INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD • INDIA

Research and Publications

Mining Frequent Item sets in Data Streams

Rajanish Dass

W.P. No. 2008-01-06
January 2008

INDIAN INSTITUTE OF MANAGEMENT
AHMEDABAD-380 015
INDIA

Mining Frequent Item sets in Data Streams

1. Introduction:

From the last decade, data mining has become the key technique to analyze and understand the data. Typical data mining tasks include association mining, classification and clustering. These techniques help find interesting patterns, regularities and anomalies in the data. However traditional data mining techniques can not directly apply to the data streams. This is because mining algorithms developed in the past target disk-resident or in-core datasets, and usually makes several passes of the data. Mining data streams are allowed only one look at the data, and techniques have to keep pace with the arrival of new data. Furthermore, dynamic data streams pose new challenges, because their underlying distribution might be changing. Recently a number of algorithms focus on approximate one-pass algorithms, mining over dynamic data streams, and mining changes or trends in data streams.

For data stream applications, the volume of data is usually too huge to be stored on permanent devices or to be scanned thoroughly more than once. Both approximation and the ability to adapt are key ingredients for execute queries and performing mining tasks over rapid data streams. The following figure shows a stream mining application example:

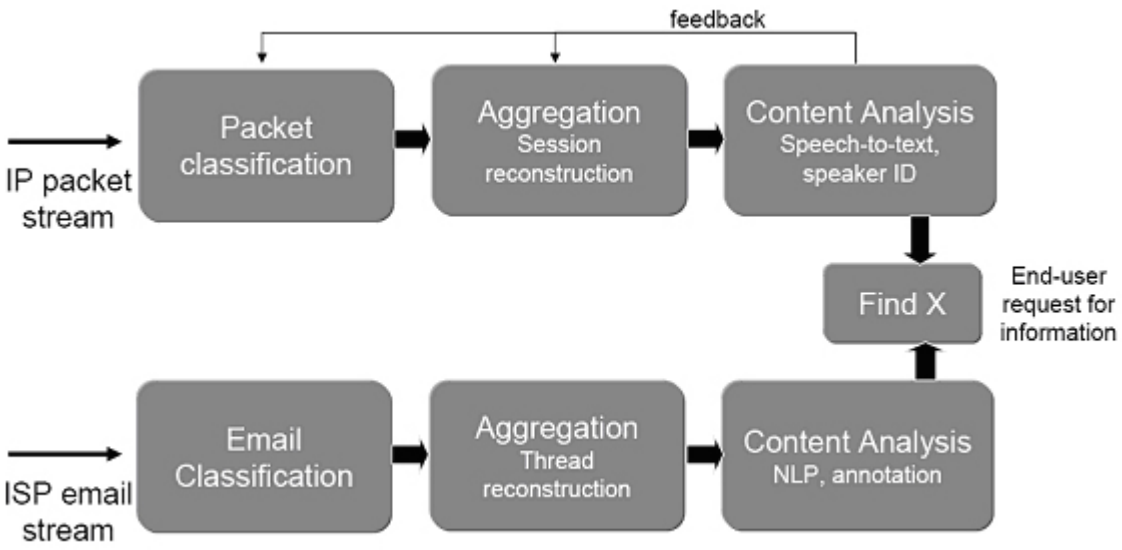

Figure 1 A stream mining application example

So, the work regarding mining of data streams addresses the need to

- capture the evolving nature of the data stream
- handle the bursty nature of the stream with limited resources.

With the pass-through feature of data streams, two types of resources, i.e., memory space and computation power, are particularly valuable in the streaming environment. An effective algorithm for data streams is expected to have the capability of resource-awareness, which means that it is highly desirable for all resources in a streaming environment to be adaptively allocated. The overall goal is to maximize precision by making the best use of available resources. Advanced scheduling and memory management algorithms are being developed.

1.1. Real world applications of Stream Data Mining:

Some of the motivating examples for stream data mining are:

- A typical ISP (internet service provider) would be interested in finding information like how much traffic went on a link in a day from a given set of

IP addresses, how much of traffic between two routers was is similar, what are the top ten heaviest traffics in a day, what are the variations of data traffic in a day, determine the total amount of bandwidth used by each source-destination pair. The above queries are very useful in rerouting users to backup servers if the primary servers are overloaded and also finding denial service attacks. All these questions can be answered by stream mining.

- There are several emerging applications in sensor monitoring where a large number of sensors are distributed in the physical world and generate streams of data that need to be combined, monitored and analyzed.
- Think of terrestrial, atmospheric and ocean surface data collected by satellites. The data collect by satellites in huge and mining such data gives enormous information about weather conditions, which helps in many ways.
- Analysis of stock prices, identifying trends and forecasting future values, and the above examples involves mining streams of data.

1.2. Problem Definition:

Mining frequent itemsets is an essential step in data mining problems. Formally presenting the problem of mining frequent itemsets:

- Let $\Sigma = \{i_1, i_2, \ldots i_n\}$ be a set of literals called items.
- A data stream a set of transactions Ti where T_i is a subset of I.
- Each transaction is associated with a unique identifier or timestamp.
- Any subset of I is called an itemset. $I = \{o_1, o_2, \ldots, o_k\}$ is an itemset for all $1 \leq i \leq k$, $o_i \in \Sigma$.
- A k-itemset is an itemset whose size is k.
- The number of times an itemset occurs in a stream is called its frequency (f_I).
- The support of an itemset is defined as the ratio between frequency of itemset and the total number of transactions. If f_I is the frequency of an itemset in a N number of transactions then its support is f_I/N.

- An itemset is called a *frequent* itemset if its support is greater than a user specified threshold value (σ).

1.3. Data Stream Model:

Stream data differs from traditional data used for DBMS applications. In DBMS applications data is stored on a disk, it can be randomly accessed at any point of time. But the stream data can neither be stored on a disk nor can be accessed randomly. Some main differences between stream data and traditional DBMS data are:

- Data elements arrive online, but not from a static storage.
- Data streams are potentially unbound, and this is the reason why they can not be stored on a static storage.
- Data should be discarded or archived as soon as we process it. To retrieve any information regarding past stream data, we need to store all the data, which is not possible with limited storage resources. So, we have to merge the data and should find new data structures for storing them.

1.4. Outline of the Paper:

In this paper, we discuss the major contributions and current techniques, methods and algorithms in data stream mining and discuss prevalent issues. The rest of the paper is arranged as follows: In sections 2-7 we discuss some of the major algorithms for mining frequent itemsets in data streams. Section 2 talks about frequent itemset mining, section 3 gives approximate techniques for mining data streams, section 4 gives some sequential pattern mining algorithms in data stream mining, section 5 talks about algoritms for finding top k-frequent itemsets, adaptive algorithms are discussed in section 6. Other related algorithms are also discussed in section 7. Challenges that all these algorithms face are explained in section 8. In section 9 and 10 we conclude our discussion.

2. Algorithms For mining frequent patterns from Stream Data:

We now present some algorithms to find frequent itemsets in data streams. The algorithms that we consider in this paper are systematized in the following figure: (algorithms are classified by the broad methodology adopted by them).

