2.1 Introduction

Unlike traditional data sets, stream data flow in and out of a computer system continuously and with varying update rates. It may be impossible to store an entire data stream due to its tremendous volume. To discover knowledge or patterns from data streams, it is necessary to develop data stream summarization techniques. Lots of work has been done to summarize the contents of data streams in order to produce an approximate but controlled result. A strong constraint both in querying and mining data streams is to process data ‘on the fly’, facing variations in input streams. For many applications, it is nevertheless necessary to keep track of information related to a maximum number of elements of the stream but in bounded space. Lots of work has been done to summarize the contents of data streams in order to produce an approximate but controlled result. Creating synopsis of data refers to the process of applying summarization techniques that are capable of summarizing the incoming stream for further analysis. Since synopsis of data does not represent all the characteristics of the dataset, approximate answers are produced when using data synopsis. Synopses structures can be used both for answering queries and applying data mining algorithms to streams.

The goal of the frequent pattern mining issues in data streams is depicted as the following: input a sequence of data item $x_1, x_2, ..., x_N, ...$, where $N$ denotes the current length of the stream, support $s \in (0,1)$ and error $\varepsilon \in (0,1)$, $\varepsilon<<s$, at any point of time, these algorithms can produce a list of item(set)s along with their estimated frequencies.

With finite space of main memory, precise frequent patterns cannot be mined when plenty of data streams arrive, which requires to drop some data items or data packets and only compute an approximate count of the data set. The algorithms on this statistical counting problem can be classified into two categories:

**Deterministic bounds**: an approximate answer which bounds on error is computed within a factor $\pm \varepsilon$ of the actual answer;

**Probabilistic bounds**: an approximate answer with high probability is computed within a factor $\pm \varepsilon$ of the actual answer;

Approximate answers with probabilistic bounds can handle the item(set)s evaluated over a sample of the data streams rather than over the entire data streams, which leads to better results when the speed of processor is fast but the speed of data structure updating is slow, we can process the data streams in batches with deterministic bounds, which is attractive because it has not any uncertainty about the accuracy of the answer, sacrificing timeliness instead, and is also a good approach when data streams are burst.
The design and choice of a particular synopsis method depends on the problem being solved with it. Therefore, the synopsis needs to be constructed in a way which is friendly to the needs of the particular problem being solved. For example, a synopsis structure used for query estimation is likely to be very different from a synopsis structure used for data mining problems such as change detection and classification. The applicability to data streams makes the efficiency issue of space and time-construction critical. There are a variety of techniques [1] like Histograms, Wavelets, Sketches, Micro-cluster, Sampling, based summarization etc which can be used for synopsis construction in data streams.

2.2 Sketches

Many data stream problems cannot be solved with just a sample. Instead, we can make use of data structures which, in effect, include a contribution from the entire input, rather than just the items picked in the sample. For example, consider trying to count the number of distinct objects in a stream. It is easy to see that unless almost all items are included in the sample, then we cannot tell whether they are the same or distinct. Since a streaming algorithm gets to see each item in turn, it can do better. Sketches are data structures[13] dedicated to a specialized task which can be updated incrementally to maintain in small space summarized information. They are often based on the use of hash functions which project randomly stream data on the small structure. Approximate answers to queries can be obtained from sketches with probabilistic bounds on errors. Since data streams produce data with a high rate, the probabilistic bounds are quite tight. “sketch” is referred as a compact data structure which summarizes the stream for certain types of query. Typically it is a linear transformation of the stream: we can imagine the stream as defining a vector, and the algorithm computes the product of a matrix with this vector (to be effective, the matrix must have a very small representation, e.g. being defined implicitly by hash functions).

In certain other applications, it may be desirable to track the frequencies of the distinct values in the data stream. In this case, if \((u_1, \ldots, u_N)\) be the frequencies of \(N\) distinct values in the data stream, then the sketch is defined by the dot product of the vector \((u_1 \ldots u_N)\) with a random vector of size \(N\). As in the previous case, the number of distinct items \(N\) may be large, and therefore the size of the corresponding random vector will also be large. A natural solution is to pre-generate a set of \(k\) random vectors, and whenever the \(i^{th}\) item is received, we add \(r_{ij}\) to the \(j^{th}\) sketch component. Therefore, the \(k\) random vectors may need to be pre-stored in order to perform the computation. However, the explicit storage of the random vector will defeat the purpose of the sketch computation, because of the high space complexity.

Popular sketch algorithms are:

**The Count-Min Sketch:** One interesting variation of the sketching method for data streams is the **count-min sketch**, which uses a hash-based sketch of the stream. The count-min sketch is proposed in [14]. In the count-min sketch, they use \([\ln(1/\delta)]\) pairwise independent hash functions, each of which map on to uniformly random integers in the range \([0, e/e]\), where \(e\) is the base of the natural logarithm. Thus, they maintain a total of \([\ln(1/\delta)]\) hash tables, and there are a total of \(O(\ln(1/\delta)/e)\) hash cells.
Apply each hash function to any incoming element in the data stream, and add the count of the element to each of the corresponding \([\ln(1/\delta)]\) positions in the different hash tables. It is noted that because of collisions, the hash table counts will not exactly correspond to the count of any element in the incoming data stream. When incoming frequency counts are non-negative, the hash table counts will over-estimate the true count, whereas when the incoming frequency counts are either positive or negative (deletions), the hash table count could be either an over-estimation or an underestimation.

**Flajolet-Martin Sketch.** The Flajolet-Martin sketch [14] is a bitmap of length approximately \(\log m\). Each item is mapped by a hash function into an entry of the bitmap: with probability \(\frac{1}{2}\) it maps into entry 1, \(\frac{1}{4}\) to entry 2, \(\frac{1}{8}\) to entry 3 and so on. For each item in the stream, we map to its bit under the hash function, and set the bit to 1. The position of the least significant 0 in the bitmap indicates the logarithm of the number of distinct items seen, \(D\); taking repetitions with randomly chosen hash functions improves the accuracy.

