates the structure of BFI-forest. It consists a set of *BFI-trees* (Buffered Frequent Itemset trees), whose count is <= n. Fig. 1 represents a forest for I = {*a,b,c,d,e*}. Each node of a BFI-tree represents an itemset. For example, node 6 in Tree1 represents the itemset {*ac*}. There are two kinds of nodes in BFI-tree - internal and external. Internal nodes consists of fields as explained in Table II. External nodes (*leaf nodes*) differ from internal nodes only in one field, i.e., instead of $childArr$, they have a field $fpRoot$ which is a pointer to an FP-tree [15]. All the external nodes are situated at depth *Max_Height*, which is a user defined parameter. Please note, that those nodes which are not at depth=*Max_Height* and don't have subtrees rooted at them,

are not considered external. For e.g., node 11 of Tree1 is not an external node.

TABLE II. FIELDS OF AN INTERNAL NODE OF BFI-TREE

Field	Definition
item	Integer identifier representing a literal *i* of set *I*. It represents the item indexed at the current node.
efreq	Frequency count of the itemset represented by this node. For node 6 in Tree 1, efreq would store the frequency count of itemset {*ac*}.
ftid	Transaction ID at which the current node was created.
ltime	Timestamp at which the current node was last accessed.
buff	This field represents the buffer at a given node. Buffers store incompletely inserted transactions, They are a key requirement for our anytime algorithm. They are implemented as hash tables with linear chaining (see Fig 1). Each bucket in the hash table is a linked list of buffer-nodes as shown in Fig. 1. Each buffer-node stores the following entries: *partial_trans*, *ftid*, *ltime* and *efreq*. The field *partial_trans* is a suffix of the partially inserted transaction. The items in this suffix are not yet inserted into the tree. The buffer has a limit on the number of buffer nodes it can hash - *buffCapacity*. The size of the hash table array (*hash_size*) is typically chosen as 10% of *n*. Simple mod function is used as the hash function, i.e. for a transaction t_j = <*abc*>, the bucket to which t_j is to be indexed is computed as: *a mod* |*hash_size*|, where *a* is the first item in t_j. Note that *a* (even *b* or *c*) is an integer ∈ *I*.
childArr	It is an array of maximum size *n*, that stores pointers to the sub-trees indexed at the current node.

D. AnyFI

AnyFI is the proposed anytime algorithm for mining frequent itemsets in a continuous data stream. The algorithm in summary consists of the following steps:

Step 1: Read incoming transactions one by one and order them lexicographically.

Step 2: Insert each transaction into the BFI-forest.

Step 3: Intermittently prune the infrequent itemsets from BFI-forest after a fixed interval of arrival of transactions.

Step 4: Mining of BFI-forest for frequent itemsets on demand depending upon the time allowance.

Insertion in BFI-forest. The insertion of an incoming transaction into the BFI-forest is an anytime algorithm, i.e. it is interruptible. The insertion process of a transaction goes on until its time allowance expires (a new transaction arrives), after which we proceed to insert the newly arrived transaction. The incoming transactions from the stream are inserted one by one into the BFI-forest. For every transaction, we first compute the suffix projections and insert them into the buffers of the corresponding trees. For example, in Fig.1, for the incoming transaction <*acde*>, the projections <*cde*>, <*de*> and <*e*> will be inserted into the buffers of the roots of Trees 1, 3 & 4 respectively, and *efreq* of the roots of Trees 1, 3, 4 & 5 will be incremented by 1, after getting decayed using equation 1 with T_1 & T_2 as *ltime* and *current time* respectively. For inserting <*cde*> into the buffer of root of Tree1, we first find the hash value to identify the bucket into which it has to be inserted, (bucket # = *c mod* \|*hash_size*\|). All the projections starting with *c* will be hashed to this bucket. If a buffer node indexing <*cde*> already exists in this bucket, we increment its frequency count by 1, after decaying it, as was done before. Otherwise, we create a new buffer node and append it at the end of the bucket. All other projections are inserted in the same way. This operation - taking suffix projections of a transaction and inserting them into a buffer, is an atomic operation (non-interruptible) in our algorithm.