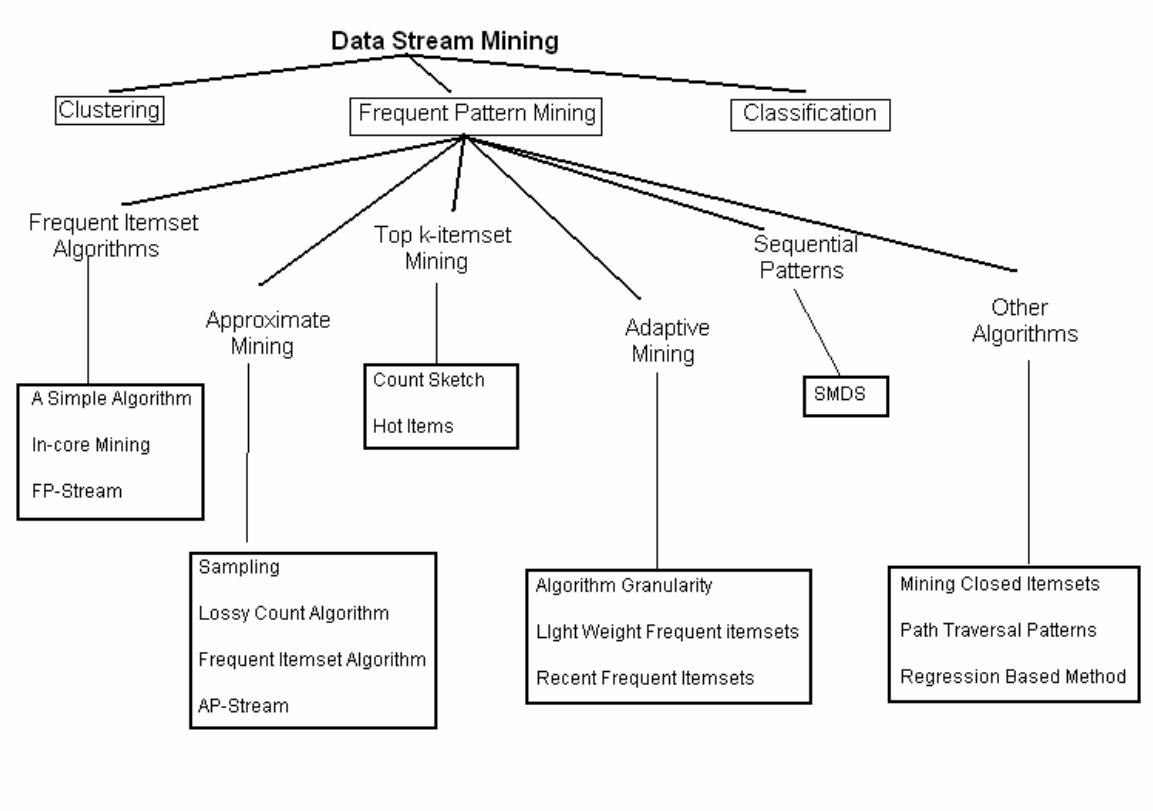

Figure 2: Systematization of Algorithms

2.1. A Simple Algorithm:

This algorithm is proposed by (Karp, Papadimitriou et al. 2003). It is a two pass algorithm, first pass finds candidate elements and the second pass removes false negatives. This algorithm generalizes the idea of finding a majority element (which occurs more than half of the time): Occurrences of two different items are found and eliminated from the sequence until only one element remains. The element left can only be the majority element and no other element can be a majority element as occurrence of

another element is also removed with an occurrence of this finally left element. By taking another pass over sequence number of occurrences of this item is found and if it is greater than half the size of sequence then it is the majority item. Extending this idea to find items with support greater than *s*:

Given a sequence of length N and a support threshold value *s*, the goal is to find items with frequency greater than *s*N. K is set of frequently occurring items in sequence.

- Initialize K to empty.
- As we read an element from the sequence we either increment its count in the set K or insert it into the set with count 1.
- If at any time the size of K, crosses 1/*s* ie if |K| > 1/*s*, counts of all elements in the set K are decremented and elements whose count becomes zero are removed from the set.
- By taking another pass over the sequence, the elements which occur less than *s*N times in stream are removed from set K.
- K gives the set of items with support frequency greater than *s*.

After first pass the elements which are not in the set K have frequencies less than N*s*(Karp, Papadimitriou et al. 2003). However, not all elements present in set K after first pass are frequent. The false negatives are eliminated with a second pass. In the second pass occurrences of elements in K are counted and if the count of any element is less than *s*N then that element is removed from K. As the space of set K is always limited to 1/*s* the space complexity of the algorithm is O(1/*s*) and for the two passes the time complexity is O(N).

When applied to find frequent itemsets(Jin and Agrawal 2005) the algorithm faces following problems: Finding candidate k-itemsets with first pass and finding exact k-itemsets from set K after second pass require large amount of spaces. If each transaction is of length l, and if we are interested in k-itemsets then the space requirement would be $(1/s).^{l}C_{k}$ and for large values l, the space required would huge. In streaming environments a second pass over a stream is not feasible.

2.2. In-core Mining:

Based on the above algorithm(Karp, Papadimitriou et al. 2003) a new algorithm for finding frequent itemsets was proposed by (Jin and Agrawal 2005). (Jin and Agrawal 2005) uses (Karp, Papadimitriou et al. 2003) to find 2-length frequent itemsets and then apply apriori property to find k-length frequent itemsets. Let s ($0 < s < 1$) be the support threshold value and ε ($0 < \varepsilon < 1$) be a factor that determines the accuracy of the algorithm. A lattice L is maintained to maintain the frequent itemsets. L_i is the set of all i length frequent itemsets. L is the union of all L_is. $L = L_1 \cup L_2 \cup ..... \cup L_k$. A buffer T is maintained to store new transactions.

- Initialize L to empty.
- If a new transaction t arrives, first it is kept in a buffer T then counts of all subsets of t which are in lattice L are incremented by unit count.
- Any m-length subset p of t is inserted into L_m, under the following cases.
    - If $m \leq 2$ or
    - If $m > 2$ and all the immediate subsets of p (subsets of p with length m-1) are present in L_{m-1}.
- The size of L_1 is bounded by the number of literals. But the size of L_m(m>1) can go up to ${}^{|L_1|}C_m$. Therefore, the size of L_m is bounded by a certain threshold (S_{thr}). This is done by a process called Crossover.
- The counts of all itemsets in lattice are decremented by unity, and itemsets whose count becomes zero are deleted from lattice. This process is called Crossover.

The threshold S_{thr} value depends on how frequently we call routine Crossover. To have a bound on the accuracy, initial the number of times Crossover is invoked is reduced by a factor of $1/\varepsilon$. Therefore any itemsets with frequency greater than εsN are included in L. Before outputting itemsets with count less than $(1 - \varepsilon)sN$ are removed. If user asks for frequent itemsets at any instant of time, the itemsets in L are outputted.

2.3. FP-Stream Algorithm:

Stream management differs from traditional DBMS applications, as lifetime of stream may be very huge sometimes it is never ending and it is not possible to store the whole stream. Thereby we may have to merge or dissipate some of the data. This gives rise to *summary data structures* with small memories which can answer most of the queries posed upon a stream. Mostly users of applications, which produce stream data, are interested in recent changes at a fine granularity and changes in long term at a coarse granularity. A *summary data structure* called *Tilted Time Window* has been proposed to serve this purpose.

Tilted time window (Giannella, Han et al. 2002; MAIDS 2003) class is a general structure that does not enforce natural time boundaries. An example to a natural tilted time window with time granularities is: 15min, 1 hour, 24 hour, 31 days, 12 months...and so on. As new transactions arrive tilted time window tables will grow. In order to make tilted time tables compact a window construction strategy is employed. One of the most famous constructions of tilted time tables is on logarithmic time scale. If the current window holds frequencies of transactions in current quarter, next window holds frequencies in next two quarters and next window has next 4 quarters and so on. By this mechanism transactions of one year at the finest precision of one hour require $\log_2(365 \times 24) + 1 = 4$, and with finest granularity of one quarter $\log_2(365 \times 24 \times 4) + 1 = 17$ frequency counts.

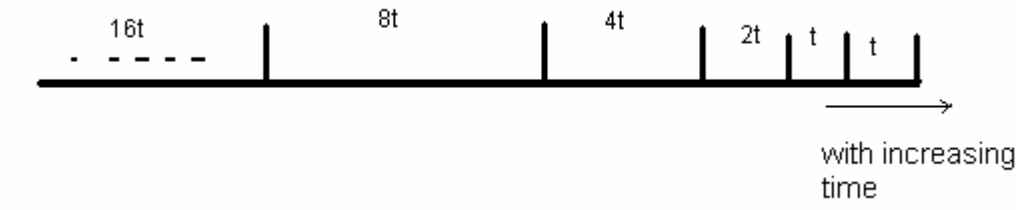

Figure 3 Tilted Time window (Logarothmic Construction)(Giannella, Han et al. 2002)

Stream is updated into tilted time windows in batches of fixed size. Each batch is identified with an integer identifier starting with unity. Let B_1, B_2,… B_n denote batches of

fixed sizes. B_n denotes the latest batch and B_1 denotes the oldest. $f_I(i, j)$ gives the frequency of itemset I in batches from B_i to B_j. At any instant of time there are $\lceil \log_2(n) \rceil + 1$ number of frequencies. The following sequence of frequencies is kept in tilted time windows.

f(n, n); *f*(n-1, n-1)[]; *f*(n-2, n-3)[]; *f*(n-4, n-7)[]; *f*(n-8,n-15)[];……

f(n, n) gives the finest granularity frequency ie the frequency in current batch. *f*(n-1, n-1) gives frequency in the next level of granularity. Every level has an intermediate window which is used in updating tilted time tables. […] denotes the intermediate window in a granularity level.

