Cardinality Estimation: An Experimental Survey

Hazar Harmouch
Hasso Plattner Institute, University of Potsdam
14482 Potsdam, Germany
hazar.harmouch@hpi.de

Felix Naumann
Hasso Plattner Institute, University of Potsdam
14482 Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Data preparation and data profiling comprise many both basic and complex tasks to analyze a dataset at hand and extract metadata, such as data distributions, key candidates, and functional dependencies. Among the most important types of metadata is the number of distinct values in a column, also known as the zeroth-frequency moment. Cardinality estimation itself has been an active research topic in the past decades due to its many applications. The aim of this paper is to review the literature of cardinality estimation and to present a detailed experimental study of twelve algorithms, scaling far beyond the original experiments.

First, we outline and classify approaches to solve the problem of cardinality estimation – we describe their main idea, error-guarantees, advantages, and disadvantages. Our experimental survey then compares the performance all twelve cardinality estimation algorithms. We evaluate the algorithms' accuracy, runtime, and memory consumption using synthetic and real-world datasets. Our results show that different algorithms excel in different in categories, and we highlight their trade-offs.

PVLDB Reference Format:
Hazar Harmouch and Felix Naumann. Cardinality Estimation: An Experimental Survey. *PVLDB*, 11(4): 499-512, 2017.
DOI: https://doi.org/10.1145/3164135.3164145

1. DATA PROFILING AND CARDINALITY

The research area of data profiling includes a large set of methods and processes to examine a given dataset and determine metadata about it [1]. Typically, the results comprise various statistics about the columns and the relationships among them, in particular dependencies. Among the basic statistics about a column are data type, the number of unique values, maximum and minimum values, the number of null values, and the value distribution. Dependencies involve for instance functional dependencies, inclusion dependencies, and their approximate versions. Data profiling has a wide range of conventional use cases, namely data exploration, cleansing, and integration. The produced metadata is also useful for database management and schema reverse engineering. Data profiling has also more recent use cases, such as big data analytics. The generated metadata describes the big data structure, how to import it, what it is about, and how much of it there is. Thus, data profiling can be considered as an important preparatory task for many big data analysis and mining scenarios to assess which data might be useful and to reveal a new dataset's characteristics. In this paper, we focus on one facet of big data profiling: finding the *number of distinct values* in a column of a large dataset.

Finding this "cardinality" is an active research area, because of its ever growing number of applications in a wide range of computer science domains. Besides its importance as a fundamental task in database query processing and optimization [33], counting distinct values is considered as one of the main studied problems in network security monitoring [12], data streams [2], search engines and online data mining [24, 27]. Moreover, the number of distinct values is used in connectivity analysis of internet topology to find the distance between a pair of nodes in the Internet graph [17].

Without doubt, given a memory size linear to the cardinality of the dataset makes finding the cardinality an easy task. Nevertheless, such memory need is too much for some applications. Therefore, many algorithms to approximate the cardinality of a dataset have been developed in a manner reducing resource/memory consumption. The other cost-dimension to consider is that of I/O. However, it can be shown that approaches that save cost by sampling cannot guarantee any reasonable degree of accuracy. Thus, research has focussed on reducing memory consumption and assumes to read all data only once. This paper presents many well-known algorithms with which the cardinality can be estimated in big datasets using small additional storage and a small number of operations per element. Our results serve as a guide to choose a suitable algorithm for a given use-case.

Outline. The rest of this paper is organized as follows: We first formally define the problem of finding the cardinality of a dataset, present general approaches used in literature to solve this problem, and discuss several classifications of concrete algorithms to estimate the cardinality of a dataset in Section 2. In Section 3, we present, discuss, and compare twelve well-known algorithms. Section 4 presents our comprehensive set of comparative experiments using both synthetic and real-world data, and reports the results of the empirical evaluation. Finally, we conclude in Section 5.

Proceedings of the VLDB Endowment, Vol. 11, No. 4
Copyright 2017 VLDB Endowment 2150-8097/17/12... $ 10.00.
DOI: https://doi.org/10.1145/3164135.3164145

2. PRELIMINARIES

In this section, we formally define and describe the problem of finding a dataset cardinality. We also discuss several previously studied general approaches to solve this problem and the limitation of each one. Finally, we present several classifications of the known algorithms to find an approximation of the number of distinct values.

