Introduction to streams and stream processing

Mining Data Streams

Florian Rappl

September 4, 2014

Abstract

Performing real time evaluation and analysis of big data is challenging. A special case focuses on incoming streams of data, which are either too large, or too frequent, to be stored and evaluated. This makes stream processing a topic, that often arises when dealing with big data. In this paper we want to give an introduction to stream processing in the context of big data. We will derive basic properties and illustrate useful algorithms that can be used to solve streaming problems.

Contents

1 Introduction **3**

2 The stream data model **3**

3 Sampling data **4**
3.1 Sliding windows 4
3.2 Obtaining a representative sample 5

4 Filtering data **5**
4.1 The Bloom filter 5
4.1.1 Introduction 5
4.1.2 Algorithm 5
4.1.3 Example 6
4.1.4 Outlook 6
4.2 The Flajolet-Martin algorithm 7
4.2.1 Introduction 7
4.2.2 Algorithm 7
4.2.3 Example 8
4.2.4 Outlook 9

5 Practical examples **9**
5.1 Google BigTable 9
5.2 Hadoop 9
5.3 Apache Cassandra 10

6 Conclusion **10**

1 Introduction

Big data may appear in many forms. From the typical appearences that are called "big data" we can deduce three unique properties, which are usually named the 3V: Volume, velocity and variety. Either we have a massive volume of data, which seems to be uncorellated or schemaless (variety), or the data has to be processed in real-time (velocity). The latter is making traditional data inspections practically useless. This is where stream processing techniques come in to help.

Streaming algorithms are based on a stream, which deals with (incoming) data. We have a type of data and the current position within the stream, which is only denoting the number of arrived data packages. In general these algorithms are constrained by memory or time. We assume that the full stream cannot be saved (in memory or on disk) and that the processing time τ needs to be faster than the time between two incoming data packages Δt.

In order to extract knowledge from incoming continous, rapid data streams, we will have to rely on approximations. The exact answer is usually hard to get without having access to the full stream. By our definition a data stream is an ordered sequence of incoming packages of data, even though it can be read only once due to limited computing and storage capabilities. Hence accessing the full stream is not possible in general.

Examples of such streams can be found in computer network traffic, phone conversations, search queries or sensor data. Techniques for filtering streams are also useful for related topics, such as high performance database queries or intrusion detection.

2 The stream data model

Most of the time one is interested in computing statistics on so-called frequency distributions. To make the problem non-trivial the assumption that the distribution is too large to be stored is considered. The vector, which is responsible for storing the data, is usually called $a = \{a_1, a_2, \dots\}$, with all entries being initialized to zero.

A famous problem that has been solved by Alon et al. [AMS96] is an estimation for the k-th frequency moments of the frequency distribution a. The k-th frequency moment of a set in this set of frequencies is defined as

$$F_k(a) = \sum_{i=1}^{n} a_i^k. \tag{1}$$

While the first moment F_1 is simply the total count, the second moment provides useful statistical properties of the data. The main issue is: How to count the frequency of the data, without being able to build a distinct set of frequencies a, i.e. without knowing the full vector a?

The answer is of course given by approximation. Without having the full information available, we cannot give error-free answers, however, we can make predictions. As outlined in [RU11] we need a strategy for sampling the data. A possible way is given by randomly determining what data to use and what data to drop. Here the law of large numbers will come to rescue our predictions. The law basically states that

$$\langle x \rangle_n = n^{-1} \sum_{i=1}^{n} x_i \to \mu, \tag{2}$$

in the limit of $n \to \infty$. Here μ is the expectation value. This means that as long as the probabilities are calculated correctly we can be as close to the real answer as we want. We only need to consider enough data points.