Updating tilted time window: If a new batch B_n arrives and the frequency of an itemset I in the current batch be *f*(B). *f*(B) replaces *f*(n, n) from the first level of granularity, and *f*(n. n) is shifted to the next level of granularity. If the intermediate window in the next granularity level is free then *f*(n-1, n-1) is shifted to the intermediate window. If the intermediate window is not free and carries a frequency *f*, then the total frequency *f*(n-1, n-1) + *f* is merged and shifted to next level of granularity. This process continues until there are no further shifts. For example:

Initially:	*f*(8, 8); *f*(7, 7)[]; *f*(6, 5)[]; *f*(4, 1)[];
When B_9 arrives:	*f*(9, 9);*f*(8, 8) [*f*(7, 7)]; *f*(6, 5)[]; *f*(4, 1)[];
When B_{10} arrives:	*f*(10, 10);*f*(9, 9)[];*f*(8, 7)[*f*(6, 5)]; *f*(4, 1)[];
	//merge *f*(8, 8) and *f*(7, 7) and *f*(8, 7) is shifted to next level
When B_{11} arrives:	*f*(11, 11);*f*(10, 10) [*f*(9, 9)]; *f*(8, 7)[*f*(6, 5)]; *f*(4, 1)[];
When B_{12} arrives:	*f*(12, 12);*f*(11, 11); *f*(10, 9); *f*(8, 5) [*f*(4, 1)];

Now coming to the FP-Stream algorithm(Giannella, Han et al. 2002), the central idea is to find frequent itemsets in every new batch with the help of FP-tree using FP-growth algorithm(Han, Pei et al. 2000), and update these frequencies in to tilted time window of FP-stream structure. Itemsets in FP-stream structure, whose approximate frequency in the given time period are greater than a threshold value are frequent in that given time period.

As, storing all the frequencies of all itemsets takes a huge amount of space, therefore some of the frequencies are pruned without loss of any knowledge.

Let t_0, t_1,…. t_n be the tilted time windows where t_n is the oldest. σ be the threshold support value and ε be the error rate. We are interested in finding frequent itemsets in time period from t_k to $t_{k'}$ or $T = t_k + t_{k+1} + \ldots..+t_{k'}$. Let w_i be the size of number of transactions in time interval t_i. $f_I(t_i)$ denotes frequency of itemset I in tilted time window *i*. One type of pruning is called tail pruning. In this, we drop frequencies $f_I(t_m)$,… $f_I(t_{k'})$ of an itemset I, where *m* satisfies the following condition:

$$\exists l, \forall i, l \leqslant i \leqslant n, f_I(t_i) < \sigma w_i \text{ and } \forall l', l \leqslant m \leqslant l' \leqslant n, \sum_{i=l}^{l'} f_I(t_i) < \epsilon \sum_{i=l}^{l'} w_i.$$

There is another type of pruning called Type-2 pruning. If I, I' are two itemsets and I is a subset of I'. If we are able to drop frequencies $f_I(t_m)$, $f_I(t_{m+1})$, …. $f_I(t_n)$ then we can also drop frequencies $f_{I'}(t_m)$, $f_{I'}(t_{m+1})$, …. $f_{I'}(t_n)$. Having known the types of frequency pruning, let us look at the steps in algorithm.

Algorithm maintains an FP-Stream structure(Han, Pei et al. 2000; Borgelt 2005). The parameters σ support threshold and ε the error rate are specified by user.

- Initialize a FP-tree to empty.
- If a new batch B of transactions gets accumulated, frequent itemsets in this batch are found by using FP-tree structure and following FP-growth algorithm(Han, Pei et al. 2000; Borgelt 2005).
- For every frequent itemset I found, there are two possible cases: If I is in the FP-Stream structure then, $f_I(B)$ (frequency of I in batch B) is added to the tilted time window of I in FP-Stream, and tail pruning on frequencies of I is conducted. If after tail pruning, the tilted time table of I is empty then the supersets of I are not mined (Type-2 pruning).
- If I is not in the FP-Stream structure and if $f_I(B) \geq \varepsilon |B|$ then I is inserted into FP-Stream structure with $f_I(B)$ as the only element in I's tilted time window.

- After all itemsets processing all itemsets, FP-Stream is scanned in depth first search manner for itemsets I, which are not updated by the new batch, and 0 is inserted into I's tilted time windows and tail pruning is done.
- In the process of scanning in depth first search manner, if any leaf's tilted time window is empty then that leaf is dropped from the FP-Stream structure. Scanning is proceeded to the siblings of dropped leaf if it does not have any siblings then its parent is scanned. If all children of a parent node are dropped then the parent node becomes the leaf.

FP-stream efficiently maintains frequent itemsets with approximate support estimation and with low memory resources.

3. Approximate Mining Frequent Itemset Mining:

Data streams are continuous and high speed flow of data. Stream management differs from traditional DBMS applications, as lifetime of stream may be very huge sometimes it is never ending and it is not possible to store the whole stream. Thereby we may have to merge or dissipate some of the data. This gives rise to *summary data structures* with small memories which can answer most of the queries posed upon a stream. Also we can not have as many passes over the streaming data sets as they are huge in number, sometimes "we may get only one look". As said above it is not possible to store the whole stream and therefore we have *summary data structures*, however high-quality approximate answers are often acceptable in lieu of exact answers. By taking such approximate data structures we also have to give bounds on the support of each pattern. Many algorithms have been proposed for approximate mining of frequent itemsets in recent years like AP_{stream} Algorithm by (Silvestri and Orlando 2005), random sampling, histograms, wavelets. There are differences storing the concise data among these algorithms. We will discuss all these algorithms according to differences in their methodologies (probabilistic, deterministic, etc) and different *summary data structures* they use.

3.1. Sampling:

In some data streaming applications data arrival rates are greater than, algorithm execution rates. Therefore some of the datasets have to be skipped without processing, so that query is processed over a sample of data rather than entire data. By this way approximate answers can be produced, but in some cases confidence bounds like maximum allowable degree of error in answers can be provided.

The central idea of sampling is frequent items and their support can be estimated by a good sample of data. Counts of each itemset are stored in a data structure. As one sample rate can not handle a potentially infinite stream. Decrease sample rate to handle more and more new data. As sample rate changes also change counts of itemsets accordingly.

Algorithm accepts three inputs, support s, error ε (ε << s) and probability of failure δ. Let N denote the current length of stream. At any point of time algorithm can be asked to produce frequent itemsets along their estimated frequencies. The answers produced will have the following guarantees:

- All itemsets whose true frequency exceeds sN are outputted.
- No itemset with true frequency less than (s-ε)N will be in output.
- Estimated frequencies are less than the true frequencies by at most εN.

If user specifies minimum support s to be 0.2% and error to be 10% of s ie, 0.02% then itemsets with actual support frequency greater than 0.2% will be in output. Itemsets with actual support count less than 0.18% (0.2 – 0.02) will not be in output. The estimated frequencies of itemsets will have a minimum error of 0.02%. Such algorithms which satisfies these three properties are called ε-deficient synopsis(Manku and Motwani 2002).

Algorithm follows the following steps:

- Maintain a set S of entries (e, f), where e is a singleton item and f is estimated frequency.
- Initially S is empty. Sampling rate r is unity.
- If an incoming item e is already present in S, then we increment its estimated frequency by unit count, otherwise we sample it with rate r.
- Sampling an item with rate r means, the item is selected with probability 1/r.
- After sampling it the item is selected, a new entry (e, 1) is inserted into S. If item e is not selected after sampling then it is discarded.
- The sampling rate is changed with over time as follows:
    - Let $t = [\log (s^{-1}\delta^{-1})] / \varepsilon$.
    - The first 2t elements are sampled with rate r = 1. Next 2t elements are sampled with sampling rate r = 2, the next 4t elements with r = 4, and so on till the life time of stream.
- Counts of each items has to be handled according to the change in the sampling rate. For each item, diminish count by unit count with a probability that follows a geometric distribution.
- If any items f becomes zero, it is pruned from S.

The above Sampling algorithm computes ε-deficient synopsis with probability 1-δ using at most $[2\log (s^{-1}\delta^{-1})] / \varepsilon$ number of entries(Manku and Motwani 2002). At any time, the items whose estimated frequencies are greater than $(\sigma - \varepsilon) N$ gives the frequent itemsets with threshold support s.

This algorithm is called *Sticky Sampling* algorithm, as it sticks elements in a stream if it already has an entry in S. Space complexity of Sticky sampling is independent of N.

In many situations sampling based approaches can not give reliable approximation guarantees, for example queries involving joins(Babcock B., Babu S. et al. 2002). Sticky

sampling performs worse in many cases like zipfian distributions(Charikar, Chen et al. 2002), because of its tendency to remember every unique element that gets sampled(Manku and Motwani 2002). In the process of skipping new data, sticky sampling might skip items which are frequent after a long time the algorithm started. Sampling methods are not applicable in finding maximum item in a stream.

Many different sampling methods have been proposed: domain sampling, universe sampling, distinct sampling, reservoir sampling etc. Different sampling algorithms can be found in (Badcock, Datar et al.; Toivonen 1996; Gibbons and Matias 1998; Babcock B., Babu S. et al. 2002; Muthukrishnan 2003; Gaber, Zaslavsky et al. 2005).

3.2. Lossy Counting Algorithm:

Lossy counting was proposed by (Manku and Motwani 2002). Lossy counting is a deterministic algorithm that computes frequency counts over a stream of singleton items. In Lossy counting algorithm , the set of the frequent itemsets in a data stream is found when minimum support s and an error parameter ε are given. The data stream is loaded into main memory and it is processed in batches. The exact current counts of all single items in the data stream are maintained main memory separately. Let N denote the number of transactions seen so far. The incoming stream is theoretically divided into buckets of size w = $\lceil 1/\varepsilon \rceil$. Each bucket is given a label starting from 1. The latest bucket $b_{current}$ has label $\lceil N/\varepsilon \rceil$. A data structure D, is maintained to store estimated frequencies of items. Each entry in D has the form (e, f, Δ), where e is an element in the stream, f is estimated frequency, and Δ represents maximum possible error in f.

