Survey of Streaming Data Algorithms

Supun Kamburugamuve
For the PhD Qualifying Exam

Advisory Committee
Prof. Geoffrey Fox
Prof. David Leake
Prof. Judy Qiu

Table of Contents

1. Introduction

The number of applications relying on data produced in real time is increasing rapidly. Network traffic monitoring systems, sensor networks for monitoring manufacturing processes or buildings are examples of such applications. The data coming from these sources must be processed before they become outdated and the decisions must be taken in real time. Since large volume of data is produced, large-scale Distributed stream processing engines (DSPEs) are developed to handle the processing in a distributed fashion. Companies are moving from batch processing to real time analysis to gain valuable information as quickly as possible. Data mining algorithms provide nice frameworks for processing data and there are streaming versions of the most popular data mining algorithms available in the literature. Other than the data mining algorithms there are various algorithms developed for finding statistics about the streaming data as well. Statistics over the most recently observed data elements are often required in applications involving data streams, such as intrusion detection in network monitoring, stock price prediction in financial markets, web log mining for access prediction, and user click stream mining for personalization.

There are some inherent challenges for stream-based algorithms. Due to the continuous, unbounded high-speed nature of the data streams, the amount of data processed by these algorithms is huge and there is not enough time to go through the data again and again iteratively. Also there is not enough space to store these unbounded data streams. Due to this, single pass algorithms that can work with a small amount of memory are necessary. In some domains underlying data distribution of a stream can change as the time pass by and the algorithms should be able to adapt to these changes in the streams, otherwise the concept-drifting problem [1] can take place. The algorithms should process data faster than they arrive to avoid shredding of data and data sampling. The results of the streaming algorithms should be available at any time. We can summarize the characteristics of the stream processing algorithms as following.

1. The data is processed continuously in single items or small batches of data
2. Single pass over the data
3. Memory and time bounded

4. The results of the processing available continuously

Zhu and Shasha [2] identified three generic models used by the steam processing algorithms. They are landmark, damping and sliding windows models. In the landmark model, data tuples over entire history of the data stream from a specific point in time (landmark point) are used. This model is not suitable for applications where recent information is more important than the past information. In the damping model, a higher weight is given to most recent data items and less weight is given to older items making the most recent data more important than the older data. In sliding windows model, the algorithms do calculations in the sliding windows. This kind of processing model allows users to answer queries for a specific time window. The use of these three methods primarily depends on the application itself. Most of the algorithms developed for the landmark model can be converted to damping model by adding decay functions to the data. Also landmark model algorithms can be converted to sliding window model by keeping the processing within a window.

Few years ago, knowledge discovery algorithms are constrained by the limited resources of time, memory and sample sizes. It was very hard to find rich sets of data collections to run the data mining algorithms. Now data is available in very large volumes and limiting factors are the time and memory available. Most of the streaming algorithms discussed in this report are developed for processing very large amount of data stored in databases. Nevertheless they can be used in a real time streaming setting as well. The streaming algorithms developed over the years were designed to work on a single stream of data. Because the volume of the data generated in real time is increasing rapidly; it is no longer sufficient to run these algorithms in a sequential manner. These algorithms should be adapted to run in parallel with streams of data partitioned to many distributed processing units.

There are many streaming algorithms available in the literature and in this report we will look at Clustering algorithms, Classification algorithms and some statistic calculation algorithms.

2. Clustering algorithms

Given a set of items clustering algorithms put the items more alike to item groups called clusters such that the items in a cluster are more close to the items in same cluster than the items in other clusters. The closeness of the items is defined by various metrics and they depend on the item set and the application. Clustering is an important process in data analysis. Most of the clustering algorithms are unsupervised learning algorithms and can find the clusters without any training. Clustering algorithms can discover the hidden relationships among the items. There are various models developed for clustering data items. Some of these models are based on Connectivity models, Centroid models, Graph based models and Group models.

