Original Research Paper

Is Single Scan based Restructuring Always a Suitable Approach to Handle Incremental Frequent Pattern Mining?

Shafiul Alom Ahmed and Bhabesh Nath

Department of Computer Science and Engineering, Tezpur University, Tezpur, India

Article history
Received: 22-12-2020
Revised: 10-03-2021
Accepted: 11-03-2021

Corresponding Author:
Shafiul Alom Ahmed
Department of Computer Science and Engineering, Tezpur University, Tezpur, India
Email: tezu.shafiul@gmail.com

Abstract: Incremental mining of frequent patterns has attracted the attention of researchers in the last two decades. The researchers have explored the frequent pattern mining from incremental database problems by considering that the complete database to be processed can be accommodated in systems' main memory even after the database gets updated very frequently. The FP-tree-based approaches were able to draw more interest because of their compact representation and requirement of a minimum number of database scans. The researchers have developed a few FP-tree based methods to handle the incremental scenario by adjusting or restructuring the tree prefix paths. Although the approaches have managed to solve the re-computation problem by constructing a complete pattern tree data structure using only one database scan, restructuring the prefix paths for each transaction is a computationally costly task, leading to the high tree construction time. If the FP-tree construction process can be supported with suitable data structures, reconstruction of the FP-tree from scratch may be less time consuming than the restructuring approaches in case of incremental scenario. In this study, we have proposed a tree data structure called Improved Frequent Pattern tree (Improved FP-tree). The proposed Improved FP-tree construction algorithm has immensely improved the performance of tree construction time by resourcefully using node links, maintained in header table to manage the same item node list in the FP-tree. The experimental results emphasize the significance of the proposed Improved FP-tree construction algorithm over a few conventional incremental FP-tree construction algorithms with prefix path restructuring.

Keywords: FP-tree, FP-Growth, Frequent Pattern, Pattern Mining, Data Mining, Frequent Itemset, Itemset Mining, Pattern Analysis

Introduction

In this 21st century, transactional databases are dynamic. Nowadays, researchers have focused on finding the hidden knowledge from these incremental databases. Frequent pattern mining is one of the most widely used knowledge retrieval techniques of data mining. The problem of mining frequent itemset was first brought to attention in the Apriori algorithm (Agrawal *et al.*, 1993). Apriori algorithm is a level-wise computation, which employs multiple database scans and generates an enormous number of candidate itemsets. Moreover, it exercises costly testing and prune out approach to discard the redundant and infrequent candidate itemsets to generate the complete set of frequent patterns. Later on, many attempts have been made by the researchers to propose an efficient method to mine the frequent itemsets from large datasets by adopting the Apriori approach. However, most of the proposed approaches sustain the same multiple database scans, computation time (candidate itemsets) and space problems.

To mitigate the multiple database scans and an enormous number of candidate itemsets generation, a group of researchers, (Han *et al.*, 2000) came up with a prefix path tree-based data structure approach called FP-Growth. FP-Growth handled the multiple scan problem by restricting it to only two. Moreover, it is capable of generating the complete set of frequent itemset without generating any candidate itemset. There have been many FP-tree based algorithms proposed by the researchers to improve the performance of FP-tree construction. However, those approaches primarily emphasize efficiently constructing the FP-tree and generating useful frequent patterns from static databases. However, in dynamic or incremental databases, FP-tree cannot directly reflect the database modifications onto the FP-tree or the already generated frequent patterns. Reconstructing afresh FP-tree and incrementally restructuring the FP-tree are two possible ways to deal with the incremental scenarios. If only a few transactions are frequently added to the database, it becomes computationally infeasible to repeatedly reconstruct the FP-tree from scratch every time transactions are added to the database. Therefore, researchers have concentrated on incrementally restructuring the FP-tree without reconstructing the FP-tree from scratch. The main aim of the restructuring operation is to maintain the basic FP-tree structure and properties. The restructuring is performed to reflect the changes in the database directly onto the FP-tree. The restructuring is achieved by performing a sequence of costly swapping and merging of FP-tree prefix paths. Though incrementally restructuring the FP-tree requires only a single database scan, the restructuring using swap and merge for each transaction becomes much costlier than afresh FP-tree construction if the size of newly added transactions is enormous. It is required to perform the restructuring operation along all the prefix paths containing an item that is to be updated in the FP-tree. Therefore, if the updated database size and the dimensionality are huge, the incremental restructuring consumes a significant amount of time compared to afresh FP-tree construction. Depending on the database size and dimensionality, both approaches have their pros and cons.

Most importantly, the data structures have a significant influence on the performance of frequent itemset mining algorithms. The data structure used by the FP-Growth algorithm is a compact prefix tree data structure named Frequent Pattern tree (FP-tree). FP-tree consists of nodes and each node contains a value pair of data or item and its count. The nodes containing the same item are maintained in the list. Therefore, every time a new node is created, it must traverse the whole same item list and add the newly created node at the list's rear position. Hence, the FP-tree data structure performs well in dense databases as many transactions will share the common prefix paths in the tree. However, for sparse databases with higher dimensions and high average transaction lengths, the FP-tree size becomes vast. Therefore, traversing a long list every time a new node is inserted into the FP-tree demotes the tree construction performance drastically. Depending on specific characteristics of databases such as updated database size, dimensionality, the average length of transactions, dense and sparse, a few approaches perform well. However, it may not be feasible to handle the incremental scenarios with all cases.

Therefore, in this study, we have addressed a new two scan based FP-tree construction algorithm named Improved FP-tree (IFP-tree) from scratch by manipulating the same item node links to handle the incremental scenarios. Instead of the linked list, we have maintained the nodes containing the same items in a stack. Stack enables to directly access the top node so that the newly created node can be inserted at the top of the stack without traversing the whole stack. Which saves a significant amount of time. Hence improves the IFP-tree construction time significantly. The IFP-tree construction algorithm's strength is that if the size of newly added transactions to the database is very high irrespective of dense or sparse, it outperforms few incremental restructuring algorithms and constructs the FP-tree from scratch approaches. The experimental results are significantly promising and establish the novelty of the proposed IFP-tree.

Frequent pattern tree construction algorithm plays an essential role in frequent pattern mining. The proposed Improved FP-tree data structure can efficiently mine frequent patterns and association rules from static and incremental datasets. Frequent patterns and association rules are used in different application domains such as market-basket analysis, risk analysis in commercial environments, disease factor analysis and patients survivability possibility analysis.

Preliminaries

The main objective of FIM is to generate the frequently occurring patterns or itemsets from transactional databases that are useful and meet users criteria in decision making. The usefulness and interestingness of patterns generated is gauged by some popular and most widely used measures discussed below:

Let D be a transactional database with items $I = [i_1, i_2 \ldots i_m\}$ and set of all transactions $T = \{t_1, t_2 \ldots t_n\}$. Each transaction t_j is a subset of I. A transaction t_j is said to contain an itemset say i, if i is a subset of t_j.

Definition 1. Support Count (σ)

Support count is the total number of transactions in the database that contain an itemset. Mathematically, the support count σ of an itemset P can be represented as:

$$\sigma(P) = \left|\left\{t_j \mid P \subseteq t_j; t_j \in T\right\}\right| \tag{1}$$

Definition 2. Support (Supp)

Support of an itemset P can be defined as the percentage of transactions in the transactional database D that contains the itemset P. Mathematically, the support of an itemset P can be represented as:

$$Supp(P) = \frac{\sigma(p)}{T} \tag{2}$$

Definition 3. Frequent Pattern or Frequent Itemset

An itemset is said to be frequent if the support of the itemset is greater than or equal to the user specified minimum support threshold *minSupp*. Formally, an itemset *P* is said to be frequent if it satisfies the following constraint:

