International Journal of Applied Engineering Research ISSN 0973-4562 Volume 14, Number 9 (2019) pp. 2238-2243

Algorithms Associated with Streaming Data Problems

Ramesh Balasubramaniam[1]*, Dr. K. Nandhini[2]

[1] Research Scholar, [2] Assistant Professor

PG and Research Department of Computer Science,
Chikkanna Govt. Arts College (Bharathiyar University), Tirupur, India.

**Corresponding Author*

Abstract:

The constant frequency and variety of high-speed data have resulted in the growth of large data sets in the last two decades. Storage of this data in memory is not possible due to storage constraints. As a result, today's streaming algorithms need to work with one pass of data. Many streaming algorithms address different problems with data streams. Data stream problems though originated around the '70s have gained popularity only in that last 15 years due to the exponential growth of data from a variety of sources. Today's researchers aim at providing solutions to specific problems that are found across many industries. Most of the problems addressed in this paper are the ones researchers have considered as oft faced problems in the industry like heavy-hitters/top k elements or frequency count or membership check or cardinality. Algorithms addressing these problems like Bloom Filter, Count-min Sketch, HyperLogLog and such are being continuously improved for faster and effective results. Hashing is a technique used by many algorithms to resolve massive data problems affecting today's applications. Hashing enables streaming data to be able to store and retrieve data resourcefully. In this paper, we analyze both the problems in streaming data and the various algorithms used to resolve them.

Keywords: Bloom Filter, Count-min Sketch, Data Stream Problems, HyperLogLog, Streaming Data

1. INTRODUCTION

A data stream is defined as a continuous sequence of high-frequency data (Volume and Velocity) from different sources (Variety) which is too big to be stored in memory. There is continuous progress being made on algorithms in recent years due to the advent of streaming data.

A more formal definition of a stream would be, an input stream arriving sequentially a_1, a_2…., item by item, and describes an underlying signal [1] The objective is to compute an output value for a function of the stream using elements from the input stream, for example, Most frequently occurring items, number of distinct items and such. At any instance *t,* an algorithm has access to $a_1, a_2 \ldots a_t$ for which the output to $f(a_1, a_2 \ldots a_t)$ can be calculated.

Since data streams are fast moving data, it is hard to store and reproduce. So, we need algorithms that do a single pass to solve data streaming algorithms. Time series model, Cash register Model, and Turnstile Model are a few streaming models. The difference between the models is how the input streams are handled based on the underlying signals [1]

- Time series model: Each a_i is equal to A[i] which is the underlying signal. They appear in increasing order of *i*. This is useful for data dependent on time series, for example, temperature checks every hour on a day, Stock price note every 5mins.
- Cash register model: Known as the most popular data stream model ai's are increments to A[j]'s. $a_i = (j, I_i), I_i \geq 0$ mean $A_i[j] = A_{i-1}[j] + I_i$ where Ai is the state of the signal after seeing the ith item in the stream. Much as in a cash register, multiple a_i's could increment a given A[j] over time. This is where each value of the signal gives a positive increment to the previous value to obtain a new value. Like a cash register adding occurrences of element values to a counter over time.
- Turnstile Model: Where increments can also be negative. Like a turnstile in an amusement park that keeps count of people entering and exiting a park, the turnstile model allows both positive and negative inputs. Here a_i's are updates to A[j]'s. $a_i = (j, U_i)$ mean $A_i[j] = A_{i-1}[j] + U_i$ where Ai is the state of the signal after seeing the ith item in the stream and $U_i$maybe positive or negative.

Though the turnstile model seems to be the best model it is only from a theoretical purpose; the cash register model is used in most algorithms from a practical purpose. [1]

Many algorithms have been developed to deal with such massive data problems, and that is what we will be discussing in our paper. The paper is sectioned as first the Introduction; the next section lists various streaming problems, and their associated algorithms formalized to resolve it. The following section includes an update on where the latest algorithms are being used today, and lastly the conclusion with our findings.