For our discussion the clustering problem can be formally defined as, given an integer k and a collection of n points in a metric space, find k medians such that each point in the data collection is assigned to the closest median point. The quality of the clusters is defined by the sum of squared distance from the assigned medians. This form of strict clustering is known as the K-Median problem and it is known to be NP-hard. One of the most famous and widely used clustering algorithms is k-Means clustering which is developed as a heuristic version to the k-Medians problem and is guaranteed to produce only a local optimum solution. The k-Means algorithms require $\Omega(n^2)$ and random access to the data. Because of the time requirements and the random access requirements k-means is not suitable for a stream setting.

2.1 Stream Algorithm

One of the early works on successful clustering for streams is done according to the above idea and is called STREAM [3] algorithm. The STREAM algorithm divides the stream in to sequence of batches. For each of the batches the algorithm uses an efficient clustering algorithm called LOCALSEARCH to produce local clusters. The local search is similar to K-Means but performs better than K-Means in terms of memory and time. These

cluster points created for each batch of data can be feed again in to the LOCALSEARCH clustering algorithm and this will give the global clustering across the whole stream. Here is the STREAM algorithm

The data stream is divided in to chunks and these chunks are $X_1, X_2, \dots, X_n$ .For each chunk the algorithms creates k centers.

For each chunk X_i

1. If a sample of size $\geq \frac{1}{\varepsilon}\log\frac{k}{\varepsilon}$ contains fewer than k distinct points, for each unique point assign a weigh of its frequency and remove the duplicates
2. Cluster X_i using LOCALSEARCH
3. Assign all the clusters created so far to X'. For each clustering point the weight of the point is the number of items in the cluster
4. Output k clusters by running LOCALSEARCH on X'

The LOCALSEARCH algorithm starts with an initial solution and refines this by making local improvements. The algorithm views the k-median problem as a facility location problem. A cluster center is named as a *facility* and grouping of the elements in to clusters incurs a *facility cost*. An exact solution to the problem is known to be NP-hard and a heuristic based solution is given.

The running time of local search algorithm is $O(nm + nk\log k)$ where m Is the number of initial facility locations opened by the initial solution which is comparatively small. The space requirement for this algorithm at a time is $O(ik)$ and this can become quite large as the time increases. So after sometime the clustering algorithm is run on the clusters and only a summery is taken from that point onward.

2.2 Evolving Data Streams

The previous clustering algorithm we discussed were using the entire data stream for calculating the clusters. These algorithms are one-pass algorithms over the data stream. There are applications where such a view of the entire stream is not suitable. For example in some applications the underlying clusters may change considerably with time. So the real clusters at the point of analysis may be different from the clusters that were seen at a previous time and taking those previous information to create the current clusters may produce incorrect results. Also users may want to look at the clusters that occurred in the previous month, previous year or previous ten years.

To cater for these requirements a two-phase clustering algorithm is developed [4]. The algorithm consists of an online phase and an offline phase. In the online phase the algorithms create micro clusters at given time intervals and these clusters are saved for the second phase. In the next phase the algorithm uses the saved micro clusters along with the summery statistics to create a macro view of the clusters. The second phase can be carried out over any given time period with input data from users.

Online Micro Clusters
The micro clusters are maintained over the whole history of the data stream. At the initial step of the algorithm a sufficient amount of incoming data is put to the disk and a traditional K-Means clustering algorithm is run on this data to get the initial micro clusters.

After this initial step when the new data points arrive, the micro clusters are updated to reflect these new data points. An existing cluster can absorb a new data point or a new data point can create a new cluster. To check whether the new data point belongs to an existing cluster the distance to each of the clusters from the new data point is calculated. If this distance is less than the distance of a cluster's maximum boundary distance and if the distance is the minimum such distance from all the clusters, the new data point can be assigned to a cluster. If the existing clusters cannot absorb the new point the new data point must be an outlier or it must belong to a new cluster. These two cases cannot be distinguished without accessing further data. So a new cluster has to be created. To create a new cluster an existing two clusters must be merged together or an existing cluster must be deleted. The micro clusters are saved to disk periodically for the

macro cluster algorithm to work. The micro clusters only contain very little information compared to the data used to create the clusters. Each micro cluster maintains how many data points belong to that cluster as a weight.