$$Supp(P) \geq minSupp \tag{3}$$

Related Work

The problem of mining frequent itemsets from static databases was first coined by Agrawal *et al*., named Apriori (Agrawal *et al*., 1993). There are several variants of Apriori based incremental algorithms have been proposed by the researchers, for instance, FUP (Cheung *et al*., 1996), FUP-2 (Cheung *et al*., 1997), Border algorithm (Aumann *et al*., 1999), Modified borders (Das and Bhattacharyya, 2004), Update With Early Pruning (Ayan *et al*., 1999) (UWEP), DEMON (Ganti *et al*., 2001), Incremental Constrained APriori (ICAP) (Ayad, 2000), Maintaining Association Rules with Apriori Property (Zhou and Ezeife, 2001) (MAAP), Maximal Frequent Trend Pattern (MFTP) (Guirguis *et al*., 2006), PRE-HU (Lin *et al*., 2014). Like Apriori, if the length of the maximum frequent itemset is K, the approaches require at least K number of scans over the database. Several limitations, such as multiple database scans and the generation of an enormous number of candidate itemset of the Apriori algorithm makes it computationally infeasible to handle the incremental scenario of frequent pattern mining. Later on, (Han *et al*., 2000) propose an efficient approach, “FP-growth,” using a prefix tree data structure called FP-tree. The researchers have exhaustively exploited the FP-tree of the FP-growth algorithm to mine frequent patterns as it can improve the mining performance compared to the candidate itemset generation and prune out mechanism of Apriori using multiple databases scans. FP-growth requires only two database scans. Since FP-tree is dependant on the user-defined minimum support threshold and the FP-tree contains information about only those items, it cannot be easily made compatible with the dynamic scenario. Reconstructing the FP-tree from scratch using two database scans every time new transactions are added to the database, or the minimum support change is not feasible. Although a significant number of approaches viz. nonordfp (Rácz, 2004), IFP Growth (Lin *et al*., 2011), LP Growth (Pyun *et al*., 2014), Incremental FP Tree (Adnan *et al*., 2006b), Alternative FP Tree (Alhajj and Barker, 2008), DB-tree (Ezeife and Su, 2002), FUFP (Hong *et al*., 2008) and Pre-FUFP (Lin *et al*., 2009) have been proposed in the last two decades. However, most of the approaches suffer from the same problems as FP-tree. Therefore, to deal with constructing afresh FP-tree, researchers have developed few new approaches, basically improvements over the FP-tree, to generate frequent patterns from incremental databases efficiently. The incremental approaches take only one database scan and apply a split, swap and merge operations sequence to construct the FP-tree incrementally. Few FP-tree based single scan incrementally restructuring approaches are discussed below.

FP-Tree based Incremental Approaches

Incremental Frequent Pattern Tree (Adnan *et al*., 2006a) does not reconstruct the FP-tree from scratch whenever the database changes. To achieve this, a complete FP-tree (assuming the minimum support threshold as one) is constructed. The complete reflects all the occurrences of items in the database onto the FP-tree. As the database gets updated, this algorithm incrementally updates the FP-tree without re-scanning the old database or reconstructing the FP-tree from scratch. This algorithm uses two primary operations: “Shuffling” and “merging” to maintain the FP-tree structure. However, this algorithm scans the original database twice to construct the initial FP-tree also scans twice the newly added set of transactions every time the database gets updated. The Fast Updated FP-tree (FUFP) (Hong *et al*., 2008) is an improvement over FP-tree based on the FUFP concept to mine incremental frequent patterns. This algorithm first divides the items into four groups based on whether the items are large or small in the old database and new transactions. Whenever these sets get changed, the Header-Table and the FUFP-tree are updated accordingly. When a sufficiently large number of transactions are inserted, then the entire tree needs to be reconstructed in a batch way. Pre-FUFP (Lin *et al*., 2009) is a modification over FUFP based on the concept of “pre-large” itemsets. This algorithm uses two threshold values; one is a lower support threshold and another one is an upper support threshold to define the pre-large itemsets. However, this algorithm requires an extra minimum support threshold input. The CATS-tree (Cheung and Zaiane, 2003) is an extension over FP-tree to improve storage compression so that it can be quickly adapted to mine frequent itemsets from incremental databases without candidate generation. The CATS-tree (Cheung and Zaiane, 2003) requires only a single pass over the dataset to construct the CATS-tree and each path from the root node to the leaf node represents a set of transactions. When the database gets updated, the new transactions are added at the root level. Then, the transactions’ items are compared with the child_nodes in each level to determine the common items in both the new transaction and the child_nodes. If there are common items, then the transaction is merged with the nodes. The frequencies of the node are incremented. However, this algorithm consumes a considerable amount of time to update the tree when the database gets incremented. Adjusting FP-tree for Incremental Mining (AFPIM) (Koh and Shieh,

2004) is also an improvement over the FP-tree structure; it uses two support thresholds to mine incremental frequent itemsets. It is an $O(n^2)$ algorithm and also it requires extra pre-minimum support. Later on, to solve the problems of AFPIM, (Leung *et al*., 2005; 2007) developed a tree data structure called CANonical-order Tree (CanTree). However, the ordering of the items in the CanTree should be unaffected even if the frequency of the items changes due to incremental database updates to maintain the items' canonical order. Which may not be possible in all cases of real-life scenarios. Though CanTree does not require any prefix path adjustment or restructuring, it requires a huge memory space to store the database information. Afterward, (Tanbeer *et al*., 2009) developed a tree data structure named CP-tree or Compact Pattern tree. CP-tree is constructed using only one database scan. A pre-defined number of transactions (slot) are inserted into the CP-tree one by one according to the pre-defined item order. After inserting a slot of transactions, if the frequency-dependent item order gets changes up to a pre-defined degree, the CP-tree construction algorithm restructures the tree by adjusting and sorting the prefix paths. Even if the CP-tree is periodically restructured, it still requires a considerable amount of time. The researchers proposed several FP-tree-based incremental approaches with both single and multiple database scans during the last two decades. A summary of a few incremental approaches found in the literature is briefly discussed in Table 1.

Proposed Method

Improved FP-tree Construction

The Improved FP-tree construction algorithm presented in this study is based on the basic paradigm of the FP-tree construction algorithm of FP-growth (Han *et al*., 2000). Unlike FP-tree, the Improved FP-tree is a complete tree, i.e., it stores all the database transactions without any information loss. The proposed Improved FP-Growth algorithm customizes the conventional FP-tree construction algorithm to enhance tree construction performance. The performance enhancement is achieved by maintaining each list of nodes containing the same item of Improved FP-tree as a linked-list implemented as stack instead of maintaining a simple linked-list as a conventional FP-tree. The same item node-link from the header table points to the most recently inserted, i.e., the stack's top node. The stack implementation of the same item node list lets us access the top node without traversing the whole list directly. The direct access eliminates the same item node list traversal for every new node inserted into the Improved FP-tree during its construction. Which saves a significant of time. For example, let *Node*(*X*) is the top node of the same item list stack of item *X*. Therefore, the header table same item node-link for item *X* will be pointing to the stack top node *Node*(*X*). Whenever a new node *Node*(*X′*) is inserted, we can directly access the top node *Node*(*X*) of the stack with header tables' node-link, without traversing the whole list for item *X*. Then the same item link of node *Node*(*X′*) is set to the existing top node *Node*(*X*). After that, the header table's node link is updated to the recently added node *Node*(*X′*). The complete step by step procedure for constructing the Improved FP-tree is illustrated in Algorithm 1 and Algorithm 2.

The working principle of Improved FP-tree construction is illustrated in this section by considering a small transactional dataset D [Table 2] and the minimum support to be 1.

Like FP-tree, the Improved FP-tree construction algorithm also requires two database scans to construct the tree. Initially, the whole transactional database D is scanned once to fetch the frequency of each item. Items with their corresponding frequencies are shown in Table 3.

Table 1: Summary of FP-tree based incremental frequent itemset mining algorithms

Algorithm	Update type	Datset rescan	Data structure
P-Tree, (Huang *et al*., 2002)	Addition	1	FP-tree
Generalized FPtree, (Ezeife and Su, 2002)	Addition	≥ 1	FP-tree
CATS-tree, (Cheung and Zaiane, 2003)	Addition, Deletion	1	FP-tree
AFPIM, (Koh and Shieh, 2004)	Addition, Deletion, Modification	≥ 0	FP-tree
IFP-tree, (Adnan *et al*., 2006a)	Addition, Deletion, Modification	2	FP-tree
CanTree, (Leung *et al*., 2007)	Addition, Deletion, Modification	0	FP-tree
FUFP, (Hong *et al*., 2008)	Addition, Deletion	≥ 1	FP-tree
Pre-FUFP, (Lin *et al*., 2009)	Addition, Deletion	≥ 0	FP-tree
CP-tree, (Tanbeer *et al*., 2009)	Addition, Deletion	1	FP-tree
Improved CPtree, (Hamedanian *et al*., 2013)	Addition	≥ 1	FP-tree
GM-tree, (Roul and Bansal, 2014)	Addition	1	FP-tree
HUPIDGrowth, (Yun and Ryang, 2015)	Addition, Deletion	1	FP-tree
SPFPtree, (Shahbazi *et al*., 2016)	Addition	1	FP-tree
VSIFP, (Yu-Dong *et al*., 2016)	Addition	1	FP-tree
SSP-tree, (Borah and Nath, 2018)	Addition, Deletion	1	FP-tree

Table 2: Database (D)

Tid	Transaction
T1	{b, d}
T2	{a, b, e}
T3	{a, c, d, e}
T4	{a, d, c}
T5	{b, d, e}
T6	{a, b, d, e}
T7	{d}
T8	{b, d, e}
T9	{a, b, d}
T10	{b, c, e}

Table 3: Item frequencies

Item	Frequency count
‘a’	5
‘b’	7
‘c’	3
‘d’	8
‘e’	6

Table 4: Sorted order

Item	Frequency count
‘d’	8
‘b’	7
‘e’	6
‘a’	5
‘c’	3

The items are then sorted with respect to their frequency count in descending order. Since the minimum support is 1, every item occurring at least once in the database will be considered to construct the complete Improved FP-tree. Hence all the items are inserted into the header table. The items after sorting in descending order according to their corresponding frequency counts are represented in Table 4.