- Initially D is empty. Let a new item e arrives.
- When there is no entry corresponding to e in D, then a new entry (e, 1, $b_{current}$-1) is inserted into D.
- If the element e is already present in D, then the frequency count is increased by unit count.

- After each bucket of transactions ie, after $\lceil 1/\varepsilon \rceil$ number of transactions, items are pruned if $f + \Delta \leq b_{current}$.
- Justification for having Δ is if an item is not found in D, then the maximum number of times it is pruned from D is $b_{current}-1$, because D is subjected to item pruning for $b_{current}-1$ times. Therefore if a new item is inserted the maximum possible error in its frequency is $b_{current}-1$, and once the entry is inserted Δ is invariable.

If an item e does not appear in D then its true frequency $f_e \leq \varepsilon N$ (Manku and Motwani 2002). Lossy counting computes an ε-deficient synopsis using at most $[\log(\varepsilon N)]/\varepsilon$ entries(Manku and Motwani 2002). Frequent items with minimum threshold s, at any point of time are entries in D, whose estimated frequency $f \geq (s - \varepsilon)N$.

Lossy counting requires space that grows logarithmically with N. Lossy counting performs better than Sticky sampling(Manku and Motwani 2002). Lossy counting can not answer queries like frequent items in a particular time period. Lossy counting same as Sticky sampling can not find recent frequent itemsets accurately over data, because the maximum allowable error increases as the bucket count increases. Items, which are frequent recently and did not, occurred in any of the transactions before, and then their approximate frequency is estimated with high error ($b_{current}-1$). Other references to Lossy counting can be found in (Chang and Lee 2004; Cormode and Muthukrishnan 2005).

Some more algorithms have been proposed by extending Lossy Counting, some of them are mentioned below.

3.3. Frequent Itemsets Algorithm:

We now discuss a Lossy counting based algorithm that finds frequency counts over a data stream of itemsets. Frequent Itemsets algorithm was first proposed by (Manku and Motwani 2002). The basic difference between Lossy counting and Frequent Itemset

algorithms is, lossy counting processes each data set at a time, but the latter processes batches of transactions at a time.

We follow same notations of Lossy counting here too. Entries in data structure have the form (*set*, f, Δ), where *set* is a subset of items, f is estimated approximate frequency and Δ represents maximum allowable error in f. Let N be the number of transactions from stream seen so far. The algorithm takes two user specified parameters as inputs, support s and error ε.

Algorithm follows the following steps:

- Available memory is filled with as many transactions as possible. Then these transactions are divided into buckets of equal size $w = \lceil 1/\varepsilon \rceil$. The space available in main memory varies with respect to time. Let β be the number of buckets present in memory at any instant of time.
- Each bucket is labeled with an integer starting from 1. $b_{current}$ gives the current bucket id and is equal to $\lceil N/\varepsilon \rceil$.
- Data structure D is initially empty.
- For each entry (*set*, f, Δ) in D, the number of occurrences of *set* in the current bucket are counted, and corresponding f is updated accordingly. If the updated entry satisfies the condition $f + \Delta \leq b_{current}$ then it is pruned form D.
- If a *set* whose frequency in the current batch is greater than β then a new entry (*set*, f, $b_{current}$-β) is inserted into D.
- If in above two steps, if a *set* is does not make into D, then all the supersets of *set* need not to be considered.

Similar to Lossy counting, if (*set*, f, Δ) Є D then the actual true frequency of *set* f_{set} satisfies the condition $f_{set} \geq \varepsilon N$. *set*s whose frequency $f \geq (s - \varepsilon) N$ are frequent itemsets at any instant of time(Manku and Motwani 2002).

The value β should be large enough so as to prune spurious itemsets, because itemsets whose occurrence is β+1 times or more, will have an entry in D. Value of β depends on the available memory and which varies with time. In this algorithm in each bucket, transactions are ordered in lexicographic order, and this step is a bottle neck in terms of time.

3.4. AP-stream Algorithm:

There is another new approximate mining algorithm AP_{stream} for mining frequent patterns over stream data, and is proposed by (Silvestri and Orlando 2005; Orlando, Perego et al. 2006). Basic idea of the algorithm is to infer frequencies of previously infrequent itemsets by interpolating present support values. As interpolation is an approximation and therefore upper and lower bounds should be returned too.

AP_{stream} is based on Partition Algorithm, which partitions dataset into different partitions and calculates local frequent itemsets, and then merges these local frequent itemsets to get global answer. The set of all locally frequent patterns is a superset of global solution. Process each partition to get local frequent itemsets. With a second pass over the data to remove all the false positives from this set of local frequent itemsets or the superset of global answer. As we can not store all the streaming data, the second pass of partition algorithm is restricted to limited datasets.

AP_{stream} (Silvestri and Orlando 2005; Orlando, Perego et al. 2006) extends Partition algorithm, and tries to remove problems faced by it. Let x be an itemset. D_i gives i^{th} partition of data steram, and σ_i gives support of an itemset in i^{th} partition. $\sigma_{[1,i)}$ gives support of an itemset in all partitions from [1,i) taken collectively. Let an itemset x is not frequent in a partition D_i, or $\sigma_i(x) < minsup.|D_i|$, then the count of x in i^{th} will not be in our knowledge as it would not be in our superset of global solution. AP_{stream} tackles this skew by interpolating present frequency to previous support. Possible cases are:

$\sigma_{[1,i)}(x)$ (Past support)	$\sigma_i(x)$ (Recent support)	Action
Known	Known	Sum recent support to past support and bounds.
Known	Unknown	Recount support on recent, still available, data.
Unknown	Known	Interpolate past support.

Table 1: Three possible cases in interpolating

In the first case, the new support $\sigma_{[1,i]}(x)$ is just sum of $\sigma_{[1,i)}(x)$ and $\sigma_i(x)$. If $\sigma_{[1,i)}(x)$ was approximated then the bounds to error in $\sigma_{[1,i]}(x)$ remain same as there is no error in $\sigma_i(x)$. In the second case, we just have to look at new partition to get $\sigma_i(x)$. The most interesting is the third case, where an itemset was not frequent and it became frequent in present partition. In this case, we need to approximate the past support so as to find the complete support $\sigma_{[1,i]}(x)$. The algorithm interpolates $\sigma_{[1,i)}(x)$ as follows:

$$\sigma_{[1,i)}(x)^{interp} = \sigma_i(x) * \min\left(\left\{\min\left\{\frac{\sigma_{[1,i)}(item)}{\sigma_i(item)}, \frac{\sigma_{[1,i)}(x \setminus item)^{interp}}{\sigma_i(x \setminus item)}\right\} \middle| item \in x\right\}\right)$$

In the previous formula the result of the first *min* is the minimum among the ratios of supports of items contained in patterns x in past and recent data and the same values computed for itemsets obtained from x removing one of its items.

For example, in a stream with items {a, b, c} let itemset 'abc' was not frequent for some partitions and became frequent afterwards. Then $\sigma_{[1,i)}(abc)$ is found in the following way.

x	$\sigma_i(x)$	$\sigma_{[1,i)}(x)^{interp}$	$\sigma_{[1,i)}(x)^{interp} / \sigma_i(x)$
abc	6	-	-
ab	12	50	4.2
ac	19	30	1.6
bc	20	100	5
a	25	160	6.4
b	30	140	4.7
c	25	160	6.4

Table 2: Example for interpolating frequency

According to the above formula minimum value is 1.6. Calculating $\sigma_{[1,i)}(abc)^{intrep}$= 6*1.6 = 9.2. We should also give bounds to the support that we are guessing with by the above formula. Each support can not be negative. If a pattern was not frequent in a time interval then its interpolated support should be less than minimum support threshold for the same interval. Support values should follow apriori property: no itemset should have more support than any of its subset. If a subpattern is missing during interpolation of a itemset it means it has been examined during a previous level and discarded. In that case all its supersets can be discarded as well.

AP_{stream} can very efficiently find recent frequent itemsets but not the itemsets which were frequent at past time.

4. Mining Sequential Patterns:

The input is a set of sequences, called data sequences, each data sequence is set of transactions and each transaction is a set of items. All transactions are attached with transaction-timestamps. Support of a sequential pattern is the fraction of transactions in which it is present. Mining for sequential patterns implies finding sequential patterns whose support is at least greater than user specified threshold. Mining sequential patterns

has a great number of applications in real world. For example in web-logs, logs with userID and the page requested by that user are stored. Each data sequence consists of sets of requests of web pages by different users with attached time stamps. By mining sequential patterns from web-logs we will be able to say which pages are visited consecutively? which web pages are revisited again and again? By predicting sequence of user requests of web pages we can intelligently manage our network frame work to reduce the traffic. The knowledge from data streams in medical treatments like the oxygen consumption, chest volume and heart rate, may indicate or predicate the patient's situation, by discovering knowledge from different sensor data streams, sensors can automatically update themselves for efficient management of data flow.

