An Overview of Count-Min Sketch and its Applications

Benedikt Sigurleifsson, Aravindan Anbarasu and Karl Kangur

April 6, 2019

The Count-Min Sketch data structure and its uses within Computer Science

Benedikt Sigurleifsson
Chalmers University of Technology
bensig@student.chalmers.se

Aravindan Anbarasu
Chalmers University of Technology
anbarasu@student.chalmers.se

Karl Kangur
Chalmers University of Technology
karlkan@student.chalmers.se

ABSTRACT

Research into data stream processing algorithms has been going on for over 20 years by now. Its relevance has rapidly grown in recent years, due to advancements in the Internet of Things (IoT), Cloud Computing and Social Networking [9]. These new technologies and devices generate a lot of data, which causes problems related to storing and processing it. This has led to a push for more efficient algorithms and new data structures that should work on top-of-the-line hardware as well as on small IoT devices with limited resources. All with the aim of handling more data than ever before. This paper mainly focuses on the Count-Min (CM) Sketch [6], which is a compact summary data structure that serves as a frequency table of events in a stream of data. The paper explains the implementation of this data structure and shows that it can be used in solving problems that appear in different fields of Computer Science, such as solving the frequent items (heavy hitters) problem, speeding up database queries, improving password security and many more. We also briefly introduce the Bloom filter [2] which is a related data structure that the CM builds upon. This paper signifies the importance of Count-Min sketch in Computer Science by describing the wide array of applications it is used in.

Keywords

Streaming Algorithms, Count-Min Sketch, Heavy hitters, Anomaly detection, Computational Biology

1. INTRODUCTION

The amount of data stored in this world is growing exponentially. This is largely due to Internet of Things (IoT), Cloud Computing and Social Networking technologies [9] generating more data than ever. Meanwhile the capacity to store data has been also been increasing. The world's storage capacity per capita has been doubling every 40 months [11]. This increase in data has caused many new challenges such as how computers process all of this data. These new challenges have driven a lot of research in the field of Streaming Algorithms.

Streaming Algorithms is a field that looks at problems that come up when processing large streams of data. Data stream refers to a sequence of data packets that carry information during transmission. When dealing with large streams of data it is often good to process it close to the source when it passes through as a data stream. This distributes the processing work, reduces data that needs to be stored and enables decision making closer to the source. If the data is not processed while it is passing through, it will need to be stored somewhere for later processing. Most likely the data will have to be sent to a database server. Depending on the amount of data that is coming in, it might not be practical to store all of it. Therefore a lot of research is happening in the field of Streaming Algorithm to better improve data stream processing, which has lead to a lot of new algorithms and data structures. The research does not only benefit high performance computers, but is also useful for cheap IoT devices that have limited processing power and are in many cases the source of data. The high performance computers can now process much more data than what they could previously. However for IoT devices, now it has suddenly become feasible to do some of the processing on them due to better algorithms and data structures that are available to them.

One of the most researched problems within the Streaming Algorithms field is the Frequent items problem. This problem revolves around processing a stream of data and find items that occur most frequently [5]. The most well known data structure that solves the Frequent items problem is the Count-Min Sketch. The word "sketch" in this case simply means a summary, which is exactly what the Count-Min Sketch does. It provides a summary of number of item occurrences that have taken place in the processed data [6]. Because of its simplicity, Count-Min Sketch has many use cases in Networking, Databases and multiple other fields. Reading this paper gives the readers an overview of problems faced while processing huge chunks of data and how efficiently we can tackle this problem using the Count-Min Sketch.

This paper intends to explain how the Count-Min Sketch works 2. Section 3 covers different use cases where the data structure is being used in different fields. Example problems that will be covered are, the Heavy hitter problem that appears in the Networking field and also in the Databases field as Iceberg queries, password security, etc. In section 4, the importance of the algorithm is discussed, along with why the algorithm appears in so many different fields. Finally, we conclude the paper in section 5.

2. COUNT-MIN SKETCH

The Count-Min Sketch is a data structure that is used to summarize data streams [6]. It stores information about how often item occurs in the data without storing all the data from the data stream and helps with answering questions like ”What items have appeared more than k times in this data stream?”

2.1 Bloom Filters

Before going into how the Count-Min Sketch works, it is worth to take a look at a solution to another related problem that the Count-Min Sketch builds upon. This problem is called the membership problem.