The frequency counts are set as zero after inserting the frequent items into the header table in frequency descending order. After that, the root node of the Improved FP-tree is created. In the second database scan, the transactions of the database are read and inserted into the Improved FP-tree one by one. The steps required to be performed to insert the transactions of D into the tree are as follows:

(a) All items of the first transaction $\{b, d\}$ are frequent are sorted according to the order of header table items and the resulting sorted transaction is $\{d, b\}$. Since the root node has no child branches yet, a node (Node(d:1)) containing item = ‘d’ and count is created and inserted as the first child_node of root. Then the frequency count of item ‘d’ in the header table is incremented by 1. Then the header table same item link is updated to Node(d:1) and set Node(d:1) as the top node of items ‘d’s stack. Similarly, for item ‘b’, Node(b:1) is created and inserted as child_node of Node(d:1). Thereafter, the frequency count of item ‘b’ in the header table is incremented by 1 and the header table same item link is updated to Node(b:1), the top node of items ‘b’s stack. The resultant Improved FP-tree after inserting first transaction is shown in Fig. 1

(b) For the second transaction $\{a, b, e\}$, the transaction is sorted as $\{b, e, a\}$. Since the root has no child_node with item label ‘b’, a new node Node(b:1) is created and inserted as a child_node of root. The frequency count of item ‘b’ in the header table is incremented by 1. Since the same item node link is already pointing to the same item stacks top node, the top Node(b:1) is directly accessed and set it as the same item next node of newly inserted node Node(b:1). The header table node link is then updated to the newly created node Node(b:1) as top. Rest of the items ‘e’ and ‘a’ are inserted as a child branch of Node(b:1) and their header table node links are also set. The Improved FP-tree generated after inserting the second transaction is shown in Fig. 2

(c) All the items of third transaction are sorted according to the order of header table and the resulting sorted transaction is $\{d, e, a, c\}$. In Fig. 2, we can see that the root node of Improved FP-tree has a child_node Node(d:1). Thus, for the first item ‘d’ of sorted transaction, the count of Node(d:1) and the corresponding frequency count in header table is incremented by 1. Since Node(d:2) has no child_node with item label ‘e’, therefore a new Node(e:1) is created and inserted as child_node of Node(d:2). The same item link of newly created Node(e:1) is set to the already existing node pointed by header table node link and header table node link is then updated to Node(e:1). Similarly, Node(a:1) and Node(c:1) are created and inserted as a child branch of Node(e:1). Their header table frequency counts, header table node links and same item node links are updated accordingly. The resultant Improved FP-tree is portrayed in Fig. 3

(d) For the first item ‘d’ of sorted forth transaction {d, a, c}, root node of Improved FP-tree in Fig. 3 already has a child_node Node(d:2). Therefore, count of the node and corresponding header table frequency count are simply incremented by 1. But Node(d:3) does not have any child_node with item label ‘a’. A new Node(a:1) is created and inserted as a child_node of Node(d:3). Their header table frequency counts, header table node links and same item node links are updated accordingly. Similarly, item ‘c’ is also handled and the resultant Improved FP-tree after inserting forth transaction is shown in Fig. 4

Algorithm 1: Improved_FP_Tree-Construction (*minsupp*, D)

```
1  input: Minimum support count (minsupp), Transactional Database D;
2  output: Improved FP-tree;
3  Header Table Item Insertion
4  Create an empty header table and the root node of the Improved FP-tree;
5  Scan the database and derive the frequency counts of each individual item;
6  Insert the items into the header table based on their frequency descending order;
7  Improved FP-tree Insertion
8  Read the transactions T of D, one at a time during the second database scan;
9  for each transaction $T_i$ do
10     Sort the frequent items $I_j$ of $T_i$ according to the header table item order;
11     Set tempRoot = root;
12     for each item $I_j$ in sorted $T_i$ do
13         Call tempRoot = insertInto-Improved FP-tree ($I_j$, tempRoot);
14     end
15 end
```

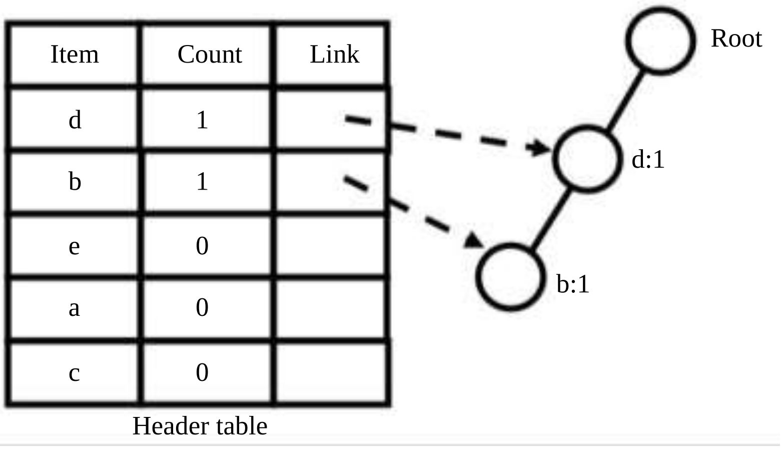
Algorithm 2: insertInto-Improved_FP-tree(I, Root)

```
1  if Root has a child_node with item label I then
2      Increment the frequency count of the child_node and corresponding header table frequency by 1;
3  else
4      Create a new child_node of Root with item label I and increment corresponding header table item frequency by 1;
5      if the node link pointer of corresponding header table position is NULL then
6          Set the header table node link pointer to child_node;
7          Set child_node as stack top;
8      else
9          Set the same item node link pointer of child_node to the top node pointed by header table node link pointer;
10         Update the header table node link pointer to child_node and set child_node as the new stack top;
11     end
12 end
13 return child_node;
```

(e) After sorting the fifth transaction as {d, b, e}, it can be seen in Fig. 4 that, items ‘d’ and ‘b’ are already shared by a prefix path. Therefore, the node counts and the corresponding header table frequency counts are simply incremented by 1. Since, Node(b:2) has no child_node, so for item ‘e’, Node(e:1) is created and inserted as child_node of Node(b:2). Thereafter, the header table frequency count, header table node link and same item node link of newly created Node(e:1) are updated. The resultant Improved FP-tree is illustrated in Fig. 5

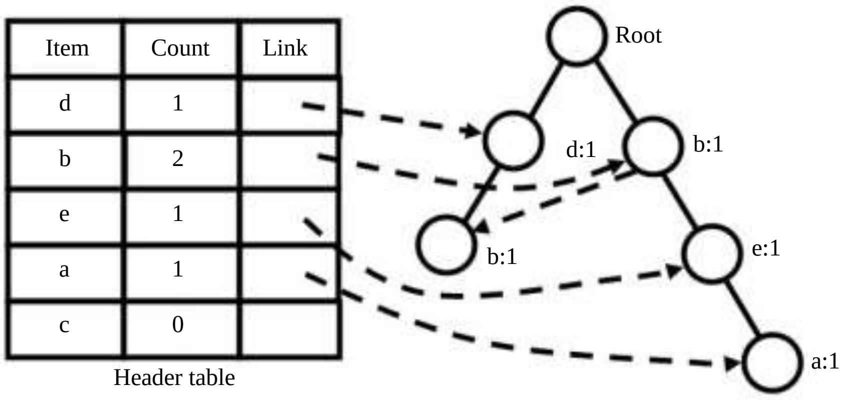
(f) Likewise, for the items ‘d’, ‘b’ and ‘e’ of sorted sixth transaction {*a*, *b*, *d*, *e*}, the counts of the shared nodes of the prefix path in Fig. 5 are just incremented by 1. The header table frequency counts of the corresponding items are also incremented by 1. Then Node(a:1) is created and inserted as child_node of Node(e:2) and the corresponding header table frequency count, header table node link and same item node link is updated. The resultant Improved FP-tree after inserting sixth transaction is illustrated in Fig. 6

(g) Identically, after inserting all transactions of the database D, the final resultant Improved FP-tree is shown in Fig. 7



