4.1 Introduction

Frequent pattern mining has been the focus of great interest among data mining researchers and practitioners. It is today widely accepted to be one of the key problems in the data mining fields. Frequent pattern mining is to find the item sets that appear frequently from plenty of data, an item set is any subset of the set of all items. Frequent pattern mining is the discovery of relationships or correlations between items in a dataset. In the case of data streams, one may wish to find the frequent item sets either over a sliding window or the entire data stream.

**Frequent pattern problem:** The frequent pattern mining problem was first introduced by [4] as mining association rules between sets of items. Let \( I = \{ i_1, \ldots, i_m \} \) be a set of items. An itemset \( X \subseteq I \) is a subset of items. We write itemsets as \( X = i_{j1} \ldots i_{jn} \), i.e., omitting set brackets. Particularly, an itemset with \( l \) items is called an \( l \)-itemset. A transaction \( T = (tid, X) \) is a tuple where \( tid \) is a transaction-id and \( X \) is an itemset. A transaction \( T = (tid, X) \) is said to contain itemset \( Y \) if \( Y \subseteq X \). A transaction database \( DB \) is a set of transactions. The support of an itemset \( X \) in transaction database \( DB \), denoted as \( sup_{DB}(X) \) or \( sup(X) \), is the number of transactions in \( DB \) containing \( X \), i.e., \( sup(X) = |\{(tid, Y)| ((tid, Y) \in DB) \land (X \subseteq Y)\}| \)

**Problem statement:** Given a user-specified support threshold \( \text{min sup} \), \( X \) is called a frequent itemset or frequent pattern if \( sup(X) \geq \text{min sup} \). The problem of mining frequent itemsets is to find the complete set of frequent itemsets in a transaction database \( DB \) with respect to a given support threshold \( \text{min sup} \).

To study frequent pattern mining in data streams, we first examine the same problem in a transaction database. To justify whether a single item \( i_1 \) is frequent in a transaction database \( D \), one just need to scan the database once to count the number of transactions that \( i_1 \) appears. One can count every single item \( i_1 \) in one scan of \( D \). However, it is too costly to count every possible combination of single items (i.e., itemset \( I \) of any length) in \( D \) because there are a huge number of such combinations. An efficient alternative proposed is the Apriori algorithm to count only those itemsets whose every proper subset is frequent. That is, at the \( k \)'th scan of \( D \), derive its frequent itemset of length \( k \) (where \( k \geq 1 \)), and then derive the set of length \( (k+1) \) candidate itemset (i.e., whose every length \( k \) subset is frequent) for the next scan.

4.2 Frequent Itemset Generation

Here the objective is to find all the itemsets that satisfy the minsup threshold. These itemsets are called frequent itemsets. The computational requirements for frequent itemset generation are quite expensive. A lattice structure is used to enumerate the list of
all possible itemsets. In general, a data set that contains \( k \) items can potentially generate up to \( 2^k - 1 \) frequent itemsets, excluding the null set. Because \( k \) can be very large in many practical applications, the search space of itemsets that need to be explored is exponentially large. To reduce the computational complexity of frequent itemset generation, the Apriori principle can be used as described below.

### 4.3 Apriori

Apriori is the first association rule mining algorithm that pioneered the use of support-based pruning to systematically control the exponential growth of candidate itemsets. The use of support for pruning candidate itemsets is guided by the following principle.

**Apriori principle:** *If an itemset is frequent, then all of its subsets must also be frequent.*

To illustrate the idea behind this principle consider the following example. Suppose \{c, d, e\} is a frequent itemset then any transaction that contains \{c, d, e\} must also contain its subsets, \{c, d\}, \{c, e\}, \{d, e\}, \{c\}, \{d\} and \{e\}. As a result, if \{c, d, e\} is frequent, then all subsets of \{c, d, e\} must also be frequent.

Conversely, if an itemset such as \{a, b\} is infrequent then all its supersets must be infrequent too. As shown below, the subgraph containing the supersets of \{a, b\} can be pruned immediately once \{a, b\} is found infrequent. This strategy of trimming the exponential search space based on the support measure is known as support-based pruning. This pruning is based on property: Support for an itemset never exceeds the support for its subsets. This property is also known as the anti-monotone property of the support measure.

![Fig: 4.1 An itemset lattice](image-url)
**Frequent itemset generation:** Apriori is the first association rule mining algorithm that pioneered the use of support-based pruning to systematically control the exponential growth of candidate itemsets. Figure 4.2 provides a high-level illustration of the frequent itemset generation part of the Apriori algorithm for the transactions TDB shown in Table 4.1. Support threshold is assumed to be 60%, which is equivalent to a minimum support count equal to 3.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{A,B}</td>
</tr>
<tr>
<td>2</td>
<td>{A,C,D,E}</td>
</tr>
<tr>
<td>3</td>
<td>{B,C,D,F}</td>
</tr>
<tr>
<td>4</td>
<td>{A,B,C,D}</td>
</tr>
<tr>
<td>5</td>
<td>{A,B,C,F}</td>
</tr>
</tbody>
</table>

Table: 4.1

Initially, every item is considered as a candidate 1-itemset. After counting their supports, the candidate itemsets {F} and {E} are discarded because they appear in fewer than three transactions. In the next iteration, candidate-2 itemsets are generated using only the frequent 1-itemsets because the Apriori principle ensures that all supersets of the infrequent 1-itemsets must be infrequent. Because there are only four frequent 1-itemsets, the number of candidate 2-itemsets generated by the algorithm is 16. Two of these are six candidate 1-Itemsets:

<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Candidate 2-Itemsets</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>{D,A}</td>
<td>2</td>
</tr>
<tr>
<td>{D,C}</td>
<td>3</td>
</tr>
<tr>
<td>{D,B}</td>
<td>2</td>
</tr>
<tr>
<td>{A,C}</td>
<td>3</td>
</tr>
<tr>
<td>{A,B}</td>
<td>3</td>
</tr>
<tr>
<td>{C,B}</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Candidate 3-Itemsets</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>{A,C,B}</td>
<td>3</td>
</tr>
</tbody>
</table>
Fig: 4.2 : Illustration of frequent itemset generation using the Apriori algorithm

candidates, \{D, A\} and \{D, B\}, are subsequently found to be infrequent after computing
their support values. The remaining four candidates are frequent, and thus will be used to
generate candidate 3-itemsets. Without support-based pruning, there are 20 candidate
3-itemsets that can be formed using six items given in this example. With the Apriori
principle, we only need to keep candidate 3-itemsets whose subsets are frequent. The
only candidate that has this property is \{A,C,B\}.  
Apriori finds the complete set of frequent itemsets as follows.

1. Scan TDB once to find frequent items, i.e. items appearing in at least 3 transactions as
   the support factor is 3. They are D, A, C, B. Each of these 4 items forms a length-1
   frequent itemset. Let L_1 be the complete set of length-1 frequent itemsets.