2.1.1 The Membership Problem

The membership problem is a common problem when working with data. It revolves around finding out whether a certain item can be found in a set of data. This problem sounds simple, but when working with large data sets, it might not be feasible to go through all the data to answer this question. It might not even be possible if the data is coming in as a stream that is not stored.

The membership problem can be solved with Bloom Filters. Bloom Filters are data structures that can be used to keep track of items, in other words they give a compact overview of what is in the data without the need to store all the data [2]. By compacting the data, Bloom Filters only provide an approximate answer. From this compact overview that Bloom Filters provide the membership query can be easily answered.

2.1.2 Bloom Filter implementation

Bloom filters are implemented using unique hash functions and a bit array that is initialized to zero. The purpose of the hash functions is to map items to certain bit fields in the array. When membership data of an item is inserted into the data structure, the item is hashed separately by all the hash functions and the respective bit array fields that the hash function maps to are set to one. An overview of how a Bloom Filter looks and how an insert operation is performed can be seen in figure 1. In the figure we have an bit array where all the bits are initialized to zero. The first value that is mapped to the bit array gets hashed by all the hash functions, which maps them to a certain field in the array. Those fields are set to one which is represented with a red color in the figure.

When retrieving from the data structure, the item is again hashed by all the hash functions and the respective bit fields are checked. If all the fields are set to one, then we can say that the item might be in the data. If one of the fields is not set, then we can say with certainty that the item does not appear in the data.

The uncertainty in the answer when all the fields are set, comes from using hash functions. This uncertainty however allows the data to be stored more compactly. Hash functions can have hash collisions, meaning that the same hash function might map two different items to the same bit array field. Two different hash functions can also map two different items to the same bit array field since the Bloom filter uses only a single array for all the hash functions. These hash collisions can lead to false positives, which is the cause of the uncertainty in the answer. If only one hash function is used the uncertainty is high because there is no redundancy that minimizes the affect of possible hash collisions. This can be fixed by adding more hash functions. However by adding more hash functions, the likelihood of having hash collisions between hash functions increases. It will also result in the bit array filling up faster, since more bit array fields are being set to one. Therefore there is a sweet spot for the number of hash functions to use.

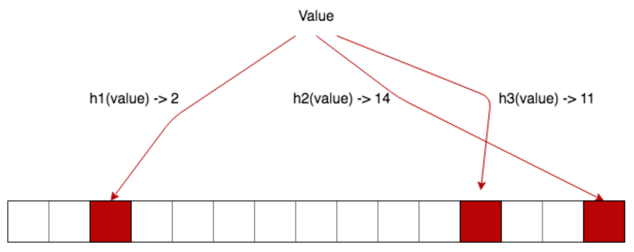

Figure 1: Bloom Filter insert operation

2.2 Count-Min Sketch implementation

Now back to the Count-Min Sketch. As mentioned in the beginning of this section the Count-Min Sketch provides answers to how often items appear in a data stream [6]. It is therefore very similar to the Bloom Filters in the sense that it provides an answer to the membership problem, but provides additionally the number of occurrences.

The Count-Min Sketch builds upon the idea of the Bloom Filter by using unique hash functions and zero initialized arrays. However instead of using a single array it uses separate array for every single hash function. This way more hash functions can be added without increasing the likelihood of collisions between hash functions. This however creates a trade off between memory usage and uncertainty. Using more hash functions leads to less uncertainty, but increases memory usage. Finally the Count-Min Sketch stores a counter unlike the membership bit that the Bloom Filter uses. This counter represents the number of occurrences of an item in the data.

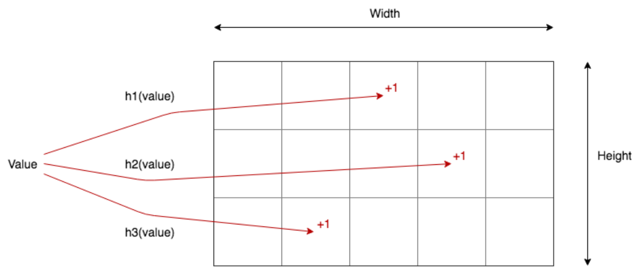

Figure 2: Count-Min Sketch insert operation

Figure 2 demonstrates what the Count-Min Sketch looks like and how the insert operation works. When an item is inserted it is hashed separately by all of the hash functions. The hash functions maps the input to the corresponding counter in their array that all get raised by one.