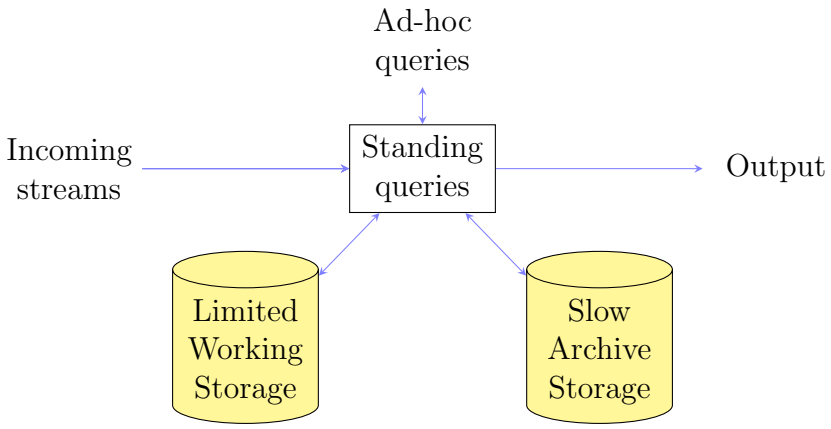

Figure 1: The stream data model

In order to conceptually do work with a stream of data the concept of a stream-management system has been developed. An early survey of research in stream-management systems is given in [BBD+02]. Such a system consists of one or many incoming streams, a limited working storage and a connection to an archival storage. The archival storage can be considered infinite, however, reading data would take too long for evaluation purposes. Writing can be seen as an asynchronous operation, i.e. non-blocking. The basic concept is also illustrated in fig. 1.

Our operations will be placed in form of standing or ad-hoc queries. In general the incoming streams do not have the same data type or data rate. Hence the splitting in two kinds of queries. While standing queries will be assigned for every (particular) kind of stream, an ad-hoc query might be run under certain circumstances, e.g., upon request. Since ad-hoc queries might require more information than available by looking only at the current stream element, we have to know what kind of questions will be asked by the ad-hoc queries. This is our preparation for feeding ad-hoc queries with all the data they require.

3 Sampling data

We use sampling to base the choice if we should process a data item or not on probabilities. Sampling is an old statistical technique that has been used for a long time. Boundaries of the error rate of the computation are given as a function of the sampling rate. We already mentioned the possibility of considering some kind of random number generator as a good source for these probabilities. See [Vit85] for the original paper.

3.1 Sliding windows

A technique that is often used for sampling data is the sliding window technique as published by Datar [DGIM02]. Here the basic idea is to rely on a subset of the incoming streaming data. The subset is moving along with the stream, just considering the last n entries, where appropriate information has been received. An alternative approach considers the entries of the entries in the last t time units. An example would be maintaining a sliding window over the last day for gaining insights on daily page requests on a webpage.

The sliding window can be combined with other techniques. While it only has access to a limited amount of information, it can still be very informative for a specific subset of queries. Of course the stream-management system must keep the buffer fresh, by replacing the oldest element with a new one.

3.2 Obtaining a representative sample

Obtaining a representitive sample by considering just a subset of the full stream is a non-trivial task. In general there is no method that may be applied for all possible queries. We already mentioned methods to estimate the k-th frequency moment of streams, without storing them. However, sometimes the opposite is more interesting: Finding only distinct values in a stream. Here the work of Gibbons [Gib01] illustrates some useful approximations.

We now consider a stream consisting of tuples with n components. A single query might only be interested in a subset of these n components. We call these components *key* components, because the selection of the representative sample will be based on them. Identifying these key components is therefore a required task that deserves full attention.

A good strategy consists of hashing the key components to a single value, which is one of B possible values. Now we only consider the incoming tuple to be included in the sample if the value is one of A values, where $A \subseteq B$. In general the number of elements a in A is much smaller than the number of elements b in B, $a \ll b$. The ratio of selected key values will be approximately a/b of all the key values appearing in the stream.

Such a method can also be extended for varying the sample size. We might consider adjusting the ratio during streaming. If we remove k elements from A, lowering the ratio from a/b to $(a-k)/b$, we just need to remove all elements whose key values hash to these k elements from the set of samples. This way ensuring consistency within our samples is fulfilled.

