MINING FREQUENT ITEMSET

3.1 Introduction

The frequent items problem is to process a stream of items and find all items occurring more than a given fraction of the time. It is one of the most heavily studied problems in data stream mining, dating back to the 1980s. Many applications rely directly or indirectly on finding the frequent items, and implementations are in use in large scale industrial systems. This abstract problem captures a wide variety of settings. The items can represent packets on the Internet, and the weights the size of the packets. Then the frequent items represent the most popular destinations, or the heaviest bandwidth users (depending on how the items are extracted from the flow identifiers). Or, the items can represent queries made to an Internet search engine, and the frequent items are now the (currently) popular terms. These are not simply hypothetical examples, but genuine cases where algorithms for this problem have been used by large corporations: AT&T [1] and Goggle [2] respectively. Given the size of the data (which is being generated at high speed), it is important to find algorithms which are capable of processing each new update very quickly, without blocking. It also helps if the working space of the algorithm is very small, so that the analysis can happen over many different groups in parallel, and because small structures are likely to have better cache behavior and hence further help increase the throughput.

Let’s assume we are given a set of items I. An itemset I belonging to S is some subset of items. A transaction is a couple T = (tid, I) where tid is the transaction identifier and I is an itemset. A transaction T = (tid, I) is said to support an itemset X, if X ⊆ I. A transaction database D is a set of transactions such that each transaction has a unique identifier. The cover of an itemset X in D consists of the set of transaction identifiers of transactions in D that support X: cover(X,D) := { tid | (tid, I) ∈ D, X ⊆ I }. The support of an itemset X in D is the number of transactions in the cover of X in D: support(X,D) = |cover(X,D)|. An itemset is called frequent in D if its support in D exceeds a given minimal support threshold σ.

In case of stream frequent itemset problem can be defined as: Given a stream S of n items t₁ . . . tₙ, the frequency of an item i is \( f_i = |\{j | t_j = i\}| \). The exact σ –frequent items comprise the set \( \{i | f_i > \sigma n\} \).

Example, The stream \( S = (a, b, a, c, c, a, b, d) \) has \( f_a = 3, f_b = 2, f_c = 2, f_d = 1 \).

For \( \sigma = 0.2 \), the frequent items are a, b and c.

The goal is now to find all frequent itemsets, given a datastream and a minimal support threshold. If frequent itemsets are long, it simply becomes infeasible to mine the set of all frequent itemsets. In order to tackle this problem, several solutions have been proposed that only generate a representing subset of all frequent itemsets. Among these, the collections of all closed or maximal itemsets are the most popular. A frequent itemset I is
called *closed* if it has no frequent superset with the same support, i.e., a frequent itemset is called *maximal* if it has no superset that is frequent. Obviously, the collection of maximal frequent itemsets is a subset of the collection of closed frequent itemsets which is a subset of the collection of all frequent itemsets. Although all maximal itemsets characterize all frequent itemsets, the support of all their subsets is not available, while this might be necessary for some applications such as association rules. On the other hand, the closed frequent itemsets form a lossless representation of all frequent itemsets since the support of those itemsets that are not closed is uniquely determined by the closed frequent itemsets.

It is important to note that the main difficulty in frequent set mining is the large number of itemsets whose frequencies need to be tracked. Given a transaction of size \( m \), the number of itemsets is exponentially proportional to its size; i.e., for a transaction of size \( m = 10 \), the number of itemsets may be up to \( 2^{10} - 1 \). This clearly makes the frequent set mining problem susceptible to overloading.

Obtaining efficient and scalable solutions to the frequent items problem is also important since many streaming applications need to find frequent items as a ‘subroutine’ of another, more complex computation. Most directly, mining frequent itemsets inherently builds on finding frequent items as a basic building block. Finding the entropy of a stream requires learning the most frequent items in order to directly compute their contribution to the entropy, and remove their contribution before approximating the entropy of the residual stream [3].

On the other hand, recent work on mining frequent itemsets over data streams can be classified into three models. The first model is the *landmark* model where frequent sets are discovered between a particular point of time (called landmark) and the current time. Lossy Counting [4] and FDPM [5] are typical algorithms in this group. The second model is the *time-fading* model where transactions are weighted on the time they arrive. This model gives more attention (fine granularity) to the recently arrived data and relaxes for the earlier ones (coarse granularity). Works that focus on this model include the estDec [6] and FP-Streaming [7] algorithms. The third model is the *sliding-window* model. Compared to the two previous ones, this model further considers the elimination of transactions. Frequent itemsets are found within a fixed portion of the stream which is pre-specified by a period of time or a number of transactions. FTP-DS (Frequent Temporal Patterns of Data Streams) [8], estWin [9], and Time-sensitive sliding window [10] are algorithms focusing on this model. A core idea underlying all these works is that they focus on designing methods that can efficiently summarize within limits of main memory.

### 3.2 Frequent Items Algorithms

Algorithms for finding the frequent items are divided into four classes [11].

- **Quantile Algorithms**: GK Algorithm, QDigest.
- **Sketch algorithms**: Count-Min Sketch, Count Sketch
- **Tree based algorithms**
- **Counter-based algorithms**: Frequent, LossyCounting, SpaceSaving

Most popular algorithms are Tree based algorithms, Counter-based algorithms and Sketch algorithms. Counter-based algorithms track a subset of items from the inputs, and
monitor counts associated with these items. For each new arrival, the algorithms decide whether to store this item or not, and if so, what counts to associate with it. Sketch algorithms, are (randomized) linear projections of the input viewed as a vector, and solve the frequency estimation problem. They therefore do not explicitly store items from the input.

A common feature of several algorithms is that when given a new item, they test whether it is one of k being stored by the algorithm, and if so, increment its count. The cost of supporting this operation depends a lot on the model of computation assumed. A simple solution is to use a hash table storing the current set of items, but this means that an otherwise deterministic solution becomes randomized in its time cost, since it takes expected $O(1)$ operations to perform this step.

### 3.3 Sketch algorithms

Sketch algorithms compute a summary that is a linear transform of the frequency vector, so departures are naturally handled by sketch algorithms. Sketches solve core problem of estimating item frequencies and hence can be used to find frequent items. Here, the term ‘sketch’ is used to denote a data structure which can be thought of as a linear projection of the input. That is, if we imagine the stream as implicitly defining a vector whose i'th entry is $f_i$, the sketch is the product of this vector with a matrix. For the algorithm to use small space, this matrix will be implicitly defined by a small number of bits. The algorithms use hash functions to define the linear projection. Algorithms are defined in terms of using hash functions are used to map items to array entries. The sketch algorithms solve the frequency estimation problem, and so need additional data information to solve the frequent items problem. Two sketching approaches are discussed below.

**CountSketch algorithm:** Count Sketch proposed in [12] uses extra hash functions to estimate frequent frequencies. Algorithm is as below:

Let $t$ and $b$ be parameters with values to be determined later.

Let $h_1, . . . , h_t$ be hash functions from objects to $\{1, . . . , b\}$ and $s_1, . . . , s_t$ be hash functions from objects to $\{+1,-1\}$. The CountSketch data structure consists of these hash functions along with a $t \times b$ array of counters, which should be interpreted as an array of $t$ hash tables, each containing $b$ buckets. The data structure supports two operations:

- **Add(C, q):** For $i \in [1, t]$, $h_i[q] += s_i[q]$.
- **Estimate(C, q):** return $\text{median}_i \{h_i[q] \cdot s_i[q]\}$.

Using the above data structure algorithm is simple to implement. For each element, CountSketch data structure is used to estimate its count, and keep a heap of the top $k$ elements seen so far. More formally:

- Given a data stream $q_1, . . . , q_n$, for each $j = 1, . . . , n$:
  1. Add(C, $q_j$)
  2. If $q_j$ is in the heap, increment its count. Else, add $q_j$ to the heap if Estimate(C, $q_j$) is greater than the smallest estimated count in the heap. In this case, the smallest estimated count should be evicted from the heap.