When checking for the number of occurrences an item has, the item gets hashed separately by all the hash functions to find the corresponding counters. These counters might have different values. This is due to the fact that hash functions are still being used and even though there are no hash collisions between hash functions, they can still occur within the same hash function. Meaning that two different items might get mapped to the same counter and thus that counter might get incremented more often. When picking a value between those potentially different counter values, it is always safest to pick the lowest value to get the most accurate results.

The memory that the Count-Min sketch uses depends on two configurable variables width (w) and depth (d), where w is the size of the arrays and d is the number of hash functions used. These two variables are chosen at the beginning and do not change over time, even though the data continues to grow. Therefore the Count-Min sketch promises fixed space usage. By promising fixed space usage, one would think that the Count-Min Sketch would not give accurate results. However every query made has at most error $2N/w$ with the probability of $1 - (1/2)^d$, where N is the total number of counts in the sketch [7]. This means that the user of the Count-Min Sketch just has to make w and d large enough to make the queries very accurate without using much space.

3. COUNT-MIN SKETCH APPLICATIONS

The following section will take a look at how the Count-Min Sketch can be used in different applications within different fields of Computer Science.

3.1 Networking

In recent years network anomaly detection, an approach to network security threat detection has become a significant area in the field of Information and Communication Technology (ICT). Anomaly detection is the identification of rare events or elements in the data stream which raise suspicions by differing significantly. These are widely used for identifying network intrusions. Nowadays, the network bandwidth is too high which makes it difficult to detect anomalies due to the computational overhead and memory requirements. Different efficient data structures have been developed for years to detect anomalies in data streams with guaranteed error bounds.

In this subsection, we focus on the detection of one important significant behavior known as heavy hitters by using the Count-Min (CM) Sketch data structure.

3.1.1 Heavy Hitter Problem

A heavy hitter in the network can be either an IP address/port or a combination of both [3]. These heavy hitters create high volume traffic beyond some predefined threshold in the network, thus creating anomalies. The heavy hitter problem [15], is summarized by considering an array of elements A, of length n and also with the parameter k. Here k is a modest number (100 or 1000) and n is very large (billions or trillions) considering all possible source and destination IP address pairs. The target here is to find the elements in the array that occur at least n/k times [15]. An important point to be noted here is the fact that there can be at most k such elements or there might be no such elements in the array [15]. But the heavy hitter problem promises the existence of a majority element. For Example, data packets that constitutes more than 15% of the entire network traffic are called heavy hitters because it violates the service agreement between the two nodes [3].

Since the Internet continues to grow in size and complexity, monitoring heavy hitters in real time is a big challenge due to processing and memory constraints [3]. Bu *et al.* [3] explains that to detect any such heavy hitters, the system should scale up to at least 2^{104} keys (keys here denote an IP address/port). "The number(2^{104}) is calculated based on the number of possible five-tuple flows: source IP address (32 bits), destination IP address (32 bits), source port (16 bits), destination port (16 bits), and protocol (8 bits). This number might be significantly smaller for realistic network flow since not all possible combination of these fields are possible" [3].

3.1.2 Heavy Hitter Detection: An Impossibility Result

The goal of the heavy-hitter detection is to efficiently identify the set of flows that represent a significantly large proportion of the link capacity with a lower error rate and memory usage [3]. The solution is quite simple if we have already a stored array of elements A in the memory , as we just need to populate the result only if the element occurs at least n/k times [15]. But the question here is that, can we solve the same heavy-hitter problem with a single pass over the array in real-time, without a local copy and by using limited memory space only [15]? Roughgarden *et al.* [15] explains the fact that there is no algorithm that can solve the heavy-hitter problem in a single pass while using limited memory space in real-time.

3.1.3 Approximate Heavy Hitter Problem and Detection

Though it was stated that there is no algorithm that can solve the heavy-hitter problem in a single pass, there are lots of applications that are still being challenged by the heavy-hitter problem. This motivates us continuously to come up with significant streaming algorithms to solve the problem. The best-case scenario is to find a relaxation of the problem that remains applicable for the significant applications and also admits a good solution [15].