4 Filtering data

We already briefly discussed the possibility of selecting data depending on one or more conditions. In the previous section we were mainly focused on selecting data due to some reduction conditions, since we wanted to obtain a representative sample. In this section we will learn two more advanced algorithms, that help us selecting data, with conditions that require more knowledge than available at the time of selection.

4.1 The Bloom filter

4.1.1 Introduction

A clever way to generate a representative sample was to use some hash function, which will give us a hint if we should keep the value or discard it. The Bloom filter [Blo70] generalizes this method. Here we take an array of n bits, initially all zeros. Additionally we consider k hash functions, $h_1, h_2, \ldots, h_k$. Each hash function maps key components to n values, represented by the n bit-array.

4.1.2 Algorithm

However, the Bloom filter does not play around with the incoming values. Instead it is initialized with a set S of m key values. Each value will be hashed with all k hash functions. In the end we set each bit to 1, that is $h_i(K)$ for some hash function h_i and a key value K in S.

The Bloom filter is especially good in testing if an incoming value does **not** fulfill the constraints specified in S. Why? We already know, that all values appearing in S will hash to the 1s in the bit-array. This is guaranteed. It is not guaranteed that an element, that is not in S, won't hash to only 1s in the bit-array. So we cannot be sure if an element is in the sequence. Nevertheless, when an element does not hash to 1 by using some hash function h_i, we can be sure that is must be an element that is not S. The principle algorithm for existence checking is shown in the flow chart presented in fig. 2.

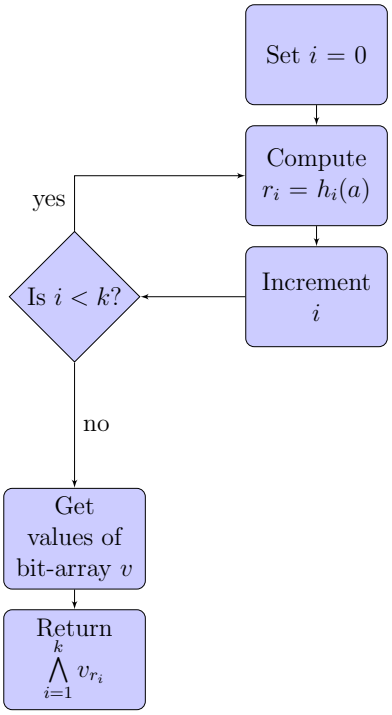

Figure 2: Flow chart for the Bloom filter algorithm

A Bloom filter offers much more practical applications that may be be obvious. Bloom filters may be used to accelerate free-text searching as shown by Ramakrishna [Ram89]. The practical performance boost is used by many tools and applications. Some of them are presented in sec. 5. By varying k and m we can change the probability for a false positive. Here the probability is basically given by

$$P \approx \left(1 - \exp\left(\frac{-mk}{n}\right)\right)^k. \tag{3}$$

Therefore we can just insert more hash functions to reduce the probability for at least a single false positive result. The optimal number of hash functions can be determined by using minimizing the false positive probability P for k with m incoming elements. We obtain

$$k = \left(\frac{n}{m}\right) \ln 2. \tag{4}$$

4.1.3 Example

Assuming the ideal number of hash functions, we can plot the probability of having at least a single false positive result with respect to the length of the bit-array n. We obtain a plot as presented in fig. 3.

4.1.4 Outlook

The Bloom filter gives a huge performance boost for filtering incoming data. The probability of a false positive can be adjusted to our needs. In general the only requirement is that m is smaller than the filter size n.