Macro Cluster Creation
The macro cluster creation is a process invoked by user and it is done for a time period. This step is done offline. The micro clusters in the previous section were created using the whole data stream. Since macro clusters are created for a specific time period the previous clusters must be subtracted from the clusters in the time widow to get the clusters created in that time window. After the micro clusters are obtained for a time window, these micro clusters are feed in to a K-Means clustering algorithm to get the overall clusters during the period of time.

3. Classification

Classification algorithms identify which of a set of categories a new item belongs to base on a training data set where the classification is known. Classification is a supervised learning task. Some of the most widely used classifiers are neural networks, support vector machines, k-nearest neighbors, naïve Bayes, decision trees and radial basis functions. The decision tree classifiers don't give good theoretical basis like in neural networks for classification but works extremely well in practice. Specially used with methods like Random Forrests, the decision trees known to outperform most of the other classifiers.

3.1 Hoeffding Trees

Domingos and Hulten [5] introduced the Hoeffding Trees for data stream classification. The main contribution from this paper was Hoeffding tree, which is a new decision tree learning method for streams. The algorithm induces a decision tree from the stream data incrementally spending very limited time on each new data item. The algorithm doesn't require the items to be stored after the item is processed and tree is updated. The algorithm only keeps the tree in the memory. The tree stores all the information required for its growth in the leaves and can classify items while it is in the building phase.

Each node of a decision tree contains a test on an attribute of the data set. This test determines which path a data item should take. A decision tree classifier sends a data item from its root to leaves based on the tests at each node along the path. The leaves are the classes of the classifier. When constructing decision trees usually the algorithms starts from an empty node set and construct nodes based on the attributes that do the best split. There are heuristics like information gain, Gini index for choosing the attribute that does the best split. In a batch algorithm, because all the training data is available the algorithm can calculate information gain or Gini index for each of the attribute to choose the best. For a stream setting because it cannot access all the data, the problem is to decide how many data points has to be seen before deciding to split based on an attribute. The Hoeffding tree gives an innovative method for making this decision in a stream setting. The Hoeffding bound [6] is a statistical result used by the Hoeffding trees to achieve this. Let's take a real valued random variable r whose range is R. Assume n observations of this variable are made and computed the mean $\bar{r}$. The Hoeffdin bound states that, with probability $1-\delta$, the true mean of the variable is at least $\bar{r}-\epsilon$ where $\epsilon = \sqrt{\frac{R^2 \ln(\frac{1}{\delta})}{2n}}$.

In the Hoeffding tree algorithm, the random variable is chosen as the difference between the gains of the two best attributes. Let $G(X_i)$ be the heuristic measure used on attribute X_i i.e. Information Gain or Gini Index. If the best attribute is X_a and second best attribute is X_b after observing n items, the random variable is taken as $\Delta\bar{G} = \bar{G}(X_a) - \bar{G}(X_b) \geq 0$. The Hoeffding bound says that X_a is the correct choice with probability $1-\delta$ if n samples have seen and $\Delta\bar{G} > \epsilon^2$. When $\Delta\bar{G} > \epsilon$, Hoeffding bound guarantees that true $\Delta G \geq \Delta\bar{G} - \epsilon > 0$. This means that the algorithm has to accumulate n samples at a leave until $\Delta\bar{G} > \epsilon$ to decide on an attribute to split. At this point algorithm can split based on the current best attribute. When new items arrive the algorithm keeps the statistics about the new item in the leaf nodes. For information gain like measure with discrete attribute values, only counts of different attribute values is sufficient at this level. Then the algorithm

checks to see if a split is possible using the Hoeffding bound. The paper proves that the trees generated by the Hoeffding algorithm are sufficiently close to the trees produce by batch algorithms.