**AMS Sketches.** The Alon-Matias-Szegedy sketch [14] can be described in terms of the Count-Min sketch. Now, when we go to update a counter, we multiply the value of the update by a hash function \(g\) on the item being updated: half the items are mapped to +1 by this hash function, and half to -1. Taking the sum of the squares of all counters in each row gives a high-quality estimate for \(F_2 = \sum_{i=1}^{m} f_i^2\), the sum of the squares of the frequency counts. This computation, or variations thereof, is at the heart of many data stream analyses.

A common feature of these sketch algorithms is that they rely on hash functions on item identifiers, which are relatively easy to implement and fast to compute. The method of sketches is a probabilistic counting method whereby a randomized function is applied to the data stream in order to perform the counting in a space-efficient way. While sketches are a good method to determine large aggregate signals, they are not very useful for counting infrequently occurring items in the stream. For example, problems such as the determination of the number of distinct elements cannot be performed with sketches. For this purpose, hash functions turn out to be a useful choice. The hash function technique is very useful for those estimations in which non-repetitive elements have the same level of importance as repetitive elements.

### 2.3 Histograms

Another key method for data summarization is that of histograms [1]. In the method of histograms, data is divided along any attribute into a set of ranges, and maintain the count for each bucket. Thus, the space requirement is defined by the number of buckets in the histogram. A naive representation of a histogram would discretize the data into partitions of equal length (equiwidth partitioning) and store the frequencies of these buckets. At this point, we point out a simple connection between the histogram representation and Haar wavelet coefficients. If we construct the wavelet representation of the frequency distribution of a data set along any dimension, then the (non-normalized) Haar coefficients of any order provide the difference in relative frequencies in equiwidth histogram buckets. Haar coefficients of different orders correspond to buckets of different levels of granularity.
It is relatively easy to use the histogram for answering different kinds of queries such as range queries, since we only need to determine the set of buckets which lie within the user specified ranges. The key source of inaccuracy in the use of histograms is that the distribution of the data points within a bucket is not retained, and is therefore assumed to be uniform. This causes inaccuracy because of extrapolation at the query boundaries which typically contain only a fractional part of a histogram. Thus, an important design consideration in the construction of histograms is the determination of how the buckets in the histogram should be designed. For example, if each range is divided into equiwidth partitions, then the number of data points would be distributed very unequally across different buckets. If such buckets include the range boundary of a query, this may lead to inaccurate query estimations. Therefore, a natural choice is to pick equiseta buckets, in which each range contains an approximately equal number of points. In such cases, the maximum inaccuracy of a query is equal to twice the count in any bucket. However, in the case of a stream, the choice of ranges which would result in equiseta partitions is not known a-priori. We note that the design of equiseta buckets is exactly the problem of quantile estimation in data streams, since the equiseta partitions define different quantiles in the data. These Equiwidth histograms [17] partition the domain into buckets such that the number of values falling into each bucket is uniform across all buckets. In other words, they maintain quantiles for the underlying data distribution as the bucket boundaries.

A different choice for histogram construction is that of minimizing the frequency variance of the different values within a bucket, so that the uniform distribution assumption is approximately held for queries. This minimizes the boundary error of extrapolation in a query.

**V-Optimal Histograms over Data Streams:** [15] showed how to compute optimal V-Optimal Histograms for a given data set using dynamic programming. The algorithm uses $O(N)$ space and requires $O(N^2B)$ time, where $N$ is the size of the data set and $B$ is the number of buckets. This is prohibitive for data streams. [16] adapted this algorithm to sorted data streams. Their algorithm constructs an arbitrarily-close V-Optimal Histogram (i.e., with error arbitrarily close to that of the optimal histogram), using $O(B^2 \log N)$ space and $O(B^2 \log N)$ time per data element. [18], removed the restriction that the data stream be sorted, providing algorithms based on the sketching technique. The idea is to view each data element as an update to an underlying vector of length $N$ that we are trying to approximate using the best $B$-bucket histogram. The time to process a data element, the time to reconstruct the histogram, and the size of the sketch are each bounded by $\text{poly}(B, \log N, 1/\epsilon)$, where $\epsilon$ is the relative error we are willing to tolerate. Their algorithm proceeds by first constructing a robust approximation to the underlying “signal.” A robust approximation is built by repeatedly adding a dyadic interval of constant value which best reduces the approximation error. In order to find such a dyadic interval it is necessary to efficiently compute the sketch of the original signal minus the constant dyadic interval. [18] showed how to construct such efficiently range-summable $p$-stable random variables. From the robust histogram they cull a histogram of desired accuracy and with $B$ buckets.

**End-Biased Histograms:** These will maintain exact counts of items that occur with frequency above a threshold, and approximate the other counts by a uniform distribution. Many applications maintain simple aggregates (count) over an attribute to find aggregate values above a specified threshold. These queries are referred to as iceberg queries [19]. Such
Iceberg queries arise in many applications, including data mining, data warehousing, information retrieval, market basket analysis, copy detection, and clustering. For example, a search engine might be interested in gathering search terms that account for more than 1% of the queries. Such frequent item summaries are useful for applications such as caching and analyzing trends. 

[19] gave an efficient algorithm to compute Iceberg queries over disk-resident data. Their algorithm requires multiple passes which is not suited to the streaming model. [20] Presented randomized and deterministic algorithms for frequency counting and iceberg queries over data streams. The randomized algorithm uses adaptive sampling and the main idea is that any item which accounts for a $c$ fraction of the items is highly likely to be a part of a uniform sample of size $1/c$. The deterministic algorithm maintains a sample of the distinct items along with their frequency. Whenever a new item is added, it is given a benefit of doubt by over-estimating its frequency. If we see an item that already exists in the sample, its frequency is incremented. Periodically items with low frequency are deleted. Thus, these algorithms report all items of count greater than $cN$. Moreover, for all items reported, they guarantee that the reported count is less than the actual count, but by no more than $cN$.