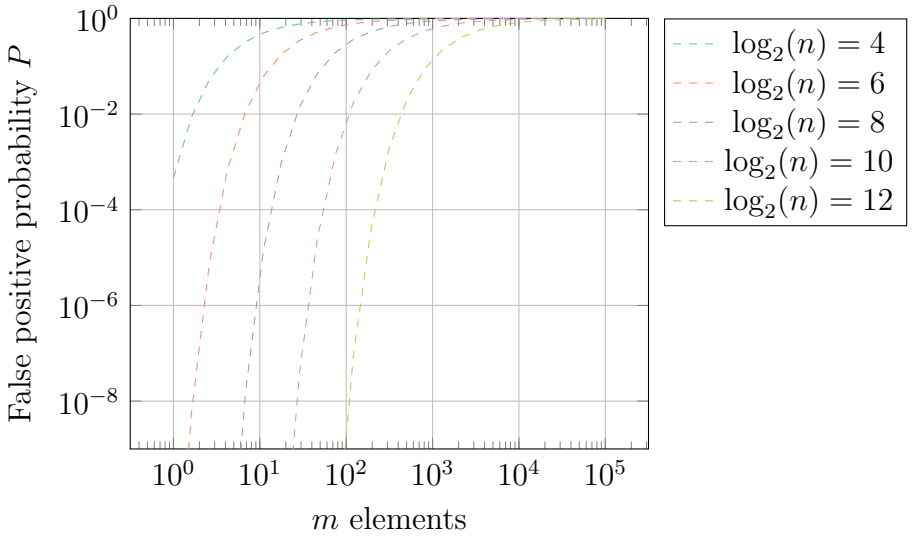

Figure 3: The false positive probability P with respect to the elements m and the filter size n assuming the optimal number of hash functions k.

4.2 The Flajolet-Martin algorithm

4.2.1 Introduction

In the previous section we have discovered another useful applications for hash functions. In general such hash functions are essential for any kind of quick estimation. The Flajolet-Martin [FM85] algorithm shows that hash functions can also be very useful for giving us an estimate about the number of distinct elements in a stream.

Hash functions are really useful, because we can shrink the information to an arbitrary size. Of course some information will be lost in the process. However, in the end we can view hash functions as random number generators, which will always produce the same result for the same arguments.

4.2.2 Algorithm

Before we can estimate the number of distinct elements we need to choose an upper boundary of distinct elements m. This boundary gives us the maximum number of distinct elements that we might be able to detect. Choosing m to be too small will influence the precision of our measurement. Choosing an m that is far bigger than the number of dinstinct elements will only use too much memory. Here the memory that is required is $\mathcal{O}(\log(m))$.

For most applications 64-bit is a sufficiently large bit-array. The array needs to be initialized to zero. We will then use one or more adequate hashing functions. They will map the input to a number, that is representable by our bit-array. This number will then be analysed for each record. If the resulting number contained k trailing zeros, we will set the k-th bit in the bit array to one.

Finally we can estimate the currently available number of distinct elements by taking the index of the first zero bit in the bit-array. This index is usually denoted by R. We can then estimate the number of unique elements N to be estimated by

$$N = 2^R, \tag{5}$$

where the result being up to 1.83 binary magnitudes off. This is easy to see once we realize that the precision of the estimate is only given in powers of two. The standard deviation of the result is, however, a constant.

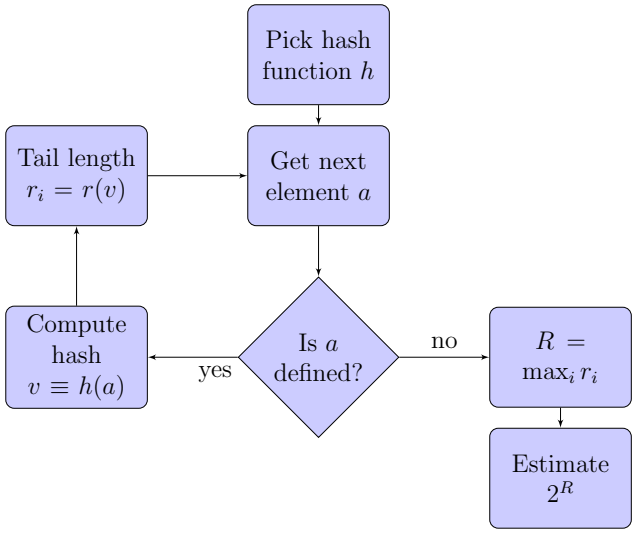

Figure 4: Flow chart for the Flajolet-Martin algorithm

The algorithm works, because the probability that a given hash function $h(a)$, as shown in fig. 4, ends in at least r zeros is 2^{-r}. In case of m different elements, the probability that $R \geq r$ is given by