4. Quantile Computation

Quantile are points taken at regular intervals from a cumulative distribution function (CDF) of a random variable. A ϕ-qunatile of an ordered sequence of N data items is the value with rank ϕN. A quantile summery consists of small number of data points from the original data sequence and use this information to calculate any quantile queries. For calculating the quantiles the whole history of the data stream or last N items seen can be used. It is proved that to calculate the exact quantiles for a stream with N items using p passes (where $p \geq 2$), an algorithm requires space of the order $\Omega\left(N^{\frac{1}{p}}\right)$ [6]. For a single pass algorithm, $\frac{N}{2}$ memory spaces are required to calculate the exact quantile summaries. Clearly this is not practical for unbounded streams of data and ε-approximation algorithms have being developed that can compute the quantiles with reasonable space requirements. An element is said to be a ε-approximation if its rank is within the range $[(\phi - \varepsilon)N, (\phi + \varepsilon)N]$.

4.1 GK-Algorithm

GK Algorithm [7] is a ε-approximation that maintains the quantiles summaries over all the data it has seen. Number of items seen so far is n and ε is the precision requirement. The algorithm maintains a data structure S over the items it has seen. S maintains an ordered subset of sequence of elements chosen from the items seen so far. Fox each observation v in S the algorithm maintains an implicit minimum rank $R_{min}(v)$ and maximum rank $R_{max}(v)$ over the n items. Given this quantile summery a ϕ quantile v_i can be found using the following property. $\lceil \phi n \rceil - \varepsilon \leq R_{min}(v_i)$ and $\lceil \phi n \rceil + \varepsilon \geq R_{max}(v_i)$.

A tuple t_i in S is of the form $\left(v_i, g_{i,} \Delta_i \right)$ where $g_{i,} = R_{min}(v_i) - R_{min}(v_{i-1})$ and $\Delta_i = R_{max}(v_i) - R_{min}(v_i)$. Every new observations is inserted to the S. Then periodically the algorithm merges some tuples in S and maintains the property $MAX_i(g_i + \Delta_i) \leq 2\varepsilon n$. A tuple is considered full if $g_i + \Delta_i = 2\varepsilon n$. A tuple's capacity is the number of tuples it can count without becoming full. In the merge step the tuples with small capacities are merged to tuples with similar or higher capacities.

The algorithm supports operations to answer quantile queries, insert an item to S, delete an item from S and compress S. The algorithm requires $O\left(\frac{1}{\varepsilon} log(\varepsilon N)\right)$ space in the worst case.

4.2 Sliding Windows

The recent data is more important to many applications than the items arrived a long time ago. Sliding window quantile summery algorithms have being designed for such applications [8]. A sliding window over a data stream is a collection of consecutive N elements. There are fixed length sliding windows where N is fixed and variable length sliding windows where N can change. Sliding windows over a time period can be taken as variable length sliding windows. The basic idea is to partition the stream in the sliding window to small blocks and maintain sketches to hold quantile summaries for the blocks. A sketch for a block is calculated using an algorithm like GK algorithm as described in the previous section. The data in the block sketches are combined to answer the quantile queries.

Fixed size windows

The algorithm maintains several levels. Levels are numbered 0,1,2. For each level, the stream is partitioned into non-overlapping, equal sized blocks. The level 0 contains $\frac{\varepsilon N}{4}$ blocks and level l contains $\frac{\varepsilon N}{4} 2^l$ blocks. Within a level blocks are numbered from 0,1,2,.. and smaller numbered blocks contain older elements. Four states are defined for blocks. A block is active if it is completely within the current window, expired if one of its elements is outside the current window, under construction if the items are still arriving, inactive if none

of the items have arrived. When a block is under construction the GK algorithm is run to create the sketch. When block becomes active the sketch is saved with the block.
To answer a query within the window the saved sketches for the active blocks are used. The values in the sketches are put together and these overall values are sorted to get the final sketch. Then these values are used as a regular sketch, described in the previous section.