2. The set of length-2 candidates, denoted as C_2, is generated from L_1. Here, we use the
   Apriori heuristic to prune the candidates. Only those candidates that consist of frequent
   subsets can be potentially frequent. An itemset xy \in C_2 if and only if x, y \in L_1. Thus,
   \begin{align*}
   C_2 &= \{DA, DB, DC, AC, AB, CB\} \\
   \text{There are 6 itemsets in } C_2.
   \end{align*}

3. Scan TDB once more to count the support of each itemset in C_2. The itemsets in C_2
   passing the support threshold form the length-2 frequent itemsets L_2. In this example,
   L_2 contains itemsets \{DC, AC, AB, CB\}.

4. Then, we form the set of length-3 candidates. Only those length-3 itemsets for which
   every length-2 sub-itemset is in L_2 are qualified as candidates. For example, \{A, C, B\}
   is a length-3 candidate since AC, AB and CB are all in L_2.

   One scan of TDB identifies the subset of length-3 candidates passing the support
   threshold and form the set L_3 of length-3 frequent itemsets. A similar process goes on
   until no candidate can be derived or no candidate is frequent.

Algorithm 4.1 : The Apriori algorithm is presented as follows.

1. \( k = 1 \).
2. \( F_k = \{i | i \in I \land \sigma(\{i\}) \geq N \times \text{minsup}\} \). \{Find all frequent 1-itemsets\}
3. \textbf{repeat}
4. \( k = k + 1 \)
5. \( C_k = \text{apriori-gen}(F_{k-1}) \). \{Generate candidate itemsets\}
6. \textbf{for} each transaction \( t \in T \) \textbf{do}
7. \( C_t = \text{subset}(C_k, t) \). \{Identify all candidates that belong to \( t \)\}
8. \textbf{for} each candidate itemset \( c \in C_t \) \textbf{do}
9. \( \sigma(c) = \sigma(c) + 1 \). \{Increment support count\}
10. end for
11. end for
12. \( F_k = \{ c \mid c \in C_k \land \sigma(c) \geq N \times \text{minsup} \} \). \{Extract the frequent k-itemsets\}
13. until \( F_k = \emptyset \)
14. Result = \( \bigcup F_k \)

- Even though Apriori can cut a lot of candidates, it could still be costly to handle a huge number of candidate itemsets in large transaction databases. For example, if there are 1 million items and only 1% (i.e. 104 items) are frequent length-1 itemsets, Apriori has to generate more than 107 length-2 candidates, test each of their support and save them for length-3 candidates generation.

- It is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is particularly true if a long pattern exists. Apriori is a level-by-level candidate-generation-and-test algorithm. To find a frequent itemset \( X = x_1 \ldots x_{100} \), Apriori has to scan the database 100 times.

- Apriori encounters difficulty in mining long patterns. For example, to find a frequent itemset \( X = x_1 \ldots x_{100} \), it has to generate-and-test \( 2^{100} - 1 \) candidates.

Based on the Apriori heuristic, two typical approaches for mining frequent patterns are proposed – the candidate generate-and-test approach and the pattern growth approach. Both approaches work in an iterative manner. They first generate all frequent 1-itemset. In each subsequent iteration of the candidate generate-and-test approach, pairs of k-itemsets are joined to form candidate \((k+1)\)-itemsets, then the database is scanned to verify the support of the entire candidate \((k+1)\)-itemsets, the set of resultant frequent \((k+1)\)-itemsets will be used as the input for next iteration. The drawbacks of the candidate generate-and-test algorithms are:

1. they all need generate lots of candidate itemsets, many of which are proved to be infrequent after scanning the database;
2. it needs scan database multiple times, in worst case, equal to the maximal length of the frequent patterns.

In contrast, the pattern growth approach avoids the cost of generating a large number of candidate itemsets by growing a frequent itemset from its prefix. It constructs a conditional database for each frequent itemset \( t \), denoted as \( D_t \), which is a collection of the projected transactions and each transaction contains only the candidate extensions of \( t \). All the patterns that have \( t \) as prefix can be mined only from \( D_t \). The key of the pattern growth approach is how to reduce the traversal and construction cost of a single conditional database, and reduce the total number of conditional databases constructed. However, it is always hard to reduce the traversal cost and the construction cost at the same time because the save of the construction cost usually invokes more traversal cost, and vice versa.
Apriori algorithm for mining frequent patterns over data stream cannot be used as; frequent itemset mining by Apriori is essentially a set of join operations as shown in [2]. However, join is a typical blocking operator [3] which cannot be performed over stream data since one can only observe at any moment a very limited size window of a data stream. Computation for any itemset cannot complete before seeing the past and future data sets. Since one can only maintain a limited size window due to the huge amount of stream data, it is difficult to mine and update frequent patterns in a dynamic, data stream environment.

Information from transaction data stream is essential for mining frequent patterns. Therefore, if we can extract the concise information for frequent pattern mining and store it into a compact structure, then it may facilitate frequent pattern mining. Motivated by this there exist compact data structure, called **FP-tree**, which is basically a prefix tree to store complete but no redundant information for frequent pattern mining.

### 4.4 Prefix Trees

A prefix tree is a data structure that provides a compact representation of transaction data set. Each node of the tree stores an item label and a count, with the count representing the number of transactions which contain all the items in the path from the root node to the current node. By ordering items in a transaction, a high degree of overlap is established. The compressed nature of this representation allows in-memory frequent pattern mining, because in most practical scenarios, this structure fits in main memory. Its design is based on the following observations:

- A transaction data set representation only needs to consist of frequent 1-items in the data set; the remaining items can be pruned away. This is a direct consequence of the anti-monotone property used in Apriori.
- If two transactions share a common prefix, as per some sorted order of the frequent items, they can be merged into one, provided a count value indicating this merge is registered. Furthermore, if frequent items in a transaction are sorted in descending order of their frequencies, there is a greater chance that more prefix strings will be shared.

With these observations, a prefix tree is constructed as follows:

1. Scan the data set to produce a list of frequent 1-items.
2. Sort the items in frequency descending order.
3. Sort the transactions based on the order from (2).
4. Prune frequent 1-items.
5. For each transaction, insert each of its items into a tree, in sequential order, generating new nodes when a node with the appropriate label is not found, and incrementing the count of existing nodes otherwise.

To compute the frequency count for an itemset, say \(ca\), using a prefix tree, proceed as follows: First, find each occurrence of item \(c\) in the tree using the node link pointers. Next, for each occurrence of \(c\), traverse the tree in a bottom up fashion in search of an occurrence of \(a\). The count for itemset \(ca\) is then the sum of counts for each node \(c\) in the tree that has \(a\) as an ancestor.