### 2.4 Wavelets

Wavelet transforms (like Discrete Cosine and Fourier transforms) are special mathematical transforms that attempt to capture the trends in numerical functions. Often, very few of the wavelet coefficients of empirical data sets are significant and majorities are small or insignificant. In practice, a small number of significant coefficients are needed to capture the trends in numerical functions.

Wavelets have traditionally been used in a variety of image and query processing applications. But nowadays Wavelets are a well known technique often used in databases for hierarchical data decomposition and summarization. The basic idea in the wavelet technique is to create a decomposition of the data characteristics into a set of wavelet functions and basis functions. The property of the wavelet method is that the higher order coefficients of the decomposition illustrate the broad trends in the data, whereas the more localized trends are captured by the lower order coefficients.

In the wavelet analysis, the hierarchical data structure is used to keep wavelet coefficient of data items. Haar wavelet is one of the simplest wavelet algorithms and is used widely. For example [10] consider the signal of length $N = 8$ given by array $A = [1, 3, 5, 11, 12, 13, 0, 1]$; its Haar wavelet transform computation is computed by convolving the signal with the low-pass filter $\{1/\sqrt{2}, 1/\sqrt{2}\}$ and the high-pass filter $\{-1/\sqrt{2}, 1/\sqrt{2}\}$, followed by down-sampling by two. In the discrete case, if there are $N$ values in the array, this process yields $N=2$ “averages” and $N=2$ “differences” (these are averages and differences, respectively, but scaled by a suitable scaling factor). We store the differences as the wavelet coefficients at this level. We then repeat this procedure on the “averages,” computing “averages” and “differences” again, until we are left with only one “average” and $N-1$ “differences” over $\log N$ scales or resolutions. The entire computation can be quite naturally represented by a binary tree over the signal array, each node in the tree representing the “average” of the nodes under it and the “difference” between the left and right child of that node as shown below.
The description above of Haar wavelet transforms is illustrative, but not conducive to streaming computations directly, especially when the signal is rendered in unordered cash register model. Haar wavelet is visualized in terms of vector computations [10]. Number the levels of the binary tree as with the bottommost level being 0 and the topmost (the array) being \( \log_2 N = 3 \) in this case. For \( j = 1, \ldots, \log_2 N \) and \( k = 0, \ldots, 2^j - 1 \), define the vector \( \varphi_{j,k}(l) = 1 \) for \( k(N/2^j) \leq l \leq k(N/2^j) + N/2^j - 1 \) and 0 otherwise. These ranges where \( \varphi_{j,k}(l) = 1 \) are known as dyadic intervals in the literature. They [10] further defined \( \psi_{j,k} = -\varphi_{j+k,2k} + \varphi_{j+k,2k+1} \) for \( 0 \leq j \leq \log_2 N - 1 \) and \( k = 0, \ldots, 2^j - 1 \). The scaling factor at level \( j \) is \( s_j = \sqrt{N/2^j} \), for all \( j = 0; \ldots; \log_2 N \). Now, we can define wavelet vectors to be \( s_j \psi_{j,k} \) for each \( j,k \), giving \( N-1 \) in all. These, respectively, yield the \( N-1 \) wavelet coefficients corresponding to the differences given by \( d_{j,k} = s_j(a, \psi_{j,k}) \), where \( (x, y) \) is the inner product of vectors \( x \) and \( y \). The final “average” is the coefficient that corresponds to all 1’s vector \( v \) with scaling factor \( s_0 = \sqrt{N} \), that is, \( c_{0,0} = s_0(a,v) \); vector \( s_0 v \) together with the N-1 wavelet vector form the N wavelet basis vectors.

The wavelet decomposition method provides a natural method for dimensionality reduction, by retaining only the coefficients with large absolute values. All other coefficients are implicitly approximated to zero. This makes it possible to approximately represent the series with a small number of coefficients. The idea is to retain only a predefined number of coefficients from the decomposition, so that the error of the reduced representation is minimized. Wavelets are used extensively for efficient and approximate query processing of different kinds of data. The efficiency of the query processing arises from the reduced representation of the data. At the same time, since only the small coefficients are discarded the results are quite accurate.

A key [1] issue for the accuracy of the query processing is the choice of coefficients which should be retained. While it may be tempting to choose only the coefficients with large absolute values, this is not always the best choice, since a more judicious choice of coefficients can lead to minimizing specific error criteria. Two such metrics are the minimization of the mean square error or the maximum error metric. The
mean square error minimizes the \( L_2 \) error in approximation of the wavelet coefficients, whereas maximum error metrics minimize the maximum error of any coefficient. Another related metric is the relative maximum error which normalizes the maximum error with the absolute coefficient value.

According to point made in [12] is that most wavelet approximation methods solve a restricted version of the problem in which the wavelet coefficient for the basis is defined to be half the difference between the left hand and right hand side of the basis vectors. Thus, the problem is only one of picking the best \( B \) coefficients out of these pre-defined set of coefficients. While this is an intuitive method for computation of the wavelet coefficient, and is optimal for the case of the Euclidean error, it is not necessarily optimal for the case of the \( L_m \)-metric. For example [1], consider the time series vector \((1, 4, 5, 6)\). In this case, the wavelet transform is \((4, -1.5, -1.5, -0.5)\). Thus, for \( B = 1 \), the optimal coefficient picked is \((4, 0, 0, 0)\) for any \( L_m \)-metric. However, for the case of \( L_\infty \)-metric, the optimal solution should be \((3.5, 0, 0, 0)\), since 3.5 represents the average between the minimum and maximum value. Clearly, any scheme which restricts itself only to wavelet coefficients defined in a particular way will not even consider this solution [12]. Almost all methods for non-Euclidean wavelet computation tend to use this approach, possibly as a legacy from the Haar method of wavelet decomposition. This restriction has been removed in [12] and proposes a method for determining the optimal synopsis coefficients for the case of the weighted \( L_m \) metric. Synopsis coefficients and wavelet coefficients are distinguish, since the latter is defined by the simple subtractive methodology of the Haar decomposition.