**Pseudocode for CountSketch Algorithm:**
Algorithm 3.1 : COUNTSKETCH (w,d)

1. \( C[1,1] \ldots C[d,w] = 0; \)
2. \( \text{for } j = 1 \text{ to } d \)
3. \( \quad \text{do Initialize } g_j, h_j; \)
4. \( \quad \text{for each } i : \)
5. \( \quad\quad n = n + 1; \)
6. \( \quad\quad \text{for } j = 1 \text{ to } d \)
7. \( \quad\quad \quad \text{do } C[j, g_j(i)] = C[j, g_j(i)] + h_j(i); \)
8. \( \quad\quad\text{endfor} \)

The COUNTSKETCH algorithm dramatically improves the speed by showing that the same underlying technique works if each update only affects a small subset of the sketch, instead of the entire summary. The sketch consists of a \( d \times w \) array \( C \) of counters, and two hash functions for each of the \( d \) rows, \( g_j \) which maps input items onto \([w]\), and \( h \) which maps input items onto \([-1, +1]\). Each input item \( i \) causes \( h_j(i) \) to be added on to entry \( C[j, g_j(i)] \) in row \( j \), for \( 1 \leq j \leq d \). The estimate is \( \text{median}_{1 \leq j \leq d} h_j(i) C[j, g_j(i)] \). The estimate derived for each value of \( j \) can be shown to be correct in expectation and has variance depending on \( F^2/w \).

CountMin Sketch. The COUNTMIN sketch algorithm \([18]\) can be described in similar terms to COUNTSKETCH. As before, an array of \( d \times w \) counters is maintained and pairwise independent hash functions \( g_j \) map items onto \([w]\) for each row. Each update is mapped onto \( d \) entries in the array, each of which is incremented. Now \( \hat{\lambda} = \min_{1 \leq j \leq d} C[j, g_j(i)] \). The Markov inequality is used to show that the estimate for each \( j \) overestimates by less than \( n/w \), and repeating \( d \) times reduces the probability of error exponentially. So setting \( d = \log(1/\delta) \) and \( w = O(1/\epsilon) \) ensures that \( \hat{\lambda} \) has error at most \( \epsilon n \) with probability at least \( 1 - \delta \).

Algorithm 3.2 : COUNTMIN(w,d)

1. \( C[1,1] \ldots C[d,w] = 0; \)
2. \( \text{for } j = 1 \text{ to } d \)
3. \( \quad \text{do Initialize } g_j; \)
4. \( \quad \text{for each } i : \)
5. \( \quad\quad n = n + 1; \)
6. \( \quad\quad \text{for } j = 1 \text{ to } d \)
7. \( \quad\quad\quad \text{do } C[j, g_j(i)] = C[j, g_j(i)] + 1; \)
8. \( \quad\quad\text{endfor} \)
Algorithm can be further explained using simple description as given below:

- Model input stream as a vector $x$ of dimension $U$
  - $x[i]$ is frequency of item $i$
- Creates a small summary as an array of $w \times d$ in size
- Use $d$ hash function to map vector entries to $[1..w]$

![Array: CM[i,j]](image)

- Each entry in vector $x$ is mapped to one bucket per row.
- Estimate $x[j]$ by taking $\min_k CM[k,h_k(j)]$

**Frequent Items using Group Testing.** The idea of “group testing” in this context [13] randomly divides the input into buckets so that we expect at most one frequent item in each group. Within each bucket, the items are divided into groups so that the “weight” of each group indicates the identity of the frequent item. This can be seen as an extension of the Count-Min sketch, since the structure resembles the buckets of the sketch, with additional information on subgroups of each bucket (based on the binary representation of items falling in the bucket); further, the analysis and properties are quite close to those of a Hierarchical Count-Min sketch. For each bucket, we keep additional counts for the total frequency of all items whose binary representation has the $i^{th}$ bit set to 1. This increases the space to $O(1/\epsilon \log U \log \delta)$ when the binary representation takes $\log U$ bits. Each update requires $O(\log 1/\epsilon)$ hashes as before, and updating $O(\log U)$ counters per hash.

**hCount Algorithm:** hCount algorithm for dynamically maintaining items over a data stream was proposed by [26]. They assumed that a sequence of transactions flows through the data stream. Here a transaction is either inserting or deleting an item $k$ at time point $i$, denoted $t_i = {\text{delete}(k)}$ or $t_i = {\text{insert}(k)}$. They assumed items as integers in the range of $[1..M]$. Let $n_k$ denote the net occurrence of the item $k$. Operation $\text{insert}(k)$ increases the net occurrence of $k$, i.e., $n_k = n_k + 1$, and operation $\text{delete}(k)$ decreases the net occurrence of $k$, i.e., $n_k = n_k - 1$. Let $N$ denote the sum of net occurrence of all items. The frequency of any item $k$ can be denoted as $f_k$, where $f_k = n_k/N$. Given three user specified parameters: a support parameter $s \in (0, 1)$, an error parameter $\epsilon \in (0, 1)$ such that $\epsilon << s$ and a probability parameter $\rho$ such that $\rho$ is near 1. At any point of time, with a small bounded memory, output a list of items along with their estimated frequencies.

They proposed an algorithm, called hCount, to find a list of most frequent items over a data stream.

A hash table, $S[m][h]$, along with $h$ hash functions, is used. Each of these $h$ hash functions maps a digit from $[0..M-1]$ to $[0..m-1]$ uniformly and independently. Hash function chosen here is as follows: $H_i(k) = ((a_i \cdot k + b_i) \mod P) \mod m$, $1 \leq i \leq h$, where $a_i$ and $b_i$ are two random numbers and $P$ is a large prime number. An item $k$ in the range has a set of associated counters: $\langle S[H_1(k)][1], S[H_2(k)][2],..., S[H_h(k)][h] \rangle$. These
associated counters increase or decrease at the same time when encountering a transaction on item $k$.

Two algorithms are proposed for handling tuples over stream and outputting final result separately. The algorithm $h\text{Count}$ maintains such a hash table and the algorithm $e\text{Freq}$ checks and outputs the items with frequency above a user-specified threshold $s$ along with their estimated frequencies. Example given in [26], assume there is a data stream, whose items are within a range of $[1...16]$. Here, in order to output the most frequent items over the data stream, a hash table $S[m][h]$ is created where $m = 5$ and $h = 4$. As per the hash function specified above, the four hash functions, $H1$, $H2$, $H3$ and $H4$, can be determined by using the following four pairs of $(a_i, b_i)$: $(a_1, b_1) = (7; 13)$, $(a_2, b_2) = (22, 6)$, $(a_3, b_3) = (24, 11)$ and $(a_4, b_4) = (14, 27)$. Suppose the prime number used is $P = 31$.

**Algorithm 3.3 : $h\text{Count}(k, \text{ttype})$**

1. **if** $\text{ttype}$ is insert **then**
2. \hspace{1em} $N = N + 1$;
3. **else**
4. \hspace{1em} $N = N - 1$;
5. **end if**
6. **for** $j = 1$ to $h$ **do**
7. \hspace{1em} \hspace{1em} $pos = ((a_j \cdot k + b_j) \mod P) \mod m$;
8. \hspace{1em} **if** $\text{ttype}$ is insert **then**
9. \hspace{1em} \hspace{1em} $S[pos][j] = S[pos][j] + 1$;
10. \hspace{1em} **else**
11. \hspace{1em} \hspace{1em} $S[pos][j] = S[pos][j] - 1$;
12. **end if**
13. **end for**

**Algorithm 3.4 : $e\text{Freq}(s)$**

1. **for** $k = 1$ to $M$ **do**
2. \hspace{1em} $c = \min_{1 \leq j \leq h} (S[H_j(k)][j])$
3. \hspace{1em} **if** $c < sN$ **then** output$(k, c/N)$;
4. **end for**

Initially, all the counters of $S[m][h]$ are initialized to zero. Assume a data stream of 38 transactions is coming as indicated, where $t_i$ indicates the $i$'th transaction and $k$ indicates the item handled by the corresponding transaction. Here, if $k$ is a positive number, then the corresponding transaction is an insertion transaction. Otherwise, it is a deletion
At time point 4, there are two transactions: \( t_5 = \text{insert} (9) \) and \( t_6 = \text{delete} (6) \). Accordingly, call \( h\text{Count}(9, \text{insert}) \) and \( h\text{Count}(6, \text{delete}) \). The four associated counters are increased by 1 (for insert) and the four associated counters are decreased by 1 (for delete). Based on the analysis, it was found that every counter contains an error, which is the sum of occurrences of other items mapped to the same counter. Algorithm is improved to remove these errors and improved version of algorithm is known as \( h\text{Count}^* \). Here the error factor is computed as the average of the estimated occurrences for all items in \([M+1...M+\Delta]\).