5. Frequent item sets mining

Frequent item set mining is one of the best-studied areas in streaming algorithms [9]. Given a sequence of items frequent item sets give the most frequently occurring k items in the sequence. Typically the most frequent items are identified as the items occurring more than a fraction of the total number of items seen. The frequent items can be calculated with respect to a time window or for the whole sequence of items. Frequent item mining in streams of data is used by search engines to figure out the most popular searches at a given time period and optimize for those queries. By figuring out the most frequent IP addresses in Internet traffic an Internet service provider can find the heavy users of the Internet services.

Solutions for frequent item set mining can be found in the literature as back in early 1980s. There are various algorithms invented and re-invented in the area of frequent item set mining. These algorithms can be classified in to three broad categories. Count based algorithms keeps the counts for subset of the inputs and updates the count and changes the items in the subset depending on the incoming tuples. The second class of algorithms is based on Quantile computations for the items. The last class of algorithms views the input as vectors and projects the input space to another space and they are called sketch-based algorithms. There are algorithms that use random sampling from the input streams and they are not widely used.

The frequent items problem in the strictest form is very hard to solve using streaming algorithms because it requires space linear to the item set size. Because of this relaxed version of the problem is defined. Most of the use cases that involve frequent items don't need the most accurate results. For example if a search engine keeps track of the 100 most frequent search queries and the difference between the 100th item and the 101st item is one it doesn't make much difference if we pick any of the two.

5.1 Exact Frequent Items

Given a stream of S of n items $s_1, s_2 \ldots s_2$ the frequency of an item k is $f_k = \left|\{r|s_j = k\}\right|$. The exact ϕ-frequent items problem finds the items comprise the set $\{i|f_i > \phi n\}$.

5.2 The ε-approximate frequent items problem

In this problem, to find the R frequent items we relax the requirement of R to be in a range. $\{i|f_i > (\phi - \varepsilon)n\}$ and for every element in the data set such that $f_i > \phi n$, that element belongs to the frequent items.

All the algorithms present here are ε-approximate frequent items finding algorithms.

5.3 Count based algorithms

In count-based algorithms the algorithm keeps a constant subset of the stream along with the counts for these items.

Frequent Algorithm

This algorithm is known as the Misra-Gries algorithm [10]. To find the k frequent items the algorithm keeps the $k - 1$ (item, count) pairs. The count for an item corresponds to the number of the item seen so far. When an item arrives it is compared to the items stored so far and it is in the store increment the corresponding counter by one. If it is not in the store and if there is less than k – 1 item in the store, item is stored and count is set to 1. If there is k – 1 item already in the store decrement the count of all the elements in the store and remove an item if its count become 0 and add the new item. Otherwise the new item is discarded. The algorithm is simple to implement and consumes a very small amount of memory and can be implemented

using hash tables for fast lookups. It is proved that by setting the $k = \frac{1}{\varepsilon}$ ensures that the count associated with each item is at most εn below the true value. The algorithm requires $O(1)$ processing time and $O(k)$ spaces.

Lossy Counting

Manku and Motwani [11] developed the Lossy Counting algorithm. In this approach the item are stored as tuples of $(item, f, \Delta)$. Here Δ is the maximum possible error and f is the current count. The stream is logically divided in to buckets of size $w = \left\lfloor \frac{1}{\varepsilon} \right\rfloor$ and these buckets have ids starting from 1 and is denoted by b_c. When ith item is encountered, the algorithm checks to see if the item is already stored. If it is stored its f value is incremented. If it is not in the stored items, we create an entry with the value $(item, 1, b_c - 1)$. The algorithm periodically deletes items from the list that have $f + \Delta \leq b_c$. Δ value of an item doesn't change after insertion. The algorithm requires $O(\frac{1}{\varepsilon} \log \varepsilon n)$ spaces in the worst case and $O(1)$ processing time. This algorithm performs better for skewed inputs than the Misra-Gries algorithm presented earlier.

5.4 Sketch Algorithms

In these algorithms the input is projected in to a data structure called a sketch and this structure is used to estimate the frequencies of the items.