The ϵ-*approximate heavy hitters* (ϵ-*HH*) problem, is summarized by considering an array of elements A, of length n and also with the user defined parameters k and ϵ. The algorithm should output a list of elements such that [15],

- The list contains every element that occurs at least n/k times in the array A
- Every element in the list occurs at least $\frac{n}{k} - \epsilon n$ times in the array A.

Here we allow the memory space used by a solution to grow as $1/\epsilon$ [15]. This clearly shows that we cannot take ϵ to be *0*

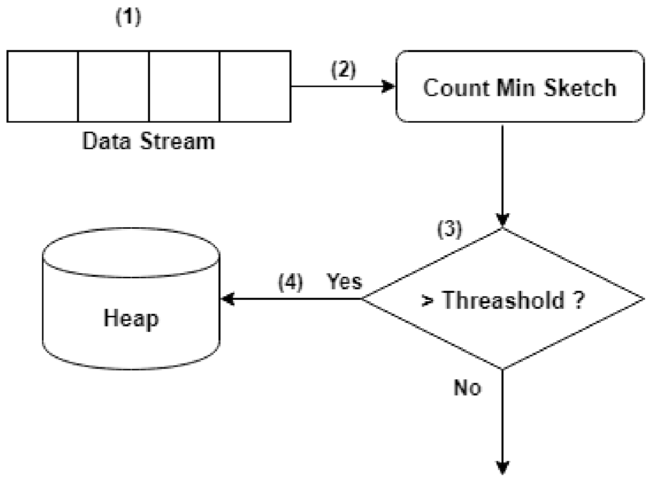

Figure 3: Approximate Heavy Hitters Detection.

and solve the exact version of *HH* problem, as the memory turns to ∞ [15]. For example, let us take $\epsilon = 1/2k$, then as per the algorithm mentioned above, the list contains every element that occurs at least n/k times in the array A. Also every element in the list occurs at least n/2k times in the array A. This approximate solution is as good as the exact solution [15].

Figure 3 explains an approximate heavy hitters problem solved by using Count-Min sketch. The Count-Min-sketch supports two functions: Inc(x) and Count(x) [15]. The function Count(x) would return the frequency count of x, which is the number of times Inc(x) function has been invoked in the past [15]. Consider a stream of data elements as the input array A. Let n be the number of elements seen so far. Assume $\epsilon = 1/2k$. All the potential heavy-hitters are stored in a heap memory.

1. We take each element x_i from the stream of data elements.
2. We add each element x_i one by one to the Count-Min sketch.
3. For each element x_i fed inside the Count-Min sketch, we invoke Inc(x) followed by Count(x). We also check for the condition if Count(x) $\geq$ n/k [15].
4. If the condition is *True* we store the x_i in heap using the *key* Count(x), else we drop x_i. If x_i was already in the heap, we delete it before re-inserting it with its new *key* value [15].

Also, whenever n grows to the point that some element x_i stored in the heap has a key less than n/k (checkable in O(1) time), we delete x_i from the heap [15]. Once the entire stream of elements are passed, we output all of the elements in the heap. Assuming that Count-Min sketch makes no big errors, we approximately consider that every element x_i in the heap has true frequency count at least $\frac{n}{k} - \epsilon n = \frac{n}{2k}$ times, as other elements would have been deleted from the heap by the end of the pass [15].

3.2 Databases

The Count-Min Sketch is not only useful for data streams, it is also very useful for databases and is used in many different applications within the field of databases. This section will cover some of those applications, especially those that were introduced in the paper "Spectral Bloom Filters" [4].

The Spectral Bloom Filter (SBF) is a type of Bloom Filter (BF) data structure that was introduced two years before the Count-Min Sketch. It provides its users with count approximations just like the Count-Min Sketch does, but is implemented differently. This paper will not go into details of the differences between those two implementations. However the main difference is that the Spectral Bloom Filter only has a single array just like the regular Bloom Filter. In this array the SBF stores counters instead of bit fields like the BF. The Count-Min Sketch on the other hand uses a separate array for every hash function it uses. Since both data structures perform the same tasks, then this paper can use the examples of different applications mentioned in the Spectral Bloom Filter paper.