2.5 Sampling methods

Sampling methods are among the simplest methods for synopsis construction in data streams. It is also relatively easy to use this synopsis with a wide variety of application since their representation is not specialized and uses the same multi-dimensional representation as the original data points. In particular reservoir based sampling methods are very useful for data streams.

Sampling is a popular tool used for many applications, and has several advantages from an application perspective. One advantage is that sampling is easy and efficient, and usually provides an unbiased estimate of the underlying data with provable error guarantees. Another advantage of sampling methods is that since they use the original representation of the records, they are easy to use with any data mining application or database operation. In most cases, the error guarantees of sampling methods generalize to the mining behavior of the underlying application. Many synopsis methods such as wavelets, histograms, and sketches are not easy to use for the multi-dimensional cases. The random sampling technique is often the only method of choice for high dimensional applications.

Assume that there is a database \( D \) containing \( N \) points which are denoted by \( X_1 \ldots X_N \). Let the function \( f(D) \) represent an operation to perform on the database \( D \). For example \( f(D) \) may represent the mean or sum of one of the attributes in database \( D \). A random sample \( S \) from database \( D \) defines a random variable \( f(S) \) which is (often) closely related to \( f(D) \) for many commonly used functions. It is also possible to estimate the standard deviation of \( f(S) \) in many cases. In the case of aggregation based functions in
linear separable form (e.g. sum, mean), the law of large numbers allows to approximate the random variable $f(S)$ as a normal distribution, and characterize the value of $f(D)$ probabilistically. However, not all functions are aggregation based (e.g. min, max). In such cases, it is desirable to estimate the mean $\mu$ and standard deviation $\sigma$ of $f(S)$. These parameters allows to design probabilistic bounds on the value of $f(S)$. This is often quite acceptable as an alternative to characterizing the entire distribution of $f(S)$. Such probabilistic bounds can be estimated using a number of inequalities which are also often referred to as tail bounds.

On-line random sampling from data streams is a problem because the size of the sample has to be bounded and the size of the data set from which elements are selected is not known when sampling begins, i.e. at the beginning of the stream. Some algorithms exist to overcome this problem: the reservoir algorithm [2] maintains a fixed size uniform random sample from the beginning of the stream to current time. Some elements are selected randomly from the stream with a decreasing probability and replace randomly elements already in the sample. The decreasing probability ensures that the sample is uniform over the period.

Some extensions and new approaches have been developed to extract random samples from a sliding window defined on the stream [3]. In this case, the problem is that elements of the sample have to be removed and replaced by new ones when they expire from the window, under the constraint of still keeping uniform sample.

On the study of frequent pattern mining in data streams, there are some probabilistic methods. The goal of these algorithms is just like this: input a sequence of data item, where $N$ denotes the current length of the stream, support $s$ and error $\varepsilon$, $\varepsilon << s$, at any point of time, these algorithms can he used to generate a list of item (set)s along with their estimated frequencies with probability $(1-\delta)$, the computed answer is within a factor $\pm \varepsilon$ of the actual answer. The key problem of the method is how many samples we need in order to satisfy this situation. If the number of sample is too small, the obtained answer is not within a factor $\pm \varepsilon$ of the actual answer with probability $(1-\delta)$; if the number of sample is too high, the precision cannot be improved (we cannot improve the probability that the answer is within a factor of the actual answer). For example, the actual frequency of item set $A$ is 5 (we can obtain it by adopting static storage techniques), if we use sampling, suppose the rate of sample is $r=0.8$, then we can obtain a frequency with probability 80%, the answer is near the true frequency within $\varepsilon=0.1$, that the estimate frequency is within $[4.5, 5.5]$.

**Reservoir Sampling:** In reservoir sampling method:

- The first $n$ points in the data stream are added to the reservoir for initialization.
- Subsequently, when the $(t+1)$th point from the data stream is received, it is added to the reservoir with probability $n/(t+1)$
- This point replaces a randomly chosen point in the reservoir

Probability of insertion reduces with stream progression. The reservoir sampling method maintains an unbiased sample of the history of the data stream. In an evolving data stream only the more recent data may be relevant for many queries. For example, if an application is queried for the statistics for the past hour of stream arrivals, then for a data stream which has been running over one year, only about 0.01% of an unbiased sample may be relevant. In general, the quality of the result for the same query will only degrade with progression of the stream, as a smaller and smaller portion of the sample remains
relevant with time. One solution is to use a sliding window approach for restricting the horizon of the sample.

The conventional reservoir sampling \cite{5} selects a uniform random sample of a fixed size, without replacement, from an input stream of an unknown size. Initially, the algorithm places all tuples in the reservoir until the reservoir (of the size of $r$ tuples) becomes full. After that, each $k^{th}$ tuple is sampled with the probability $r/k$. A sampled tuple replaces a randomly selected tuple in the reservoir. It is easy to verify that the reservoir always holds a uniform random sample of the $k$ tuples seen so far.