After this step, we take each of the affected trees (trees that were accessed in the above step – Trees 1, 3, 4 & 5 in Fig. 1) and start refining them in depth first order (DFS) of tree nodes. Depending on available time allowance, we refine these trees one after the other, i.e., if a new transaction arrives before refining all the trees, we quit in between and proceed with processing of the newly arrived transaction. Consider Tree1 in Fig. 2a, where <*cde*> is inserted in its root's buffer in the above step. The refinement starts with root node of this tree (root becomes *curr_node*). We first remove the first projection (or partially inserted transaction) from a randomly chosen bucket of its buffer. This projection is the oldest and expected to have highest frequency count among buffer nodes in the bucket. Let's say that <*cde*> is the partial transaction that was removd. We first update its frequency count as explained before using eqn. 1. If *curr_node* is an external node, we would insert this projection in to the FP-tree beneath it and exit the insert function. Otherwise, we take suffix projections of <*cde*> and insert them into the buffers of the corresponding children of *curr_node* as shown in Fig 2b (nodes 6 & 9 get projections into their buffers). The insertion into these buffers is same as explained before. We also increment the frequency counts of nodes 6 & 9 with respect to the frequency counts of

Fig. 1. Structure of BFI-Forest and BFI-Trees

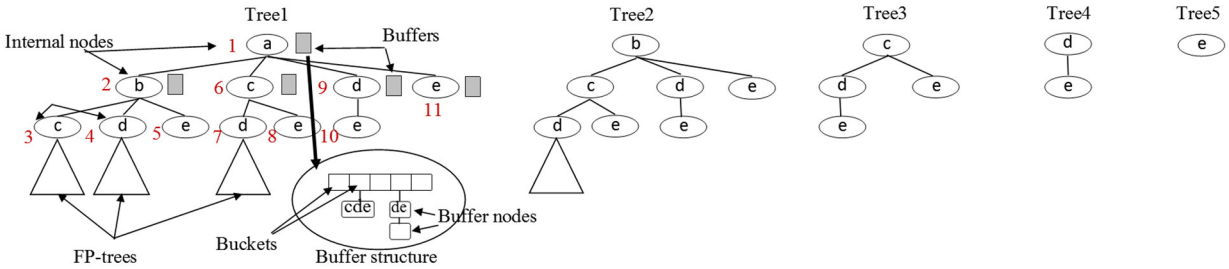
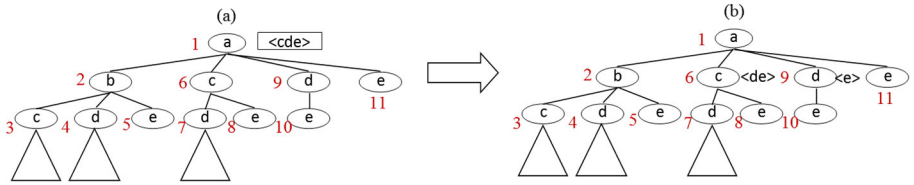


Fig. 2. Insertion in BFI-Tree



the partial transactions being inserted into their buffers, after decaying them. If any child into which a projection has to be inserted, does not exist, we create that child first, insert it into its buffer and assign it a frequency count. Also, while inserting into the buffer, we take care that the buffers do not overflow. If they exceed a pre-defined capacity- *buffCapacity*, the oldest (first) item is removed from the bucket in which we are trying to append the current projection. The oldest is chosen because it has the maximum probability of becoming infrequent as we are decaying the frequency counts with time. After this step, we check if a new transaction has arrived. If yes, we return back to the caller function and move on to process the next transaction; otherwise we continue processing the current transaction. Please note that we check this condition at the beginning of refinement of every node.

After insertion of projections into the sub-trees, buffer pruning is conducted. *Buffer pruning* prunes away infrequent projections lying in the buffers. So, the buffers of affected children of *curr_node* (children into which suffix projections were inserted in the previous step – nodes 6 & 9 in Fig. 2b) are pruned before we proceed with further refinement. Buffer pruning is not done every time we visit a given node in our traversal. This is because, there may not be many infrequent projections accumulated every time we visit a node's buffer. So, by delaying it, we let the node accumulate a few more infrequent projections and then remove them in one go. So, buffer pruning is conducted in intervals of some minimum time decided by a parameter γ and the height of the node. It can be observed that closer the node to the root, more filled will its buffer be. So, buffers at lesser depth must be pruned more often than the buffers at greater depth. So, the pruning interval (*PI*) for each node is computed using the following formula - $PI = \left\lceil \frac{batch\_size}{10} * \gamma * height(node) \right\rceil$; *batch_size* is the # of transactions after which we perform intermittent pruning (explained next). Whenever we are visiting a node, we prune its buffer only when - *it was last pruned at least PI transactions earlier*. To prune a given buffer, we visit every buffer-node (nodes from all the buckets) and check if *partial_trans* in it is ϵ-frequent or not (after decaying its frequency count). If it is not, then we check if the current *affected_child* (node for which buffer pruning is being conducted) has a child in its *childArr* which corresponds to the head of this *partial_trans*. If it does, then we don't delete this partial transaction, as we might lose a potential FI by removing it. Otherwise, it is safely deleted.