2. DATASTREAM PROBLEMS AND ASSOCIATED ALGORITHMS

In this section, we will discuss a few problems or frequently asked questions in data streams and different algorithms used to answer them. All One-pass data stream algorithms have:

1. First, an *initialization* section which is executed before the incoming of the stream

2. Second a *processing* section executed each time a signal comes in and
3. Third an *output* section which answers a query about the stream

Streaming algorithms need to manage and analyze the data stream efficiently. The algorithms should be able to handle a rapid, continuous and large volume of streaming data without the privilege of several passes over the data. The below algorithms claim to address all these requirements and are continuously being enhanced by several research authors in the recent decade.

3. FINDING FREQUENT ITEMS/ HEAVY HITTERS/ FREQUENCY OF TOP HITTERS/ FREQUENCIES

The frequency count is all items whose frequency exceeds a specified fraction of the total number of items. Tracking approximate counts for objects arises in many scenarios such as most popular queries in a search engine, commonly purchased items by customers, frequently visited websites, most accessed topics in news streams or social media and such. Solutions to this problem are based on the count and prune strategy of the Misra-Gries algorithm to find approximate frequent itemsets.

3.1. Misra-Gries Algorithm

The first algorithm to solve it was developed in the early 1980s by David Gries of Cornell University and Jayadev Misra of the University of Texas, Austin.

Let D be a stream and α be a parameter. The index i ∈ [n] is an α-heavy element if $f_i^k \geq \alpha F_k$

The Misra-Gries algorithm stores *k-1* (item, counter) pairs instead of a single counter of items from the input. The algorithm compares each new item against the stored items *T* and increments the corresponding counter if it is present. Else, if the item does not exist then some counter with count zero is allocated to the new item, and the counter set to 1. If all *k - 1* counter is allocated to distinct items, then all are decremented by 1. A grouping argument is used to argue that any item which occurs more than n = k times must be stored by the algorithm when it terminates [2].

3.2. Apriori Algorithms

Apriori algorithm was first introduced to find the frequent itemsets based on the Apriori property (i.e., if an itemset frequently occurs then all the subsets of the item set, also occur frequently). As in a supermarket scenario is knowing which items are bought together (frequent itemsets) helps stores shelve them and discount them efficiently increasing sales and help customers purchase their items with more ease.

Algorithm: Apriori

1. Itemset generation: Generating frequent candidate itemsets or which the support is higher than the threshold support. Support is the number of times the item appears.
2. Prune step: Scanning the dataset to find the support of each candidate itemset and prune those whose support is less than the threshold.
3. k = k + 1, go to step 1.

The Apriori algorithm needs at most *l* + 1 scans if the maximum size of a frequent itemset is *l*.

3.3. FP Algorithm

Though the Apriori algorithm achieves excellent performance by reducing the candidate size, for large streaming datasets, they are very costly in terms of candidate generation and scanning for frequencies. Han, Pei & Yin [3] introduced the FP-Growth algorithm to remove the bottlenecks of the Apriori-Algorithm. They construct a compressed tree data structure called FP-tree (Frequent-Pattern tree) that stores the actual transactions from the database as a linked list going through all transactions that contain an item. Only frequent length-1 items will have nodes in the tree, and only frequently occurring nodes will have better chances of sharing nodes than less frequently occurring ones [4]. The search technique employed is a partitioning-based, divide-and-conquer method that looks for shorter patterns and concatenates the suffix hence reducing search costs.

Algorithm: The FP-tree [5]

1. Scan the transaction database DB once. Collect the set of frequent items *F* and their supports. Sort *F* in support descending order as *L*, the list of frequent items.
2. Create the root of an FP-tree, *T*, and label it as "*null*." For each transaction *Trans* in DB do the following:
   - Select and sort the frequent items in *Trans* according to the order of *L*. Let the sorted frequent item list in *Trans* be [*p* | *P*], where *p* is the first element and *P* is the remaining list. Call insert_tree([*p* | *P*], *T*).
   - The function insert_tree([*p* | *P*], *T*) is performed as follows. If *T* has a child *N* such that *N.item-name* = *p.item-name*, then increment *N*'s count by 1; else create a new node *N*, and let it count be 1, its parent link is linked to *T*, and its node-link be linked to the nodes with the same *item-name* via the node-link structure. If *P* is nonempty, call *insert_tree(P, N)* recursively

The FP-tree construction process needs exactly two scans of the transaction database, first to collect the set of frequent items, and second to construct the FP-tree. Time taken to insert a transaction into the FP-tree is O(I*Trans*I).