**Algorithm 2.1 : conventional reservoir sampling\cite{5}**

1: \textbf{Inputs:} $r$ \{reservoir size\}
2: \textbf{1: } $k = 0$
3: \textbf{2: for each tuple arriving from the input stream do}
4: \textbf{3: } $k = k + 1$
5: \textbf{4: if } $k \leq r$ \textbf{then}
6: \textbf{5: add the tuple to the reservoir}
7: \textbf{6: else}
8: \textbf{7: sample the tuple with the probability $r/k$ and replace a randomly selected tuple in the reservoir with the sampled tuple}
9: \textbf{8: end if}
10: \textbf{9: end for}

**Uniformity confidence:** A sample is a uniform random sample \cite{5} if it is produced using a sampling scheme in which all statistically possible samples of the same size are equally likely to be selected. In this case, we say the uniformity confidence in the sampling algorithm equals 100%. In contrast, if some statistically possible samples cannot be selected using a certain sampling algorithm, then we say the uniformity confidence in the sampling algorithm is below 100%. Thus, we define uniformity confidence as follows.

\[
\text{Uniformity confidence} = \frac{\text{the number of different samples of the same size possible with the algorithm}}{\text{the number of different samples of the same size possible statistically}} \times 100
\]

For reservoir sampling, the uniformity confidence in a reservoir sampling algorithm which produces a sample $S$ of size $r$ (denoted as $S[r]$) is defined as the probability that $S[r]$ is a uniform random sample of all the tuples seen so far. That is, if $k$ tuples have been seen so far, then the uniformity confidence is 100% if and only if every statistically
possible $S[r]$ has an equal probability to be selected from the $k$ tuples. If the reservoir size is decreased from $r$ to $r - \delta$ ($\delta > 0$), then there is a way to maintain the sample in the reduced reservoir such that every statistically possible $S_{[r-\delta]}$ has an equal probability of being in the reduced reservoir, whereas if the reservoir size is increased from $r$ to $r+\delta$ ($\delta > 0$), then some statistically possible $S_{[r+\delta]}$'s cannot be selected.

Reservoir sampling is also a well-known technique for sequential random sampling over data streams. Conventional reservoir sampling assumes a fixed-size reservoir. There are situations, however, in which it is necessary and/or advantageous to adaptively adjust the size of a reservoir in the middle of sampling due to changes in data characteristics and/or application behavior. [5] Proposed an algorithm called *adaptive reservoir sampling* for maintaining the reservoir sample after the reservoir size is adjusted. If the size decreases, the algorithm maintains a sample in the reduced reservoir with a 100% uniformity confidence by randomly evicting tuples from the original reservoir. If the size increases, the algorithm finds the minimum number of incoming tuples that should be considered in the input stream to refill the enlarged reservoir such that the resulting uniformity confidence exceeds a given threshold. Then, the algorithm decides probabilistically on the number of tuples to retain in the enlarged reservoir and randomly evicts the remaining number of tuples. Eventually, the algorithm fills the available room in the enlarged reservoir using the incoming tuples. The number of incoming tuples used to fill the enlarged reservoir is denoted as $m$ and called the uniformity confidence recovery tuple count.

**Algorithm 2.2 : Adaptive reservoir sampling**

**Inputs:** $r$ {reservoir size}
$k$ {number of tuples seen so far}
$\zeta$ {uniformity confidence threshold}

1: while true do
2:     while reservoir size does not change do
3:         conventional reservoir sampling (Algorithm 1).
4:     end while
5:     if reservoir size is decreased by $\delta$ then
6:         randomly evicts $\delta$ tuples from the reservoir.
7:     else
8:         {i.e., reservoir size is increased by $\delta$}
9:         Find the minimum value of $m$ that causes the $UC$ to exceed $\zeta$.
10:        flip a biased coin to decide on the number, $x$, of tuples to retain among $r$ tuples already in the reservoir
11:        randomly evict $r - x$ tuples from the reservoir.

23
12: \text{select } r + \delta - x \text{ tuples from the incoming } m \text{ tuples using conventional reservoir sampling.}

13: \text{end if}

14: \text{end while}

To summarize if the reservoir size decreases, sample can be maintained in the \textit{reduced} reservoir with a 100% confidence uniformity. In contrast, if the reservoir size increases, it is not possible to maintain the sample in the \textit{enlarged} reservoir with a 100% uniformity confidence. The above algorithm maintains the reservoir sample after the reservoir size is adjusted such that the resulting uniformity confidence exceeds a given threshold.

\textbf{Unbiased reservoir sampling}: In the case of streams, the number of records \(N\) is not known in advance and the sampling must be performed dynamically as the records are read. To obtain an unbiased sample of size \(n\) from the data stream, the algorithm maintains a reservoir of size \(n\) from the data stream. The first \(n\) points in the data streams are added to the reservoir for initialization. Subsequently, when the \((t + 1)\)th point from the data stream is received, it is added to the reservoir with probability \(n/(t+1)\). In order to make room for the new point, any of the current points in the reservoir are sampled with equal probability and subsequently removed. This sampling approach maintains the unbiased character of the reservoir. The probability of the \((t + 1)\)th point being included in the reservoir is \(n/(t + 1)\). The probability of any of the last \(t\) points being included in the reservoir is defined by the sum of the probabilities of the events corresponding to whether or not the \((t + 1)\)th point is added to the reservoir. From the inductive assumption, we know that the first \(t\) points have equal probability of being included in the reservoir and have probability equal to \(n/t\). In addition, since the points remain in the reservoir with equal probability of \((n - 1)/n\), the conditional probability of a point (among the first \(t\) points) remaining in the reservoir given that the \((t+1)\) point is added is equal to \((n/t) \cdot (n-1)/n = (n-1)/t\). By summing the probability over the cases where the \((t+1)\) point is added to the reservoir (or not), we get a total probability of \(((n/(t+1)) \cdot (n-1)/(t+1)) \cdot (n/t) = n/(t+1)\). Therefore, the inclusion of all points in the reservoir has equal probability which is equal to \(n/(t+1)\). As a result, at the end of the stream sampling process, all points in the stream have equal probability of being included in the reservoir, which is equal to \(n/N\).