After pruning the buffer, we now select nodes which are to be refined further. We prefer to deliberately delay the refinement of certain nodes to save space and time. We let the nodes accumulate more itemsets in their buffers before they are refined or expanded for insertion into sub trees. This step is critical in making the insertion fast. By doing this, we save time by avoiding repeated insertions and removals of infrequent itemsets. We let many infrequent itemsets to be pruned away from buffers itself, rather than getting expanded into a large number of infrequent subtrees. This delayed refinement is achieved by a tuning parameter θ and this process is known is θ-*deferring*. So, after inserting the suffix projections at a node and pruning its buffer, every affected child is checked whether it is ϵ-frequent or not. If yes, then we check whether the subtree corresponding to the head of the projection to be inserted into it, is present in its *childArr* or not. If this is so, we keep this node in a *stack* for its refinement in subsequent iterations. For example, consider node 6 in Fig 2b. If it is ϵ-frequent and there exists a subtree with root *d* present in its *childArr* (node 7), we would want this node to be refined further and thus, push it into the stack. However, if the subtree doesn't exist, then we check if the node is θ-frequent or not. If it is so, only then we would want this node to be refined further and we push it into the stack. Else, we don't refine this node and let it accumulate more transactions in its buffer. This reduces creation of infrequent subtrees in the forest. After this we proceed for the next iteration, in which the nodes accumulated in the stack are refined.

Intermittent Pruning. After a fixed batch (*batch_size*) of transactions are received, we conduct intermittent pruning. This is for timely pruning of infrequent subtrees in the forest, which makes insertion fast & efficient. In this step, we visit each node of every BFI-Tree in DFS order, and delete it along with all its child subtrees, if it is *infrequent*. Removing the subtrees beneath infrequent nodes doesn't affect the accuracy of the algorithm because all itemsets in those subtrees can never be frequent. If however, the node is ϵ-frequent, we do not delete it. If the node we visit in the traversal is a leaf, we prune the FP-tree beneath it, as explained in [5].

In principle, deletion of sub-tree in BFI-tree does not affect the other branches of the tree (and other trees as well). Unlike other algorithms, e.g. DSM-FI [4], we do not need to update or remove items from other branches. This is because AnyFI enumerates all possible suffix projections of the incoming transactions, which get stored in independent branches. The frequency counts of itemsets represented by the other branches have no connection with the branch being deleted. Hence, our pruning step is accurate.

Mining BFI-forest. Mining for FIs in BFI-forest is very simple and straight forward. In the insertion step, we have inserted suffix projections of all incoming transactions in the trees, and within each tree we have enumerated suffix projections of those partial transactions and inserted them either into the tree as subtrees or into the buffers. This makes mining task easy, without the need of enumerating candidate itemsets like in apriori or generating conditional trees like in FP-growth. This makes AnyFI mine faster than the existing algorithms (see section 4). All that is needed is to do is to empty all the buffers in the trees (traversing in DFS or BFS order) and then traverse the tree again in DFS order accumulating the itemsets whose frequency counts are greater than σ. So, as we traverse down, at every node we check if it is σ-frequent. If yes, we store the itemset represented by that node in $FIset$, which is a list of FIs. Note that each node in the BFI-tree represents an itemset with items in the path from root to this node. Whenever an external node is encountered, we simply mine the FP-tree beneath and concatenate all the FIs that come from FP-tree with the itemset represented by the current node, and add all of them to $FISet$. Please note that when we mine a BFI-tree with root node having an item say *a*, all the FIs that start with *a* are mined from this tree only. This is because, while inserting a given transaction say <*abcd*>, we insert all its suffix projections into their

respective trees. So, <*bcd*> is inserted into a tree with root as *b* and <*cd*> is inserted into a tree with root *c*. The contribution of transaction <*abcd*> to the frequency count of itemset {*bcd*} is taken care by this process. So, to mine for {*bcd*}, it is not required to traverse the tree with root *a*.