3.4. Count-Min Sketch Algorithm

Count-min sketch (CMS) probabilistic data structure also can be used to resolve the frequency heavy hitters problem. The count-min sketch uses hash functions for hashing each element

in the stream and keeps a count of the number of times each bucket is hashed to. [6]

Skeletal pseudocode to initialize and update the sketch is shown in the following functions 3 to 5. The code in function 3 initializes the array C of $w \times d$ counters to 0 and picks values for the hash functions based on the prime p. For each Update(i, c) shown in function 4, the total count N is updated with c, and the *for* loop hashes i to its counter in each row, and updates the counter also. The procedure for Estimate(i) shown in function five is almost identical to this loop: given i, we perform the hashing in the first line of the *for* loop and keep track of the smallest value of $C[j, h_j(i)]$ over the d values of j [4].

Function 1: CM_Init (d, p)

$C[1,1] \dots C[d, w] \leftarrow 0;$

$for\ j \leftarrow 1\ to\ d\ do$

$Pick\ a_j, b_j\ uniformly\ from\ [1 \dots p];$

$N = 0$

Function 2: CM_Update (i, c)

$N \leftarrow N + c;$

$for\ j \leftarrow 1\ to\ d\ do$

$h_j(i) = (a_j \times i + b_j\ mod\ w);$

$C[j, h_j(i)] \leftarrow C[j, h_j(i)] + c;$

Function 3: CM_Estimate (i)

$e \leftarrow \infty;$

$for\ j \leftarrow 1\ to\ d\ do$

$h_j(i) = (a_j \times i + b_j\ mod\ p)\ mod\ w;$

$e \leftarrow (e, C[j, h_j(i)]);$

$return\ e$

Using a heap data structure and Count() and Inc() functions of the count-min sketch we get a probabilistic answer to the heavy hitters problem. Count(x) returns the frequency count of x, that is the number of times Inc(x) has been called on in the past. When an item x arrives, querying the data structure, its estimated frequency is obtained. If the count is above a user-defined threshold or m/k insert count (x) into the heap and delete other occurrences of x from the heap. If x has a count less than m/k, it is deleted using *Find-min* and *Extract-Min* operations. In the end, the heap is scanned, and every element that has a frequency >= m/k is returned as a heavy hitter [6].

4. COUNTING DISTINCT ELEMENTS

Counting distinct elements in a stream is a calculation using hashing functions. 2 raised to the power that is the longest sequence of 0's seen in the hash value of any stream element is an estimate of the number of different elements.[7] Using these estimates and taking averages an approximation to the distinct element count is obtained.

4.1. Flajolet-Martin (FM) Algorithm

Probabilistic counting by Flajolet & Martin (FM) is the first method used for approximating the number of distinct elements in a stream in one pass. By hashing elements of a stream to a bit-string that is sufficiently long (more than the possible results of the hash function), it is possible to estimate the number of distinct items. For example, 64 bits is sufficient to hash the URL's [7].

Suppose a data stream has repeated elements { $x_1, x_2, \dots, x_s$ }. Let d be the number of distinct elements, namely d = | { $x_1, x_2, \dots, x_s$ } | and elements be { $e_1, e_2, \dots, e_n$ }, the FM algorithm runs in O(s) time and requires O(log(d)) memory. So, memory usage is the focus in the FM algorithm since other algorithms like hashing would need O(d) memory.