### 3.4 Tree-based Algorithms

**Mining recent Frequent Itemsets:** Generally, knowledge embedded in a data stream is more likely to be changed as time goes by. Identifying the recent change of a data stream quickly, specially for an online data stream, can provide valuable information for the analysis of the data stream. In addition, monitoring the continuous variation of a data stream enables to find the gradual change of embedded knowledge, so that it can be timely utilized. In order to achieve this, the effect of obsolete information in old transactions on the current mining result of a data stream should be eliminated effectively. As a simple solution, it is possible to consider a sliding window approach. It restricts the target transactions of data mining to those transactions that are generated within the most recent period of a fixed-sized window. However, its current mining result totally depends on recently generated transactions in the range of the window. Due to this reason, this approach is a primitive way of disregarding obsolete information. In addition, all the transactions in the window need to be maintained in order to remove their effects on the current mining result when they are out of the range of a sliding window.

In terms of information differentiation, the \( SWF \) algorithm [14] uses a sliding window to find frequent itemsets in the fixed number of recent transactions. The sliding window is composed of a sequence of partitions. Each partition maintains a number of transactions. The candidate 2-itemsets of all transactions in the window are maintained separately. When the window is advanced, the oldest partition is disregarded and a new partition containing newly generated transactions is appended to the window. At the same time, the candidate 2-itemsets of the advanced window are adjusted. Subsequently, all possible candidate itemsets are generated by these candidate 2-itemsets. The new set of frequent itemsets is identified by scanning the transactions of the sliding window. A more flexible way of information differentiation is presented in [15] where correlations among co-evolving time sequences are analyzed. The missing values of the sequences are estimated and their future values are predicted. In order to identify the recent change of correlations adaptively, a forgetting factor is used to diminish the effect of old correlations among sequences. A forgetting factor determines how fast the effect of old information is faded away. This type of an information decay model is also introduced in NIDES [16] for anomaly intrusion detection. NIDES models the historical behavior of a user’s activities in terms of various measures and generates a long-term profile containing a statistical summary for each measure. In order to concentrate on the recent behavior of the user, the statistics of old activities in the long-term profile are decayed as new activities are performed by the user. [6] Examines each transaction in a data stream one-by-one without any candidate generation. The occurrence count of a significant itemset that appears in each transaction is maintained by a prefix-tree lattice structure in main
memory. The effect of old transactions on the current mining result is diminished by decaying the old occurrence count of each itemset as time goes by. In addition, the rate of decay old information is flexibly defined as needed. The total number of significant itemsets in main memory is minimized by delayed-insertion and pruning operations of an itemset. As a result, its processing time is flexibly controlled while sacrificing its accuracy. A method of finding recent frequent itemsets adaptively over an online data stream is based on the decay mechanism. The different combinations of items that appear in each transaction are maintained in a prefix-tree lattice structure as a monitoring lattice. A node in a monitoring lattice contains an item and it denotes an itemset composed of items that are in the nodes of its path from the root. Method proposed by [6] is estDec Method. According to this method not all of itemsets that appear in a data stream are significant for finding frequent itemsets. An itemset which has much less support than a predefined minimum support is not necessarily monitored since it cannot be a frequent itemset in the near future. Therefore, the insertion of a new itemset can be delayed until it can possibly be a frequent itemset in the near future. When the estimated support of a new itemset is large enough, it is regarded as a significant itemset and it is inserted to a monitoring lattice. On the other hand, an efficient pruning technique is obviously another way of reducing the usage of memory space. Although an itemset in a monitoring lattice was significant enough to be monitored in the past, if its current support becomes much less than a predefined minimum support, it can be eliminated from the monitoring lattice.

**FP-growth method:** The FP-growth method by [17] uses a data structure called the FP-tree (Frequent Pattern tree) and an AFOPT algorithm. The FP-tree is a compact representation of all relevant frequency information in a database. Every branch of the FP-tree represents a frequent itemset, and the nodes along the branches are stored in decreasing order of frequency of the corresponding items, with leaves representing the least frequent items. Compression is achieved by building the tree in such a way that overlapping itemsets share prefixes of the corresponding branches. The FP-tree has a header table associated with it. Single items and their counts are stored in the header table in decreasing order of their frequency. The entry for an item also contains the head of a list that links all the corresponding nodes of the FP-tree. Compared with Apriori and its variants which need several database scans, the FP-growth method only needs two database scans when mining all frequent itemsets. The first scan counts the number of occurrences of each item. The second scan constructs the initial FP-tree which contains all frequency information of the original dataset. Mining the database then becomes mining the FP-tree.

To construct the FP-tree, first find all frequent items by an initial scan of the database. Then insert these items in the header table, in decreasing order of their count. In the next (and last) scan, as each transaction is scanned, the set of frequent items in it are inserted into the FP-tree as a branch. If an itemset shares a prefix with an itemset already in the tree, the new itemset will share a prefix of the branch representing that itemset. In addition, a counter is associated with each node in the tree. The counter stores the number of transactions containing the itemset represented by the path from the root to the node in question. This counter is updated during the second scan, when a transaction causes the insertion of a new branch. Given below is an example of a database and Figure 3.8 the FP-tree for that database.
Note that there may be more than one node corresponding to an item in the FP-tree. The frequency of any one item $i$ is the sum of the count associated with all nodes representing $i$, and the frequency of an itemset equals the sum of the counts of the least frequent item in it, restricted to those branches that contain the itemset. For instance, from Figure 3.8 we can see that the frequency of the itemset $\{c, a, g\}$ is 5.
Thus the constructed FP-tree contains all frequency information of the database. Mining the database becomes mining the FP-tree. The FP-growth method relies on the following principle: if X and Y are two itemsets, the count of itemset XUY in the database is exactly that of Y in the restriction of the database to those transactions containing X. This restriction of the database is called the *conditional pattern base* of X, and the FP-tree constructed from the conditional pattern base is called X’s *conditional FPtree*, which is denoted by $T_X$. We can view the FP-tree constructed from the initial database as $T_{Ø}$, the conditional FP-tree for Ø. Note that for any itemset Y that is frequent in the conditional pattern base of X, the set XUY is a frequent itemset for the original database.

Given an item i in the header table of an FP-tree $T_X$, by following the linked list starting at i in the header table of $T_X$, all branches that contain item i are visited. These branches form the conditional pattern base of $X\cup\{i\}$, so the traversal obtains all frequent items in this conditional pattern base. The FP-growth method then constructs the conditional FP-tree $T_{XU\{i\}}$, by first initializing its header table based on the found frequent items, and then visiting the branches of $T_X$ along the linked list of i one more time and inserting the corresponding itemsets in $T_{XU\{i\}}$. Note that the order of items can be different in $T_X$ and $T_{XU\{i\}}$. The above procedure is applied recursively, and it stops when the resulting new FP-tree contains only one single path. The complete set of frequent itemsets is generated from all single-path FP-trees.

**An array technique:** The main work done in the FP-growth method is traversing FP-trees and constructing new conditional FP-trees after the first FP-tree is constructed from the original database.