**Ascending frequency ordered prefix tree (AFOT)** [9] : It's a pattern growth algorithm which uses prefix-tree structure to organize the conditional databases. Contrary to the FP-
tree structure, AFOT structure adopts the top-down traversal strategy and the ascending frequency ordering method. The top-down traversal strategy is capable of minimizing the traversal cost of a conditional database; it does not need to maintain the parent links and node-links at each node. The ascending frequency order method is capable of minimizing the total number of conditional databases. One drawback of this ordering method is that it reduces node sharing among different transactions compared with the descending frequency ordering method. To alleviate this problem, arrays are used to store single branches. This representation leads to great space saving and also reduces the tree construction cost.

Prefix-tree is only a compact representation of the conditional databases, so it contains the complete information for mining frequent itemsets from the original database. The size of the prefix-tree is bounded by, but usually much smaller than, the total number of frequent item occurrences in the database. After the prefix-tree is constructed, the remaining mining is performed only on the prefix-tree. Algorithm first traverses the database to find the frequent items, and sort them in ascending order of their frequencies. Then the database is traversed the second time to construct a prefix-tree to represent the conditional databases of these frequent items. Only the frequent items are included in the prefix-tree. Each node in the prefix-tree contains three pieces of information: the item id, the support of the itemset corresponding to the path from the root to the node, and the pointers pointing to the node's children. Each entry of the arrays only keeps the first two pieces of information.

**Frequent-Pattern Tree (FP-tree):** FP-tree is a prefix tree that provides a compact representation of transaction data set. It is a compact data structure for efficient frequent-pattern mining. An FP-tree is a compressed representation of the input data. It is constructed by reading the data set one transaction at a time and mapping each transaction onto a path in the FP-tree. As different transactions can have several items in common, their path may overlap. The more the paths overlap with one another, the more compression can be achieved using FP-tree structure. Following example illustrates the concept in depth.

Example: Let T1 be the transaction database, as shown in Table 4.2 and the minimum support threshold be 3 (i.e., $min sup = 3$).

<table>
<thead>
<tr>
<th>Tid</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a,b}</td>
</tr>
<tr>
<td>2</td>
<td>{b,c,d}</td>
</tr>
<tr>
<td>3</td>
<td>{a,c,d,e}</td>
</tr>
<tr>
<td>4</td>
<td>{a,d,e}</td>
</tr>
<tr>
<td>5</td>
<td>{a,b,c}</td>
</tr>
<tr>
<td>6</td>
<td>{a,b,c,d}</td>
</tr>
<tr>
<td>7</td>
<td>{a}</td>
</tr>
<tr>
<td>8</td>
<td>{a,b,c}</td>
</tr>
<tr>
<td>9</td>
<td>{a,b,d}</td>
</tr>
<tr>
<td>10</td>
<td>{b,c,e}</td>
</tr>
</tbody>
</table>

Table 4.2: transaction database.
Transaction database T1 contains ten transactions and five items. The structure of the FP-tree after reading the first transaction will be as shown below.

(i) After reading Tid=1

(ii) After reading Tid=2

(iii) After reading Tid=3

(iv) After reading Tid=10

Fig 4.3 Construction of an FP-tree
With the above observations, one may construct a frequent-pattern tree as follows.

1. A scan of $T_1$ derives a list of frequent items, $h(a:8); (b:5); (c:3); (d:1)$; etc (the number after “:” indicates the support), in which items are ordered in frequency descending order. (In the case that two or more items have exactly same support count, they are sorted alphabetically.) This ordering is important since each path of a tree will follow this order.

2. Then, the root of a tree is created and labeled with “null”. The FP-tree is constructed as follows by scanning the transaction database $T_1$
   (a) The scan of the first transaction leads to the construction of the first branch of the tree: $h(a:1), (b:1)$. Notice that the frequent items in the transaction are listed according to the order in the list of frequent items.
   (b) For the third transaction, since its (ordered) frequent item list $a,c,d,e$ shares a common prefix ‘a’ with the existing path $a:b;$, the count of each node along the prefix is incremented by 1, and three new nodes $(c:1),(d:1),(e:1)$ is created and linked as a child of $(a:2)$
   (c) For the seventh transaction, since its frequent item list contains only one item i.e. ‘a’ shares only the node ‘a’ with the f-prefix subtree, a’s count is incremented by 1.
   (d) The above process is repeated for all the transactions.

To facilitate tree traversal, an item header table is built in which each item points to its first occurrence in the tree via a node-link. Nodes with the same item-name are linked in sequence via such node-links.

Based on this example, a frequent-pattern tree can be designed as follows.
1. It consists of one root labeled as “null”, a set of item-prefix subtrees as the children of the root, and a frequent-item-header table.
2. Each node in the item-prefix subtree consists of three fields: item-name, count, and node-link, where item-name registers which item this node represents, count registers the number of transactions represented by the portion of the path reaching this node, and node-link links to the next node in the FP-tree carrying the same item-name, or null if there is none.
3. Each entry in the frequent-item-header table consists of two fields, (1) item-name
and (2) head of node-link (a pointer pointing to the first node in the FP-tree carrying the item-name).

**Analysis.** The FP-tree construction takes exactly two scans of the transaction database:
1. The first scan collects the set of frequent items; and
2. The second constructs the FP-tree.

The cost of inserting a transaction \( t \) into the FP-tree is \( O(|\text{freq}(t)|) \), where \( \text{freq}(t) \) is the set of frequent items in transaction \( t \).

FP-tree is a prefix tree that provides a compact representation of transaction data set. Each node of the tree stores an item label and a count, with the count representing the number of transactions which contain all the items in the path from the root node to the current node. By ordering items in a transaction, a high degree of overlap is established. The compressed nature of this representation allows in-memory frequent pattern mining, because in most practical scenarios, this structure fits in main memory.

**Compactness of FP-tree:** To test the compactness of FP-trees, sizes of the following structures are compared.

- *Alphabetical* FP-tree. It includes the space of all the links. However, in such an FP-tree, the alphabetical orders of items are used instead of frequency descending order.
- *Ordered* FP-tree. Again, the size covers that of all links. In such an FP-tree, the items are sorted according to frequency descending order.
- *Transaction database.* Each item in a transaction is stored as an integer. It is simply the sum of occurrences of items in transactions.
- *Frequent transaction database.* That is the sub-database extracted from the original one by removing all infrequent items.

CATS tree[6]. (Compressed and Arranged Transaction Sequences tree) CATS Tree extends the idea of FPTree to improve storage compression and allow frequent pattern mining without generation of candidate itemsets. It allows frequent pattern mining with different supports without rebuilding the tree structure.

CATS Tree is a prefix tree and it contains all elements of FP-Tree including the header, the item links etc. Paths from the root to the leaves in CATS Tree represent sets of transactions. It contains all items in every transaction as compared to FP-tree which contains only frequent items.