The critical property of a hash function is that when applied to the same element, it always produces the same result. This property estimates unique elements within a large dataset by recording the longest sequence of zeroes within that set. The algorithm works on the principle that the maximum number of leading zeros that occurs for all hash values is less likely and indicates a larger cardinality [8].

4.2. HyperLogLog

Using less memory and cost the hyperloglog algorithm provides an estimate to the count distinct problem. This algorithm is an extension derived from the Flajolet-Martin algorithm. The HyperLogLog algorithm can estimate cardinalities of > 10^9 with a typical accuracy of 2%, using 1.5kb of memory [9].

Aiming to minimize memory space required in such a continuous computation much research has been done to present a unique space and time efficient data structure (sketch) by building on the FM algorithm. The result is continuously maintaining a sketch (consisting of several sub-sketches) over a data stream S such that for any given time t, the sketch can be used to return a ϵ -approximate answer to $n_{S,t}$.

Using a hashing table with the sliding window technique [10] an efficient data structure is built. A hashmap stores the elements of the current window of the input stream {$x_1, x_2, \dots, x_s$ } and a counter 'C' is used to calculate the number of distinct elements. Using the count from the previous window elements the next window's elements are checked for membership. As the window slides one element is removed, and a new element is added to the hash. This way the runtime is also kept at O(n) [11].

Algorithm: HyperLogLog

Step 1: Create an empty hash map hM and initialize distinct element counter C=0

Step 2: Insert elements of the first window to hM. Since hashing uses (key, value) pairs, in hM elements are the key and their counts the value. Counter C is also updated.

Step 3: The window slides

i. the first element of the previous window is removed

1. If the removed element appeared only once remove it from hM and decrement C

2. Else decrement its count hM

ii. Add the last element of next window

1. If the added element is not in hM add it and increment C

2. Else increment its count hM

Using an auxiliary memory of *m* units (typically, "short bytes"), HyperLogLog performs a single pass over the data and produces an estimate of the cardinality such that the relative accuracy (the standard error) is typically about $\frac{1.04}{\sqrt{m}}$. This improves on the best previously known cardinality estimator, LOGLOG, whose accuracy can be matched by consuming only 64% of the original memory [12].

5. MEMBERSHIP TEST

To test for membership is a commonly occurring problem in many real-life scenarios. For example, identify malicious URLs, reduction of disk lookups for non-existent rows or columns, detect spam emails, and such. Using the filtering technique, we can screen elements belonging to a particular set and store them, and non-members removed easier to check for membership.

5.1. Bloom Filter

Bloom Filter is a probabilistic data structure which saves memory space and is time efficient, but the trade-off is false positives. It tells us if the value is definitely not in the input stream or maybe in the stream. Bloom Filters do not allow false negatives, that is, returning false when the element is not in the set. To determine whether a value is not present in a search takes a long time compared to determine whether the value exists. Bloom Filter reduces some failed searches to O(1) compared to O(log n) in a binary search or O(n) in a sequential search. So even though Bloom Filters allow false positives the simplicity, space saving, and speed factors often compensate this downside. [13]

In today's world since just checking for membership is not sufficient Bloom filter variants to count the number of occurrences, allow deletions for faster retrieval, minimum false positive probability and such have all resulted in variations to boom filters like counting Bloom Filter (CBF), Cuckoo filter, Stable Bloom Filter, Scalable Bloom Filters and more.

6. OTHER PROBLEMS

6.1. Problems using Graph Streams

So far, all our algorithms are based on streams that have a metric structure. Another essential class of structured large data sets is large graphs. This motivates the study of graph algorithms that operate in streaming fashion: the input is a stream that describes a graph [14]. In computer science, the flow of computation is represented by graphs. A graph is a non-linear data structure consisting of vertices (nodes) connected by edges (arcs) [15] Transportation companies use graphs to plan out their delivery systems. Using vertices (intersection of roads) and edges (roads connecting two vertices) the shortest path between two vertices is found as the ideal delivery route.