3.2.1 *Database queries*

Database queries are used to retrieve information from database tables. When dealing with large tables, then running a simple query can be computationally expensive. Let's take the following query as an example

```
SELECT count(a1) FROM R WHERE a1 = v
```

This query counts all elements a1 in the table R that meet a certain condition $a1 = v$. This requires the database to go through every row in the table R and count the items depending on the condition. This is a very simple task, but can be very time consuming depending on the size of the table. If this query is required to be run often, then it can become very costly. Running this query can be made considerately cheaper by using the Count-Min Sketch. Like mentioned in section 2 the Count-Min Sketch provides a relatively precise frequency approximation of items in the table R, using fixed memory. The precision and memory usage depends on user defined parameters and which parameter the user prefers over the other. By having this approximation of the table R, the query can simply sum up the frequency count for those elements that meet the condition, instead of iterating over the whole table R.

Count is not the only database query operation that can benefit from using the Count-Min Sketch, it can also be used for operations like average, sum, max and many more.

3.2.2 *Iceberg queries*

Iceberg queries refer to queries that search for items that occur more often than some specified threshold. These queries are mostly useful to detect high frequency items in the data. It is essentially the same problem as the Heavy Hitter problem that is described in section 3.1.1 and is solved in the same way. The problem is briefly mentioned here again to show how it can be used in databases.

An example of how an iceberg query could be used is an online store that want's to give frequent customers special deals. When a frequent customer contacts a sales representative, the sales representative gets an event that this is a frequent customer and that he is allowed to give the customer better deals in order to close the purchase.

Using the Count-Min Sketch for this is fairly straight forward. It can be used to keep count of occurrences. Every time a customer contacts a sales representative the Count-Min Sketch is checked for whether this is a frequent customer. This is done by checking the number of purchases and see if it exceeds some dynamic threshold that the company decides. If it exceeds the threshold the sales representative gets a notification about it.

3.2.3 Spectral Bloomjoins

When querying data from databases it is very common to want data from two database tables in the same query. The join operation is used to do just that. In a distributed setting, these database tables might reside on different database servers. When doing a distributed join between two database servers, either database server will have to send the contents of its table to the other server. This results in a slow response time and high communication cost.

The Bloomjoin was presented as a method of doing a fast distributed join between two database tables that are located on different database servers. As the name suggests the Bloomjoin uses a Bloom Filter (BF) to make these communications more efficient. To be able to use the Bloomjoin, both databases need to use Bloom Filters to summarize what items are in their tables. Section 2.1 gives an good overview of how Bloom Filters work. Instead of sending all the data that resides on one of the database server to the other server and then send the result of the joined data back to the first server, the database servers can simply send the summary of their data. By sending the summary of the data a lot less time is spent on sending data back and fourth between the database servers.

Lets look at an example of how a Bloomjoin works. Lets say that there are two database servers S_1 and S_2. They get a query that wants to do a join operation between two tables residing on those two servers. One of those servers S_1 starts by sending the other server S_2 its BF_1 that contains a summary of the requested table. S_2 then performs the join operation between its table and BF_1. S_2 then replies with only the matches from the join operation back to S_1. Now S_1 can produce the final results from the matches that were sent back from S_2. These communications are much more efficient than sending a whole table between the servers.

The Spectral Bloom Filter paper [4], suggests that the SBF can be used to further improve some Bloomjoin operations. The join operations in question are those join operations that rely on counting or filtering the results based on some threshold value. These are the same operations that are mentioned in section 3.2.1, that discusses what database queries can be improved.

An example of this is the following query, which performs a count on the joined results. Both SBF and the Count-Min Sketch work very well for this problem since they contain count of item occurrences.

```
SELECT R.a,count(*) FROM R,S
WHERE R.a = S.a GROUP BY R.a
```

Another example is the following query that faces the Iceberg Queries problem discussed in section 3.2.2. The query filters the results of the join operation based on a threshold.

```
SELECT R.a,count(*) FROM R,S
WHERE R.a = S.a GROUP BY R.a
HAVING count(*) [>,=] T
```

3.3 Other areas

While Networks and Databases are the most well-known areas for applications of the Count-Min Sketch, the need to process a data stream in a fast and efficient manner has also grown to become a significant problem in many other fields such as Computational Biology [21], Security [16], Games [10], Machine Learning [18], Social networks [20] etc. In this section we cover the application of Count-Min Sketch in some of those fields.