There are extensive research works for mining frequent itemsets and association rules from stream data. There is a little research in mining sequential patterns in data streams. Sequential pattern mining was first introduced by (Srikant and Agrawal 1996). We now discuss some of the sequential pattern mining algorithms in data streams.

4.1. SMDS:

SMDS (Sequence Mining in Data Streams)(Marascu and Masseglia 2005) is based on sequence alignment of approximate sequential patterns in data streams. Let I = $\{i_1, i_2,\ldots i_m\}$ be a set of m items. A sequence is an ordered list of itemsets. Typically a data sequence is denoted by $<s_1, s_2,\ldots s_n>$. A data sequence $<a_1, a_2,\ldots a_m>$ is a subsequence of another data sequence $<b_1, b_2, \ldots b_k>$ if there exists a sequence of integers $i_1 < i_2 < \ldots < i_n$ such that $a_1 b_{i1}, a_1 b_{i2}, \ldots, a_m b_{im}$. If a data sequence is of the form S = < (a) (be) (f) (g) >, then the itemsets in this sequence are a, be, f and g.

- Stream is processed in batches of fixed sizes. For each batch of transactions frequent sequential patterns are extracted and stored in a prefix tree.
- The data sequences are first clustered according to the similarity between two sequences.
- If s_1 and s_2 are two data sequences, then LCS[s_1, s_2] represent the longest common subsequence of $s_1$and s_2.

- Similarity between two sequences $s_1$and s_2, denoted by sim(s_1, s_2) and is found by:

$$sim = \frac{2 \times |LCS(s_1, s_2)|}{length(s_1) + length(s_2)}.$$

- Algorithm is initiated with one cluster containing the first transaction. Each new transaction is compared with each cluster and is inserted into a cluster c if there exists $s_c \in c$ such that sim(s, s_c) ≤ minSim, where minSim is specified by user.
- If no such cluster satisfying the above condition exists then a new cluster is made with this new sequence as the only sequence in it.
- After clustering a batch of sequences, for every cluster c, first two data sequences in c are aligned. After the second sequence each other sequence is aligned with the previously aligned sequence (will be clear after we take an example). This process is called summarizing a cluster.
- Each aligned sequence is represented in a weighted sequence as follows:
  SA = <I_1:n_1, I_2:n_2, …. I_m:n_m>, where I_k is an itemset, n_k is number of occurrences of I_k in alignment. I_k has the form <x_{i1} :m_{i1}, x_{i2} :m_{i2}, …, x_{it} :m_{it}> m_{it} is the number of sequences containing the item x_{ip} at the p^{th} position in the aligned sequence.
- All sequences belonging to a cluster are aligned, to get one aligned sequence in the above form, and if this aligned sequence is not present in the prefix tree data structure then it is inserted into it, otherwise its count is incremented by unit count.
- Prefix tree maintains tilted time windows(Giannella, Han et al. 2002) for each itemset, and after each batch of transactions Tailpruning (Giannella, Han et al. 2003) is done so as to drop frequencies from fine granularities so as to store least number of frequencies and without loosing any knowledge.
- Prefix tree has a null root. Each path from the root to any node in tree gives a sequence. Each node has the support values in corresponding to its sequence (example to prefix tree structure is given below).

Let us take an example to show how sequences are aligned in a batch. Consider the following sequences: S_1 = < (a,c) (e) (m,n) >, S_2 =< (a,d) (e) (h) (m,n) >, S_3 =< (a,b) (e) (i,j) (m) >, S_4 =< (b) (e) (h,i) (m) >. At first S_1 is inserted into an empty set and then S_1 and S_2 are aligned to get SA_{12}, and then SA_{12} and S_3 are aligned to get SA_{13}, after that SA_{13} and S_4 are aligned to get SA_{14}. The procedure is explained in the figure.

Step 1 :				
S_1:	<(a,c)	(e)	()	(m,n)>
S_2:	<(a,d)	(e)	(h)	(m,n)>
SA_{12}:	(a:2, c:1, d:1):2	(e:2):2	(h:1):1	(m:2, n:2):2
Step 2 :				
SA_{12}:	(a:2, c:1, d:1):2	(e:2):2	(h:1):1	(m:2, n:2):2
S_3:	<(a,b)	(e)	(i,j)	(m)>
SA_{13}:	(a:3, b:1, c:1, d:1):3	(e:3):3	(h:1, i:1, j:1):2	(m:3, n:2):3
Step 3 :				
SA_{13}:	(a:3, b:1, c:1, d:1):3	(e:3):3	(h:1, i:1, j:1):2	(m:3, n:2):3
S_4:	<(b)	(e)	(h,i)	(m)>
SA_{14}:	(a:3, b:2, c:1, d:1):4	(e:4):4	(h:2, i:2, j:1):3	(m:4, n:2):4

Figure 4 Steps in Sequence Alignment (Marascu and Masseglia 2005)

SA_{14} is the summary of the corresponding cluster. To get frequent sequential patterns, summary of each cluster can be filtered with a user specified minimum support. If sequence SA_{14} is filtered with minimum support 2, then we obtain sequence <(ab)(e)(hi)(mn)>. Prefix tree for sequences (<(a c)>, <(a d)>, <(b)>, <(c d)>, <(c)(e)>, <(d)(a)>) is as follows:

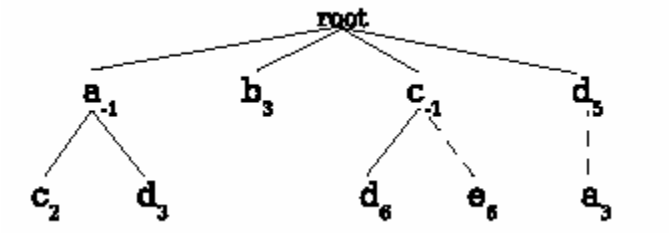

Figure 5 Prefix Tree Construction (Marascu and Masseglia 2005)

Let us consider the sequence <(a)>, the support of sequence a is -1, which means that sequence a is extracted from more than one cluster. Consider the sequence <(d)(a)>, the support of this sequence is 3 meaning it was filtered with minimum support of 3. The dotted lines in tree (between c and e and between d and a), shows that the two sequences (<(c)(a)>, <(d)(a)>) are not same.

SMDS processes stream in batches. The number of candidates generated in any batch can be huge in number, leading to a blocking operator. Related works on sequential patterns can be seen in (Agrawal and Srikant 1995; Srikant and Agrawal 1996; Zheng, Xu et al. 2002; Chen, and et al. 2005; Yu and Chen 2005; Jiang and Gruenwald 2006).

There exists an algorithm which mines sequential patterns over multiple data streams proposed by (Chen, Wu et al. 2005). They have proposed a recursive algorithm MILE which makes new patterns with the knowledge of existing patterns.

5. Mining k-Top Frequent Items:

Mining frequent items in a data stream has its applications iceberg queries(Fang, Shivakumar et al. 1998; Manku and Motwani 2002), iceberg cubes (Manku and Motwani 2002) and also in finding association rules. Frequent items in market data are highly influential in decision making. There are other areas such as data warehousing, and information retrieval where frequent items find applications. Knowing frequent items is very useful in skew data. In many applications stated above, the frequency count of items change quite rapidly. Keeping track of frequent data also arises in application domains outside traditional databases. For example, in telecommunication networks such as the Internet telephone, it is of great importance for network operators to see meaningful statistics about the operation of the network.

Let us assume that we are looking at a data stream of transactions, and at any time we want to find top k frequently occurring items. Saying in the formal words, Let S be a

sequence transactions. Each transaction is a subset of a set of items O = $\{i_1, i_2, \ldots i_t\}$. Then S = $\{T_1, T_2, \ldots T_n\}$ and each $T_i \in O$. The frequency of an item i_k is defined as the ratio of number of transactions it appears in and to the total number of transactions. Now if items are arranged in decreasing order of their frequencies, then we are interested in first k items of that ordered list.

Some of the algorithms that find top k frequent items and majority items (Boyer and Moore 1982; Fischer and Salzberg 1982; Misra and Gries 1982; Gilbert, Guha et al. 2002). We now discuss some new algorithms for finding k-top candidate items in data streams.

5.1. Count Sketch:

In algorithm Count Sketch (Charikar, Chen et al. 2002), a randomized sketch structure is constructed and updated which guarantees with high probability that the count of an item can be approximated up to a value which is function of the frequencies of infrequent items. A heap is maintained to storage frequent items. If a new frequent item arrives and the heap is full, then the item with least frequency is removed from heap and the new item is inserted.