We have seen that, mining for FIs has two steps – 1) emptying of buffers and 2) traversing to accumulate FIs. We had empirically observed that emptying buffers is the step that takes maximum time. Traversal for FIs takes time in order of milli seconds. So, we have made emptying of buffers step *anytime*, i.e. it proceeds as long as the time is available and once the quantum of time allotted by the user expires, the algorithm exits from this step and quickly traverses the forest for accumulating FIs. In this traversal, the residual partial transactions in the buffer, if any, are ignored. Consequently, we get a less accurate result when time allowance is less.

E. *Why AnyFI is efficient?*

Since, *AnyFI* tries to enumerate all possible suffix projections of transactions at all levels of BFI-Trees and insert them, the mining of FIs merely requires a traversal of its trees. The insertion of a transaction into BFI-forest is a DFS traversal of multiple trees in the forest, where each traversal cost would be *O(L)*, where *L* denotes # of nodes accessed in the tree = $\sum_{k=1}^{ht} \frac{m!}{(m-k)!}$. Here *ht* is the avg. height of a BFI-tree and *m* is the avg. length of transactions. In our algorithm, we have applied techniques like θ-deferring, buffer pruning, intermittent pruning and usage of FP-tree to save space and time (traversal cost). We used FP-trees at a certain height (*Max_height*) in the BFI-trees. This is because due to enumeration of so many suffix projections, the branching in the tree becomes very high at greater depths of the tree; and the itemsets indexed in those branches would mostly be infrequent. If the branching is allowed to grow beyond *Max_height*, it would lead to repeated creation and deletion of such infrequent subtrees. FP-trees are compressed trees and are efficiently mined when their sizes are small. That is why we store the suffix projections beyond *Max_height* into FP-trees, without enumerating their projections further. This saves memory and time. In addition, θ-deferring helps in avoiding creation of infrequent subtrees and thus saving their repeated creation and deletion. Similarly, buffer pruning and intermittent pruning also aid in reduction of infrequent subtrees. As a consequence, the worst case number of nodes visited would become: $L = \sum_{k=1}^{Max\_Height} \frac{m!}{(m-k)!}$. In fact, the actual number of nodes visited is much lesser than this figure because of pruning techniques used. The value of *Max_Height* is usually chosen <= 5, thus making *L* a polynomial of *m* of order <= *Max_Height*.

IV. Experimental Analysis

All experiments were performed on a Linux workstation with an i7 processor & 32 GM RAM. All programs were implemented in C. Both synthetic and real data sets were used for experimentation. The synthetic dataset *1MD1000T10I4* has 1M transactions, drawn from 1000 items, with avg. transaction length of 10 and average FI length of 4. It was generated using IBM Synthetic Data Generator [17]. The details of the real datasets are given in Table III. *Retail* dataset [18] contains market basket data from a Belgian retail store. *MSNBC* [19] is a click stream dataset describing page visits on msnbc.com. We evaluate the results produced by our algorithm using precision and recall.

TABLE III. REAL DATA SETS

Dataset	#Trans	#Items	Avg Trans Len
Retail	88162	16470	10.3
MSNBC	989818	17	1.71

In Any-FI, we mine BFI-Forest for FIs, with support σ. At this support, we get precision = 1, i.e. the mined result will not have any itemset which is not σ-frequent. However, we may miss out some of the FIs, leading to reduction in recall. In our experiments, we measure the quality of our results by studying the effect on recall with varying avg. speed of transactions. To simulate a stream with varying inter-arrival rate, we use *Poisson streams*, which is a stochastic model used to model random arrivals [20]. It takes in a parameter λ, which controls the speed of the stream. For $\lambda = 1/x$, the model generates an expected number of λ tps, with expected inter-arrival time of x seconds between two contiguous transactions. The values of the parameters chosen for experimentation are as follows: *Max_Height*=4, θ =0.05, γ =2, *batch_size*=10000, *f*=0.9, *buffCapacity* = 100 and *hash_size* is 10% of $|I|$.