CountS-Sketch

The CountS-Sketch algorithm of Charikar et al [12] provides a space and time efficient algorithm for finding frequent items. This algorithm maintains a data structure called count sketch. The count sketch consists of $t \times b$ matrix of counters C. Also it has t hash functions $h_1, h_1, \dots, h_t$ which maps items to $\{1, \dots, b\}$ and another t hash functions $s_1, s_2, \dots, s_t$ which maps items to $\{+1, -1\}$. Each input item i is hashed using all s_j and these values are added to $C[j, h_j(i)]$ where $1 \leq j \leq b$. Now to estimate the frequency of a value i it uses the following formula $median_j\left(s_j(i) C\left[j, h_j(i)\right]\right)$ where $1 \leq j \leq b$.

Algorithm maintains a fixed size queue of items, which contains the most frequent items along with a count. A new item is first added to the count sketch. Then if the item is in the queue, its count is incremented. Else, it uses the count sketch to estimate the frequency of the new item and if this estimation is greater than the min count in the queue, delete the min item from the queue and add the new item to the queue. The algorithm assumes that the hash functions s_i and h_i are pairwise independent. Also $t = \log \frac{4}{\delta}$ where δ is the probability of algorithm failing and $b = O(\frac{1}{\varepsilon^2})$. The algorithm has $O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta})$ space complexity and $O(\log \frac{1}{\delta})$ time complexity for updates.

CountMin Sketch

The CountMin Sketch algorithm by Cormode [13] is similar to the CountS-Sketch algorithm described above. It also maintains a $t \times b$ matrix of counters C. It also has t hash functions $h_1, h_1, \dots, h_t$ which maps items to $\{1, \dots, b\}$. The hash functions are pairwise independent. The CountMin sketch is updated with $C\left[j, h_j(i_t)\right] = C\left[j, h_j(i_t)\right] + c_t$ for $\forall\ 1 \leq j \leq b$ when an update (i_t, c_t) arrives. A frequency estimation for item i is given by $min_j\left(C\left[j, h_j(i)\right]\right)$ where $1 \leq j \leq b$. Here $t = \log \frac{1}{\delta}$ and $b = \frac{e}{\varepsilon^2}$.

The rest of the algorithm is same as the previous algorithm. The algorithm maintains a fixed size queue of items, and for each new item the CountMin sketch is updated. If the new item is in the queue, its count is incremented. Else it uses the CountMin sketch to estimate the frequency of the new item and if this estimation is greater than the min frequency item in the queue, the new item is added to the queue and min item is removed from the queue. The algorithm has $O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ space complexity and $O(\log \frac{1}{\delta})$ time complexity for updates.

Sliding Windows

The algorithm described above can be used with the sliding windows [8] protocols described in the section 4.2 to calculate the frequent items over time windows. For example, the items along with the counts

described in the Misra-Gries [10] algorithm can be saved for each of the active window. The counts of the same item occurred in multiple blocks are summed up to get an overall count for an item for the window. Then this information is used to answer the approximate queries.

5.5. Association Rule Mining

Association rule mining [15] can be used to discover interesting relationships among the attributes in a data set. The technique is widely used to discover the buying patterns of customers in large-scale transaction data recorded in supermarkets and online shopping websites like Amazon. For example an association rule may say $\{phone\} \Rightarrow \{phone\ cover\}$. This rule implies that people who buy phones also buys phone covers with it. Other than the market basket analysis, association rule mining can be used to estimate missing data in sensor networks, monitor-manufacturing flows, predict failure using web log streams.

Let $I = \{i_1, \dots, i_n\}$ be a set of binary attributes and these represent the items and $D = \{t_1, \dots, t_m\}$ be set of transactions. Each transaction contains a set of items from I. An association rule is of the form $X \Rightarrow Y\ where\ X, Y \subset I\ and\ X \cap Y = \emptyset$. To discover the rules support and confidence of an item set is used. $support(X)$ of an item set is the number of transactions with X divided by the total number of transactions. $confident(X \Rightarrow Y) = \frac{support(X \cup Y)}{support(X)}$.