Given three parameters data stream S, an integer k(the number of top items) and a real number ε, if n_m denotes the count of an element *m* in the stream then (Charikar, Chen et al. 2002) finds a list of k-elements from stream S, such that every element i in the list has $n_i > (1-\varepsilon)\, n_k$. t and b be two parameters depending on ε. Let $h_1, h_2 \ldots h_t$ be hash functions from objects to $\{1, 2, \ldots, b\}$(means h_i maps any object p to $m \in [1,b]$) and $s_1, s_2, .. s_t$ be hash functions from objects to {1, -1}. These hash functions can be interpreted as t hash functions each containing b buckets.

- For each transaction q in data stream, update hash table as $h_i[q] \mathrel{+}= s_i[q]$ for all $i \in [1, t]$.

- If that transaction q is in heap increment its count.
- Otherwise add q to heap if median$\{h_i[q].s_i[q]\}$ i ∈ [1, t] is greater than the smallest estimated count in heap and eject the smallest estimated count from the heap.

The value b depends on ε. t is taken as O(log(n/δ)), where δ the maximum probability with which algorithm fails. To estimate frequency of an element median is used because median is sufficiently robust to eliminate collisions of high frequency elements. By adopting this algorithm we can also find items whose frequencies change the most(Charikar, Chen et al. 2002).

5.2. Finding Hot Items in Dynamic Streams:

(Cormode and Muthukrishnan 2005) have proposed an algorithm based on group testing for finding k top most frequent items in dynamic data streams (transactions either insert or delete an itemset). The algorithm deterministically maintains the majority item in data stream and then finds frequent items with group testing (explained later in detail). The basic approach of this algorithm is: each item belongs to a different subset of items. The count of a subset is incremented or decremented by unity whether an item belonging to that subset is inserted or deleted. If a subset count exceeds a certain threshold then there will be a frequent item in that subset otherwise there are no frequent items in that subset. Now get information from all the subsets to find the global solution. This is exactly is idea of group testing: to arrange a number of test, each of which groups together a number of the *m* items in order to find up to *k* items which test "positive".

Without loss of generality the items are represented with integers 1,2,….m. The items are mapped onto integers 1,2,…W with a family of hash functions. Let P (P>m>W) be a prime number, define a family of hash functions $f_{a,b}$ where a and b ranges from [0, P-1]. Then $f_{a,b}(x)$ = ((ax+b) mod P) mod W). The non adaptive group testing data structure is initialized with two parameters W and T and its other components are:*c* is a three dimensional array of counters, of dimensions TxWx(logm + 1), universal hash functions

h, defined by a[1,..T] and b[1,…T]. The values c[i][j][0] are used to store the counts of subsets $G_{i,j}$. Each subset $G_{i,j}$ has items $G_{i,j,l} = \{x| x \in G_{i,j} \text{ and } bit(x,l) = 1\}$. bit(x, j) gives the binary digit at position j, when x is converted into binary number. Initially all values in c are initialized to zero. a[1,..T] and b[1,…T] are initialized with uniformly random in the range [0, P-1]. If we are looking for top k frequent items, then we need to find items whose occurrences are greater than n / (k+1).

- If a input item x, it is determined to which subset it belongs to and the appropriate *log(m)* counters are updated based on bit representation of x. If transaction is an insertion then counters are incremented by unit count, otherwise if it is a deletion then decremented by unit count. The item count is also appropriately incremented or decremented by one.

```
If (insertion)
        d = 1;          // if item x is inserted then make d as 1.
else
        d = -1;         // if x is deleted then make d as -1.
For i = 1 to T
        c[i][h_i(x)][0] = c[i][h_i(x)][0] + d;
                // increment or decrement the count accordingly.
        For j = 1 to log(m)
                c[i][h_i(x)][j] = c[i][h_i(x)][j] + bit(x, j) * d
                        // the counters are updated at the respective 1bits in
                        // the binary form of x.
```

- To find frequent items, each group is checked whether there exists any frequent item in it or not. If the count of a subgroup is less than n/(k+1) [minimum frequency of top k frequent items], then there exists no frequent item in that subgroup(group testing). Discard those subgroups $G_{i,j}$ if c[i][j][0] $\leq$ n/(k+1).
- For an item to be frequent should satisfy the following conditions:

  - The candidate item found in that group must belong to that group ie $h_i(x) = j$.
  - The occurrences of frequent item in every group must be greater than the threshold, ie, $c[i][h_i(x)][0] > n/(k+1)$ for all i.
- Frequent items should give "positive" to the above tests (group testing).

Choosing $W \geq 2k$ and $T = \log_2(k/\delta)$, for a user specified parameter δ ensures that the probability of all hot items being output is at least 1- δ.(Cormode and Muthukrishnan 2005).

6. Adaptive Mining:

In stream data mining, data arrives sequentially and processed by an online algorithm whose workspace is limited. The system has no control over the order in which data elements arrive to be processed, either within a data stream or across data streams. Most of the real world data stream applications generate data with very high speeds. For example, network traffic management systems for large network. Such systems monitor a variety of continuous data streams that may be characterized as unpredictable and arriving at a high rate, including both packet traces and network performance measurements. Mining data streams has to be online, and we may only get a single look at this high speed data. Therefore our mining technique should adapt to the data flow. There have been number of techniques to approximately mine data streams(Manku and Motwani 2002; Silvestri and Orlando 2005), methods to merge knowledge into small data structures like tilted time windows(Giannella, Han et al. 2002; Marascu and Masseglia 2005). Here we present some adaptive techniques for mining data streams.

6.1. Algorithm Granularity based Mining Techniques:

Our aim is to increase the accuracy of output of an algorithm with limited availability of resources like memory, processing speed. Algorithm output granularity(AG) is defined as

the amount of generated results kept in memory before doing incremental integration in order to catch up with the high data rate(Gaber, Krishnaswamy et al. 2003; Gaber, Krishnaswamy et al. 2004).

- The algorithm rate (AR) is function of data rate (DR). ie AR = *f*(DR).
- The time needed to fill the memory (TM) is dependent on algorithm rate (AR).
- The algorithm accuracy is function in TM. That is if the time needed to fill the available memory is enough to the algorithm at the highest data rate without any sampling or algorithm granularity, this would be the best solution.

Therefore higher the algorithm granularity, the more accurate the algorithm output will be(Gaber, Krishnaswamy et al. 2003).

6.2. Light Weight Frequent Items (LWF):

The algorithm(Gaber, Krishnaswamy et al. 2003) outputs frequent items over a data stream. By the definition of AG, AG represents the number of frequent items that the algorithm can calculate and the some counters that the algorithm uses. Algorithm follows the following steps:

- Data elements are processed one by one.
- When a new element arrives it is looked up in data structure and if the look up is successful then the counter allocated to that item is incremented by unity.
- If the lookup is a failure, then if any of the counters are free then, new item is inserted with counter 1.
- If the item is new and all the counters are full, now if the present time is more than a threshold time then a number of items which are least in count are pruned and the new item is inserted.
- But if the algorithm time is not greater than a threshold value, then the new items are dropped till a threshold time.

6.3. Finding Recent Frequent Itemsets Adaptively:

(Chang and Lee 2003) have proposed an adaptive algorithm to find recent frequent patterns in a data stream. The algorithm diminishes effect of old transactions on the present knowledge thereby effectively mining recent frequent itemsets. The basic ideas behind the (Chang and Lee 2003) algorithm are:

1) All itemsets are not frequent at all times. We need not to monitor itemsets whose occurrences are much less than a predefined value since it can not be frequent in near future.
2) Insertion of any itemset into a *monitoring lattice* is done only when the itemset becomes frequent enough.
3) The effect of past transactions should be decayed so as to efficiently find recent frequent itemsets.
4) If an itemset in *monitoring lattice* (it means it was frequent in past time) becomes less significant or its recent frequency is much less than a specified value then it has to be pruned from *monitoring lattice*.

Before explaining the algorithm let us look at some preliminary derivations:

The upper bound for frequency count of any itemset is the minimum of the frequency count of its immediate subsets. Let C(e) denote count of itemset e, then

$C^{max}(e) = \min(C(e'))$ where e' is immediate subset of e. (If e is of length n, then subsets of e with length n-1 are called its immediate subsets).

Let $C(e_1 U e_2)$ be the count of transactions in which at least one of the itemsets e_1 or e_2 occur. The minimum value $C^{min}(e_1 U e_2)$ as follows

$$C^{min}(e_1 \cup e_2) = \begin{cases} Max[\,0, C(e_1) + C(e_2) - C(e_1 \cap e_2)] & \text{if } e_1 \cap e_2 \neq \Theta \\ Max[\,0, C(e_1) + C(e_2) - |D|] & \text{if } e_1 \cap e_2 = \Theta \end{cases}$$

where |D| is the total number of transactions in D.