2.1 Problem statement

The problem of finding the number of distinct values of a multiset is polyonymous: In statistics, it is known as the problem of estimating the number of species in a population. It is also know as the cardinality of a column or the "COUNT DISTINCT" in database literature. Furthermore, the number of distinct values in a multiset is referred to as the zeroth-frequency moment by Alon, Marias, and Szegedy, who introduced the frequency moments of a multiset [3]:

Definition 1. Consider a multi-set $E = (e_1, e_2, \ldots, e_n)$ of n items where each e_i is a member of a universe of N possible values and multiple items may have the same value. Let $m_i = |\{j : e_j = i\}|$ denote the number of occurrences of $i \in N$ in the multi-set E. The *frequency moments* F_k for each $k \geq 0$ are

$$F_k = \sum_{i=1}^{n} m_i^k$$

The number of distinct values in E, called the zeroth-frequency moment F_0 of the multi-set E, is the number of elements from universe N appearing at least once in E. For most applications NULL values are discarded. The algorithms for finding F_0 are the main topic of this survey.

The cardinality F_0 has a wide variety of applications. Each application has its special requirements for designing an algorithm determining F_0. Some of these applications require a very accurate estimation of F_0. However, others accept a less accurate estimation. To give an illustration, the number of distinct visitors of a website influences the price of showing advertisements. So allowing only a small error in measuring F_0 is important. In comparison, a good estimation of the number of distinct connections is enough to detect a potential denial of service attack. To address these differing needs, some applications focus on high accuracy but have a high memory consumption and runtime. Others do the opposite and accept lower accuracy and can better limit memory and runtime. As a result, the key requirements for F_0 estimation algorithms for a specific application can be specified by trading off among accuracy, memory consumption, and runtime.

To quantify the accuracy of such algorithms, we check how close the estimation is to the true cardinality. There are many error metrics to evaluate the accuracy of an estimation algorithm; the most popular ones are standard and relative error:

Definition 2. The *standard error* of an estimation $\hat{F_0}$ is the standard deviation of F_0 divided by F_0:

$$E_{standard}(\hat{F_0}) = \sigma_{\hat{F_0}}(F_0)/F_0 \tag{1}$$

Definition 3. The *relative error* of an estimation $\hat{F_0}$ is:

$$E_{relative}(\hat{F_0}) = |\hat{F_0} - F_0|/F_0 \tag{2}$$

To address the strength of the algorithm guarantee, a new estimation error metric emerged:

Definition 4. The (ε, δ) *approximation scheme* of an estimation $\hat{F_0}$ means that the estimator guarantees a relative error of ε with probability $\geq 1 - \delta$ where $\varepsilon, \delta < 1$.

Besides the accuracy, we also need to quantify the memory consumption of the cardinality estimation algorithm. The data structure maintained by the estimation algorithm in main memory is called *synopsis*. An estimation algorithm should find an estimate $\hat{F_0}$ of the dataset cardinality close to the true F_0 as a function of the synopsis size. The synopsis size is essentially proportional to the number of elements in the multiset to exactly determine F_0 by sorting. However, multisets today tend to be too big to fit in main memory of one machine or in the allotted memory for the profiling process. Consequently, the synopsis of the estimation algorithm can be seen from two different perspectives. First, the synopsis is only a temporary data structure used to estimate F_0 in static scenarios, i.e., the whole dataset is stored on disk. Second, the synopsis is a replacement or a compact representation of the real data in scenarios where we cannot store the dataset, such as in streaming applications.

To sum up, the goal of this experimental survey is to present and analyze the efficiency of a wide range of known algorithms for estimating F_0, the number of distinct elements/cardinality of a multiset. We also take into account the application requirements determined by the three factors: accuracy, memory consumption, and runtime.

2.2 General approaches

Here we present several broad approaches that are used to estimate the number of distinct values in a multi-set.