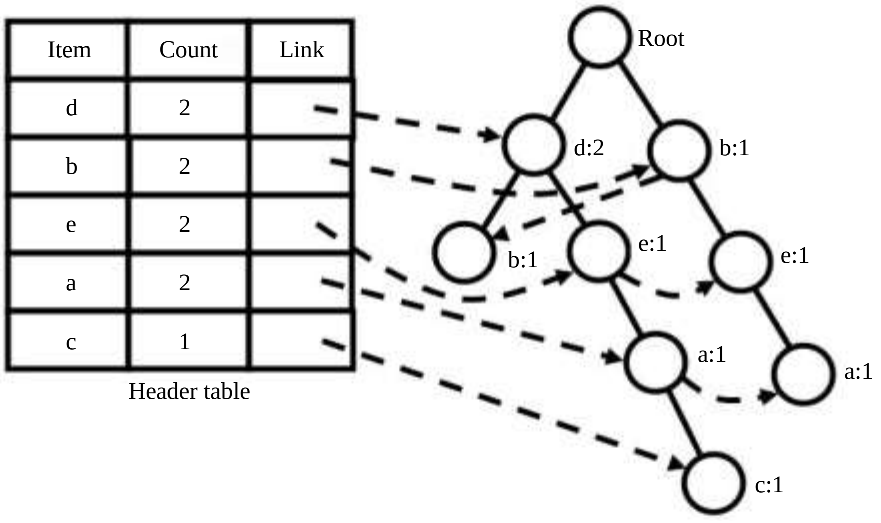
Item	Count	Link
d	1	
b	1	
e	0	
a	0	
c	0	

Fig. 1: Improved FP-tree after inserting T_1



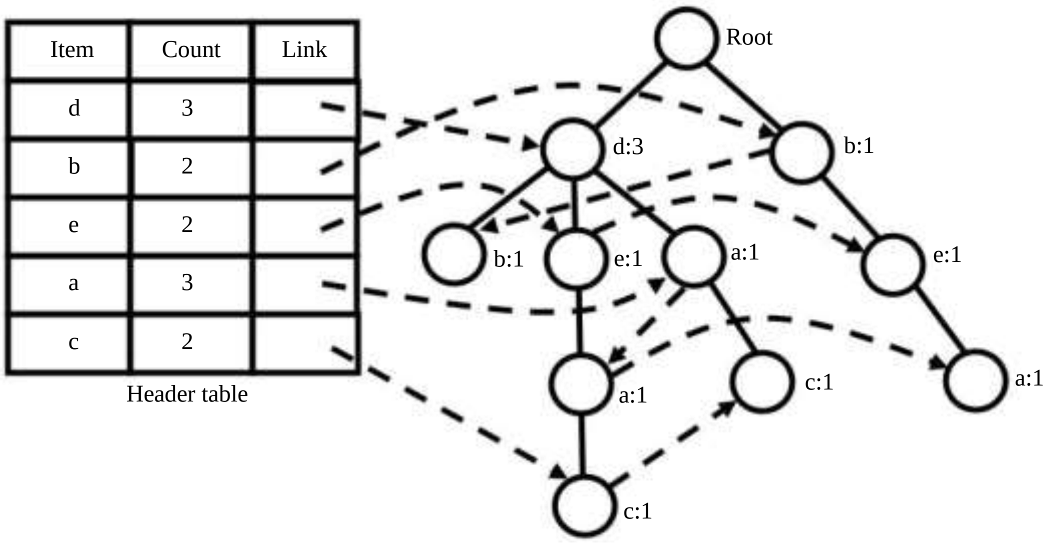
Item	Count	Link
d	1	
b	2	
e	1	
a	1	
c	0	

Fig. 2: Improved FP-tree after inserting T_2



Item	Count	Link
d	2	
b	2	
e	2	
a	2	
c	1	

Fig. 3: Improved FP-tree after inserting T_3



Item	Count	Link
d	3	
b	2	
e	2	
a	3	
c	2	

Fig. 4: Improved FP-tree after inserting T_4

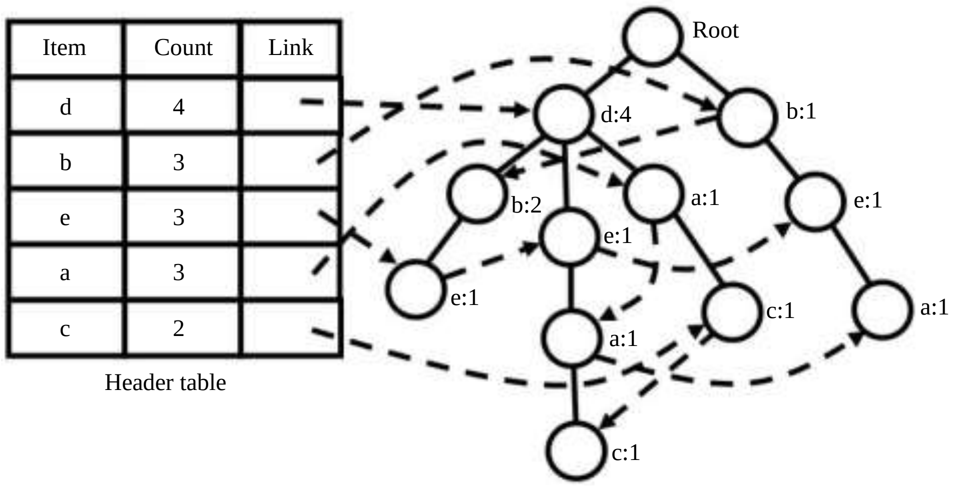


Fig. 5: Improved FP-tree after inserting T_5

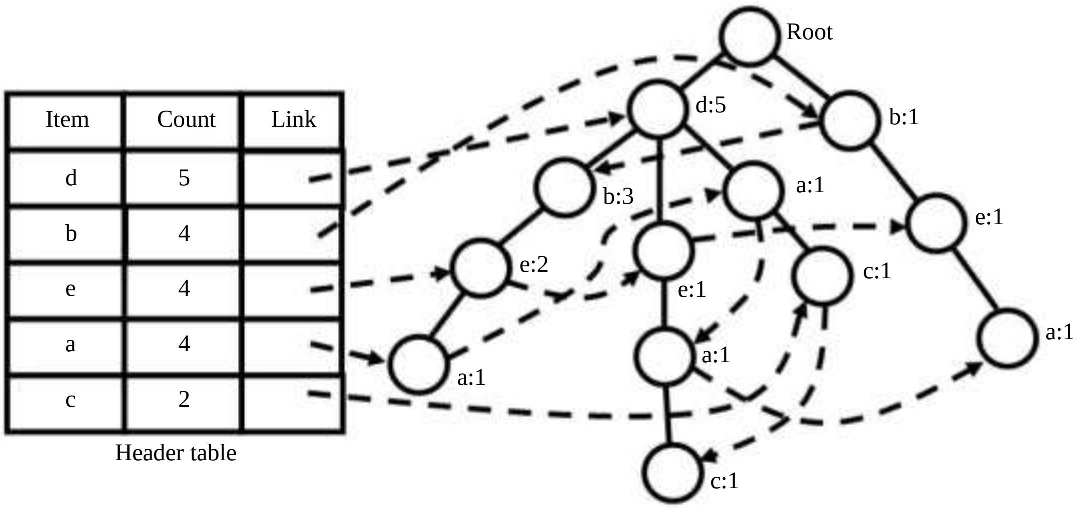


Fig. 6: Improved FP-tree after inserting T_6

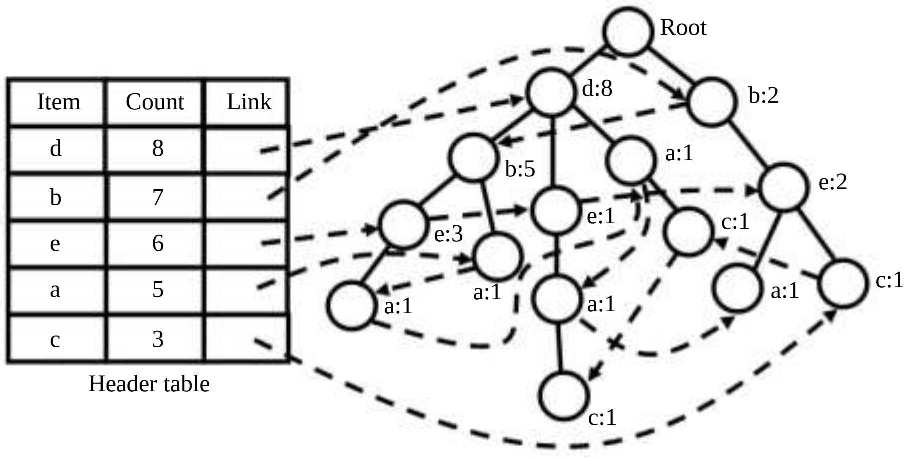


Fig. 7: Improved FP-tree after inserting all transactions

Experimental Results Evaluation

In this section, we are going to analyze the performance of the proposed Improved FP-tree construction. The Improved FP-tree algorithm's significance is established by comparing its performance with FP-tree with two databases scan-based tree construction and two incrementally restructuring single scan-based approaches viz. CP-tree and SSP-tree. Several experiments have been carried out to assess the Improved FP-tree construction algorithm's effectiveness by considering the total tree construction time, the effect of updated database size and minimum support threshold change.