The minimum count of an itemset can be found as follows:

$C^{min}(e) = \max(\{C^{min}(s_i \cup s_j)\} \mid$ for all s_i, s_j immediate subsets of e).

As we have lower and upper bounds for counts of an itemset therefore, the difference between them gives the *estimated error*.

Having some basic idea, let us get back to the algorithm. The algorithm maintains a data structure called *monitoring lattice* in which maintains entries of the form (cnt, err, MRtid) for its corresponding itemset e. d is a given decay rate. D denotes the stream data. |D| denotes the number of transactions. Each transaction is provided with a transaction id which starts from one. k denotes the current transaction id.

- Initialize monitoring lattice to empty.
- For each new transaction T_k in data stream total number of transactions |D| is updated as

  $$|D|_k = |D|_{k-1} * d + 1$$
- The itemsets in monitoring lattice (cnt_{pre}, err_{pre}, $MRtid_{pre}$), that are present in the new transaction are updated to (cnt_k, err_k, $MRtid_k$)as follows:

  $$cnt_k = cnt_{pre} * d^{(k - MRtid_{pre})} + 1.$$
  $$err_k = err_{pre} * d^{(k - MRtid_{pre})}$$
  $$MRtid_k = k.$$
- As the cnt of itemsets in monitoring lattice changes, we prune those itemsets whose count values (cnt_k) are less than a specified threshold. This threshold value should be less than minimum support S_{min} specified by user.

- If any new 1-itemset (itemset with unit length) appears in transaction T_k, then it is immediately inserted into monitoring lattice with corresponding entry (1, 0, k)[cnt = 1, err = 0, MRtid = k].
- For any n-itemset (itemset with length n) ($n \geq 2$), if the estimated support ($cnt/|D_k|$) is greater than a specified threshold S_{ins}, then it is inserted into monitoring lattice with corresponding entry values, cnt = $C^{max}(e)$, err = $C^{max}(e) - C^{min}(e)$, MRtid = k.
- An itemset in monitoring lattice is frequent if its support is greater than a predefined minimum support S_{min},

  Support is given by $[cnt * d^{(k - MRtid)}] / |D|_k$

  Error in support is given by $[err * d^{(k - MRtid)}] / |D|_k$.

The complete details of the algorithm can be found in (Chang and Lee 2003).

7. Other Related Algorithms:

7.1. Mining Closed Frequent Itemsets:

An itemset is closed if none of its proper superset has the same support as it has. Frequent itemsets can be derived from the set of closed itemset as every frequent itemset *I* must be a subset of one or more closed frequent itemset, and *I*'s support is equal to the maximal support of these closed itemsets that contain *I*. An algorithm for finding closed frequent itemsets has been proposed by (Chi, Wang et al. 2004).

A *Closed Enumeration Tree* similar to prefix tree maintains closed frequent itemsets and itemsets that form a boundary between closed frequent itemsets and other itemsets. A sliding window slides maintains recent itemsets. The closed enumeration tree is incrementally updated(Chi, Wang et al. 2004) when newly arrived transactions change the content of the sliding window. Other related works for mining closed itemsets are (Zaki and Charm 2002).

7.2. Path Traversal Patterns:

Traversal patterns can be best explained with an example of web mining. Consider a network monitoring application which maintains the web page addresses that are visited by an internet user in the exact order that the user visits. These patterns will have forward as well as backward references. A backward reference means that page is revisited by the same user. A maximal forward reference means a forward reference sequence without any backward references. Finding frequent traversal patterns can be viewed as finding frequent maximal forward references.

(Li, Lee et al. 2004) have proposed an algorithm for mining frequent traversal patterns in a data stream. This algorithm is derived from FP-growth algorithm(Han, Pei et al. 2000). Algorithm maintains a Stream Header table and Stream FP-tree. Header table contains starting items of a pattern. The web click sequences by all users are divided into maximal forward references. These maximal forward references are then inserted into Stream FP-tree with starting of pattern same as the header item of tree.

Other frequent tree pattern mining algorithm is proposed by (Hsieh, Wu et al. 2006).

7.3. Regression Based Mining:

Discovering temporal relations among huge databases has its applications in market basket analysis, for understanding customer behaviors by finding frequent sets of items in market transactions. Most of the methods designed for mining frequent sets in traditional databases can not be directly applied to mine data streams due to several obvious reasons. A regression based scheme is given by (Teng, Chen et al. 2003) for mining temporal pattern mining from data streams. FTP-DS(Frequent temporal patterns of Data streams) algorithm, proposed by (Teng, Chen et al. 2003) analyses synopsis of frequency variations of frequent temporal patterns by a regression. It processes stream slot by slot.

The data of new slot is scanned with the previous candidate patterns and a set of candidate patterns for next time slot is created.

There are many other algorithms related to frequent pattern mining in data streams, like Hierarchical Heavy hitters (Cormode, Korn et al. 2003), Sliding window queries over data streams (Hammad, Aref et al. 2003) Load shedding for Aggregation queries (Babcock B., Babu S. et al. 2002; Babcock, MayurDatar et al. 2004). Research issues regarding stream mining can be found in (Domingos and Hulten 2001; Aggarwal 2002; Babcock B., Babu S. et al. 2002; Ganti, Gehrke et al. 2002; Garofalakis, Gehrke et al. 2002; Indyk, Datar et al. 2002; Zhu and Shasha 2002; Dong, Han et al. 2003; Golab and Ozsu 2003; Koudas and Srivastava 2003; Muthukrishnan 2003; Gaber, Krishnaswamy et al. 2004; Gaber, Zaslavsky et al. 2005; Jiang and Gruenwald 2006).

8. Challenges in Stream Data Mining:

• Unbounded requirement of Memory:

Data streams are potentially unbound and the amount of storage requirements to process such data can grow unboundedly. For example, if the data has many items and each transaction in stream has a considerable length of items, then just enumerating possible itemsets it self takes lot of time and not only that maintaining counters to all such itemsets will definitely require huge amounts of spaces.

• We may get only one look:

In any streaming environment large volumes of data gets produced continuously at high rate over time. These huge data can not be stored on any static storage as discussed above. It is highly impossible to have another look at stream data. At most we can only have another pass over recent data and whose bounds are very close.

• Retrieve knowledge from discarded data:

As we can not store all the stream data we have to discard data elements after we process them. If a user is interested in knowledge regarding past data, there has to be a mechanism to retrieve knowledge of discarded data by some estimation or approximation.

• Speed at which data arrives Vs Speed with which Algorithm processes:

In any data stream large volumes of data that are being continually produced at a high rate over time. New data is constantly arriving even as the old data is being processed; the amount of computation time per data element must be low, or else the latency of the computation will be too high and the algorithm will not be able to keep pace with the data stream. Most of the time, we might have to discard data without processing it but we do not know which data to process and which to discard?

• Adapting to data:

Mining data streams is done online. The algorithm that mines a stream data should adapt itself to the pace at which data arrives. Data adaptation techniques to catch up with the high-speed data stream and at the same time to achieve the optimum accuracy according to the available resources have to be designed.

• Approximate answering:

User can pose queries on past data which is discarded so we need to retrieve knowledge from discarded data. As storing streams is not possible we have to merge some of the data. This gives rise to the concept of summary data structures with small memories. Summary data structures can only find approximate answers, however high quality approximate answers are often acceptable. With the approximate answers we should also provide bounds on the errors in answers.

9. Discussion:

We have discussed some of the major algorithms for mining frequent itemsets in stream data. Most of the algorithms that we discussed only talk about mining frequent itemsets in recent data. Many of them (Charikar, Chen et al. 2002; Manku and Motwani 2002; Chang and Lee 2003; Karp, Papadimitriou et al. 2003) fail if we ask for knowledge about frequent itemsets in the stream at a past time or in a particular interval of time. Few algorithms(Karp, Papadimitriou et al. 2003; Jin and Agrawal 2005) require second pass over data. Algorithms like (Giannella, Han et al. 2002; Manku and Motwani 2002; Marascu and Masseglia 2005; Silvestri and Orlando 2005) process the transactions in a batches or buckets of fixed size. In any real world stream mining application, fixing batch size is not realistic. If we only process transactions batch wise, then we are restricting stream to only have fixed multiples of transactions in fixed intervals of time and it is not going to be the scenario, as the algorithm has no control over the stream, and number of transactions in a fixed interval of time can not exactly be the multiples some fixed batch size. So, to process transactions in batch wise, the batch size should not be fixed as it is more realistic.

10. Conclusion:

There are many emerging applications which generate stream data. We have discussed the need of stream data mining by exemplifying some real world applications. Related work, algorithms for mining frequent itemsets and the challenges faced by them have been discussed. We classified all these works according to their methodology. However, there is no algorithm that uses variable batch size processing.

References:

1. Aggarwal, C. C. (2002). An Intuitive Framework for Understanding Changes in Evolving Data Streams. In Proceedings of the 18th International Conference on Data Engineering (ICDE'02), San Jose, CA.