It is found that [18] about 80% of the CPU time was used for traversing FP-trees. Traversal time can be reduced by using a simple data structure array. Recall that for each item i in the header of a conditional FP-tree $T_X$, two traversals of $T_X$ are needed for constructing the new conditional FP-tree $T_{XU\{i\}}$. The first traversal finds all frequent items in the conditional pattern base of $XU\{i\}$, and initializes the FP-tree $T_{XU\{i\}}$ by constructing its header table. The second traversal constructs the new tree $T_{XU\{i\}}$. The first scan of $T_X$ can be omitted by constructing an array $A_X$ while building $T_X$. The following example will explain the idea. In Figure 3.3 (a), supposing that the minimum support is 20%, after the first scan of the original database, we sort the frequent items as e:8, c:8, a:8, g:5, b:2, f:2, d:2. This order is also the order of items in the header table of $T_{Ø}$. During the second scan of the database construct $T_{Ø}$ and an array $A_{Ø}$. This array will store the counts of all 2-itemsets. All cells in the array are initialized as 0.

```
   6
 a 6 8
 g 4 5 5
 b 2 2 2 0
 f 2 2 2 0 2
 d 1 2 2 1 0 0
```

(a) $A_{Ø}$

```
   5
 c 4 4
 e a c
```

(b) $A_{Ø}$
Fig- 3.3 Two array examples

In $A_\emptyset$, each cell is a counter of a 2-itemset, cell $A_\emptyset[d, e]$ is the counter for itemset $\{d, e\}$, cell $A_\emptyset[d, c]$ is the counter for itemset $\{d, c\}$, and so forth. During the second scan for constructing $T_\emptyset$, for each transaction, first all frequent items in the transaction are extracted. Suppose these items form itemset $I$. To insert $I$ into $T_\emptyset$, the items in $I$ are sorted according to the order in header table of $T_\emptyset$. When we insert $I$ into $T_\emptyset$ at the same time $A_\emptyset[i, j]$ is incremented by 1 if $\{i, j\}$ is contained in $I$. For example, for the first transaction, $\{a, b, c, e, f\}$ is extracted (item o is infrequent) and sorted as $e, c, a, b, f$. This itemset is inserted into $T_\emptyset$ as usual, and at the same time, $A_\emptyset[f, e], A_\emptyset[f, c], A_\emptyset[f, a], A_\emptyset[f, b], A_\emptyset[b, a], A_\emptyset[b, c], A_\emptyset[b, e], A_\emptyset[a, e], A_\emptyset[a, c], A_\emptyset[c, e]$ are all incremented by 1. After the second scan, array $A_\emptyset$ keeps the counts of all pairs of frequent items.

Next, the FP-growth method is recursively called to mine frequent itemsets for each item in header table of $T_\emptyset$. However, now for each item $i$, instead of traversing $T_\emptyset$ along the linked list starting at $i$ to get all frequent items in $i$’s conditional pattern base, $A_\emptyset$ gives all frequent items for $i$. For example, by checking the third line in the table for $A_\emptyset$, frequent items $e, c, a$ for the conditional pattern base of $g$ can be obtained. Sorting them according to their counts, we get $a, c, e$. Therefore, for each item $i$ in $T_\emptyset$ the array $A_\emptyset$ makes the first traversal of $T_\emptyset$ unnecessary, and $T_{\{i\}}$ can be initialized directly from $A_\emptyset$. For the same reason, from a conditional FP-tree $T_X$, when we construct a new conditional FP-tree for $X \cup \{i\}$, for an item $i$, a new array $A_{X \cup \{i\}}$ is calculated. During the construction of the new FP-tree $T_{X \cup \{i\}}$, the array $A_{X \cup \{i\}}$ is filled. For instance, in Figure 1, the cells of array $A_{\{g\}}$ is shown in table (b) of Figure 2. This array is constructed as follows. From the array $A_\emptyset$, we know that the frequent items in the conditional pattern base of $\{g\}$ are, in order, $a, c, e$. By following the linked list of $g$, from the first node we get $\{e, c, a\} : 4$, so it is inserted as $(a : 4, c : 4, e : 4)$ into the new FP-tree $T_{\{g\}}$. At the same time, $A_{\{g\}}[e, c], A_{\{g\}}[e, a]$ and $A_{\{g\}}[c, a]$ are incremented by 4. From the second node in the linked list, $\{c, a\} : 1$ is extracted, and it is inserted as $(a : 1, c : 1)$ into $T_{\{g\}}$. At the same time, $A_{\{g\}}[c, a]$ is incremented by 1. Since there are no other nodes in the linked list, the construction of $T_{\{g\}}$ is finished, and array $A_{\{g\}}$ is ready to be used for construction of FP-trees in next level of recursion. The construction of arrays and FP-trees continues until the FP-growth method terminates.

The array technique works very well especially when the dataset is sparse. The FP-tree for a sparse dataset and the recursively constructed FP-trees will be big and bushy, due to the fact that they do not have many shared common pre-fixes. The arrays save traversal time for all items and the next level FP-trees can be initialized directly. In this case, the time saved by omitting the first traversals is far greater than the time needed for accumulating counts in the associated array. However, when a dataset is dense, the FP-trees are more compact. For each item in a compact FP-tree, the traversal is fairly rapid, while accumulating counts in the associated array may take more time. In this case, accumulating counts may not be a good idea. Even for the FP-trees of sparse datasets, the first levels of recursively constructed FP-trees are always conditional FP-trees for the most common prefixes. We can therefore expect the traversal times for the first items in a header table to be fairly short, so the cells for these first items are unnecessary in the array.
Note that the datasets (the conditional pattern bases) change during the different depths of the recursion. In order to estimate whether a dataset is sparse or dense, during the construction of each FP-tree number of nodes at each level of the tree can be counted. Based on experiments [18], it is found that if the upper quarter of the tree contains less than 15% of the total number of nodes, dataset is dense. Otherwise the dataset is likely to be sparse. If the dataset appears to be dense, array for the next level of the FP-tree is not calculated. Otherwise, array for each FP-tree in the next level is calculated, but the cells for the first several (say 5) items in its header table are not set.

**FPgrowth**: [18] It has an FP-tree T as parameter. The tree has attributes: base, header and array. T.base contains the itemset X, for which T is a conditional FP-tree, the attribute header contains the head table, and T.array contains the array AX.

Algorithm 3.5: $FPgrowth^*(T)$

**Input:** A conditional FP-tree T

**Output:** The complete set of FI’s corresponding to T.

**Method:**
1. **if** T only contains a single path P
2. **then for each** subpath Y of P
3. output pattern $Y \cup T.base$ with count = smallest count of nodes in Y
4. **else for each** i in T.header
5. output $Y = T.base \cup \{i\}$ with $i.count$
6. **if** T.array is not NULL
7. construct a new header table for Y’s FP-tree from T.array
8. **else** construct a new header table from T;
9. construct Y’s conditional FP-tree $T_Y$ and its array $A_Y$
10. **if** $T_Y \neq \emptyset$
11. call $FPgrowth^*(T_Y)$;

**CFP-Tree:** A Compressed FP-Tree is a prefix tree [19] with the following properties:
1. It consists of an ItemTable and a tree whose root represents the index of the item with the highest frequency and a set of subtrees as the children of the root.
2. The ItemTable contains all frequent items sorted in descending order by their frequency. Each entry in the ItemTable consists of four fields, (1) index, (2)
item-id, (3) frequency of the item, and (4) a pointer pointing to the root of the subtree of each frequent item.

3. If \( I = \{i_1, i_2, \ldots, i_k\} \) is a set of frequent items in a transaction, after being mapped to their index-id, then the transaction will be inserted into the Compressed FP-Tree starting from the root of a subtree to which \( i_1 \) in the ItemTable points.

4. The root of the Compressed FP-Tree is the level 0 of the tree.

5. Each node in the Compressed FP-Tree consists of four fields: node-id, a pointer to the next sibling, a pointer to the next node with the same id, and a count array where each entry corresponds to the number of occurrences of an itemset. If \( C = \{C_0, C_1, \ldots, C_k\} \) is a set of counts in the count array attached to a node and the index of the array starts from zero, then \( C_i \) is the count of a transaction with an itemset along the path from the node at level \( i \) to the node where \( C_i \) is located.

Compared to FP-Tree, CFP-Tree has some important differences, as follows:

1. FP-Tree stores the item id in the tree while, in CFP-Tree, item ids are mapped to an ascending sequence of integers that is actually the array index in ItemTable.

2. The FP-Tree is compressed by removing identical subtrees of a complete FP-Tree and succinctly storing the information from them in the remaining nodes. All subtrees of the root of the FP-Tree (except the leftmost branch) are collected together at the leftmost branch to form the CFP-Tree.

3. Each node in the FP-Tree (except the root) consists of three fields: item-id, count and node-link. Count registers the number of transactions represented by the portion of the path reaching this node. Node-link links to the next node with the same item-id. Each node in the CFP-Tree consists of three fields: item-id, count array and node-link. The count array contains counts for item subsets in the path from the root to that node. The index of the cells in the array corresponds to the level numbers of the nodes above.