Bitmap. The trivial method to determine F_0 exactly is by using a *bitmap* of size N, the size of the universe, as the synopsis. The bitmap is initialized to 0s. Then, we scan the dataset item-wise and set the bit i to 1 whenever an item with the i-th value of the universe is observed. After a single scan of the dataset, F_0 is the number of 1s in the bitmap. The synopsis size is a function of the universe size N, which is potentially much larger than the size of the dataset itself. Thus, in general, this approach is infeasible, but bitmaps in general do play a role in other approaches.

Sorting. The traditional method of determining F_0 was through *sorting* to eliminate duplicates [32]. However, sorting is an expensive operation that requires a synopsis size at least as large as the dataset itself. So, sorting is an impractical approach especially for the current big datasets.

Hashing. A straightforward approach to obtain an exact F_0 and scale-down the synopsis size is *hashing*. Hashing eliminates duplicates without sorting and requires only one pass over the dataset to build a hash table. However, a simple application of hashing can be worse than sorting in terms of memory consumption. To accurately capture datasets with high column cardinalities, the hash table would need to be too large to fit in normal main memory.

While the methods above are exact, they are also expensive in both size and runtime. If we relax the need for an exact solution, other approaches are available.

Bitmap of hash values (Bloom filter). Instead of storing the hash table, Bloom filters combine the bitmap approach with hashing to keep track of the hashed values of the dataset items [25]. However, the main problem of this approach is that it requires a prior knowledge of the maximum expected cardinality to choose a good bitmap size.

Sampling. Another common general approach is *sampling* to reduce the synopsis size. Various studies used this approach for cardinality estimation [13,18,22]. Obviously, F_0 is difficult to estimate from a sample of the dataset. Charikar et al. presented powerful negative results for estimating F_0 from a sample of a large table [8]: for every estimator based on a small sample, there is a dataset where the ratio between the cardinality estimate and the exact cardinality is arbitrarily large. I.e., if the estimator does not examine a large fraction of the input data, there is no guarantee of low error across all input distributions. These results match the results obtained in [22, 23]. There, Haas et al. highlighted that to bound the estimation error within a small constant, almost all the dataset needs to be sampled. Therefore, we can admit that any approach based on sampling is unable to provide a good guaranteed error and we need to read the entire dataset to determine an accurate estimation, , as we show in the experimental section..

Observations in hash values. Another approach relies on making *observations* on the hashed values of the input multiset elements to reduce the size of synopses, such as the length of a particular prefix in the binary representation of the hashed values. The observations are linked only to cardinality and are independent of both replication and order of the items in the dataset. These observations are then used to infer an estimation $\hat{F_0}$ of the dataset cardinality. Most of the algorithms presented in this survey follow this approach to estimate F_0. More details are given in the following section.

2.3 Classification

The backbone of most modern cardinality estimation algorithms is the work of Flajolet and Martin in the mid-1980's [15]. For that reason, Gibbons in his survey for the literature on distinct-values estimation has a separate family of algorithms named Flajolet and Martin's Algorithms [19]. Under this family, he classified FM & PCSA [15], AMS [3], and LogLog & SuperLogLog [11]. To understand the similarity among the larger set of algorithms presented in this survey, we present two more fine-grained previous classifications of cardinality estimation algorithms. In addition, we provide a new classification that distinguishes the core method of the algorithms. Table 1 gives a summary of these algorithms and their class according to each classification. We discuss them in detail in the following section.

The first classification is by Flajolet et al. [14]. The authors classified algorithms in two categories corresponding to the type of observations *bit-pattern observables* and *order statistic observables*. In the first category, the hash values are seen as bit-strings. The algorithms are based on the occurrence of particular bit patterns at the binary string representation of the dataset values. On the other hand, the order statistic observable algorithms consider the hash values as real numbers. The estimation is based on the order statistics rather than on bit patterns in binary representations. The order statistic of rank k is the k-th-smallest value in the dataset, which is not sensitive to the distribution of repeated values. So, the minimum of the hashed values is a good observable. The hash function distributes the hash values uniformly in the interval [0-1]. The minimum of n uniform random variables taking their values in [0-1] is an estimate of $1/(n+1)$. So we are able to retrieve an approximation of n from this value. All the algorithms in our survey belong to the first category except BJKST, MinCount, and AKMV, which are associated with the second category. LC and Bloom filter do not use any observable. The second classification is a high-level classification given by Metwally et al. [27] (Figure 1). The authors distributed the algorithms into two broad categories: *Sampling algorithms* and *sketch-based algorithms*. The first category contains the algorithms that take advantage of not scanning the entire dataset, but estimate the cardinality by sampling (discussed in the previous section). Algorithms of the second category scan the entire dataset once, hash the items, and create a sketch. The sketch, also called synopsis, is queried later on to estimate the cardinality.