Experimental Environment and Datasets

All the tree construction and pattern growth algorithms are coded in C and run on Ubuntu-18.04.2 with 2.67 GHz CPU and 8 GB main memory. To assess the significance of the Improved FP-tree construction algorithm over other alternative tree construction algorithms, we have conducted experiments on both real and synthetic datasets, as well as dense and sparse datasets. The datasets presented in Table 5 are retrieved from the UCI Machine Learning Repository and FIMI Repository.

Complete Tree Construction Time

The initial experiment has been carried out to analyze the total time taken by the proposed tree construction algorithm to construct the Improved FP-tree. The total time taken by the Improved FP-tree construction algorithm has been compared to the total tree construction time of FP-tree, CP-tree and SSP-tree for different datasets mentioned in Table 5. As mentioned above, Improved FP-tree, as well as CP-tree and SSP-tree, are complete trees. That means all three tree data structures are independent of the user-defined minimum support threshold. Therefore, all the items appearing in a database are considered for constructing the trees, irrespective of their frequency counts. Hence for performing a proper comparison, though the FP-tree construction algorithm takes two database scans, we have considered the minimum support 1 to construct the FP-tree. SSP-tree construction algorithm constructs the tree in a single scan over the database and processes the database transactions one by one. The algorithm performs some restructuring operations based on the updated header table item counts to maintain the FP-tree properties for each transaction. Which consumes a significant amount of time. On the other hand, the CP-tree algorithm also constructs the tree by taking a single database scan. However, instead of performing restructuring for each transaction periodically, i.e., after inserting a certain number of transactions (slot), the restructuring is performed to improve the tree construction time. For this experiment, we have considered the slot size to be 10K for CP-tree construction. Therefore, the CP-tree construction algorithm performs the restructuring after inserting 10K transactions into the CP-tree. Table 6 depicts the total time taken by all the above-mentioned tree construction algorithms to construct or restructure the complete trees.

From Fig. 8, it can be observed that Improved FP-tree outperforms all the other three tree construction algorithms for both dense and sparse databases. However, for sparse databases, the performance of the Improved FP-tree is more prominent. The number of items in a sparse database is very high as compared to a dense database. The possibility of sharing the tree prefix paths less, resulting in expanse the same item lists, increases the tree size. Therefore, every time a new node is inserted into the tree, the FP-tree construction algorithm has to traverse a long list of the same item nodes. In addition to the same item list traversals, CP-tree and SSP-tree construction algorithms also have to restructure the tree prefix paths. Table 6 represents the number of same item list traversals performed by different algorithms to construct the trees for different databases.

From Table 7, it can be observed that all three algorithms perform several traversals over the same item lists. Simultaneously, the proposed Improved FP-tree construction algorithm does not perform the traversal a single time. Though it constructs the Improved FP-tree from scratch using two database scans, neither it performs any same item list traversal nor requires incremental restructuring of the Improved FP-tree to handle incremental scenarios.

Effect of Updated Database Size

To assess the Improved FP-tree construction algorithm's performance concerning database update, i.e., in the incremental scenario, we have initially conducted experiments in one dense real-life database, "Connect-4" and one sparse synthetic database, "T40I10D100K". For the "Connect-4" database, to begin with, we have considered the first 15K transactions as input and executed the tree construction algorithm. Each time, the size of the input database size is updated with the next set of 15K transactions. Finally, the remaining 7557 transactions are set as input. For a fair comparison, we have considered the minimum support to be 1 for FP-tree, i.e., if an item occurs at least once in the database, it will be taken into account to construct the tree. Since the CP-tree performs the restructuring periodically, we have performed the restructuring after inserting every 2.5K transactions in this experiment. The time taken by the proposed Improved FP-tree construction algorithm and the other tree construction algorithms for a different-sized set of transactions of "Connect-4" is illustrated in Fig. 9.

Table 5: Databases used

Database	Category	Average length	Number of transaction	Number of item	Type
Mushroom	Real	23	8,124	119	Dense
Connect-4	Real	43	67,557	129	Dense
Retail	Real	10	88,162	16,470	Sparse
T10I4D100K	Synthetic	10	100,000	870	Sparse
T40I10D100K	Synthetic	40	100,000	1000	Sparse

Table 6: Execution time (in seconds) of different algorithms

Algorithm	Mushroom	Connect-4	Retail	T10I4D100K	T40I10D100K
FP-tree	0.11	19.608	29.108	20.014	724.816
SSP-tree	0.562	71.372	52.434	45.18	794.574
CP-tree	0.294	53.404	41.212	29.036	714.944
Improved FP-tree	0.076	0.972	25.314	2.058	7.514

Table 7: Same item node list traversed by different algorithms

Algorithm	Mushroom	Connect-4	Retail	T10I4D100K	T40I10D100K
FP-tree	27269	359292	677466	714185	3562155
SSP-tree	27358	392270	677469	714733	3562959
CP-tree	81859	769332	728643	752914	3619722
Improved FP-tree	0	0	0	0	0

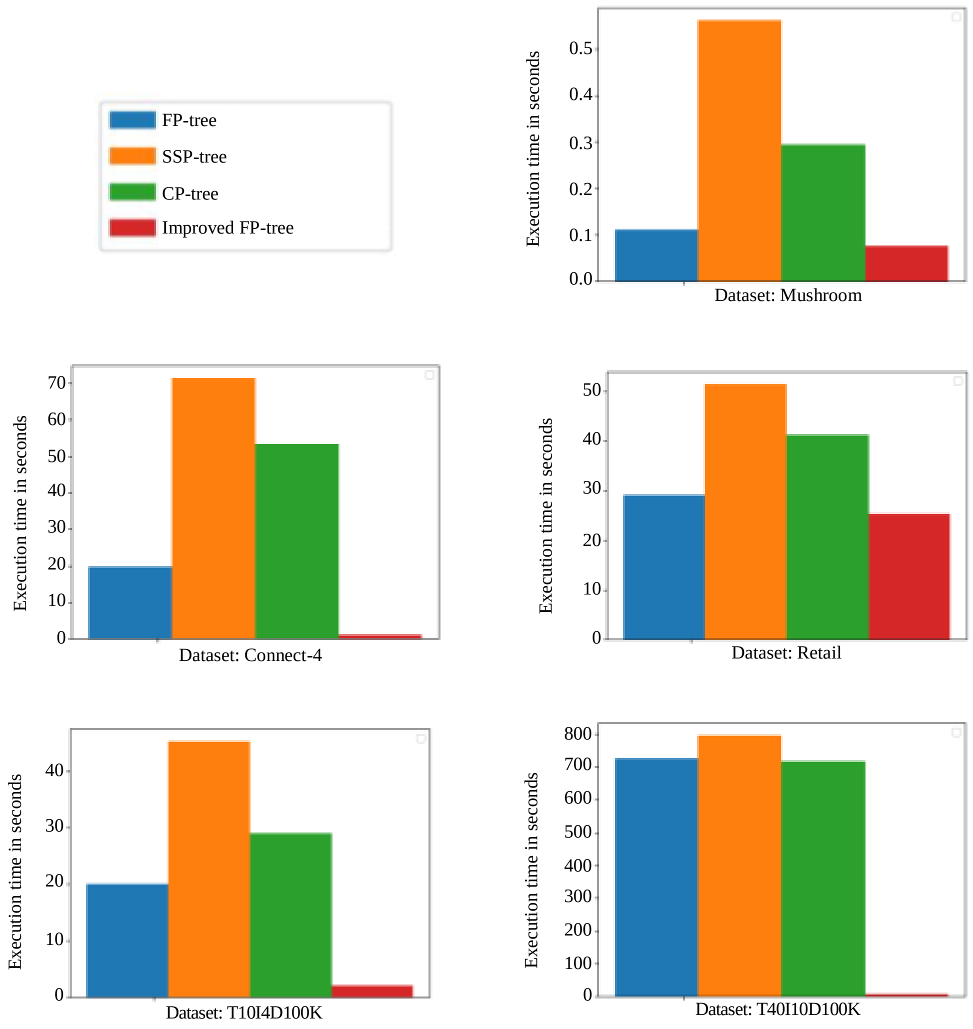


Fig. 8: Execution time comparison of FP-tree, SSP-tree, CP-tree and proposed Improved FP-tree construction algorithms for all the datasets of Table 5