6.2. Estimating Moments

A generalization of the counting distinct problems (discussed previously) is the estimating moments problem. The problem, called computing "moments," involves the distribution of frequencies of different elements in the stream. The base for all moments is a simple extension of second moments [7].

6.3. Counting 1's

This is counting how many 1's is there in a window in the last k bits for any k≤N. This is used to know an approximate number in a binary stream.

6.4. Most-Common 'recent' Elements Decaying Window

Instead of fixed window size, we could consider the data stream over arrived t minutes ago. This allows computing the summary of an exponentially decaying window efficiently. [7]

7. AREAS OF ADVANCEMENTS IN CURRENT ALGORITHMS

Machine learning is the field of science where algorithms teach the computer how to perform a task successfully. Machine learning algorithms focus on mining data using supervised (under observation) or unsupervised (without observation) or both learning methods to provide data scientists with analytics to solve business problems. The type of machine learning algorithms being used is based on the type of task being solved, the volume, velocity, and variety of input data and the output required. According to Ahmad, Lavin, Purdy, and Agha [16], the increase in connected real-time sensors has resulted in increasing the importance of anomaly detection in streaming data. Early anomaly detection though valuable is challenging to execute in practice. In this space algorithms that learn continuously are preferred.

8. ANALYSIS AND FINDINGS

All the above algorithms (Count-min Sketch, HyperLogLog, and Bloom filter) used to solve problems (Heavy Hitter,

Membership check, Association and such) use techniques (such as hashing, clustering) to provide solutions. After extensive research and analysis, we can confidently say that most queries revolve around a few basic questions:

- How many distinct elements are in the data set (cardinality)?
- What are the most frequent elements (heavy-hitters/top k elements)?
- What are the frequencies of the most frequent elements? (frequency count)
- Does an element exist in the data set (membership check)?

Today's researchers aim at providing solutions to specific problems that are found across many industries. Shah, Shah, Shetty, et al. [17] in their paper provide a comparative study of pattern recognition algorithms on sales data. Using 2 of the association mining algorithms – Apriori and FP-Growth discussed in the Frequent Items /Heavy Hitters section of our paper they have compared their performance on sales data. Their results show that the FP-Growth algorithm is much more consistent and quicker in performance due to its divide and conquer method. Anderson, Bevin, Lang, et al. [16] provide a high-performance Misra-Gries algorithm that they say overcomes significant shortcomings of current streaming frequency approximation methods. First, their algorithm handles weighted updates in amortized constant time and employs an optimized data structure with minimal memory and maximum throughput. Secondly, they provide a method for merging summaries produced by any counter-based algorithm (like Misra-Gries (MG) and Space Saving (SS)).

Mohamadi, Khan, and Birol [18] introduce ntCard a streaming algorithm used to estimate the k-mer coverage frequency histogram for high throughput sequencing genomics data. The algorithm uses Bloom filters for faster categorization and tools for optimal memory size and number of hash functions. This way the ntCard algorithm can get the total number of distinct k-mers F_0, as well as the number of k-mers above a certain multiplicity threshold.

By using not just one streaming technique but a combination of different techniques a richer understanding of the dataset can be generated, like sampling filtered data or sketching a cluster or using a hashing table over a sliding window [19]. Rusu & Dobra in their paper [20] improve the time performance of sketches by computing the sketch over a sample of the stream. Thus, using sketching and the sampling technique. Their results show that the accuracy of the sketch computed over a small sample is close to the accuracy over the entire data even when the sample size is 10% or less of the data size.

9. CONCLUSION

The key to success is to see challenges as opportunities. This paper aims to showcase algorithms used for solving streaming data issues. With the growth of large data sets from IoT, sensors, social networks, and such data streams have gained popularity as a data format. The origins of data stream problems have been traced to the '70s but have only gained popularity in the last 15 years due to the growth in data volume, many theories, and application-oriented research. Most algorithms and their variations/improvements revolve around answering few basic queries – cardinality or heavy-hitters/top k elements or frequency count or membership check. In this paper, we have listed various streaming problems, and their associated algorithms formalized to resolve it. We have also provided an update on where the latest algorithms are being used today. As next steps, deep diving into a particular area its application, implementation, and findings will result in improvements for the specific area.