As the length of the stream increases, an unbiased sample contains a larger and larger fraction of points from the distant history of the stream. In reservoir sampling, this is also reflected by the fact that the probability of successive insertion of new points reduces with progression of the stream [2]. A completely unbiased sample is often undesirable from an application point of view, since the data stream may evolve, and the vast majority of the points in the sample may represent the stale history of the stream. For example, if an application is queried for the statistics for the past hour of stream arrivals, then for a data stream which has been running over one year, only about 0.01% of an unbiased sample may be relevant. The imposition of range selectivity or other constraints on the query will reduce the relevant estimated sample further.

\textbf{Biased reservoir sampling}: In many cases the stream data may evolve over time. Then the corresponding data mining or query results may also change over time. Thus, the results of a query over a more recent window may be quite different from the results of a
query over a more distant window. Similarly, the entire history of the data stream may not be relevant for use in a repetitive data mining application such as classification. Recently, the reservoir sampling algorithm was adapted to sample from a moving window over data streams [6]. This is useful for data streams, since only a small amount of recent history is more relevant than the entire data stream. However, this can sometimes be an extreme solution, since one may desire to sample from varying lengths of the stream history. While recent queries may be more frequent, it is also not possible to completely disregard queries over more distant horizons in the data stream. A method in [4] designs methods for biased reservoir sampling, which uses a bias function to regulate the sampling from the stream.

One interesting characteristic of reservoir maintenance algorithm is that it is extremely efficient to implement in practice. When new points in the stream arrive, we only need to decide whether or not to insert into the current sample array which represents the reservoir. The sample array can then be overwritten at a random position. The biased formulation of sampling problem[4] is given below:

The bias function associated with the r-th data point at the time of arrival of the t-th point (r ≤ t) is given by f(r, t) and is related to the probability p(r,t) of the r-th point belonging to the reservoir at the time of arrival of the t-th point. Specifically, p(r,t) is proportional to f(r, t). The function f(r, t) is monotonically decreasing with t (for fixed r) and monotonically increasing with r (for fixed t). Therefore, the use of a bias function ensures that recent points have higher probability of being represented in the sample reservoir. Concept of a bias-sensitive sample S(t), is defined by the bias function f(r, t).

Let f(r, t) be the bias function for the r-th point at the arrival of the t-th point. A biased sample S(t) at the time of arrival of the t-th point in the stream is defined as a sample such that the relative probability p(r,t) of the r-th point belonging to the sample S(t) (of size n) is proportional to f(r, t).

This bias function is quite effective since it regulates the sampling in a smooth way so that queries over recent horizons are more accurately resolved. While the design of a reservoir for arbitrary bias function is extremely difficult, it is shown in [4], that certain classes of bias functions (exponential bias functions) allow the use of a straightforward replacement algorithm. The advantage of a bias function is that it can smoothly regulate the sampling process so that acceptable accuracy is retained for more distant queries. The method in [4] can also be used in data mining applications so that the quality of the results does not degrade very quickly. The aim of bias function is to regulate the choice of the stream sample. In other words, the bias function modulates the sample in order to focus on recent or old behaviors in the stream following application specific constraints.

As indicated in [7] a new approach based on biased reservoir sampling to achieve a more efficient mining of sequential patterns is introduced. Biased reservoir size is always bounded whatever the size of the stream is. This property often allows keeping the entire relevant reservoir in main memory. They gave a simple algorithm to build the biased reservoir for the special case of sequential pattern mining. The algorithm was built to achieve exponential bias with the λ parameter. Unlike the algorithms presented in [4], this approach needs to take into account the constraint of the lower bound of the size of the reservoir. Here the reservoir size is defined in term of customer’s number and not in
term of transaction numbers. That means that insertion and delete operations must be done at the customer’s level and at the itemsets level.

**Concise Sampling:** The effectiveness of the reservoir based sampling method can be improved further with the use of concise sampling [4]. We note that the size of the reservoir is sometimes restricted by the available main memory. It is desirable to increase the sample size within the available main memory restrictions. For this purpose, the technique of concise sampling is quite effective. The method of concise sampling exploits the fact that the number of distinct values of an attribute is often significantly smaller than the size of the data stream. This technique is most applicable while performing univariate sampling along a single dimension. For the case of multi-dimensional sampling, the simple reservoir based method discussed above is more appropriate. The repeated occurrence of the same value can be exploited in order to increase the sample size beyond the relevant space restrictions. It is noted that when the number of distinct values in the stream is smaller than the main memory limitations, the entire stream can be maintained in main memory, and therefore sampling may not even be necessary. For current desktop systems in which the memory sizes may be of the order of several gigabytes, very large sample sizes can be main memory resident, as long as the number of distinct values does not exceed the memory constraints. On the other hand, for more challenging streams with an unusually large number of distinct values, we can use the following approach.

The sample is maintained as a set S of <value, count> pairs. For those pairs in which the value of count is one, count is not maintained explicitly, but value is maintained as a singleton. The number of elements in this representation is referred to as the footprint and is bounded above by n. It is noted that the footprint size is always smaller than or equal to than the true sample size. If the count of any distinct element is larger than 2, then the footprint size is strictly smaller than the sample size. A threshold parameter \( \tau \) is used to define the probability of successive sampling from the stream. The value of \( \tau \) is initialized to be 1. As the points in the stream arrive, we add them to the current sample with probability \( 1/\tau \). It is noted that if the corresponding value-count pair is already included in the set S, then only count needs to be incremented by 1. Therefore, the footprint size does not increase. On the other hand, if the value of the current point is distinct from all the values encountered so far, or it exists as a singleton then the footprint increases by 1. This is because either a singleton needs to be added, or a singleton gets converted to a value-count pair with a count of 2. The increase in footprint size may potentially require the removal of an element from sample S in order to make room for the new insertion. When this situation arises, we pick a new (higher) value of the threshold \( \tau ' \), and apply this threshold to the footprint in repeated passes. In each pass, value of count is reduced with probability \( \tau '/\tau ' \), until at least one value-count pair reverts to a singleton or a singleton is removed. Subsequent points from the stream are sampled with probability \( 1/\tau ' \). As in the previous case, the probability of sampling reduces with stream progression, though we have much more flexibility in picking the threshold parameters in this case.