All CATS Trees have the following properties:
1) The compactness of CATS Tree measures how many transactions are sharing a node. Compactness decreases as it is getting away from the root. This is the result of branches being arranged in descending order.
2) No item of the same kind could appear on the lower right hand side of another item. If there were items of the same kind on the right hand side, they should have been merged with the node on the left to increase compression. Any items on the lower right hand side can be switched to the same level as the item, split nodes as required if switching nodes violates the structure of CATS Tree. After that they can be merged with the item on the left. Because of the above properties, a vertical downward boundary is formed beside each node and a horizontal rightward boundary is formed.
at the top of each node. The vertical and horizontal boundaries combine to form a step-like individual boundary. Boundaries of multiple items can be joined together to form a refined boundary for a particular item. Items of the same kind can only exist on the refined boundary.

New transactions are added at the root level. At each level, items of the transaction are compared with those of children nodes. If the same items exist in both the new transaction and that of the children nodes, the transaction is merged with the node at the highest frequency. The frequency of the node is incremented. The remainder of the transaction is added to the merged nodes and the process is repeated recursively until all common items are found. Any remaining items of the transaction are added as a new branch to the last merged node.

Algorithm to built CATS tree has to consider not only the immediate items of that level, but also all possible descendants. The frequency of a descendant node can become larger than that of its ancestor, once the frequency of the new transaction is added. If the frequency becomes larger, the descendant has to swap in front of its previous ancestor to maintain the structural integrity of CATS Tree.

There are few properties of CATS Tree that can be used to prune the search space.
1) Inherited from FP-Tree, the sum of frequencies of all children nodes can only be smaller or equal to that of their parent.
2) Children of a node are sorted. Based on these properties, if a node cannot have local frequency greater than that of its parent, none of its sibling after it or any of its children can. As soon as an invalid node is found, it can abort the search and pursue other paths or insert the new transaction as a new branch. Since the frequency of the new transaction is 1, this implies that the frequency of descendant node must be equal to that of its ancestor. There can only be one node if we need to search downward. If the ordering of sibling node becomes out of order after merging, the offending node is repositioned to maintain the structural integrity of the tree.

The CATS tree extends the idea of the FP-tree to improve storage compression, and allows frequent-pattern mining without the generation of candidate itemsets. The aim is to build a CATS tree as compact as possible.

**Depth-First Search Algorithm for mining frequent patterns:** FP-growth is a prominent algorithm in the field of frequent pattern mining. On the one hand, it promotes the efficiency of dataset scanning by constructing FP-tree to compress dataset; on the other hand, it constructs conditional FP-sub-tree for every frequent prefix $\beta$ to reduce the search space. FP-growth has stronger selectivity than others, but the efficiency of selectivity is different for different dataset. As far as sparse dataset is concerned, the relationship between items is low, so the selectivity of FP-growth is efficient; as for dense dataset, the relationship between items is higher, so the selectivity of FP-growth algorithm is worse, and because of the long pattern, the recursion depth is deep, so on every layer of recursion the conditional FP-sub-tree would be constructed again and again, and finally much more place and time are consumed. By comparing the two cases with each other, [7] discovered that in the case of the latter, the depth-first search on FP-tree directly is more efficient than constructing recursive conditional FP-sub-tree for either place or time. To execute depth-first search on FP-tree directly, FP-tree* is defined, which is the modified version of FP-tree, to include tree node Node* ([L, U], NL, S, IN, A) and head node Header (IN, NL), in which IN stands for the current node number, S
refers to its frequency, NL refers to the next node of the tree with the same item number, and A stands for current active node with initial value of 0. The difference to FP-tree is that FP-tree* replaces the former parent-children pointer set with \([L, U]\), in which \(L\) stands for the node number of pre-order of the tree, \(U\) stands for the maximum number of the child tree. So by intersecting \([L,U]\) of two nodes, it can be shown whether they are on the same path directly. Another difference to FP-tree is that the node link of FP-tree* should be sorted by the value of \(L\).

Depth first search algorithm outperforms FP-growth when the data set is dense as FP-tree generated by dense dataset has narrower width and higher depth, so it needs to be constructed time after time, and the pruning of it is not efficient, consequently the overall efficiency is poor. However, on sparse dataset, the pruning of FP-tree is more efficient, since the depth of FP-tree is lower because of its shorter patterns, so the algorithm needs fewer recursions, but it is not fit for Depth first search algorithm. Both depth first search and FP-growth have their own advantage on different character of dataset, and both of them mine on the basis of FP-tree, in the manner of depth-first mining, and search pattern space by continually increasing current frequent prefix. So it’s easy to combine these two algorithms. Based on the theory above, [7] designed a Self-Adaptive FP Mining algorithm (SAFP). At first SAFP is the same as FP-growth, but replaces FP-growth with SAFP to mine. Virtually SAFP is an improved version of FP-growth algorithm: Before the next step mining, it judges the denseness of conditional FP-sub-tree by the variable \(C\): if it is bigger than certain threshold value, then SAFP calls NDFS to mine; otherwise it calls FP-growth to complete the remaining mining process. The mining algorithm is determined by corresponding distributing character of the dataset.

**CanTree:** [8] proposed tree structure, called CanTree (Canonical-order Tree) that captures the content of the transaction database and orders tree nodes according to some canonical order. By exploiting its properties, the CanTree can be easily maintained when database transactions are inserted, deleted, and/or modified. For example, the CanTree does not require adjustment, merging, and/or splitting of tree nodes during maintenance. No rescan of the entire updated database or reconstruction of a new tree is needed for incremental updating.

CanTree is designed for incremental mining. The construction of the CanTree only requires one database scan. This is different from the construction of an FP-tree where two database scans are required (one scan for obtaining item frequencies, and another one for arranging items in descending frequency order). In CanTree, items are arranged according to some canonical order, which can be determined by the user prior to the mining process or at runtime during the mining process. Specifically, items can be consistently arranged in lexicographic order or alphabetical order. Alternatively, items can be arranged according to some specific order depending on the item properties (e.g., their price values, their validity of some constraints). For example, items can be arranged according to prefix function order \(R\) or membership order \(M\) for constrained mining. With the canonical ordering of items, there are some nice properties, as described below.

**Property 1** The ordering of items is unaffected by the changes in frequency caused by incremental updates.

**Property 2** The frequency of a node in the CanTree is at least as high as the sum of
Features of CanTree:

(i) In CanTree, items are arranged according to some canonical order that is unaffected by the item frequency. Hence, searching for common items and mergeable paths during the tree construction is easy. No extra downward traversals are needed during the mining process.

(ii) The construction of CanTree is independent of the threshold values. Thus, it does not require such user thresholds as $\text{preMinsup}$.