REFERENCES

[1] Muthukrishnan, S., 2005, "Data Streams: Algorithms and Applications," Foundations and Trends® in Theoretical Computer Science, vol. 1, no. 2, pp. 117–236.

[2] Cormode, G., 2014, "Misra-Gries Summaries," Encyclopedia of Algorithms, pp. 1–5.

[3] Han, J., Pei, J., Yin, Y., and Mao, R., 2004, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," Data Mining and Knowledge Discovery, vol. 8, no. 1, pp. 53–87.

[4] Cormode, G., and Muthukrishnan, M., 2012, "Approximating Data with the Count-Min Sketch," IEEE Software, vol. 29, no. 1, pp. 64–69.

[5] Pramod, P. S., and Vyas, O., 2010, "Survey on Frequent Itemset Mining Algorithms," International Journal of Computer Applications, vol. 1, no. 15, pp. 94–100.

[6] Balasubramaniam, R., and Nandhini, N., 2019, "Efficient Count-Min Sketch to solve Frequent Items problem" Proceedings of the First International Conference on Electrical and Computer Technologies (ICAECT 2019), vol. 3, pp. 1120-1125.

[7] Leskovec, J., Rajaraman, A., and Ullman, J. D., 2016, "Mining of massive datasets." Delhi: Cambridge University Press.

[8] Heule, S., Nunkesser, M., and Hall, A., 2013, "HyperLogLog in practice," Proceedings of the 16th International Conference on Extending Database Technology - EDBT 13.

[9] Yang, F., 2012, "Fast, Cheap, and 98% Right: Cardinality Estimation for Big Data," Druid. [Online]. Available: http://druid.io/blog/2012/05/04/fast-cheap-and-98-right-cardinality-estimation-for-big-data.html. [Accessed: 5-Feb-2019].

[10] "notes3,", 2008, Scribd. [Online]. Available: http://www.scribd.com/document/347534981/notes3. [Accessed: 6-Mar-2019].

[11] Aggarwal, C. C., 2007, "Data Streams Models, and Algorithms." Boston (MA): Springer.

[12] Flajolet, P., Fusy, E., Gandouet, O., and Meunier, F., 2007, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," Discrete Mathematics and Theoretical Computer Science, pp. 127–146.

[13] Balasubramaniam, R., and Nandhini, K., 2019, "Malicious Website Detection Using Probabilistic Data Structure Bloom Filter" Proceedings of the Third International Conference on Computing Methodologies and Communication (ICCMC 2019), 27-29 March 2019, pp.340-345.

[14] Chakrabarti, A., 2014, "CS49: Data Stream Algorithms," 14-Oct-2014.

[15] "Applications of Graph Data Structure," GeeksforGeeks, 16-Aug-2018. [Online]. Available: http://www.geeksforgeeks.org/applications-of-graph-data-structure. [Accessed: 10-Mar-2019].

[16] Anderson, D., Bevan, P., Lang, K., Liberty, E., Rhodes, L., and Thaler, J., 2017, "A high-performance algorithm for identifying frequent items in data streams," Proceedings of the 2017 Internet Measurement Conference on - IMC 17.

[17] Shah, M., Shah, N., Shetty, A., Shah, D., and Gotmare, P., 2016, "A Comparative Study of Pattern Recognition Algorithms on Sales Data," International Journal of Computer Applications, vol. 141, no. 1, pp. 38–41.

[18] Mohamadi, H., Khan, H., and Birol, I., 2017, "ntCard: a streaming algorithm for cardinality estimation in genomics data," Bioinformatics.

[19] Rozenbaum, I., 2007, "Filtering Techniques for Data Streams." Graduate School-New Brunswick Rutgers, The State University of New Jersey.

[20] Rusu, F., and Dobra, A., 2009, "Sketching Sampled Data Streams," 2009 IEEE 25th International Conference on Data Engineering.