One of the interesting characteristics of this approach is that the sample S continues to remain an unbiased representative of the data stream irrespective of the choice of \( \tau \). In practice, \( \tau ' \) may be chosen to be about 10% larger than the value of \( \tau \). The choice of different values of \( \tau \) provides different tradeoffs between the average (true)
Density Biased Sampling: Density Biased Sampling is proposed to probabilistically under-sample dense regions and oversample light regions. A weighted sample is used to preserve the densities of the original data. Density biased sampling naturally includes uniform sampling as a special case.

Density biased sampling requires that the data be partitioned into groups. Numerical attributes are divided into $G$ bins and categorical attributes have a bin for each category. When dealing entirely with $d$-dimensional numerical data, the space is divided into bins by placing a $d$-dimensional grid over the data. If the data is drawn from a low dimensional space and the number of occupied bins, $B$, is small, we can compute the bin counts using $O(d \times B)$ bytes. This is an exact density biased sample. If too many of the bins are occupied, an approximate histogram algorithm can be used. But, in higher dimensions, it becomes prohibitively expensive to merely represent the occupied bins. Those potential implementations suffer from the lack of available memory. [8] Proposed a hashing based approach where all available memory is used to create an array of bin counts. Call this array $n$ and assume that it has $H$ entries (indexed from 0). To index into this array, use a hash function from the bin label to array index (see Figure 2). The bin labels are integers (either in the range $0,...,G-1$ for numerical data or $0,...,c-1$ for categorical data with $c$ categories). Hashing bin labels should be similar to hashing strings since each element is expected to be drawn from a relatively small range and we expect values to frequently differ in only one position (adjacent bins). For simplicity, assume that $h(x)$ is a function that takes value $x$, quantizes it and then invokes hash on the quantized version. Then the two pass algorithm using a hash function to approximate density biased sampling is trivial. The second pass over the data makes this an unappealing algorithm. If $0 \leq \varepsilon \leq 1$, this algorithm can be converted into a one pass reservoir style algorithm.

To eliminate second pass over the data, densities and samples are built in parallel. Obviously this one pass algorithm can be used for any representation of the bin densities. Hashing is only used to map from input point to group size and any other mapping could be used here instead. Density biased sampling naturally includes uniform sampling as a special case. Here density biased sampling is implemented using a hashing function to map bins in space to a linear ordering, allowing it to work with very limited memory. They define density preserving to mean that the expected sum of the weights of the sampled points for each group is proportional to the group's size. That is, if group $i$ contains the points $\{x_1, x_2, ..., x_n\}$, point $x_j$ is included in the sample (with weight $w_j$) with probability $P(x_j)$.

Density biased sampling technique based on the reservoir method was proposed by [9]. The inherent advantage of efficient memory usage in the reservoir scheme is adopted and extended with the additional capability of dealing with data that are much different in density distribution. The proposed technique is designed to lessen the effect of noise as it is the case in the hash-based approach.
Proposed algorithm generalizes the reservoir scheme for the case of data with different density distribution. In the initial step, data is partitioned into groups. But the subsequent steps are not based on a hashing scheme in order to avoid the effect of noise and collision problems. After the initial step of dividing the data space into bins of equal size, the information of the first \( n \) groups are put into the \( n \) reservoirs residing in main memory. The collected information includes the number of points in each group and the id of the group. The algorithm performs a single scan on a data set in a random manner controlled by a random variable \( S \) with the distribution \( W \). The density biasing is achieved through the consideration of two consecutive data groups. Here Sampling is used for clustering.

**Algorithm 2.3** Density-biased reservoir sampling[9]

Input: a data set of \( N \) objects

Output: a density-biased sample of size \( n \) (\( n \leq N \)) associated with weight \( w \)

1. Partition data into \( g \) groups (with group-id 1, 2, ..., \( g \)), \( g \geq n \)
2. Initialize the reservoir \( X_1, ..., X_n \) to be the first \( n \) \(<\text{group-id, density}>\)-pairs of the data groups
3. Set \( W = \exp(\log(\text{random()}) / n) \) // initialize \( W \) that will be used in the
4. Set $S = \log(\text{random}()) / \log(1-W)$.
5. While $S < g$ do
6. Read data groups $g_S$ and $g_{S+1}$ // read two consecutive data groups
7. If ($||\text{density}(g_S) - \text{density}(g_{S+1})|| > \delta$) OR
   ($\text{density}(g_S) + \text{density}(g_{S+1}) > \varepsilon$) // $\delta$ and $\varepsilon$ are predefined density threshold values
   Then $X_{1+n*\text{random}} = <\text{group-id, density}>$ of maximum
   density$\{g_S, g_{S+1}\}$
   // randomized the reservoir area to be updated
8. $W = W * \exp(\log(\text{random}()) / n)$ // update $W$ for the skipping process
9. $S = \lfloor \log(\text{random}()) / \log(1-W) \rfloor$ // generate $S$ to denote the number of
   // groups to be skipped over
10. Return $X_1, ..., X_n$

2.6 Synopsis construction over transaction stream

Algorithm 2.1 (Conventional reservoir sampling) can be modified to work with data stream. In this section the conventional reservoir sampling method is modified for synopsis construction over transaction stream. The reservoir sampling method is very easy to understand as it generates a sample of the original data representation. However, the classical unbiased reservoir method is inaccurate for transaction streams, this is due to the fact that when the stream length increases, the accuracy of the reservoir decreases as it will contain a large portion of points from the distant history of the stream (the probability of successive insertions of new points reduces with the progression of the stream) and in an evolving data stream only the more recent data may be relevant for many mining tasks. To overcome this problem stream is divided into fixed size windows of transactions. Samples from each window are taken and stored for future processing. Advantage of this algorithm is it gives samples of historical as well as recent data.