4. FP-Tree has a HeaderTable consisting of two fields: item-id and a pointer to the first node in the FP-Tree carrying the nodes with the same item-id. CFP-Tree has an ItemTable consisting of four fields: index, item-id, count of the item and a pointer to the root of the subtree of each item. The root of each subtree has a link to the next node with the same-item-node. Both HeaderTable and ItemTable store only frequent items.
COFI-tree (Co-Occurrence Frequent-Item-trees): The COFI-tree approach [25] consists of two main stages. Stage one is the construction of a modified Frequent Pattern tree. Stage two is the repetitive building small data structures, the actual mining for these data structures, and their release.

The first stage is to build the compact data structure called Frequent Pattern Tree. This construction is done in two phases, where each phase requires a full I/O scan of the dataset. A first initial scan of the database identifies the frequent 1-itemsets. The goal is to generate an ordered list of frequent items that would be used when building the tree in the second phase. This phase starts by enumerating the items appearing in the transactions. After enumeration these items (i.e. after reading the whole dataset), infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This list is organized in a table, called header table, where the items and their respective support are stored along with pointers to the first occurrence of the item in the frequent pattern tree. Phase 2 would construct a frequent pattern tree.

This phase 2 requires a second complete I/O scan from the dataset. For each transaction read, only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted transaction items are used in constructing the FP-Trees as follows: for the first item on the sorted transactional dataset, check if it exists as one of the children of the root. If it exists then increment the support for this node. Otherwise, add a new node for this item as a child for the root node with 1 as support. Then, consider the current item node as the new temporary root and repeat the same procedure with the next item on the sorted transaction. During the process of adding any new item-node to the FP-Tree, a link is maintained between this item-node in the tree and its entry in the header table. The
header table holds as one pointer per item that points to the first occurrences of this item in the FP-Tree structure.

Computation of the frequencies relies first on building independent, relatively small trees for each frequent item in the header table of the FP-Tree called COFI-trees. The small COFI-trees are similar to the conditional FP-Trees [18] in general in the sense that they have a header with ordered frequent items and horizontal pointers pointing to a succession of nodes containing the same frequent item, and the prefix tree with paths representing sub-transactions. However, the COFI-trees have bidirectional links in the tree allowing bottom-up scanning as well, and the nodes contain not only the item label and a frequency counter, but also a participation counter. The COFI-tree for a given frequent item \( x \) contains only nodes labeled with items that are more frequent than \( x \) or as frequent as \( x \).

COFI-trees are built for the FP-Tree. For example, if the item F is the least frequent item in the header table than the first Co-Occurrence Frequent Item tree is built for item F. In this tree for F, all frequent items, which are more frequent than F, and share transactions with F, participate in building the tree. This can be found by following the chain of item F in the FP-Tree structure. The F-COFI-tree starts with the root node containing the item in question, then a scan of part of the FP-Tree is applied following the chain of the F item in the FP-Tree. The first branch FA has frequency of 1, as the frequency of the branch is the frequency of the test item, which is F. The goal of this traversal is to count the frequency of each frequent item with respect to item F.

By applying the anti-monotone constraint property it can be predicted that item F will never appear in any frequent pattern except itself. Consequently there will be no need to continue building the F-COFI-tree. The same process is repeated each next least frequent items.

A pruning technique is applied to remove all nonfrequent items with respect to the main frequent item of the tested COFI-tree. Then each of the trees is mined separately as soon as they are built, minimizing the candidacy generation and without building conditional sub-trees recursively. The trees are discarded as soon as they are mined. At any given time, only one COFI-tree is present in main memory.

**FP-Stream:** Mining frequent itemsets over a stream of transactions presents difficult new challenges over traditional mining in static transaction databases. Stream transactions can only be looked at once and streams have a much richer frequent itemset structure due to their inherent temporal nature. Recently, the mining and management of stream data has received considerable attention. In this model the data arrives as a potentially infinite stream of elements. It is assumed that the stream can only be scanned once, hence, once an element has passed, the element cannot be revisited unless it is stored in main memory. Some applications for which this model is appropriate include network traffic analysis, web click stream mining, power consumption measurement, sensor network data analysis, and dynamic tracing of stock fluctuation.

Previous work [4] studied the problem of maintaining all frequent itemsets over the entire history of the stream. This will not be desirable in circumstances where the goals of mining the stream are time sensitive. For example, consider a shopping transaction stream that began a year ago. Itemsets frequent over the life of the stream will be of no use in detecting purchasing trends or combinations of items that have become popular only recently. As another example, in network monitoring, changes in the
frequent patterns in the past several minutes are valuable and can be used for detection of network intrusions.

**Sliding Window Model:** In many cases, the data stream may evolve over time, as a result of which it is desirable to determine all the frequent patterns over a particular sliding window. A method for determining the frequent patterns over a sliding window is discussed in [7]. The inspiration behind sliding window is that the user is more concerned with the analysis of most recent data streams. Thus the detailed analysis is done over the most recent data items and summarized versions of the old ones. Here, one specifies a window size w, and explicitly focuses only on the most recent stream of size w, i.e., at time t, only consider updates \( a_{t-w+1}, \ldots, a_t \). Items outside this window fall out of consideration for analysis, as the window slides over time.

FP-stream data structure was proposed by [7], for maintaining information about itemset frequency histories. This data structure is dynamically updated after each new batch of transactions arrives. At any time, a request for itemsets frequent over a user-defined interval can be serviced by scanning the maintained FP-stream. An approximate answer is given whose error is guaranteed to be no greater than user-specified frequency and temporal thresholds. Itemset frequency histories are maintained using logarithmic tilted-time window tables. These tables store frequencies over exponentially increasing time granularities (e.g., every second for the last minute, every two seconds for the previous minute, every four seconds for the minute before that, etc.). Moreover, entries in the time window table are weighted by an aging function. Older entries are weighted less. A tree structure (similar to the FP-tree) is used to store a collection of itemsets with their tilted-time window tables. With the arrival of a new batch of transactions, the FP-stream is updated. Pre-existing itemsets may be dropped or part of their tilted-time window table entries may be dropped. New itemsets may be added. The dropping condition guarantees that no itemset which is frequent over arbitrary time intervals consistent with pre-defined exponential time granularities will be absent from the FP-stream. Moreover, for each itemset in the structure, its frequency approximation will not be different than the true frequency by more than a user-defined fraction (taking into account age weighting).

A logarithmic tilted-time window table is used to maintain history information for a single itemset. But, stream of transactions contains information about a potentially large number of itemsets. An efficient method of storing a large collection of itemsets and their tilted-time window tables is needed. Here collection is represented using a tree structure. As an example, Figure 3.11 depicts the representation of a collection of itemsets with a single associated frequency (support). Each node in the tree represents a itemset (from root to node). The frequency is recorded at each node.
A tilted-time window table is embedded at each node. Figure 3.12 depicts an example of an itemset tree with tilted-time window tables embedded. This structure along with a global partition table is called an FP-stream.

FP-stream model as a foundation upon which frequent itemset change mining queries can be answered. For example, the change in frequency of itemsets across time intervals can be computed. Moreover, the user can first examine the frequent itemsets at a low granularity (less details), then drill-down to examine the frequent itemsets at higher granularities (more details).

**BFI-tree**: Another tree based algorithm to mine recent frequent itemsets over data stream is based on BFI-trees. [20] Bounded Frequent Itemsets stream (abbreviated as BFI-stream) algorithm, which uses a prefix-tree based structure, called BFI-tree, to maintain all accurate frequent itemsets from sliding windows over data streams. By monitoring the boundary between frequent itemsets and infrequent itemsets, it restricts the update process on a small part of the tree. Mining all frequent itemsets with accurate frequencies is just to traverse the tree. It is time efficient even when the user-specified minimum support threshold is small.
BFI-tree is a prefix-tree based data structure. A node in the BFI-tree stores the following information: item, frequency is frequent, and children. Similar to a prefix tree, a path from the root to a node in the BFI-tree represents an itemset. \( nX \) is used to denote a node, where \( X \) is the itemset it represents. The status property, called is frequent, denotes whether the itemset is frequent. The children property represents its direct descendants. A node is (in) frequent if and only if its represented itemset is (in) frequent. BFI-tree maintains all frequent itemsets and a selected part of infrequent itemsets. Because users are only interested in itemsets with support bigger than the pre-defined minimum support threshold, there is no need to maintain all itemsets in the BFI-tree. However, it is impossible to know a node changing from infrequent to frequent if we only maintain frequent itemsets. To solve this problem, a boundary between the frequent nodes and infrequent nodes is record to monitor node status changes. The infrequent nodes in BFI-tree can be divided into two kinds, the first is the children of the root of BFI-tree, and the second is created by joining its frequent parent with the parent’s frequent right siblings.