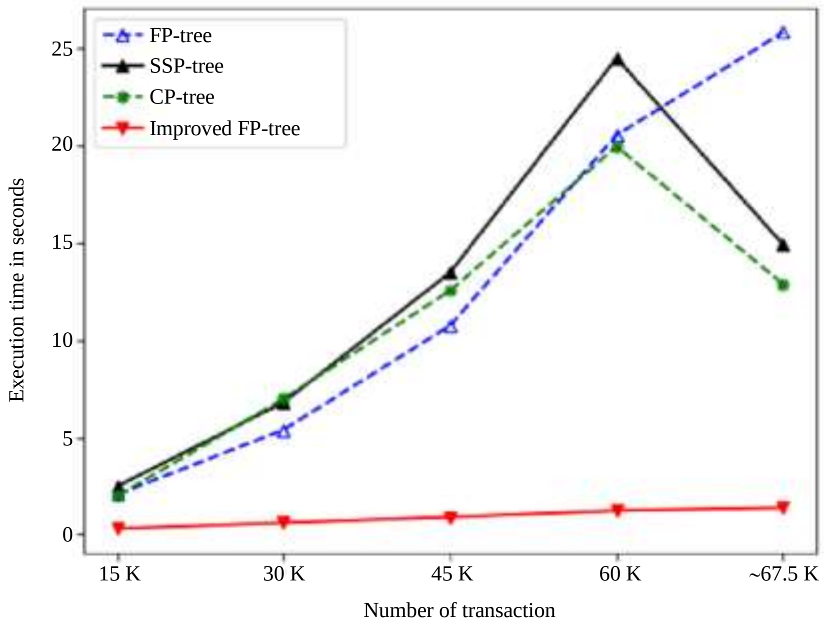


Fig. 9: Execution time (Connect-4)

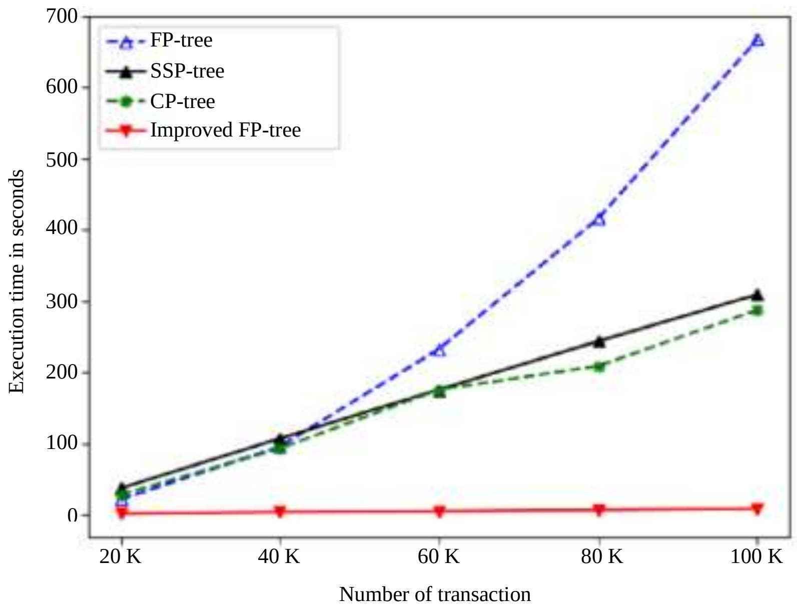


Fig. 10: Execution time (T40I10D100K)

From Fig. 9, it can be observed that in the incremental scenario, our proposed Improved FP-tree construction the algorithm outperforms the approaches with reconstruction from scratch as well as incrementally restructuring approaches. The dense databases contain a small number of attribute values or items. Therefore, the possibility of sharing the items in transactions is very high. Therefore, a tree prefix path or its sub-part is shared by multiple database transactions. Which maximizes the compactness of the tree. Since most of the tree nodes are shared by multiple transaction items, a minimum number of tree nodes creation and relatively smaller same item node lists lead to a minimum number of tree nodes creation. The same item node list is traversed if a new node is created and inserted into the tree. Though the same item node lists are relatively smaller for dense databases, it does not prevent the FP-tree, SSP-tree, or CP-tree from traversing the list whenever a new node is inserted into the tree. On the contrary, in our proposed Improved FP-tree construction algorithm, it is not required to traverse the whole same item node list every time a new node is inserted into the Improved FP-tree. The stack implementation of the same item node list prevents it from traversing the same item node list. As the same item node-link points to the most recently inserted or the top node of the stack, we can directly access the last top node and insert the new node as the new top node. It saves a significant amount of time compared to conventional FP-tree, SSP-tree and CP-tree.

A sparse database can be considered as contradictory to a dense database. That means the database contains a relatively large number of distinct items. Therefore, the possibility of sharing a prefix path is significantly less as compared to the dense database, which increases the size (breadth) of the FP-tree and leads to relatively longer same item node lists. Therefore, in the case of a sparse database, the performance of the FP-tree, SSP-tree and Cptree construction algorithms atrophy drastically. For the sparse database "T40I10D100K", we have considered the minimum support to be 1 and each time 20K transactions increment the input database slot size. Similarly, for CPtree construction, we have performed the restructuring after inserting 5K transactions. The time is taken by the proposed Improved FP-tree construction algorithm and the conventional FP-tree, SSP-tree and CP-tree construction algorithms for the different sized sets of transactions for the "T40I10D100K" database is illustrated in Fig. 10.

From Fig. 10, it can observe that in the case of the sparse database, the performance of the Improved FP-tree construction algorithm is much prominent than FP-tree, SSP-tree and CP-tree construction algorithms. For a sparse database, the same item node lists' length is relatively more extensive compared to dense databases. Therefore, FP-tree, SSP-tree and CP-tree consume a considerable amount of time to traverse those lists. Moreover, SSP-tree and CP-tree need to restructure the tree data structures. Fig. 10 shows that when the CP-tree is restructured after inserting every 5K transactions, SSP-tree and CP-tree take almost the same amount of time to construct the trees in the incremental scenario. Therefore, if the CP-tree will be structured after inserting a lesser number of transactions, i.e., it will increase the number of CP-tree restructuring and demote the tree construction performance. At some point in time, CP-tree may require more time than SSP-tree restructures the tree. On the other hand, Improved FP-tree acquired great convenience concerning execution time over other tree construction algorithms by avoiding the same item node lists' repetitive traversal. Figure 9 and 10 show that though the Improved FP-tree is reconstructed from scratch every time the database gets updated, it takes significantly less time to construct the Improved FP-tree than other tree construction algorithms.

Effect of Minimum Support Threshold Change

Except for FP-tree, SSP-tree, CP-tree and the proposed Improved FP-tree are complete trees. But in our experiment, we have constructed the FP-tree also by considering the minimum support to be 1. Therefore, like complete trees, FP-tree also maintains all the database transactions without any information loss. Hence, even if the support threshold changes, it does not affect the tree construction algorithms' performance. The significant advantage of constructing a complete tree is that neither it is required to reconstruct the tree from scratch nor restructure the tree even if the support threshold changes. Moreover, it enables generating frequent patterns for any set of minimum support threshold without intrusion to the tree data structure. However, the complete tree has a significant disadvantage also. If the minimum support threshold is very high, only a few database items will be frequent. Therefore, the complete tree will consume a considerable amount of physical memory to maintain less interesting or infrequent items. Which will unnecessarily increase the tree size.