Algorithm 2.4 : reservoir sampling

Inputs: $r$ {reservoir size}
1. $k = 0$
2. for each tuple arriving from the input stream do
3. $k = k + 1$
4. if \( k < r \) then
5. add the tuple to the reservoir
6. else
7. sample the tuple with the probability \( \text{prob}(r/k) \) and
8. generate a random number between \( \text{rnd}[0,1] \)
9. if \( (\text{prob}<\text{rnd}) \) then
10. replace a randomly selected tuple in the reservoir with the sampled tuple
11. endif
12. end if
13. end for

Algorithm 2.4 is executed for each transaction window and output is stored on a secondary storage for further analysis if required. Formally, we assume that the stream of transactions is broken up into fixed sized batches \( t_1, t_2, t_3, \ldots t_n, \ldots \), where \( t_n \) is the most current batch and \( t_1 \) the oldest.

**Experimental results and Performance study**

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-A. We report our experimental results on the chess dataset downloaded from [fimi.cs.helsinki.fi/data](http://fimi.cs.helsinki.fi/data).

Experiments are carried out on different window sizes with different sample sizes. In first experiment transaction stream data set is divided into windows (fixed batches), each containing 100 transactions. Out of 100 transactions only 10 transactions are sampled. Following graph shows number of times the reservoir is updated when the reservoir size is 10.

![Graph showing number of updates](image)
Second experiment was carried out on window size of 1000 and reservoir size 100. Following graph shows the frequency of updates to the reservoir.

Fig- 2.3

Program uses a double dimension array (Reservoir); each row corresponds to one transaction. Implementation is static requiring continuous memory locations. Dynamic implementation requires building a link list of items and then storing only the pointer to the first node in the reservoir. Dynamic method is time consuming requiring allocation and deallocation of memory. As we have divided the stream into windows it requires very less memory and due to static data structure execution time is also very less.

2.7 Mining Frequent Itemset Using reservoir sampling

In the present section we propose a new algorithm for mining frequent patterns over data stream. Proposed algorithm belongs to the class of combination of reservoir sampling and counting. Studies show that reservoir sampling can also be used for mining frequent items over data stream. Here algorithm maintains a reservoir of distinct items and uses a random function to decide whether to add the current transaction to the reservoir or no. If the new transaction is to be added then counters of the items corresponding to new transaction are incremented by one and the transaction is discarded.

Algorithm 2.5 :

Input: \(r \{\text{reservoir size}\} \ 1: k = \text{number of distinct items in the data stream}\)
Output: frequent item sets

1. for each tuple arriving from the input stream do
2. sample the tuple, generate a random number between $rnd[0,1]$
3.   if ($rnd==0$) then
4.     increment the counters of its items in the reservoir
5.     if (New item found) then
6.       select an item with minimum counter and replace it with new item
7.     endif
8.   endif
9. end for

Here if there is no empty slot to add a new item, one of the items from the reservoir is selected for replacement. Selection of an item for replacement is carried out using LRR (Least Recently Replaced) technique. Here with each item a weight is associated which indicates number of times the item is replaced. Whenever a new item needs to be added to the reservoir then the item with least weight is selected for replacement. Thus at the end reservoir consists of the most recent frequent items.

Experimental results and Performance study

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-B. We report our experimental results on the chess dataset downloaded from [fimi.cs.helsinki.fi/data](http://fimi.cs.helsinki.fi/data).

First experiment selected 3112 transactions from total 3,196 transactions where number of distinct items are 75 and average transaction length is 37. Following graph shows number of frequent items found in the stream. Frequency range of frequent items is put into following ranges Greater than 400, 400>= frequency >300, 300>=frequency >200, 200>=frequency>100 and less than 100.
Second experiment was carried where 836 transactions were added to the reservoir out of 3196 transactions. Following graph shows the frequency of frequent items.

![Graph 2.4](image)

Third experiment was carried where 383 transactions were added to the reservoir out of 3196 transactions. Following graph shows the frequency of frequent items.

![Graph 2.5](image)

Program uses a single dimension array (Reservoir), corresponding to one transaction. Implementation is static requiring continuous memory locations. Approximate frequent items are the output as sampling is used for selection.

### 2.8 Conclusion

Studies show that the algorithm 2.4 and 2.5 gives good results depending on the efficiency of random function, in the program. Random function was initialized using time function every time before execution to get better results. These algorithms require very less memory as the reservoir size is small. As the stream is divided into windows...
Algorithm 2.4 gives unbiased random uniform sample representing historical as well as current data. It also forms a preliminary step towards clustering of data.

If algorithm 2.5 is implemented on market basket data it can predict the items which are currently in more demand, which in turn helps in deciding the reorder level of the stock. It also gives some type of sequential association between items in the stream. As it was observed purchasing of item 1 was mostly followed by item 3 and item 5.

Algorithm 2.5 can be further extended to be more interactive by taking the number of frequent items to be mined from the user. Algorithm works well if the numbers of distinct items are known in advance. Some extra memory space for new items can be added in case of data stream with unknown distinct items in advance to the existing reservoir.

An article based on algorithm 2.4 is published in the proceedings of a National conference [21].

The work based on algorithm 2.5 has been submitted to a National journal for possible publication.
2.9 References


Workshop on data stream analysis – san leucio, italy - march, 15-16, 2007