BFI-tree will monitor the boundary movements to efficiently maintain the selected part of infrequent itemsets. If a node status changes, either from infrequent to frequent or vice versa, it must come through the boundary and result in boundary movements. Boundary movements may cause recursive updates, which will be restricted in a small subtree. It may also cause creating new nodes, which need an additional scan on all transactions in the sliding window to compute their frequencies. In order to return accurate frequent itemsets, all transactions in the sliding window must be maintained in a highly compact structure. However, the boundary is stable at most time, which means the update cost is very small. BFI-tree uses the Apriori property in construction and updates to prune infrequent nodes, and there are two corollaries for us to use the Apriori property more efficiently.

**Trie tree:** A trie is a rooted, labeled tree. The root is defined to be at depth 0, and a node at depth \( d \) can point to nodes at depth \( d + 1 \). A pointer is also referred to as edge or link. If node \( u \) points to node \( v \), then we call \( u \) the parent of \( v \), and \( v \) is a child node of \( u \). For the sake of efficiency – concerning insertion and deletion – a total order on the labels of edges has to be defined.

Tries are suitable for storing and retrieving not only words, but any finite sets (or sequences). Tries also called a lexicographic tree. They are used to quickly determine the support of itemsets whose size is greater than 2. The link is labeled by a frequent item, and a node represents an itemset, which is the set of the items in the path from the root to the leaf. The label of a node stores the counter of the itemset that the node represents. Figure 3.7 presents a trie (without the counters) that stores the itemsets \{A\}, \{C\}, \{E\}, \{F\}, \{A,C\}, \{A,E\}, \{A,F\}, \{E,F\}, \{A,E,F\}. 
As a preprocessing the database is scanned and the global frequencies of the items are counted. Using the minimum support infrequent items are erased and frequent items are renumbered in frequency-descending order. During a second scan of the database all transactions are preprocessed: infrequent items are erased, frequent items are translated and sorted according to the new numbering. Then the itemset is inserted into a temporary trie. This trie is similar to the classic FP-tree: each node contains an item identifier, a counter, a parent pointer and a children map. The children map is an unordered array of pairs (child item identifier, child node index). Lookup is done with linear scan. Though this is asymptotically not an optimal structure, the number of elements in a single children map is expected to be very small, linear scan has the least overhead compared to ordered arrays with binary search, or search trees/hash maps.

The core algorithm [22] consists of a recursion. In each step the input is a condition (an itemset), a trie structure and an array of counters that describe the conditional frequencies of the trie nodes. In the body we iterate through the remaining items, calculate the conditional counters for the input condition extended with that single item, and call the recursion with the new counters and with the original or a new, projected structure, depending on the projection configuration and the percentage of empty nodes. The core recursion is given below.

**Algorithm 3.6 : Core algorithm**

Recursion(condition, nextitem, structure, counters):

1. for citem=nextitem-1 downto 0 do
2. if support of citem < min_supp then
3. continue at next citem
4. end if
5. newcounters=aggregate conditional pattern base for condition U citem
6. if projection is beneficial then
7. newstructure=projection of structure to newcounters
8. Recursion(condition U citem, citem, newstructure, newcounters)
The recursion has four different implementations that suit differently sized FP trees:

- Very large FP trees that contain millions of nodes are treated by *simultaneous projection*: the tree is traversed once and a projection to each item is calculated simultaneously. This phase is applied only at the first level of recursion; very large trees are expected to arise from sparse databases, like real market basket data; conditional trees projected to a single item are already small in this case.

- *Sparse aggregate* is an aggregation and projection algorithm that does not traverse those part of the tree that will not exist in the next projection. To achieve this, a linked list is built dynamically that contains the indices to non-zero counters. This is similar to the header lists of FP-trees. This aggregation algorithm is used typically near the top of the recursion, where the tree is large and many zeroes are expected. The exact choice is tunable with parameters.

- *Dense aggregate* is the default aggregation algorithm. Each node of the tree is visited exactly once and its conditional counter is added to the counter of the parent.

- *Single node optimization* is used near the last levels of recursion, when there is at most one node for each item left in the tree. (This is a slight generalization of the tree being a single chain.) In this case no aggregation and calculation of new counters is needed, so a specialized very simple recursive procedure starts that outputs all subsets of the paths in the tree as a frequent itemset.

The core [22] data structure is a trie. Each node contains a counter and a pointer to the parent. As the trie is never searched, only traversed from the bottom to the top, child maps are not required. The nodes are stored in an array, node pointers are indices to this array. Nodes that are labeled with the same item occupy a consecutive part of this array, this way we do not need to store the item identifiers in the nodes. Furthermore, we do not need the header lists, as processing all nodes of a specified item requires traversing an interval of this array. This also allows faster execution as only contiguous memory reads are executed. We only need one memory cell per frequent item to store the starting points of these intervals (the *itemstarts* array). The parent pointers (indices) and the counters are stored in separate arrays (*parents* and *counters* rsp.) to fit the core algorithm's flexibility: if projection is not beneficial, then the recursion proceeds with the same structural information (parent pointers) but a new set of counters.

The item intervals of the trie are allocated in the array ascending, in topological order. This way the bottom-up and top-down traversal of the trie is possible with a descending respectively ascending iteration through the array of the trie, still only using contiguous memory reads and writes. This order also allows the truncation of the tree to a particular level/item: if the structure is not rebuilt but only a set of conditional counters is calculated
for an item, then the recursion can proceed with a smaller sized newcounters array and the original parents and itemstarts array.

Tries can be implemented in many ways. In compact representation the edges of a node are stored in a vector. Each element of a vector is a pair; the first element stores the label of the edge, the second stores the address of the node, which the edge points to. This solution is very similar to the widespread “doubly chained” representation, where edges of a node are stored in a linked list.

In the non compact representation (also called tabular implementation) only the pointers are stored in a vector with a length equal to that of the alphabet (frequent items in our case). An element at index i belongs to the edge whose label is the ith item. If there is no edge with such a label, then the element is NIL. This solution has the advantage of finding an edge with a given label in O(1) time, instead of O(log n) required by a binary search –which is the case in compact representation. Unfortunately for nodes with few edges this representation requires some more memory than compact representation. On the contrary, if a node has many edges (exact formula can be given based on the memory need of pointers and labels), then the non-compact representation needs less memory since labels are not stored explicitly. If there are many nodes with single edges, then memory can be saved by using patricia tries [21].

![Fig- 3.14 standard trie for a sample dataset](image1)

![Fig- 3.8 Patricia trie for the sample dataset](image2)

The Patricia trie for a dataset D’ is a modification of the standard trie: namely, each maximal chain of nodes v₁ → v₂ → · · · → vₖ, where all vᵢ’s have the same count value c
and (except for \(v_k\)) exactly one child, is coalesced into a single node that inherits count value \(c\), \(v_k\)'s children, and stores the sequence of items previously stored in the \(v_i\)'s.

3.5 Counter-based Algorithms

Here counters are associated with each distinct item and are either incremented or decremented during the mining process. Few counter based algorithms are discussed below.

**Majority Algorithm**: MAJORITY algorithm solves the problem in arrivals only model. Start with a counter set to zero. For each item:

- If counter is zero, pick up the item, set counter to 1
- Else, if item is same as item in hand, increment counter
- Else, decrement counter

After processing all items, the algorithm guarantees that if there is a majority vote, then it must be the item stored by the algorithm.

In short MAJORITY can be stated as follows: store the first item and a counter, initialized to 1. For each subsequent item, if it is the same as the currently stored item, increment the counter. If it differs, and the counter is zero, then store the new item and set the counter to 1; else, decrement the counter. The correctness of this algorithm is based on a pairing argument: if every non-majority item is paired with a majority item, then there should still remain an excess of majority items. Although not posed as a streaming problem, the algorithm has a streaming flavor: it takes only one pass through the input (which can be ordered arbitrarily) to find a majority item. To verify that the stored item really is a majority, a second pass is needed to simply count the true number of occurrences of the stored item.