3.3.1 K-mer counting in Computational Biology

Computational Biology is the development and application of data-analytical and theoretical methods, mathematical modeling and computational simulation techniques to the study of biological, behavioral, and social systems [12]. In other words, it is the science of using biological data to develop algorithms or models to understand biological systems and relationships.

For example, one of the challenges in computational genomics, is counting all the different k-mers (subsequences of lenght k) obtained through DNA sequencing. K-mer counting has been widely used in bioinformatics and with the increase in sequencing data set sizes, efficient processing of the reads has become more important.

Khmer is a k-mer counting solution that uses the Count-Min Sketch data structure in its implementation [21]. The advantage of khmer compared to other k-mer counting solutions is that it enables memory- and time-efficient online counting directly as data is loaded, without a need for disk access. This enables khmer to retrieve the counts of individual k-mers significantly faster than previous solutions.

One of the important properties of the Count-Min sketch is that its memory usage is fixed, meaning it will not increase as data is loaded. The memory usage, as well as the accuracy of the count, is determined by the size and number of hash tables used. Therefore, in khmer, the user has control over the memory usage, based on the desired accuracy of the results. This is critical in order to run the system on commodity hardware with limited resources.

Another important property of the Count-Min sketch is that it is a probabilistic data structure with one-sided error. This means that khmer gives random overestimates of k-mer frequency, but never generates underestimates. The probability of an inaccurate count can be estimated based on the hash table load. The size of the miscount depends on the details of the frequency distribution of k-mers. These inaccuracies are usually acceptable since for many applications, an approximate k-mer count is sufficient.

3.3.2 *Password Security*

One of the more popular techniques of attack on passwords is a dictionary attack, which is a form of brute force attack for defeating an authentication mechanism. The attacker attempts to guess a password by trying all the words in some kind of a list (often words in a dictionary)[19]. This kind of an attack is especially dangerous if the attacker has information about the most popular passwords used in the system. In order to protect users from a dictionary attack where the attacker tries to gain access to an account by trying the most popular passwords, it is important to:

1. limit the amount of tries the attacker gets to issue when trying to guess a password of an account and
2. minimize the amount of accounts with the most popular passwords [16].

While the first problem is quite straight-forward to solve, no perfect solution has been found to achieve the second goal. In an attempt to influence user selection of passwords, numerous different tools and rules have been created. Most sites nowadays have password-composition policies, forcing users to use long passwords that include uppercase and lowercase letters, use special characters etc. These policies are often criticized for unintended consequences and little added entropy.

Schechter et al. [16] have proposed to take the direct approach of preventing users from choosing passwords that are dangerously popular. To achieve that, it is necessary to identify those undesirably popular passwords - the authors of the aforementioned paper have come up with a solution where they use Count-Min sketch to create an oracle.

Their proposal is to strengthen user selected passwords against statistical dictionary attacks by allowing users to select any kind of password they want as long as it is not already too popular among other users. The purpose of the oracle is to identify dangerously popular passwords using Count-Min sketch data structure that is populated with existing users' passwords. Interestingly, a minimum acceptable false-positive rate is set in order to obfuscate attackers that might get access to the oracle.

According to their paper [16], a password is considered too popular if it occurs at a rate that exceeds the fractional popularity threshold - r. An attacker who would be able to issue G guesses against each account, and who has information about the G most popular passwords, could compromise a fraction of at most rG accounts. For example, if the fractional popularity threshold was set to $\frac{1}{1,000,000}$, an attacker with knowledge about the most popular password would be able to compromise only 0.0001% of the accounts. In contrast, an attacker could compromise 0.22% of MySpace accounts [17] or 0.9% of RockYou accounts [13] if he knew the most popular password of those sites.

4. DISCUSSION

Count-Min sketch is a solution to a problem that exists in a wide range of areas - how to efficiently approximate the count of items in a vast stream of data without having to store every item. It is a problem that is quite simple in its nature and appears in many different fields. In this survey paper we have provided an overview of some of the key areas where the Count-Min sketch data structure is used and how it is implemented. One of the pivotal properties of the method is that it requires memory space which is sub-linear to the data size being considered [1]. This enables its use in cases where storing all of the data is not feasible due to the large amount of data and limited hardware resources.