$$P(R \geq r) = 1 - \left(1 - 2^{-r}\right)^m . \tag{6}$$

Since 2^{-r} is small, we can estimate the probability to be approximately $1 - \exp\left(-2^{-r}m\right)$. There are two edge cases that can easily be checked:

1. We have $2^r \gg m$. In this case we would dramatically over-estimate the result. However, it is easy to show that the probability for this is close to zero,

$$1 - (1 - 2^{-r})^m \approx 1 - (1 - 2^{-r}m) \approx \frac{m}{2^r} \approx 0. \tag{7}$$

2. We have $2^r \ll m$. In this scenario we would be under-estimating the result by far. Here the probability that $R \geq r$ is going to be 1. We compute,

$$1 - (1 - 2^{-r})^m \approx 1 - \exp\left(-2^{-r}m\right) \approx 1. \tag{8}$$

Nevertheless, there are reasons why the algorithm won't work with just a single hash function. The probability halves when $R \to R + 1$, however, the value doubles. In order to get a much smoother estimate, that is also more reliable, we can use many hash functions. In that case we are getting many samples, which may then be grouped and averages. The median of the group averages is then the final result.

4.2.3 Example

As an example let's assume that we have a dictionary consisting of w unique words. We choose a subset of size m words, where $m \leq w$. Then we start streaming elements, where we pick words from our subset of size m. We choose elements by randomly selecting them with redraw. Picking a uniform distribution is sufficient.

In the end we want to know how many unique words have been found. Depending on the stream length and the random process we will end up with a number $m' \leq m$. We are now interested in how good we could estimate m'.

The plot in fig. 5 illustrates the major drawback of the most trivial algorithm. Even though we may be quite close to the result in some cases, we can only be exact for $m' = 2^l$ with $l \in \mathbb{N}$. Usually, however, we won't have an exact result in these cases. The result depends on the quality of the used hash function, the stream length and the presented unique words m'.

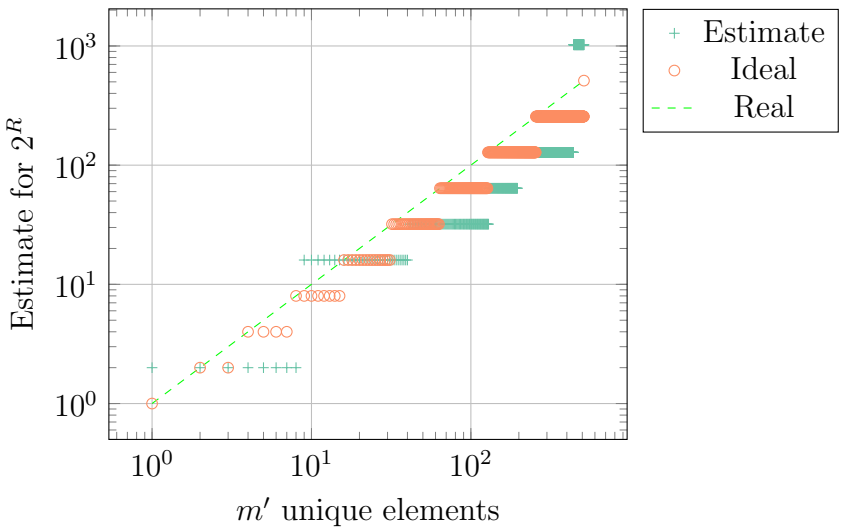

Figure 5: The estimate for a sample consisting of m' unique elements with a stream length of 10^7.