In the first experiment, we analyze the effect of varying stream speed (λ) with different support thresholds (σ) on recall (Fig. 3a), memory (Fig. 3b) & mining time (Fig. 3c) on 1MD1000T10I4 dataset with ϵ=0.005. For measuring the mining time in this experiment, we have let all the buffers in the forest get emptied without any interruption. The results show that recall (close to 100%) is not affected by increase in stream speed. The memory consumption however, has reduced as more transactions get buffered at higher speeds and thus stopping the forest to grow large. Mining time has also reduced with increase in stream speed due to reduction in

Fig. 3. Effect of varying stream speed (λ) and σ on (a) Recall (b) Memory (c) Mining time for 1MD1000T10I4 dataset; (d) Recall for real datasets.

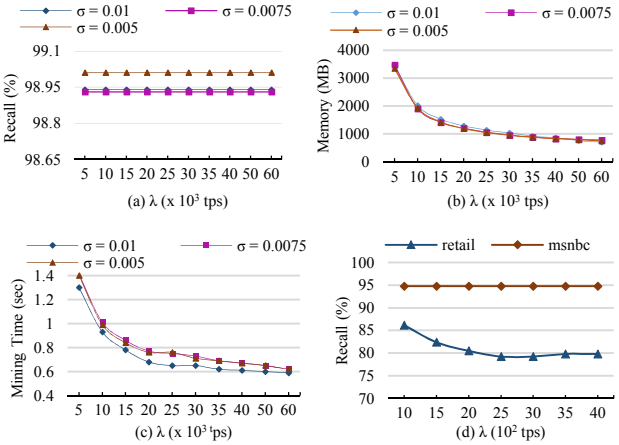
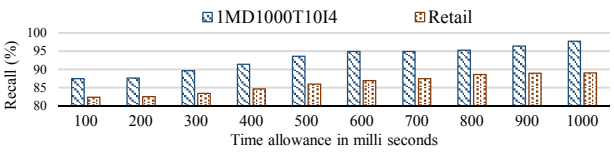


Fig. 4. Effect of increasing time allowance for mining on Recall



number of nodes visited in its traversal. Fig. 3d shows the variation in recall with increase in stream speed for two real datasets – *retail* and *msnbc*, for ϵ=0.002 and σ=0.01, with *max_height*=2 for retail and *max_height*=4 for msnbc. The retail dataset has a large dictionary size due to which choosing *max_height*>2 leads to high memory consumption. The reduction in recall observed for retail dataset with increase in stream speed is because of loss of transactions from buffers occurring at higher stream speeds.

In the second experiment, we study the quality of FI mining result with variation in time allowance to mine. Fig. 4 presents the effect on recall with increase in time allowance for mining, which is the anytime mining feature of our algorithm. We conduct this experiment on 1MD1000T10I4 and retail datasets with λ=20000, ϵ=0.0025 and σ=0.005. The result clearly shows that AnyFI is able to output mining result with compromised accuracy, within a few milli seconds. And then it is able to improve its recall with increase in time allowance to mine, for both the datasets.

In the next experiment, we compare the speed handling capacity and memory consumption of AnyFI with the existing algorithms for 1MD1000T10I4 dataset (ϵ=0.005 & σ=0.01). The batch size in all of them is chosen to be 10,000. The results given in Table IV, clearly show that these algorithms have limited budget, whereas Any-FI is able to perform for speeds upto 60,000 tps for the same dataset.

TABLE IV. SPEED AND MEMORY OF BUDGET ALGORITHMS

Algorithm	Category	Speed (tps)	Memory (MB)
FPStream	Tilted Time Window	27000	99
SWP-tree	Sliding Window	12000	530
DSM-FI	Landmark Window	9200	1200
VSW	Sliding Window	900	700
AnyFI	Damped Window	upto 60000	600 -3400

Finally, we compare the mining time of FP-growth, DSM-FI and AnyFI for 100KD100T10I4 dataset (results in Table V). For fair comparison, we insert the complete dataset into the summary structures of all the three algorithms without conducting any pruning, and then mine for FIs. Mining in AnyFI is running in complete mode, i.e. without anytime interruptible feature. The results show that AnyFI performs faster. It is worth noting that FP-growth is used in SWP-tree and FP-Stream.