(iii) Since items are consistently ordered in CanTree, any insertions, deletions, and/or modifications of transactions have no effect on the ordering of items in the tree. As a result, swapping of tree nodes—which may lead to merging and splitting of tree nodes—is not needed.

4.5 Mining Temporal Patterns in a Data Stream

In a temporal database, frequent patterns are usually targets of mining tasks. In many applications, a time-constraint is usually imposed during the mining process to meet the respective constraint. Specifically, the sliding window model is employed in this study, i.e., data expires after exactly $N$ time units after its arrival where $N$ is the user-specified window size. Consequently, a temporal pattern is frequent if its support, i.e., occurrence frequency, in the current window is no less than the threshold. There are several models of temporal patterns, including the inter-transaction association rule, the causality rule, the episode, and the sequential pattern.

In essence, the frequency counting process is similar to that of incremental mining [12] in that new arriving transactions are being dealt with while some obsolete transactions are discarded due to the sliding window constraint, and is, however, different from the latter in that approximate answers in the model of data stream are allowed as a trade-off of having only one data scan. As pointed out in [14], there are two major approaches to dealing with the frequency counting problem of data streams, i.e., one with a probabilistic error bound and the other with a deterministic error bound. Due to the limitation in processing data streams, both approaches are designed for obtaining approximate answers. The one with a probabilistic error bound is based on the sampling technique and that with a deterministic error bound is based on data segmentation technique. Note that although these approaches work successfully for counting supports of singleton items, as the number of items increases, the rapidly increasing number of temporal patterns can cause severe problems which include prohibitive storage and computing overheads. Explicitly, if the lossy counting scheme proposed in [14] is adopted, patterns with supports no less than $\epsilon$ are maintained during the mining process to guarantee the error range to be within $\epsilon$. However, since the threshold $\epsilon$, whose value could be one-tenth of $\text{min_sup}$, is usually too small to filter out uninteresting patterns, the storage space could be quite large. To address this point, in the sliding window model employed, only the occurrences of singleton items are being counted in the first time window. After the counting iteration, frequent items which have supports no less than the specified threshold are identified. These frequent items can be joined to generate candidate patterns of size two, which are then being counted in later iterations. After some patterns of size two are identified frequent, the candidate patterns of size three are
generated and counted subsequently. As a result, longer candidate patterns are gradually generated, counted and verified to be frequent during the counting iterations. From the downward closure property, it follows that only patterns whose subpatterns are all frequent are taken as candidates and to be counted subsequently.

A regression based algorithm was devised by [11], called algorithm FTP-DS (Frequent Temporal Patterns of Data Streams), to mine frequent temporal patterns for data streams. While providing a general framework of pattern frequency counting, algorithm FTP-DS has two major features, namely one data scan for online statistics collection and regression based compact pattern representation. To attain the feature of one data scan, the data segmentation and the pattern growth scenarios are explored for the frequency counting purpose. Algorithm FTP-DS scans online transaction flows and generates candidate frequent patterns in real time. The second important feature of algorithm FTP-DS is the regression-based compact pattern representation. Specifically, to meet the space constraint, they devised for pattern representation a compact ATF (standing for Accumulated Time and Frequency) form to aggregate all the information required for regression analysis. In addition, they developed the techniques of the segmentation tuning and segment relaxation to enhance the functions of FTP-DS. With these features, algorithm FTP-DS is able to not only conduct mining with variable time intervals but also perform trend detection effectively.

FTP-DS: FTP-DS (Frequent Temporal Patterns of Data Streams) to mine frequent temporal patterns for data streams. While providing a general framework of pattern frequency counting, algorithm FTP-DS has two major features, namely one data scan for online statistics collection and regression-based compact pattern representation, which are designed to address, respectively, the time and the space constraints in a data stream environment. To attain the feature of one data scan, the occurrence frequency of a temporal pattern is first defined in accordance with the time constraint of sliding windows. Specifically, the data segmentation and the pattern growth scenarios are explored for this frequency counting purpose. Algorithm FTP-DS then scans online transaction flows and generates candidate frequent patterns in real time. With the downward closure property, longer patterns are gradually formed from their subsets as time advances. As such, frequent patterns are incrementally discovered and recorded by only one database scan. Since a pattern is not deemed frequent until all its subsets are found frequent, the efficient frequent pattern identification with one data scan by FTP-DS is in fact at the cost of having some patterns recognized with delays due to the candidate pattern generation process. This phenomenon is referred to as delayed pattern recognition.

The second important feature of algorithm FTP-DS is on the regression-based compact pattern representation which is designed to address the space constraint of a data stream environment. To maintain the synopsis of frequency variations for frequent temporal patterns, the regression analysis is utilized. After being transformed into a time series, the data stream is segmented and represented by one or more segments. Each segment of a time series is identified through the regression process. This regression analysis is employed to capture the trends of frequent patterns. Specifically, to meet the space constraint, they have devised for pattern representation a compact ATF (Accumulated Time and Frequency) form to comprise all the information required for regression analysis. Using ATF forms to represent patterns, not only is the amount of
storage space significantly reduced but also the trends can be efficiently detected. Consequently, only required synopses rather than historical details of frequent patterns are maintained during discovering process.

**DSFP-tree:** In order to improve the mining efficiency of frequent patterns in data streams, [10] presented an algorithm DS-FPM for mining frequent patterns in data streams. First, a data structure DSFP-tree, a tree of frequent patterns in data streams (abbreviated as DSFP-tree), is constructed and the data stream is divided into a set of segments, then potential frequent itemsets on each segment are obtained, while the generated itemsets and the remaining itemsets of DSFP-tree generated by the earlier segment and sampled by fading factor are stored in new DSFP-tree, finally, the frequent patterns in the data stream can be rapidly found by a breadth-first search strategy.

In order to mine frequent pattern from the entire data stream and focus on mining frequent patterns from the most recent data stream, introduced a fading factor $\lambda$, DSFP-tree was constructed to preserve potential frequent itemsets in data stream. First, the algorithm DS-FPM, in which data streams are partitioned, samples from DSFP-tree generated by the earlier segment using fading factor $\lambda$ and gets potential frequent itemsets from the latest segment, then it inserts both of them into the new DSFP-tree, finally, it obtains frequent itemsets from DSFP-tree.

DSFP-tree is an extended prefix tree-based summary data structure. DSFP-tree is defined as follows:

1. DSFP-tree is composed of root (defaulted by null), prefix subtrees constituted by potential frequent itemsets and a potential frequent item list (abbreviated as head-list).
2. Each node excepted root in the DSFP-tree consists of seven fields: data, f, reval, pnode, par, leftchild, rightchild, where data is the item name which the node stands for, f registers the number of transactions represented by a portion of the path reaching the node with the data, reval is a counter which avoids duplicate itemsets’ insertion when DSFP-tree is restructured, pnode is a pointer which points to next node which has the same item name with this node, par is a pointer which points to the parent node, leftchild is a pointer which points to the first child node, rightchild is a pointer which points to the brother node.
3. Each entry in the head-list consists of two fields: item name and head-link, where head-link is a pointer which points to the first node with the item name of the DSFP-tree.