To find the association rules a frequent item set mining algorithm can be used. First the algorithm can find the support of an attribute set by finding the frequent items. For these items confidence can be calculated again using the frequent items mining algorithms. Jiang and Gruenwald [16] have explained some of the challenges in association rule mining in a streaming environment.

6. Discussion

In this report we have identified some of the key areas where streaming algorithms are developed and used. The algorithms described in this report are all assume to run on a sequentially arriving stream of data. In practice however this is not the case and large number of tuples arrives simultaneously in parallel. Single node sequential processing is not sufficient for executing these algorithms under a reasonable time bound. The solution is to run these algorithms in parallel using large clusters. Running these algorithms in parallel using distributed stream computing environment is a non-trivial task. Distributed stream processing engines provide streaming frameworks on which these algorithms can be executed. Not much research is done on how to parallelize these algorithms for such environments. The distributed processing engines have different programming models and stream partitioning models. It is interesting to explore the possibilities of running these algorithms on the programming models imposed by these computation frameworks. At this moment it is not even clear weather the programming models provided by the distributed stream processing engines are sufficient for efficiently executing these algorithms.

7. References

[1] Haixun, Wei Fan, Philip S. Yu, and Jiawei Han Wang, "Mining concept-drifting data streams using ensemble classifiers," in *In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 226-235*, 2003.

[2] Yunyue, and Dennis Shasha Zhu, "Statstream: Statistical monitoring of thousands of data streams in real time," in *In Proceedings of the 28th international conference on Very Large Data Bases, pp. 358-369*, 2002.

[3] Liadan, Nina Mishra, Adam Meyerson, Sudipto Guha, and Rajeev Motwani O'callaghan, "Streaming-data algorithms for high-quality clustering.," in *In Data Engineering, 2002. Proceedings. 18th International Conference on, pp. 685-694*, 2002.

[4] Charu C., Jiawei Han, Jianyong Wang, and Philip S. Yu Aggarwal, "A framework for clustering evolving data streams," in *In Proceedings of the 29th international conference on Very large data bases-Volume 29,*

pp. 81-92, 2003.

[5] Pedro, and Geoff Hulten Domingos, "Mining high-speed data streams," in *In Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 71-80*, 2000.

[6] J. Munro and M. Paterson, "Selection and sorting wiith lim- ited storage.," in *In TCS12*, 1980.

[7] Michael, and Sanjeev Khanna Greenwald, "Space-efficient online computation of quantile summaries," in *In ACM SIGMOD Record, vol. 30, no. 2, pp. 58-66.* , 2001.

[8] Arvind, and Gurmeet Singh Manku Arasu, "Approximate counts and quantiles over sliding windows," in *In Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 286-296*, 2004.

[9] Graham, and Marios Hadjieleftheriou Cormode, "Finding frequent items in data streams," in *Proceedings of the VLDB Endowment 1, no. 2 (2008): 1530-1541*.

[10] Jayadev, and David Gries Misra, "Finding repeated elements," in *Science of computer programming 2, no. 2 (1982): 143-152*.

[11] Gurmeet Singh, and Rajeev Motwani Manku, ""Approximate frequency counts over data streams," in *In Proceedings of the 28th international conference on Very Large Data Bases, pp. 346-357*, 2002.

[12] Moses, Kevin Chen, and Martin Farach-Colton Charikar, "Finding frequent items in data streams," in *In Automata, Languages and Programming, pp. 693-703. Springer Berlin Heidelberg*, 2002.

[13] Graham, and S. Muthukrishnan Cormode, "An improved data stream summary: the count-min sketch and its applications," in *Journal of Algorithms 55, no. 1 (2005): 58-75*.

[14] Nan, and Le Gruenwald Jiang, "Research issues in data stream association rule mining," in *ACM Sigmod Record 35, no. 1 (2006): 14-19*.