4.2.4 Outlook

Improving the estimation is just a question of including more hash functions and adding more samples. The desired estimation precision can be controlled quite easily. In practice this does not matter so much, as we are only interested in estimates and ratios.

Additionally more sophisticated methods exist, which attack outliers or invalid results by dropping values. Most of these methods do not come with memory cost, even though some might require slightly more computational work.

5 Practical examples

The following examples have been chosen to illustrate usage of streaming algorithms. These applications also provide a good basis or extension for actual streaming implementations.

5.1 Google BigTable

One of the leading data mining companies is Google. Not only is their main product completely based on web crawling, the also earn their money by matching streams of requests to web profiles. Creating, maintaining and improving these profiles requires a lot of different techniques.

As an example the BigTable [CDG+06] data storage system is Google's high performance archival storage, based on the Google FileSystem (GFS). In order to reduce write cycles to a minimum, the system uses, e.g., a bloom-filter for detecting if a given key is not already stored in the system.

There are also offsprings from the proprietary project. The probably most famous one is called Hypertable. Nearly all of these offsprings are quite close to the ideas published in the papers about BigTable.

5.2 Hadoop

Google published some quite influential papers for big data or data processing in general. Two famous ones focus on the Google File System [GGL03] and the MapReduce algorithm [DG04]. Since Google's server software is proprietary, open source offsprings have been created. A

popular implementation of the MapReduce algorithm is Hadoop [Whi09], which has been designed for storage and large-scale processing of data-sets on clusters of commodity hardware.

While the MapReduce algorithm is a topic that has to be discussed on its own, the technology that is provided in form of Hadoop is certainly a great tool for data streaming. For the end-users any programming language can be used with "Hadoop Streaming" to implement the two parts, *map* and *reduce*, of the user's program. The Hadoop framework itself is mostly written in the Java programming language, with some native code in C and command line utilities written as shell-scripts.

5.3 Apache Cassandra

The Apache Cassandra [LM10] project is also using a Bloom filter for reducing the number of disk lookups by detecting non-existent rows or columns. Apache Cassandra is an open source distributed database management system, that performs really well as an archival storage for stream processing.

The project originated at Facebook, where its initial purpose has been to power their Inbox Search feature. When Facebook abandoned the project, it has been released as an open source project. Since then a lot of improvements, features and capabilities have been added. Today, Cassandra is considered one of the fastest and most reliable distributed DBMS. Cassandra also places a high value on performance.

One of the key factors for Cassandra's performance is the ability to avoid costly disk lookups by using streaming algorithms. This considerably increases the performance of a database query operation. Since tables may be created, dropped, and altered at runtime without blocking updates and queries, Cassandra presents an ideal archival storage.

6 Conclusion

The topic of streaming algorithms is much deeper than one might realize at first glance. The introductory only covers the most basic algorithms and ideas. The general method is to reduce the incoming amount of data to a number, which may then be used for estimation. Quite often the reduction is possible by using hash functions or reservoirs of (random) numbers.

A comprehensive review of the current research progress can be found in the paper by Gaber [GZK05]. This paper exceeds this introduction by far and represents a good starting point for any detailed studies on the subject.

The book by Witten [WFH11] can be recommended as an introduction and reference to the general subject of Data Mining. Even though streaming algorithms are not discussed in this book, many other algorithms are. The most important algorithms are all presented in great detail. Additionally implementation details and optimization notes are included.

The most important question, however, has to be asked before starting to use any algorithm. What exactly do we want to know?

References

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 20–29, New York, NY, USA, 1996. ACM.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, June 2002.

[FM85] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, September 1985.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[Gib01] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 541–550, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[GZK05] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34(2):18–26, June 2005.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[Ram89] M. V. Ramakrishna. Practical performance of bloom filters and parallel free-text searching. *Commun. ACM*, 32(10):1237–1239, October 1989.

[RU11] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[Vit85] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985.

[WFH11] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[Whi09] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.