In order to improve the effect of mining frequent pattern in data streams, algorithm DS-FPM constructs a data structure DSFP-tree for storing potential frequent itemsets and introduces fading factor to control the memory used by algorithm in a certain size. The space efficiency of DS-FPM has been improved, and the result is not only from the entire data stream, but also focusing on the current transactions in the data stream. The data stream is divided into a set of segments, then potential frequent itemsets on each segment are obtained by an existing algorithm, while the generated itemsets and the DSFP-tree generated by the earlier segment are sampled by fading factor and the remaining itemsets are stored in new DSFP-tree, finally, the frequent itemsets in the data stream can be rapidly found by a breadth-first search strategy.

**Block Depth First Search:** (BDFS(b)) was proposed by [13] is an efficient technique for frequent pattern mining in real-Time business applications. Association rule mining in
real time is of increasing thrust in many business applications such as e-commerce, supply chain management, group decision support system etc. To show the details of their algorithm let us consider some market basket data to illustrate its working stepwise.

Let the following table 4.3 represent a set of 12 transactions, where the items are represented by a, b, c…

<table>
<thead>
<tr>
<th>Transaction id</th>
<th>items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b c d e</td>
</tr>
<tr>
<td>2</td>
<td>a c d e</td>
</tr>
<tr>
<td>3</td>
<td>a d e</td>
</tr>
<tr>
<td>4</td>
<td>b c d e</td>
</tr>
<tr>
<td>5</td>
<td>b d e</td>
</tr>
<tr>
<td>6</td>
<td>a b d</td>
</tr>
<tr>
<td>7</td>
<td>a b d</td>
</tr>
<tr>
<td>8</td>
<td>a b c d</td>
</tr>
<tr>
<td>9</td>
<td>d e</td>
</tr>
<tr>
<td>10</td>
<td>a c d e</td>
</tr>
<tr>
<td>11</td>
<td>a b c d e</td>
</tr>
<tr>
<td>12</td>
<td>a c e</td>
</tr>
</tbody>
</table>

Table- 4.3

**Step I.** Given this set of transactions D, create a two-dimensional lower triangular matrix M and the diff-set transaction_id lists as shown below.

<table>
<thead>
<tr>
<th>item</th>
<th>Transaction ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 2 3 6 7 8 10 11 12</td>
</tr>
<tr>
<td>B</td>
<td>1 4 5 6 7 8 11</td>
</tr>
<tr>
<td>C</td>
<td>1 2 4 8 10 11 12</td>
</tr>
<tr>
<td>D</td>
<td>1 2 3 4 5 6 7 8 9 10 11</td>
</tr>
<tr>
<td>E</td>
<td>1 2 3 4 5 10 11 12</td>
</tr>
</tbody>
</table>

Table 4.4 The tid-list of the items
This diff-set tid-list (table 4.4) contains the transaction numbers corresponding to which the particular item does not occur. The created matrix M is depicted in fig.4.6. This creating of the matrix M and the diff-set tid-list and storing in the hard-drive is a support independent step, which is a preprocessing step.

**Step II.** Let the absolute support be 3. Cells of Matrix M are visited to find F(1) and F(2) [where F(n) is frequent pattern of length n]. With the frequency being in parentheses, we have:

\[
F(1) = \{ a(9), b(7), c(7), d(11), e(9) \} \quad \ldots \quad (1)
\]

\[
F(2) = \{ ab(5), ac(6), ad(8), ae(6), bc(4), bd(7), be(4), cd(6), ce(6), de(8) \} \quad \ldots \ldots \quad (2)
\]

**Step III.** Two 2-length patterns are merged if their first elements match. Thus

\[
\text{Newly merged patterns} = \{ abc, abd, abe, acd, ace, ade, bcd, bce, bde, cde \} \quad \ldots \ldots \quad (3)
\]

**Step IV.** Find if all the subsets of new merged patterns are frequent. If all its 2-length subsets are not present, then the pattern is pruned (using the support monotonic property), else the pattern becomes a *candidate-pattern* and it is moved to the *global-pool* of candidate patterns \( C( ) \). The global-pool of candidate patterns is sorted on length and any tie between two same length patterns is resolved arbitrarily.

\[
C( ) = \{ abc, abd, abe, acd, ace, ade, bcd, bce, bde, cde \} \quad \ldots \ldots \quad (4)
\]

**Step V.** Let the block size \( b \) is 4. This means as the 3-length candidate patterns are pushed into the global pool, 4 of these patterns namely, \( abc, abd, abe \) and \( acd \), will be put in the next block \( b \).

**Step VI.** From the diff-sets of the two-length patterns, calculate the diff-sets of the three length patterns as follows: If \( d(ab) \) and \( d(ac) \) represents the diff-set of \( ab \) and \( ac \) respectively, then we can get \( d(abc) = d(ac) - d(ab) \) and the frequency of the pattern \( abc \) can be found from \( \text{freq}(abc) = \text{freq}(ab) - |d(abc)| \). Now check the frequency of these patterns by intersecting the diff-set tid-lists of the items.
\[ b = \{abc (3), abd (5), abe (2), acd(5)\} \ldots (5) \]

As frequency of \( abe \) is less than the support threshold, it gets pruned.
\[ F (3) = \{abc (3), abd (5), acd (5)\} \ldots (6) \]

**Step VII.** Now merge the newly found frequent patterns in \( F(3) \) and test these newly merged patterns generated for the presence of their immediate subsets.

*Newly merged patterns* = \{ \( abcd \) \} …. (7)

All immediate subsets of the pattern \( abcd \) are not present in \( F(3) \). Hence move the pattern \( abcd \) to border set of length 4, \( BS (4) \), with a sub-itemset counter of 3.

\[ BS (4) = \{ abcd (sub-itemset = 3) \} \ldots (8) \]

Patterns \( ace, ade, bcd, bce \) are taken in the next block \( b \) from the global-pool of candidate patterns.
\[ b=\{ace(5),ade(5),bcd(4),bce(3)\} \ldots (9) \]

All these items have frequency greater than \( \text{abs} = 3 \) and are hence frequent. Thus from the new block \( F(3)=\{ ace(5),ade(5),bcd(4),bce(3)\} \ldots (10) \)

For each pattern in the current \( F(3) \), search \( BS (4) \) to see if any of the immediate supersets are waiting in the border set. Pattern \( abcd \) is in \( BS (4) \) with sub-itemset counter = 3. Hence increase the sub-itemset counter of \( abcd \) and make it 4. The pattern \( abcd \) is of the highest length among the candidate patterns in the global-pool and is put in the next block \( b \). Merge newly found \( k \)-length frequent patterns with previously found \( k \)-length frequent patterns to make patterns of higher length.