**Frequent Algorithm.** It is generalization of the Majority algorithm to solve the problem of finding all items in a sequence whose frequency exceeds a \(1/k\) fraction of the total count. Instead of keeping a single counter and item from the input, the FREQUENT algorithm stores \(k-1\) (item, counter) pairs. The natural generalization of the Majority algorithm is to compare each new item against the stored items \(T\), and increment the corresponding counter if it is amongst them. Else, if there is some counter with count zero, it is allocated to the new item, and the counter set to 1. If all \(k-1\) counters are allocated to distinct items, then all are decremented by 1. A grouping argument is used to argue that any item which occurs more than \(n/k\) times must be stored by the algorithm when it terminates. In short FREQUENT generalizes MAJORITY to find up to \(k\) items that occur more than \(1/k\) fraction of the time.

Keep \(k\) different candidates in hand. For each item in stream:

- If item is monitored, increase its counter
- Else, if \(<k\) items monitored, add new item with count 1
- Else, decrease all counts by 1

Pseudo code [11] to illustrate this algorithm is given below:

**Algorithm 3.7**: Frequent(k)

1. \(n = 0; T = \emptyset\)
2. **for each** \(i\):
3. \( n = n + 1; \)
4. \( \text{if } i \in T \)
5. \( \text{then } c_i = c_i + 1; \)
6. \( \text{else if } |T| < k - 1 \)
7. \( \text{then } T = T \cup \{i\}; \)
8. \( c_i = 1; \)
9. \( \text{else for all } j \in T \text{ do} \)
10. \( c_j = c_j - 1; \)
11. \( \text{if } c_j = 0 \)
12. \( \text{then } T = T \setminus \{j\}; \)
13. \( \text{endfor} \)

set notation are used to represent the operations on the set of stored items \( T \): items are added and removed from this set using set union and set subtraction respectively, each item \( j \) stored in \( T \) has an associated counter \( c_j \). For items not stored in \( T \), then \( c_j \) is defined as 0 and does not need to be explicitly stored.

It is sometimes stated that the FREQUENT algorithm does not solve the frequency estimation problem accurately, bound on the true frequency of the items it retains, but this is erroneous.

**Lossy Counting algorithm:** The LOSSYCOUNTING algorithm was proposed by [4], in addition to a randomized sampling-based algorithm and techniques for extending from frequent items to frequent itemsets.

Simplified version:
- Track items and counts
- For each block of \( 1/\varepsilon \) items, merge with stored items and counts
- Decrement all counts by one, delete items with zero count
- Counts are accurate to \( \varepsilon N \)

The algorithm stores tuples which comprise an item, a lower bound on its count, and a 'delta' (\( \Delta \)) value which records the difference between the upper bound and the lower bound. When processing the \( i \)th item, if it is currently stored then its lower bound is increased by one; else, a new tuple is created with the lower bound set to one, and \( \Delta \) set to \( \lceil i/k \rceil \). Periodically, all tuples whose upper bound is less than \( \lceil i/k \rceil \) are deleted. These are correct upper and lower bounds on the count of each item, so at the end of the stream, all items whose count exceeds \( n/k \) must be stored. As with FREQUENT, setting \( k = 1/\varepsilon \) ensures that the error in any approximate count is at most \( \varepsilon n \). Storing the delta values ensures that highly frequent items which first appear early on in the stream have very accurate approximated counts. But this adds to the storage cost. A variant of this algorithm dispenses with explicitly storing the delta values, and instead has all items sharing an implicit value of \( \Delta(i) = \lceil i/k \rceil \). The modified algorithm stores (item, count) pairs, for each item in the stream, if it is stored, then the count is incremented; otherwise, it is initialized with a count of 1. Every time \( \Delta(i) \) increases, all counts are decremented by
1, and all items with zero count are removed from the data structure. Following is the pseudocode for Lossy counting algorithm [11].

**Algorithm 3.8 : LOSSYCOUNTING (K)**

1. \( n=0; \Delta =0; T=\emptyset; \)
2. \( \text{for each } i : \)
3. \( n = n + 1; \)
4. \( \text{if } i \in T \)
5. \( \text{then } c_i = c_i + 1 \)
6. \( \text{else } T = T \cup \{ i \}; \)
7. \( c_j = 1 + \Delta; \)
8. \( \text{if } \lfloor n/k \rfloor \neq \Delta \)
9. \( \text{then } \Delta = n/k; \)
10. \( \text{for all } j \in T \)
11. \( \text{do if } c_j < \Delta \)
12. \( \text{then } T = T / \{ j \} \)
13. \( \text{endfor} \)

Here again \( T \) represents the set of currently monitored items, updated by set operations, and \( c_j \) are corresponding counts.

Implementation of above algorithm uses 3 modules. Buffer, Trie, and setGen. Buffer repeatedly reads in a batch of transactions into available main memory. Transactions are set of item-id’s. They are laid out one after the other in a big array. A bitmap is used to remember transaction boundaries. After reading in a batch, Buffer sorts each transaction by its item-id’s.

Trie maintains the data structure. It is a forest consisting of labeled nodes. Labels are of the form \(< \text{item}_id, f, \Delta, \text{level}>\), where \( \text{item}_id \) is an item-id, \( f \) is its estimated frequency, \( \Delta \) is maximum possible error in \( f \), and \( \text{level} \) is the distance of this node from the root of the tree it belongs to. The root node has level 0. The level of any other node is one more than its parent. The root nodes in the forest are also ordered by \( \text{item}_id \)’s. SetGen module operates on the current batch of transactions. It enumerates subsets of these transactions along with their frequencies.

Buffer repeatedly fills available main memory with as many transactions as possible and sorts them. SetGen operates on the current batch of transactions. It generates sets of itemsets along with their frequency counts in lexicography order. It limits the number of subsets using the pruning rule. Together Trie and SetGen implements the Update_Set and New_Set. In the end, Trie stores the updated data structure and Buffer gets ready to read in the next batch.
**Space Saving**: A set of items and counters are kept, and various simple rules are applied when a new item arrives. Here, \(k\) (item, count) pairs are stored, initialized by the first \(k\) distinct items and their exact counts. As usual, when the next item in the sequence corresponds to a monitored item, its count is incremented. But when the next item does not match a monitored item, the (item, count) pair with the smallest count has its item value replaced with the new item, and the count incremented. So the space required is \(O(k)\) (resp. \(O(1/\epsilon)\)), and the counts of all stored items solve the frequency estimation problem with error \(n/k\) (resp. \(\epsilon n\)). It also shares the nice property of LOSSYCOUNTING that items which are stored by the algorithm early in the stream and not removed have very accurate estimated counts. “SpaceSaving” algorithm merges Lossy Counting and FREQUENT algorithms, the algorithm in simplest form appears as below:

- Keep \(k = 1/\epsilon\) item names and counts, initially zero
- Count first \(k\) distinct items exactly
- On seeing new item:
  - If it has a counter, increment counter
  - If not, replace item with least count, increment count
- Smallest counter value, \(\text{min}\), is at most \(\epsilon n\)
  - Counters sum to \(n\) by induction
  - \(1/\epsilon\) counters, so average is \(\epsilon n\): smallest cannot be bigger
- True count of an uncounted item is between 0 and \(\text{min}\)
  - Proof by induction, true initially, \(\text{min}\) increases monotonically
  - Hence, the count of any item stored is off by at most \(\epsilon n\)
- Any item \(x\) whose true count > \(\epsilon n\) is stored
  - By contradiction: \(x\) was evicted in past, with count \(\leq \text{min}\)
  - Every count is an overestimate, using above observation
  - So est. count of \(x > \epsilon n \geq \text{min} \geq \text{mint}\), and would not be evicted

Following is the pseudocode for space saving algorithm

**Algorithm 3.9**: SpaceSaving (\(k\))

1. \(n = u;\quad T = \emptyset;\)
2.  
3.     **for each** \(i\):
4.         \(n= n + 1;\)
5.         **if** \(i \in T\)
6.             **then** \(c_i = c_i + 1;\)
7.         **else if** \(|T| < k\)
8.             **then** \(T = T \cup \{i\};\)
9.             \(c_i = 1;\)
10.         **else** \(j = \arg\min_{j \in T} c_j;\)
11.         \(c_i = c_j + 1;\)
12.         \(T = T \cup \{i\} / \{j\};\)
13.  **endfor**
Counter algorithms are very efficient for arrivals-only case. Our work is also based on this approach. It uses $O(1/\epsilon)$ space, guarantee $\epsilon N$ accuracy and very fast in practice (many millions of updates per second)

### 3.6 A new counter based approach to mine frequent itemsets

In the present section a new algorithm for mining frequent itemsets is proposed. Compared to above discussed algorithms proposed algorithm is also a counter based algorithm to count frequent itemsets of length one and two. Here Input stream is split into fixed-size windows and each these windows are processed sequentially. Size of window is $N$ (total number of items stored in the window collected over one hour). For each element in a window it inserts an entry into a table, which monitors the number of occurrences of the elements or if the element is already in the table it updates its frequency. At end of each window the algorithm removes all elements of small frequency from table.