When analysing the different applications described in sections 3.1, 3.2 and 3.3 it is possible to draw parallels between some of the solutions. In principle, the heavy hitter problem (3.1.1), Iceberg queries (3.2.2) and the oracle in password security (3.3.2), are all dealing with the same problem. They are trying to identify outliers in the data - high frequency items that occur more often than some predefined threshold. Count-Min sketch is a great tool for solving this problem as efficiently as possible.

While Count-Min sketch is without a doubt an excellent sketch, it is still not perfect. One potential problem is that it can be quite biased by overestimating the frequencies of elements with a low number of observations. This becomes a significant problem if we are interested in the count of the low-frequency elements. This flaw is quite well known and several improvements have been suggested to compensate for this [8], [14].

We have shown in this paper that the Count-Min sketch has already found its use in many different applications. Considering the growing importance of data stream processing, there is no reason to believe that the Count-Min sketch is going away anytime soon.

5. CONCLUSIONS

In this paper, we have given the readers an overview of the Count-Min Sketch and its applications. Probabilistic data structures, such as Count-Min sketch, are tools that have become increasingly important in helping us cope with the growth of big data in an efficient and adequately accurate way. We have explained how Count-Min sketch works and how it is used to process data streams in the areas of networking, databases and others fields. Though Count-Min sketch has its own pros and cons, it is one of the most effective and simplest ways to do approximation based queries on streaming data.

6. ACKNOWLEDGMENTS

The authors would like to thank Marina Papatriantafilou and Karl Bäckstrőm for valuable discussions, helpful pointers and constructive criticism during the writing of this survey paper.

7. REFERENCES

[1] C. C. Aggarwal and S. Y. Philip. A survey of synopsis construction in data streams. In *Data Streams*, pages 169–207. Springer, 2007.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] T. Bu, J. Cao, A. Chen, and P. P. Lee. A fast and compact method for unveiling significant patterns in

high speed networks. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1893–1901. IEEE, 2007.

[4] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252. ACM, 2003.

[5] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2):1530–1541, 2008.

[6] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[7] G. Cormode and S. Muthukrishnan. Approximating data with the count-min data structure. *IEEE Software*, 2012.

[8] A. Goyal, H. Daumé III, and G. Cormode. Sketch algorithms for estimating point queries in nlp. In *Proceedings of the 2012 joint conference on empirical methods in natural language processing and computational natural language learning*, pages 1093–1103. Association for Computational Linguistics, 2012.

[9] M. S. Hajirahimova and A. S. Aliyeva. About big data measurement methodologies and indicators. *International Journal of Modern Education and Computer Science*, 9(10):1, 2017.

[10] B. A. Harrison. *Move prediction in the game of go*. PhD thesis, Citeseer, 2010.

[11] M. Hilbert and P. López. The world's technological capacity to store, communicate, and compute information. *science*, page 1200970, 2011.

[12] M. Huerta, G. Downing, F. Haseltine, B. Seto, and Y. Liu. Nih working definition of bioinformatics and computational biology. *US National Institute of Health*, 2000.

[13] A. Imperva. Consumer password worst practices. Technical report, Technical report, Imperva ADC, 2010.

[14] G. Pitel and G. Fouquier. Count-min-log sketch: Approximately counting with approximate counters. *arXiv preprint arXiv:1502.04885*, 2015.

[15] T. Roughgarden and G. Valiant. Cs168: The modern algorithmic toolbox lecture# 2: Approximate heavy hitters and the count-min sketch. 2015.

[16] S. Schechter, C. Herley, and M. Mitzenmacher. Popularity is everything: A new approach to protecting passwords from statistical-guessing attacks. In *Proceedings of the 5th USENIX conference on Hot topics in security*, pages 1–8. USENIX Association, 2010.

[17] B. Schneier. Myspace passwords aren't so dumb. *Wired. com*, 2006.

[18] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10(Nov):2615–2637, 2009.

[19] R. Shirey. Internet security glossary, version 2. Technical report, 2007.

[20] H. H. Song, T. W. Cho, V. Dave, Y. Zhang, and L. Qiu. Scalable proximity estimation and link prediction in online social networks. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 322–335. ACM, 2009.

[21] Q. Zhang, J. Pell, R. Canino-Koning, A. C. Howe, and C. T. Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS one*, 9(7):e101271, 2014.