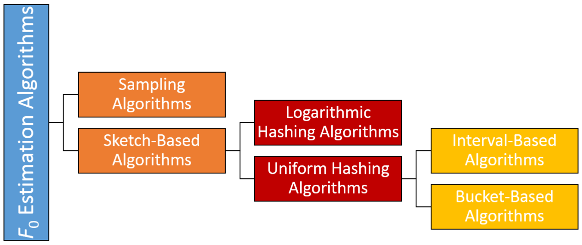

F_0 Estimation Algorithms
Sampling Algorithms
Sketch-Based Algorithms
Logarithmic Hashing Algorithms
Uniform Hashing Algorithms
Interval-Based Algorithms
Bucket-Based Algorithms

Figure 1: Classification of F_0 estimation algorithms [27].

Metwally et al. further classified the sketch-based algorithms according to their hashing probabilities into *logarithmic hashing* and *uniform hashing* algorithms. The former keeps track of the most uncommon element observed so far, using a bitmap and a hash function. The hash function maps each element to a bit in the bitmap with a hashing probability that decreases exponentially as the bit significance increases. FM, PCSA, AMS, LogLog, SuperLogLog, HyperLogLog, and HyperLogLog++ are in this category of the sketch-based algorithms. The insight of the latter class, uniform hashing, is to employ a uniform hash function to hash the entire dataset into an interval or a set of buckets. Thus, this class comprises two classes: *Interval-based* algorithms and *Bucket-based* algorithms. F_0 is estimated in the interval-based algorithms based on how packed the interval is. But F_0 is estimated based on the probability that a bucket is (non)empty by the bucket-based algorithms. BJKST, LC, and Bloom filter are examples of the bucket-based category. AKMV and MinCount are members of the interval-based class.

The first classification by Flajolet et al. [14] depends on the observable which an estimation algorithm uses. The second classification by Metwally et al. [27] is based on the intuition of the algorithm and how it maps the hash values to a bit in the bitmap. We added a third classification based on the core method an algorithm uses to estimate F_0, as explained in the next section.

Table 1: Classifications of algorithms studied in this survey

Algorithm	**Ref.**	**Year**	**Observables** [14]	**Intuition** [27]	**Core method** (Sec. 3)
FM	[15]	1985	Bit-pattern	Logarithmic hashing	Count trailing 1s
PCSA	[15]	1985	Bit-pattern	Logarithmic hashing	Count trailing 1s
AMS	[3]	1996	Bit-pattern	Logarithmic hashing	Count leading 0s
BJKST	[4]	2002	Order statistics	Bucket-based	Count leading 0s
LogLog	[11]	2003	Bit-pattern	Logarithmic hashing	Count leading 0s
SuperLogLog	[11]	2003	Bit-pattern	Logarithmic hashing	Count leading 0s
HyperLogLog	[14]	2008	Bit-pattern (order statistics)	Logarithmic hashing	Count leading 0s
HyperLogLog++	[24]	2013	Bit-pattern	Logarithmic hashing	Count leading 0s
MinCount	[21]	2005	Order statistics	Interval-based	k-th minimum value
AKMV	[7]	2007	Order statistics	Interval-based	k-th minimum value
LC	[32]	1990	No observable	Bucket-based	Linear synopses
BF	[28]	2010	No observable	Bucket-based	Linear synopses

3. REVIEW OF TWELVE CARDINALITY ESTIMATION ALGORITHMS

After introducing the motivation, the problem, and general approaches to solve it, we provide an overview of concrete cardinality estimation algorithms and how they trade the accuracy for runtime and memory consumption. Table 2 summarizes this section.