TABLE V. COMPARISION OF MINING TIME

Algorithm	Mining Time
FP-growth	5.8 sec
DSM-FI	3.3 sec
Any-FI	1.2 sec

V. CONCLUSIONS & FUTURE WORK

In this paper, we presented AnyFI which is the first anytime FI mining algorithm for data streams. AnyFI incorporates both the functionalities of an anytime algorithm – (i) ability to handle variable stream speeds, & (ii) ability to give an immediate mining result and improve its accuracy with increase in time allowance. AnyFI uses a novel data structure known as *BFI-forest*, which handles varying inter-arrival rates of data in the stream. Also, mining BFI-forest for FIs requires a simple traversal of its trees without generating any candidate itemsets in apriori like methods or conditional trees in FP-growth like methods, thus making it very efficient. The experimental analysis presented show that AnyFI can handle higher stream speeds while maintaining high recall.

In future, we plan to parallelize AnyFI for efficient FI mining of multi-port data streams.

REFERENCES

[1] G. S. Manku and R. R. Motwani, "Approximate Frequency Counts over Data Streams," in *VLDB '02 - Proceedings of the 28th VLDB Conference*, 2002, pp. 346–357.

[2] C. Giannella, et al., "Mining Frequent Patterns in Data Streams at Multiple Time Granularities," in *Data Mining: Next Gen. Challenges and Future Directions*, AAAI/MIT Press, 2003, pp. 191–212.

[3] S. K. Tanbeer, et al. , "Sliding window-based frequent pattern mining over data streams," *Inf. Sciences*, vol.179, no.22, pp. 3843–3865, 2009.

[4] H.-F. Li, M.-K. Shan, and S.-Y. Lee, "DSM-FI: an efficient algorithm for mining frequent itemsets in data streams," *Knowledge and Information Systems*, vol. 17, pp. 79–97, 2008.

[5] H. Chen, L. Shu, J. Xia, and Q. Deng, "Mining frequent patterns in a varying-size sliding window of online transactional data streams," *Information Sciences*, vol. 215, pp. 15–36, 2012.

[6] M. Deypir, M. H. Sadreddini, and S. Hashemi, "Towards a variable size sliding window model for frequent itemset mining over data streams," *Computers & Industrial Engineering*, vol. 63, no. 1, pp. 161–172, 2012.

[7] P. Kranen, I. Assent, C. Baldauf, and T. Seidl, "The ClusTree: Indexing micro-clusters for anytime stream mining," *Knowledge and Information Systems*, vol. 29, no. 2, pp. 249–272, 2011.

[8] M. Hassani, P. Kranen, and T. Seidl, "Precise anytime clustering of noisy sensor data with logarithmic complexity," in *Proceedings of the Fifth International Workshop on Knowledge Discovery from Sensor Data - SensorKDD '11*, 2011, pp. 52–60.

[9] K. Ueno, et al., "Anytime classification using the nearest neighbor algorithm with applications to stream mining," *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 623–632, 2006.

[10] P. Kranen, M. Hassani, and T. Seidl, "BT* - An advanced algorithm for anytime classification," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012, vol. 7338 LNCS, pp. 298–315.

[11] Shichao Zhang and Chengqi Zhang, "Anytime mining for multiuser applications," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 32, no. 4, pp. 515–521, Jul. 2002.

[12] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data - SIGMOD '93*, 1993, vol. 22, no. 2, pp. 207–216.

[13] P.-Y. Hsu, Y.-L. Chen, and C.-C. Ling, "Algorithms for mining association rules in big databases," *Information Sciences*, vol. 166, no. 1, pp. 31–47, 2004.

[14] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 3, pp. 372–390, 2000.

[15] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery*, vol. 8, no. 1, pp. 53–87, Jan. 2004.

[16] H.-F. Li and S.-Y. Lee, "Mining frequent itemsets over data streams using efficient window sliding techniques," *Expert Systems with Applications*, vol. 36, no. 2, pp. 1466–1477, 2009.

[17] "Market-Basket Synthetic Data Generator." [Online]. Available: https://synthdatagen.codeplex.com/. [Accessed: 19-May-2017].

[18] "Frequent Itemset Mining Dataset Repository." [Online]. Available: http://fimi.ua.ac.be/data/. [Accessed: 04-Jun-2017].

[19] "UCI Machine Learning Repository: MSNBC.com Anonymous Web Data Data Set." [Online]. Available: http://archive.ics.uci.edu/. [Accessed: 04-Jun-2017].

[20] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*, 2nd ed. Wiley London, 2000.