*Newly merged patterns* (4) = \{ \( acde \), \( bcde \) \} ……(11)

The number of frequent immediate subsets of \( acde \) and \( bcde \) are 3 and 2 respectively. Hence they are moved to \( BS (4) \).

\[ BS (4) = \{ acde (sub-itemset = 3), bcde (subitemset= 2)\} \ldots (12) \]

The patterns \( abcd, bde \) and \( cde \) go to the current block \( b \). After intersecting the diff tid-list of these patterns,
\[ F (4) = \{abcd (3)\} \ldots \ldots (13) \]
\[ F (3) = \{bde (3),cde (5)\} \ldots \ldots (14) \]

Similarly search the \( BS (4) \) with newly found \( F(3) \) patterns and merge the patterns in the newly found \( F(3) \)'s with themselves and also with previous \( F(3) \)'s to generate higher length patterns. \( acde \) and \( bcde \) move from \( BS (4) \) to global pool of patterns and moves into the block \( b \). By intersecting the diff tid-lists of the items,
\[ F(4)=\{acde (4), bcde (3)\} \ldots \ldots (15) \]

As no higher length patterns can be generated and the number of patterns in block \( b \) becomes zero and also the number of candidate patterns in the global pool of candidate patterns becomes zero, the algorithm stops executing here. Thus, the set of all frequent patterns are:
\[ F(1) = \{ a(9), b(7), c(7), d(11), e(9)\} \]
\[ F(2) = \{ ab(5), ac(6), ad(8), ae(6), bc(4), bd(7), be(4), cd(6), ce(6), de(8)\} \]
\[ F(3) = \{ abc (3), abd (5), acd (5), ace(5) ,ade(5),bcd(4), bce(3), bde (3),cde (5)\} \]
\[ F(4) = \{ abcd (3), acde (4), bcde (3)\} \]

The block size \( b \) can now be varied to show how it affects the execution time of the algorithm. Whenever it is stopped before its natural completion, it outputs frequent patterns of various lengths it had obtained up to that point of execution time.

**FP-Stream:** The landmark model, mines frequent patterns in data streams by assuming that patterns are measured from the start of the stream up to the current moment. The
landmark model may not be desirable since the set of frequent patterns usually are time-sensitive and in many cases, changes of patterns and their trends are more interesting than patterns themselves. For example, a shopping transaction stream could start long time ago (e.g., a few years ago), and the model constructed by treating all the transactions, old or new, equally cannot be very useful at guiding the current business since some old items may have lost their attraction; fashion and seasonal products may change from time to time. Moreover, one may not only want to fade (e.g., reduce the weight of) old transactions but also to find changes or evolution of frequent patterns with time. Frequent patterns are maintained under a tilted-time window framework [1] in order to answer time-sensitive queries. The frequent patterns are compressed and stored using a tree structure similar to FP-tree and updated incrementally with incoming transactions. An FP-tree is used for storing transactions for the current time window; on the other hand, a similar tree structure, called pattern-tree, is used to store frequent patterns in the past windows. This time-sensitive stream mining model is called FP-stream, which includes two major components:

(1) a global frequent pattern-tree held in main memory, and
(2) tilted-time windows embedded in this pattern-tree.

The design of the tilted-time window is based on the fact that people are often interested in recent changes at a fine granularity, but long term changes at a coarse granularity. Fig- 4.4 shows such a tilted-time window: the most recent 4 quarters of an hour, then the last 24 hours, and 31 days. Based on this model, one can compute frequent itemsets in the last hour with the precision of quarter of an hour, the last day with the precision of hour, and so on, until the whole month. This model registers only 4+24+31 = 59 units of time, with an acceptable trade-off of lower granularity at a distant time.

For each tilted-time window, one can register window-based count for each frequent pattern. A compact tree representation of frequent pattern set, called pattern-tree is used each node in the frequent pattern tree represents a pattern (from root to this node) and its frequency is recorded in the node. This tree shares the similar structure with FP-tree. The difference is that it stores frequent patterns instead of streaming data. Usually frequent patterns do not change dramatically over time. Therefore, the tree structure for different tilted-time windows will likely have considerable overlap. To save the space, tilted –time window structure can be embedded into each node.
Here only one frequent pattern tree is used, where at each node; the frequency for each tilted-time window is maintained. Figure 4.5 shows an example of a frequent pattern tree with tilted-time windows embedded (FP-stream).

4.6 Dynamic FP-tree

In this section we introduce a new data structure which is constructed on the basis of FP tree, named Dynamic FP-tree. As compared to conventional FP-tree Dynamic FP-tree has following properties.

- Does not have a separate root node with null value
- Nodes at same level are linked to form a list.
- It is non-recursive FP-tree.
- Here as transactions are read, they are directly stored on the tree instead of counting the number of occurrences, sorting in descending order and then inserting into the FP tree. This saves execution time and also avoids use of another data structure to store the counts.
- Does not require prior knowledge of distinct items over data stream.
- Data structure is memory and time efficient as dynamic tree is constructed per window, patterns are mined and the tree is discarded by freeing the memory.

Dynamic FP tree is collection of linked list at each level. As soon a transaction is read a linked list of its item is added to the tree. Each node of tree consists of data, counter and two pointers next and down pointer. Next pointer points to the item at the same level of
other transaction, where as down pointer points to the next item in the same transaction. Following fig. 4.6 shows the dynamic FP tree after inserting transactions $t_1$, $t_2$, $t_3$ and $t_4$ in the order of arrival.

$t_1= \{a,b,c,d\}$  
$t_2= \{a,c,d,f,g\}$  
$t_3= \{b,f,h\}$  
$t_4= \{a,b,c,e,g\}$

Insertion to dynamic FP tree: As soon as first transaction is added first pointer will be made to point to its first node. This is the only pointer through which all the nodes of the tree can be accessed. As the next transaction is added its first item is matched with the existing first item, if matches then the counter of the node is increased and data is discarded. If the items are different then a new node is added at the same level and existing nodes next pointer is made to point it. If $t_1$, $t_2$, $t_3$ and $t_4$ are transactions of the database as incoming transactions then as soon as transaction $t_1$ is read a link list of its items is created with all the counters of its items, set as one. When the next transaction $t_2$ is read, it has common prefix $a$ so counter of $a$ is increased by one and data item $a$ is discarded next item is $c$ so $c$ is added at the next level by making the existing nodes next pointer to point it and then rest of the items are added to it using down pointer. Insertions are made in ascending order suppose at level 2 if the current nodes data is $d$ and the new
node to be added at that level has data $b$ than new node will be added in front of it and the
down pointer of upper level will manipulated accordingly to point to newly added node.
This tree is created for each time window and once the patterns are retrieved it is
discarded.