Computational Complexity Analysis

The conventional FP-tree and the proposed Improved FP-tree construction algorithms can be defined in three phases: Header table management, sort transaction and transaction insertion. On the other hand, prefix-path restructuring based SSP-tree and CP-tree construction algorithms require an additional phase, "prefix-path restructuring" to maintain the FP-tree properties. Therefore, the time complexity for each phase is analysed subsequently. Let D be a dataset of T transactions and containing N number of items. Let M is the longest transaction size, where $2 \leq M \leq N$. Therefore, M will also be the tree height:

- Header Table Management: The frequent pattern tree is a compact representation of a dataset's required information. The header table plays a vital role in the pattern generation phase. The construction of the Conditional FP-trees from the FP-tree to generate frequent patterns will be very time consuming without the header table's support. Hence, the header table is to be constructed during the construction of the FP-tree. The header table items are always maintained in descending order of their frequency counts to maintain the FP-tree properties. In the case of FP-tree and Improved FP-tree, the dataset is scanned once to compute each item's frequency counts. The items are then sorted in descending order concerning their frequency counts. After sorting, all the items are accordingly inserted into the header table. In the worst-case scenario, the sorting costs $O(N \log N)$ and inserting the header table items require linear time. Therefore, this phase's total time complexity is $O(N \log N)$ and additionally the time taken for a complete scan of the dataset

On the contrary, for each transaction, the SSP-tree construction algorithm re-arranges the header table items in frequency descending order due to the relative changes in the header table's items. It helps in reducing the dataset scan to one but at the cost of on memory computation. In the worst-case scenario, the longest transaction of the dataset will contain N number of items. Hence, the SSP-tree construction algorithm's total time complexity for this phase is $O(TN^2)$

CP-tree performs the header table restructuring periodically after inserting some transactions into the tree to reduce the execution time. Let CP-tree re-arranges the header table after inserting P transactions into the tree. Therefore, the total time complexity of the CP-tree construction algorithm for this phase is $O\left(\frac{T}{P}N^2\right) = O(QN^2)$, where $Q < T$. However, in the worse case, $P = 1 \Rightarrow Q = T$, i.e., the restructuring will be performed for each transaction. Hence the total time complexity of the CP-tree construction algorithm for this phase is $O(TN^2)$

- Prefix-path Restructuring: In the worst-case scenario, the tree nodes' order will be in the reverse order of the corresponding header table item order. If the longest path of the tree contains N items, for each item, it will perform N-1 number of restructuring operation. For N items, it will require $N \times (N-1)$, i.e., $O(N^2)$ time. Therefore, for all T transactions, the total time complexity of SSP-tree construction for this phase is $O(TN^2)$.

  On the other hand, CP-tree performs the restructuring only $\frac{T}{P} = Q$ times. Hence, the total time complexity of this phase is $O(QN^2)$. In the worst-case scenario, the total time complexity of the CP-tree construction algorithm for this phase will be $O(TN^2)$
- Transaction Sorting: Since the items of a transaction are sorted according to the descending frequency order of the header table, searching an item into the header table requires linear time and for each item of the transaction, it will take the same amount of time. In the worst-case scenario, the length of each transaction will be N. Therefore, the total time complexity of this phase for all the tree construction algorithms again lead to $O(TN^2)$.
- Transaction Insertion: The time complexity of inserting a transaction into a tree depends on searching each item of the transaction in the header table, the length of the same item list of each item in the tree and the depth of the tree. The maximum depth of the tree is upper-bounded by N for each of the prefix sub-trees. The maximum length of the same item list is upper-bounded by T, i.e., the total number of transactions.

  Therefore, the transaction insertion phase's total time complexity can be represented as O(Total number of transactions × Number of items in header table × depth of the tree × maximum length of same item list)

  Since FP-tree, SSP-tree and CP-tree traverse the whole same item list to insert a transaction item into the tree. Therefore, the total time complexity of all these three tree construction algorithms for this phase is $O(T \times N \times N \times T) = O(T^2N^2)$.

  On the other hand, the proposed Improved FP-tree can directly insert a node without traversing the same item node list (constant time). Therefore, the total time complexity of Improved FP-tree construction algorithm for this phase is $O(T \times N \times N \times 1) = O(TN^2)$.

Therefore, the total time complexity of all the tree construction algorithms can be asymptotically represented as:

- For FP-tree: $O(N \log N + TN^2 + T^2N^2) = O(T^2N^2)$
- For SSP-tree: $O(TN^2 + TN^2 + TN^2 + T^2N^2) = O(T^2N^2)$
- For CP-tree: $O(TN^2 + TN^2 + TN^2 + T^2N^2) = O(T^2N^2)$
- For Improved FP-tree: $O(N \log N + TN^2 + TN^2) = O(TN^2)$

From the time complexity analysis of the tree construction algorithms, it can be observed that the proposed Improved FP-tree construction algorithm is much faster than the other tree construction algorithms.

Discussion

The requirement of two database scans and its dependency on the user-defined minimum support threshold makes the conventional FP-tree technically infeasible for incremental frequent pattern mining. The FP-tree maintains only those database information or items which satisfy the user-defined minimum support threshold value. The items having a frequency count greater than or equal to the minimum support threshold are called frequent items. So, FP-tree excludes the infrequent items, which do not meets the minimum support threshold. Later on, if the database gets updated, it must reconstruct a fresh FP-tree from scratch. The infrequent items excluded while constructing the FP-tree for the original database may become frequent after the database gets updated. Therefore, to solve these problems, researchers came up with a new concept called the complete tree, which uses only single database scans.

The main idea behind constructing a complete tree is maintaining all the database information without any information loss. A complete tree's most significant advantage is that it is not required to reconstruct the FP-tree from scratch even if the database gets updated or the support threshold changes. Whenever the database gets updated, the tree can be incrementally updated by performing some prefix path restructuring operations. It has been observed that the approaches use split, swap and merge operations to reflect the database update to the complete tree. Most of the incremental approaches perform the restructuring before inserting each transaction. Since restructuring is a very costly computation in terms of execution time, a few approaches construct the complete tree by periodically restructuring the tree data structure to minimize the computation cost. Figure 8 shows that though the incrementally restructuring approaches take only one database scan, they still require more time to construct the complete trees than the complete FP-tree. However, in cases of incremental scenario, from Fig. 9 and 10, it can be observed that the incremental approaches take comparatively less amount of time than FP-tree. Therefore, we initially experimented with analyzing the impact of database scans on tree construction. It has been observed that though FP-tree performs two database scans, but the total time required only to scan the database twice is very nominal as compared to complete tree construction time. Therefore, we have proposed and demonstrated an effective tree data structure (Improved FP-tree) construction algorithm in this manuscript.

Improved FP-tree is a complete tree constructed using two database scans. It is an improvement over the conventional FP-tree data structure; therefore, it is named Improved FP-tree. The main aim of our experiment is to minimize tree construction time. The Improved FP-tree construction algorithm has achieved a remarkable performance gain in terms of total tree construction time. It has been gained by intelligently maintaining the same item list as a stack instead of a simple linked list. The stack implementation of the same item list bypasses the whole list traversal and directly accesses the most recently inserted same item node in the tree. A new item can be directly added as the new top same item node since it removes the overhead of traversing the whole same item list every time a new node is inserted into the Improved FP-tree. It saves a significant amount of time. From Fig. 8, it can be observed that Improved FP-tree outperforms all the other three tree construction algorithms concerning complete tree construction. The Improved FP-tree construction algorithm constructs the tree from scratch using two database scans over the updated (original database + newly added transactions) database. Still, from the experimental results shown in Fig. 9 and 10, it can be observed that the performance of the Improved FP-tree construction algorithm is much prominent in the case of incremental scenario also. The Improved FP-tree outperforms conventional FP-tree, SSP-tree and CP-tree construction algorithms in terms of runtime in all cases of incremental updates for both sparse and dense databases. However, there may be a situation when the updated database size is huge, then reconstructing the Improved FP-tree from scratch may not be a suitable approach to handle the incremental scenario.

Conclusion

As mentioned above, the FP-tree-based incremental frequent pattern mining approaches perform frequent pattern mining by considering that the complete database to be processed can be accommodated in the systems main memory even after the database gets updated very frequently. Most of the existing algorithms use at least two database scans to construct the FP-tree. A few methods have been found, using only a single scan over the newly added transactions to restructure tree data structure to handle incremental databases. In this research work, we have proposed a two scan based tree data structure called Improved FP-tree. From the experimental results, it can be observed that though it is required to reconstruct the Improved FP-tree from scratch whenever the database gets updated, it still requires less time to construct the tree. The proposed Improved FP-tree construction algorithm outperforms all the FP-tree, SSP-tree and CP-tree construction algorithms in incremental scenarios for dense and sparse databases. Since all the approaches and the Improved FP-tree is main memory dependent and it is not always a suitable approach to use the restructuring approaches to construct the tree. Only if a few transactions are added to the database can the restructuring approaches be recommended. Nevertheless, if the number of newly added transaction is very high, then constructing the tree using our proposed Improved FP-tree construction algorithm will save a significant amount of time. The computational complexity analysis also shows that our proposed Improved FP-tree construction algorithm outperforms the other frequent pattern tree construction algorithms. The main limitation of the proposed Improved FP-tree is that it is main memory dependent. If the Improved FP-tree tree size exhausts the available main memory during construction, the algorithm will fail to construct the complete Improved FP-tree. To solve the main memory dependent problem of the proposed Improved FP-tree, shortly we will make an effort to develop an approach that can efficiently mine the frequent patterns from large scan databases even if the tree data structure cannot be accommodated in the computer's main memory.