2. Agrawal, R. and R. Srikant (1995). Mining Sequential Patterns. Data Eng. (ICDE '95).

3. Babcock, B., MayurDatar, et al. (2004). Load Shedding for Aggregation Queries over Data Streams. 20th International Conference on Data Engineering (ICDE 2004).

4. Babcock B., Babu S., et al. (2002). Models and issues in data stream systems. Twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, Madison, Wisconsin, ACM Press New York, NY, USA.

5. Badcock, B., M. Datar, et al. "Sampling from a moving window over Streaming data."

6. Borgelt, C. (2005). An Implementation of the FPgrowth Algorithm. Workshop Open Source Data Mining Software. New York, NY, USA.

7. Boyer, B. and J. Moore (1982). A fast majority vote algorithm, Institute for Computer Science, University of Texas.

8. Chang, J. H. and W. S. Lee (2003). Finding recent frequent itemsets adaptively over online data streams. Ninth ACM SIGKDD international conference on Knowledge discovery and data mining, Washington, D.C.

9. Chang, J. H. and W. S. Lee (2004). "A Sliding Window Method for Finding Recently Frequent Itemsets over Online Data Streams." Journal of Information Science and Engineering **20**: 753-762.

10. Charikar, M., K. Chen, et al. (2002). Finding Frequent items in data streams. The 29th International Colloquium on Automata, Languages and Programming, Springer-Verlag London, UK.

11. Chen, G., X. W. and, et al. (2005). Sequential Pattern Mining in Multiple Streams. ICDM conf/icdm/ChenWZ05.

12. Chen, G., X. Wu, et al. (2005). Mining Sequential Patterns Across Data Streams, University of Vermont.

13. Chi, Y., H. Wang, et al. (2004). Catch the Moment: Maintaining Closed Frequent Itemsets over a Data Stream Sliding Window, IBM Research Division**:** RC23312 (W0408-155).

14. Cormode, G., F. Korn, et al. (2003). Finding hierarchical heavy hitters in data streams. The 29th VLDB Conference, Berlin, Germany.

15. Cormode, G. and S. Muthukrishnan (2005). "What's hot and what's not: tracking most frequent items dynamically." ACM Transactions on Database Systems (TODS) **30**(1): 249 - 278.

16. Domingos, P. and G. Hulten (2001). Catching Up with the Data: Research Issues in Mining Data Streams. Workshop on Research Issues in Data Mining and Knowledge Discovery, Santa Barbara, CA.

17. Dong, G., J. Han, et al. (2003). "Online Mining of Changes from Data Streams:Research Problems and Preliminary Results." ACM SIGMOD MPDS.

18. Fang, M., N. Shivakumar, et al. (1998). Computing iceberg queries efficiently. 1998 Intl. Conf. on Very Large Data Bases.

19. Fischer, M. and S. Salzberg (1982). "Finding a majority among n votes: Solution to problem." Journal of Algorithms.

20. Gaber, M., M. Krishnaswamy, et al. (2003). Adaptive Mining Techniques for Data Streams Using Algorithm Output Granularity. The Australasian Data Mining Workshop (AusDM 2003).

21. Gaber, M. M., S. Krishnaswamy, et al. (2003). Adaptive Mining Techniques for Data Streams using Algorithm Output Granularity. It was a workshop named- The Australasian Data Mining Workshop (AusDM 2003), Held in conjunction with the 2003 Congress on Evolutionary Computation (CEC 2003).

22. Gaber, M. M., S. Krishnaswamy, et al. (2004). Cost-Efficient Mining Techniques for Data Streams. the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation, Dunedin, New Zealand,

23. Gaber, M. M., S. Krishnaswamy, et al. (2004). Ubiquitous Data Stream Mining. The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining, Sydney, Australia.

24. Gaber, M. M., A. Zaslavsky, et al. (2005). "Mining data streams - a review." SIGMOD Record **34**(2).

25. Ganti, V., J. Gehrke, et al. (2002). "Mining Data Streams under Block Evolution." SIGKDD Explorations 3(2) **3**(2): 1-10.
26. Garofalakis, M., J. Gehrke, et al. (2002). Querying and mining data streams: you only get one look a tutorial. The 2002 ACM SIGMOD international conference on Management of data, Madison, Wisconsin, ACM Press New York, NY, USA.

27. Giannella, C., J. Han, et al. (2002). Mining Frequent Patterns in Data Streams at Multiple Time Granularities. Proceedings of the NSF Workshop on Next Generation Data Mining, November 2002.

28. Giannella, C., J. Han, et al. (2003). Mining Frequent Patterns in Data Streams at Multiple Time Granularities. Next Generation Data Mining AAAI/MIT.

29. Gibbons, P. and Y. Matias (1998). New sampling-based summary statistics for improving approximate query answers. ACM SIGMOD International Conference on Management of Data.

30. Gilbert, A., S. Guha, et al. (2002). Fast, small-space algorithms for approximate histogram maintenance. 34th ACM Symposium on Theory of Computing.

31. Golab, L. and M. T. Ozsu (2003). "Issues in Data Stream Management." ACM SIGMOD Record **Volume 32**(2): 5--14.
32. Hammad, M. A., W. G. Aref, et al. (2003). Efficient Execution of SlidingWindow Queries Over Data Streams, Department of Computer Sciences, Purdue University.**:** CSD TR#03-035.

33. Han, J., J. Pei, et al. (2000). Mining Frequent Patterns without Candidate Generation. ACM SIGMOD, Dallas,TX.

34. Hsieh, M. C.-E., Y.-H. Wu, et al. (2006). Discovering Frequent Tree Patterns over Data Streams. SIAM conference on Data mining.

35. Indyk, P., M. Datar, et al. (2002). Maintaining Stream Statistics Over Sliding Windows (Extended Abstract). 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002).

36. Jiang, N. and L. Gruenwald (2006). "Research Issues in Data Stream Association Rule Mining." ACM SIGMOD **vol -35** (1): 14-19.

37. Jin, R. and G. Agrawal (2005). "An algorithm for in-core frequent itemset mining on streaming data." ICDM 2005: 210-217.

38. Karp, R. M., C. H. Papadimitriou, et al. (2003). "A Simple Algorithm For Finding Frequent Elements In Streams And Bags." ACM Transactions on Database Systems (TODS) **28**(1): 51-55.

39. Koudas, N. and D. Srivastava (2003). Data Stream Query Processing: A Tutorial. 29th International Conference on Very Large Databases (VLDB), Berlin, Germany.

40. Li, H.-F., S.-Y. Lee, et al. (2004). On mining webclick streams for path traversal patterns. 13th international World Wide Web conference on Alternate track papers & posters, New York, NY, USA.

41. MAIDS (2003). Tilted-Time Window Implementation, NCSA and Department of Computer Science, University of Illinois at Urbana-Champaign.

42. Manku, G. S. and R. Motwani (2002). Approximate frequency counts over data streams. The 28th International Conference on Very Large Data Bases, Hong Kong, China.

43. Marascu, A. and F. Masseglia (2005). Mining Data Streams for Frequent Sequences Extraction, First International Workshop on Mining Complex Data 2005 (IEEE MCD'2005).

44. Misra, J. and D. Gries (1982). "Finding repeated elements." Science of Computer Programming,.

45. Muthukrishnan, S. (2003). Data Streams: Algorithms and Applications. Fourteenth annual ACM-SIAM symposium on discrete algorithms.

46. Orlando, S., R. Perego, et al. (2006). Algorithms and frameworks for stream mining and knowledge discovery on Grids, Institute on Knowledge and Data Management.

47. Silvestri, C. and S. Orlando (2005). Approximate Mining of Frequent Patterns on Streams. the Second International Workshop on Knowledge Discovery in Data Streams in conjunction with 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD), Porto, Portogallo.

48. Srikant, R. and R. Agrawal (1996). Mining Sequential Patterns: Generalizations and Performance Improvements. 5th Int. Conf. Extending Database Technology, {EDBT}.

49. Teng, W.-G., M.-S. Chen, et al. (2003). A Regression-Based Temporal Pattern Mining Scheme for Data Streams. 29th ACM VLDB International Conference on Very Large Data Bases.

50. Toivonen, H. (1996). Sampling large database for association rules. 22nd Intl. Conf. on Very Large Data Bases.

51. Yu, C.-C. and Y.-L. Chen (2005). "Mining Sequential Patterns from Multidimensional Squence." IEEE Transactions on Knowledge and Data Engineering.

52. Zaki, M. J. and C. H. Charm (2002). An effcient algorithm for closed itemset mining. 2nd SIAM Int'l Conf. on Data Mining.

53. Zheng, Q., K. Xu, et al. (2002). When to Update the sequential patterns of stream data.

54. Zhu, Y. and D. Shasha (2002). StatStream: Statistical monitoring of thousands of data streams in real time. The 28th VLDB Conference, Hong Kong, China.