**Deletion and traversal:** Tree is a recursive data structure but its traversal is made in non
recursive way by using a stack. Once the nodes are traversed memory is freed and only
nodes having the counters more than or equal to minimum support threshold are stored as
frequent pattern. As the next time window starts all the memory is ready to be used so
using very less memory all the transactions in the batches can be processed efficiently.

An algorithm is proposed to which uses Dynamic FP-tree to mine frequent
patterns. Proposed algorithm works for both dense as well sparse streams of transactions.
It does not require knowing number of items per transaction .It works for fixed as well as
variable length transactions. Here a transaction stream, arriving as a time ordered series
of transitions is considered for analysis and is denoted as $D = \{ t_1, t_2, \ldots, t_n, \ldots \}$. Each
transaction $t_i$ contains a set of items $a_i$ and $a \in I$, where $I = \{ a_1, a_2, \ldots, a_3 \}$ is a set of
items or objects and $t_n$ is called the current transaction arriving on the stream. Here let
$TS_0$, $TS_1$, $\ldots, TS_{i-k+1}$, $\ldots$, $TS_i$ denote the time period or time slot which contains
multiple transactions arriving in that interval and thus they form a partition of
transactions in the stream. Given an integer $k$, the time based sliding window is defined
as the set of transactions arriving in the last $k$ time periods and denoted by $SW = \{ TS_i-
k+1, \ldots, TS_{i-1}, TS_i \}$. $TS_i$ is called the latest time slot and $TS_{i-k}$ is called the expiring one in
the sliding window. When the time shifts to the new slot $TS_i$ the effect of all transactions
in $TS_{i-k}$ will be eliminated from the mining model.

**Input:** an incoming batch of transactions arriving one at a time from the buffer and
min_support threshold $\sigma$

**Output:** Frequent patterns written to a secondary storage, and run time of the program.

**Algorithm 4.2 :**
1. Initialize the dynamic FP tree by assigning null value to first pointer.
2. Insert each incoming transaction as a link list without pruning any item.
   a. Initialize the first pointer with the first item of first transaction.
   b. If the first pointer is not null then if the prefix of the transaction matches
      with the existing prefix of the tree then increase the counters of the
      prefixes.
      else
      Add new link list of transaction at that level.
3. Traverse the tree by adding its data to stack and storing only those items to the
   file satisfying min_support threshold accepted by user. As the data is popped
   out of stack it is deleted by freeing the memory.
4.7 Performance Study and Experiments

All the experiments are performed on a Pentium PC machine with 2 GB main memory, running Microsoft Windows-XP. All the methods are implemented using Microsoft Visual C++. Program details are available in Appendix-D. As the tree is dynamic it does not need to know in advance number of items. As a new item comes a node is created and inserted. Once the node is traversed it is deleted by freeing the memory, so data structure used is memory efficient and can create patterns of any length. It is also time efficient as it uses only one data structure as compared to [1] which uses FP-tree as well FP stream. There is no need to create a flist as it is done in [1] items are directly inserted in to the tree no intermediate data structure required.

For experiments three datasets are downloaded from fimi.cs.helsinki.fi/data, Frequent Item set mining dataset repository, and are treated as transaction stream to test the data structure as well as algorithm.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#items</th>
<th># Avg. Length</th>
<th># Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess</td>
<td>75</td>
<td>37</td>
<td>3,196</td>
</tr>
<tr>
<td>Connect</td>
<td>129</td>
<td>43</td>
<td>67,557</td>
</tr>
<tr>
<td>Pumsb</td>
<td>2113</td>
<td>74</td>
<td>49,046</td>
</tr>
</tbody>
</table>

Table 4.5 Dataset Characteristics

The details of selected datasets are summarized in table 4.5. The frequent item set discovery programs have been tested on each dataset with varied minSup. Two user-specified parameters are accepted: a support threshold $s \in (0, 1)$, and an error parameter $e \in (0, 1)$ such that $e < s$. The error bound and the threshold parameter is used in determining which pattern to archive. A pattern is archived if the sum of its frequency is greater than or equal to difference between error and support threshold multiplied by the number of elements in the time window i.e. $(f \geq (s-e)N)$. Experiments are conducted using support values varying from 0.5 to 0.1 and error bound is kept 0.01 throughout. Window size i.e. number of transactions per window are kept 50. Following graph shows runtime of chess, connect and Pumsb datasets with minimum support 0.5 and error bound 0.01. As it is observed that run time increases if the distinct items are more in the database. Run time of chess dataset is two seconds as it has only 75 distinct items and that of connect is three seconds which has 129 different items and Pumsb has 2113 distinct items so its run time is four seconds.
Fig 4.10 Comparison of Chess, Connect and Pumsb transaction streams

Following graphs shows number of frequent patterns generated over time window t₁ to t₁₅₀ i.e. 150 * 50 = 7,500 transactions. In fig-4.11 we can observe number of patterns of length one at support 0.1 is zero as all the patterns generated are of length greater than one. In fig- 4.12 and fig-4.13, it is observed that, as the support factor decreases number of patterns generated increases.

Fig-4.11 number of output patterns of length one w.r.t. min_sup, Chess transaction stream

Fig –4.12 number of output patterns w.r.t. min_sup, Connect transaction stream
4.8 Conclusion

Results in the experiments reported above are obtained by executing the program on Windows XP platform which gives the run time in seconds. Frequent patterns are stored on a secondary storage with each line pertaining to one time window. Infrequent patterns are discarded by freeing the memory. Here pointers are used as main programming component requiring some time for allocation and freeing of memory. Algorithm is time efficient as the whole tree is not traversed only part of it, satisfying the min_support threshold is traversed and rest is discarded.

Above experiments prove that execution time is increased with increased number of distinct items. This is due to the data structure used, as depth of the tree increases with more distinct items. If the first few item of the transactions are different it will give rise to another link added to the tree which increase the breadth of the tree. Future work will be to merge (cluster) the branches which are almost similar thus resulting in reduced height as well as breadth of the tree and will make the data structure more efficient.
4.9 References


[8] Carson Kai-Sang Leung, Quamrul I. Khan, Tariqul Hoque CanTree: A Tree Structure for Efficient Incremental Mining of Frequent Patterns, Proceedings of the Fifth IEEE International Conference on Data Mining (ICDM’05)


[10] Jia-Dong Ren, Hui-Ling He, Chang-Zhen Hu, Li-Na Xu, Li-Bo Wang Mining Frequent Pattern Based On Fading Factor In Data Streams, Proceedings of the Eighth International Conference on Machine Learning and Cybernetics, Baoding, 12-15 July 2009