Acknowledgment

The authors would like to acknowledge Tezpur University for supporting the whole research. Moreover, we would like to thank all friends and colleagues in the Department of Computer Science and Engineering, Tezpur University, for their supports. The authors would also like to acknowledge Maulana Azad National Fellowship (MANF), UGC, Govt. of India for the financial support to successfully conduct the research.

Author’s Contributions

Shafiul Alom Ahmed: Participated in all experiments, coordinated the data-analysis and contributed to the writing of the manuscript.

Bhabesh Nath: Designed the research plan, organized the study and participated in complexity and result-analysis.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References

Adnan, M., Alhajj, R., & Barker, K. (2006a, September). Alternative Method for Incrementally Constructing the FP-Tree. In 2006 3rd International IEEE Conference Intelligent Systems (pp. 494-499). IEEE. https://doi.org/10.1109/IS.2006.348469

Adnan, M., Alhajj, R., & Barker, K. (2006b, June). Constructing complete FP-tree for incremental mining of frequent patterns in dynamic databases. In International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (pp. 363-372). Springer, Berlin, Heidelberg. https://doi.org/10.1007/11779568_40

Agrawal, R., Imieliński, T., & Swami, A. (1993, June). Mining association rules between sets of items in large databases. In Proceedings of the 1993 ACM SIGMOD international conference on Management of data (pp. 207-216). https://doi.org/10.1145/170036.170072

Alhajj, R., & Barker, K. (2008). Alternative method for incrementally constructing the fp-tree. In Intelligent Techniques and Tools for Novel System Architectures (pp. 361-377). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-77623-9_21

Aumann, Y., Feldman, R., Lipshtat, O., & Manilla, H. (1999). Borders: An efficient algorithm for association generation in dynamic databases. Journal of Intelligent Information Systems, 12(1), 61-73. https://doi.org/10.1023/A:1026482903537

Ayad, A. M. (2000). A new algorithm for incremental mining of constrained association rules (Doctoral dissertation, Master Thesis, Department of Computer Sciences and Automatic Control, Alexandria University). http://pages.cs.wisc.edu/~ahmed/publications/Thesis.pdf

Ayan, N. F., Tansel, A. U., & Arkun, E. (1999, August). An efficient algorithm to update large itemsets with early pruning. In Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 287-291). https://doi.org/10.1145/312129.312252

Borah, A., & Nath, B. (2018). Identifying risk factors for adverse diseases using dynamic rare association rule mining. Expert systems with applications, 113, 233-263. https://doi.org/10.1016/j.eswa.2018.07.010

Cheung, D. W., Han, J., Ng, V. T., & Wong, C. Y. (1996, February). Maintenance of discovered association rules in large databases: An incremental updating technique. In Proceedings of the twelfth international conference on data engineering (pp. 106-114). IEEE. https://doi.org/10.1109/ICDE.1996.492094

Cheung, D. W., Lee, S. D., & Kao, B. (1997). A general incremental technique for maintaining discovered association rules. In Database Systems For Advanced Applications' 97 (pp. 185-194). https://doi.org/10.1142/9789812819536_0020

Cheung, W., & Zaiane, O. R. (2003, July). Incremental mining of frequent patterns without candidate generation or support constraint. In Seventh International Database Engineering and Applications Symposium, 2003. Proceedings. (pp. 111-116). IEEE. https://doi.org/10.1109/IDEAS.2003.1214917

Das, A., & Bhattacharyya, D. K. (2004, December). Rule mining for dynamic databases. In International Workshop on Distributed Computing (pp. 46-51). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30536-1_6

Ezeife, C. I., & Su, Y. (2002, May). Mining incremental association rules with generalized FP-tree. In Conference of the Canadian society for computational studies of intelligence (pp. 147-160). Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-47922-8_13

Ganti, V., Gehrke, J., & Ramakrishnan, R. (2001). Demon: Mining and monitoring evolving data. IEEE Transactions on Knowledge and Data Engineering, 13(1), 50-63. https://doi.org/10.1109/69.908980

Guirguis, S., Ahmed, K. M., El Makky, N. M., & Hafez, A. M. (2006, December). Mining the Future: Predicting Itemsets' Support of Association Rules Mining. In Sixth IEEE International Conference on Data Mining-Workshops (ICDMW'06) (pp. 474-478). IEEE. https://doi.org/10.1109/ICDMW.2006.116

Hamedanian, M., Nadimi, M., & Naderi, M. (2013). An efficient prefix tree for incremental frequent pattern mining. International Journal of Information and Communication Technology Research, 3(2), 49-55. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.457.3669&rep=rep1&type=pdf

Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Proceedings of ACMSIGMOD, Dallas, TX, pages 1–12. https://doi.org/10.1145/335191.335372

Hong, T. P., Lin, C. W., & Wu, Y. L. (2008). Incrementally fast updated frequent pattern trees. Expert Systems with Applications, 34(4), 2424-2435. https://doi.org/10.1016/j.eswa.2007.04.009

Huang, H., Wu, X., & Relue, R. (2002, December). Association analysis with one scan of databases. In 2002 IEEE International Conference on Data Mining, 2002. Proceedings. (pp. 629-632). IEEE. https://doi.org/10.1109/ICDM.2002.1184015

Koh, J. L., & Shieh, S. F. (2004, March). An efficient approach for maintaining association rules based on adjusting FP-tree structures. In International Conference on Database Systems for Advanced Applications (pp. 417-424). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-24571-1_38

Leung, C. K. S., Khan, Q. I., Li, Z., & Hoque, T. (2007). CanTree: a canonical-order tree for incremental frequent-pattern mining. Knowledge and Information Systems, 11(3), 287-311. https://doi.org/10.1007/s10115-006-0032-8

Leung, C. S., Khan, Q. I., & Hoque, T. (2005, November). CanTree: a tree structure for efficient incremental mining of frequent patterns. In Fifth IEEE International Conference on Data Mining (ICDM'05) (pp. 8-pp). IEEE. https://doi.org/10.1109/ICDM.2005.38

Lin, C. W., Hong, T. P., Lan, G. C., Wong, J. W., & Lin, W. Y. (2014). Incrementally mining high utility patterns based on pre-large concept. Applied intelligence, 40(2), 343-357. https://doi.org/10.1007/s10489-013-0467-z

Lin, C. W., Hong, T. P., & Lu, W. H. (2009). The Pre-FUFP algorithm for incremental mining. Expert Systems with Applications, 36(5), 9498-9505. https://doi.org/10.1016/j.eswa.2008.03.014

Lin, K. C., Liao, I. E., & Chen, Z. S. (2011). An improved frequent pattern growth method for mining association rules. Expert Systems with Applications, 38(5), 5154-5161. https://doi.org/10.1016/j.eswa.2010.10.047

Pyun, G., Yun, U., & Ryu, K. H. (2014). Efficient frequent pattern mining based on linear prefix tree. Knowledge-Based Systems, 55, 125-139. https://doi.org/10.1016/j.knosys.2013.10.013

Rácz, B. (2004, November). nonordfp: An FP-growth variation without rebuilding the FP-tree. In FIMI.

Roul, R. K., & Bansal, I. (2014, December). GM-Tree: An efficient frequent pattern mining technique for dynamic database. In 2014 9th International Conference on Industrial and Information Systems (ICIIS) (pp. 1-6). IEEE. https://doi.org/10.1109/ICIINFS.2014.7036626

Shahbazi, N., Soltani, R., Gryz, J., & An, A. (2016, July). Building fp-tree on the fly: Single-pass frequent itemset mining. In International Conference on Machine Learning and Data Mining in Pattern Recognition (pp. 387-400). Springer, Cham. https://doi.org/10.1007/978-3-319-41920-6_30

Tanbeer, S. K., Ahmed, C. F., Jeong, B. S., & Lee, Y. K. (2009). Efficient single-pass frequent pattern mining using a prefix-tree. Information Sciences, 179(5), 559-583. https://doi.org/10.1016/j.ins.2008.10.027

Yu-Dong, G., Sheng-Lin, L., Yong-Zhi, L., Zhao-Xia, W., & Li, Z. (2016). Large-scale dataset incremental association rules mining model and optimization algorithm. International Journal of Database Theory and Application, 9(4), 195-208. https://doi.org/10.14257/ijdta.2016.9.4.18

Yun, U., & Ryang, H. (2015). Incremental high utility pattern mining with static and dynamic databases. Applied intelligence, 42(2), 323-352. https://doi.org/10.1007/s10489-014-0601-6

Zhou, Z., & Ezeife, C. I. (2001, June). A low-scan incremental association rule maintenance method based on the apriori property. In Conference of the Canadian Society for Computational Studies of Intelligence (pp. 26-35). Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45153-6_3