Let $s$ be the support threshold. Let $e$ be error parameter and $N$ be the size of window . Let $t_i$ be the window where $i$ indicate the index of window, e.g. $t_2$ indicates window containing items of second batch i.e. next 50 transactions of items. 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 element to archive. An element 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. $f \geq (s-e)N$. This condition results in elements with less frequency being overwritten by the elements of next time window.

Results of processing will be stored on a secondary storage which can be queried for decisions. Here only 24hr processed data at a time since the set of frequent items usually are time sensitive and in many cases, change of patterns and their trends are more interesting than patterns themselves so authors also propose to store patterns over stream of length two. For example in intrusion detection system last 10 minutes data plays an important role to detect the intruder.

Proposed work is carried out on a Chess database available at FIMI repository. This database consists of set of transactions with item_id’s only. This work is designed to answer short term queries like.

1. What is the frequent item set over the time period $t_2$ and $t_5$?
2. What are the periods when itemset (4,7) is frequent?
3. Does the support of item_id 45 change dramatically from time period $t_3$ to $t_{10}$? And so on.

To facilitate above queries two files are maintained, File1 containing frequent itemsets of length one and File2 containing frequent itemsets of length two. Longer patterns are not generated because over a short period of time longer patterns may not be of interest but, if required; longer patterns can be generated using these patterns of length one & two using the algorithm proposed in [24].
**Algorithm 3.10:** LossyCounting algorithm works for frequent item sets of length one, but proposed algorithm works for frequent item sets of length one as well two. It uses an array of single dimension to store patterns of length one. Size of array is equal to number of distinct items in the stream. Let’s call this array as A1 which works as a counter.

1. initialize all the items of A1 to zero
2. Read an item from stream and increment its counter by one
   \[ A1[item] = A1[item] + 1 \]
3. Repeat step 2 till all the items of window \( t_i \) are processed.
4. At the end of the time window, accept a support threshold \( s \) and error \( e \) from user and archive only those items to a secondary storage (File1) whose frequency \( f \) is
   \[ f \geq (s-e)N. \]
5. Rest of items are discarded and counters are again initialized to zero to collect frequencies from next time window.

File1 will contain data of 24 time windows. Each row in File1 will indicate frequencies of frequent item sets of length one. Suppose item 5 occurs 10 times in 2nd time window \( (t_2) \) then it will be stored as second row of File1. \(<item\- frequency> 1-10 2-15 5-20\)

This file can be queried for most frequently occurring itemsets of length one over a time period in the range \( t_1 \) to \( t_{24} \). Each time window will collect stream data of an hour. It can also be queried to detect if any item sets frequency is changing drastically over time period \( t_1 \) to \( t_3 \) etc.

To count itemset of length 2 a double dimension array \( A2 \) is maintained. Suppose there are \( n \) distinct items in the stream then \( A2 \) will be declared as \( A2[n][n] \). It will store sequential patterns of length two. Advantage of using array is random access; any update will require only one unit of time.

\[
A2[n][n] = \\
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 & \ldots & n \\
1 & 0 & 5 & 25 & 10 & 65 & \ldots & 0 \\
2 & 12 & 0 & 45 & 0 & 12 & \ldots & 4 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\end{bmatrix}
\]

In the above array \( A2 \), item sets of length two starting with item one are stored in the \( A2[1][1] \) i.e. \(<item\-item\-frequency> 1,2 – 5 1,3- 25 1,4- 10 1,5-65\)

Here pattern \(<1,2>\) is occurring 5 times in a time window \( t_i \). After pruning the item sets ie whose frequency \( f \geq (s-e)N \), are archived in File2 for further querying or processing. Here the updation of counters is done in the same loop along with the single item counters. Frequency of these patterns is updated in one unit of time.

\[
A2[i][j] = A2[i][j] + 1
\]

File2 can be queried for the queries related to patterns of length two over 24 time windows. E.g. What are the periods when itemset \( (4, 7) \) is frequent.

### 3.7 System issues and optimization

Experiments are carried out using Chess data base available at FIMI repository which consists of 75 distinct item_id’s, average length of transactions is 37 and number of
transactions are 3,196. This file itself is treated as a data stream. Total numbers of items processed are: \((37 \times 3,196 = 1,18,252 \text{ items})\). To generate itemsets of length two a trie tree is implemented using array data structure. One dimensional array acts as a root of the trie and second array \(A2\) is level 2 of trie tree. Use of pointers is avoided. Chess data base consist of only item_id 's and these item_id’s are used as an index of the array so updates can be carried out in one unit of time. In case of [4, 23] removing an element from the table can introduce a subsequent error in its estimated frequency. If a removed element later re-enters the table, then its new frequency does not reflect the amount removed earlier. Obviously, this error can only underestimate the true frequency of an element, as a counter is only incremented when a corresponding element is observed. But in proposed algorithm frequent itemsets of each time window is maintained on a secondary storage, which avoids the loss of previous frequent data.

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-C. We report our experimental results on the chess dataset downloaded from \(\text{fimi.cs.helsinki.fi/data/}\). Experiments are carried out with:

- Support threshold =0.01%  
- error =0.001%  
- size of time window \(t_i = 4927\) items

Fig- 3.9 Frequent itemsets over time window \(t_1\) to \(t_{24}\)

Fig- 3.10 Frequency of item_id \(<9>\) and \(<75>\) over time windows \(t_1\) to \(t_{24}\)
Fig-3.11 Frequent itemsets over time window $t_1$ to $t_{24}$ with support 0.02

Fig-3.12 Frequency of pattern $<1,3>$ and $<5,7>$ over time windows $t_1$ to $t_{24}$
Fig- 3.13 Frequency of pattern <1,3> and <5,7> over time windows t₁ to t₂₄

Fig- 3.14 Frequent pattern over time window t₁ to t₂₄ of size two

3.8 Conclusion

Here basic LossyCount algorithm [4] is modified, so that along with itemset of length one it also generates itemsets of length two. Instead of using pointers as compared to LossyCount array data structure is used to store the item counts which is time efficient as updations can be carried out within one unit of time. For real time systems time efficiency plays an important role. Eg. Intrusion detection, stock fluctuation etc. LossyCount algorithm does not store frequent items at each time window where as this new algorithm allows user to store frequent itemsets of each hour. It helps in answering short time queries efficiently.
Limitation of this algorithm is, it is based on array so if the number of distinct items are more in number than limitation will be main memory as it will require a large array for storage which will require continuous allocation of memory which may not be possible. The proposed algorithm is efficient for arrival case only. In future this algorithm can be extended to work for large number of distinct data items with dynamic data structure.

An article based on this work is published in an international journal. [27].
3.9 References


[17] Jiawei Han, Jian Pei, Yiwen Yin, Mining frequent patterns without candidate generation, Proceedings of the 2000 ACM SIGMOD international conference on Management of data, p.1-12, May 15-18, 2000, Dallas, Texas, United States


[20] Kun Li, Yong-yan Wang, Manzoor Ellahi, Hong-an Wang, Mining Recent Frequent Itemsets in Data Streams, Fifth International Conference on Fuzzy Systems and Knowledge Discovery, IEEE - 2008


[26] Cheqing Jin, Weining Qian, Chaofeng Sha, Jeffrey X. Yu, Aoying Zhou, *Dynamically maintaining frequent items over a data stream*, CIKM’03, November 3-8-2003, New Orleans, Louisiana, USA. ACM