3.1 Trailing 1s algorithms

This algorithm family uses the number of trailing 1s in the bit pattern observable's bitmap as the core method to estimate F_0.

3.1.1 Flajolet and Martin (FM)

Flajolet and Martin designed the first algorithm to estimate F_0 in a single pass using less than one hundred binary words additional storage and only a few operations per elements (what would nowadays be called a data stream scenario) [15]. This algorithm, also known as *probabilistic counting*, uses the observations approach. Flajolet and Martin observed that if a hash function h maps the elements into uniformly distributed integers (binary strings y of length L) over the range $[0 \dots 2^L - 1]$, the pattern $0^k 1 \dots$ appears with probability $\frac{1}{2^{(k+1)}}$.

They formalized this observable as a function $\rho(y)$ that represents the position of the least significant 1-bit in y, i.e., k if $y > 0$ and L otherwise. Then, they recorded these observable values using a bitmap B initialized to all 0. All items with same value set at random the same bit $\rho(y)$ in B to 1. After the algorithm scans the entire dataset, B is independent of any duplication and $B[i] = 1$ if there are at least $2^{(i+1)}$ distinct values. If $B[i+1]$ is still 0, F_0 is likely greater than $2^{(i+1)}$ but less than $2^{(i+2)}$.

Therefore, Flajolet and Martin used R, the position of least significant bit that was not flipped to 1 in the bitmap B as an indicator of $log_2(\varphi * F_0)$ with standard deviation close to 1.12. In other words, R is the number of trailing 1s in B. So the estimation $\hat{F_0}$ of the cardinality F_0 is given by $\hat{F_0} = 2^R/\varphi$ where $\varphi = 0.77351$ is a statistical correction factor. To reduce the variance of R, the FM algorithm is improved to take the average of m runs of the previous procedure using a set of m hash functions to compute m bitmaps, i.e., $\hat{F_0}$ is

$$\hat{F}_0^{FM} = 2^{\overline{R}}/\varphi \qquad with \qquad \overline{R} = \frac{1}{m} * \sum_{i=1}^{m} R_i \tag{3}$$

Hence, the standard error of the estimator is reduced by a factor of $\mathcal{O}(\sqrt{m})$ to become $\mathcal{O}(1/\sqrt{m})$, yet CPU usage per element processing is multiplied by m. FM's accuracy is directly proportional to the synopsis size namely to the design parameter m. The size L of the bitmap is also an important design parameter and depends on the maximum cardinality N_{max} to which we safely want to count up to, and selected to be larger than $log_2(N_{max}/m) + 4$. However, error analysis of the FM algorithm is based on the assumption that explicit family of hash functions with some ideal random properties is used.

3.1.2 Probabilistic counting with stochastic averaging (PCSA)

In the same article, Flajolet and Martin pointed out that the use of what is called *stochastic averaging* can achieve the same effect as when using direct averaging in the FM algorithm. The result was a new variant of the FM algorithm called *probabilistic counting with stochastic averaging* (PCSA) [15]. PCSA uses the same observable of FM but reduces the processing time per element to $\mathcal{O}(1)$ as well as reducing the synopsis size.

The intuition behind PCSA is to distribute the dataset items into buckets hoping that F_0/m items fall into each bucket. Then, R as described in FM of each bucket should be close to F_0/m and the average of those values can be used in the right hand side of the Equation (3) as $\overline{R}$ to derive a reasonable approximation of F_0/m.

This intuition is implemented using m bitmaps, one per bucket, and a single hash function h that is used to distribute the dataset elements into one of the bitmaps. When PCSA observes a new item x, the $log_2(m)$ least significant bits of the binary representation of $h(x)$ are used to determine the bitmap to be updated and the remaining bits are used to find the observable ρ (same in FM), and then set the corresponding bit to 1 within the previously determined bitmap. So the estimation $\hat{F_0}$ of the cardinality F_0 is given by:

$$\hat{F}_0^{PCSA} = m * 2^{\overline{R}}/\varphi \tag{4}$$

where φ, L, the size of each bitmap, and $\overline{R}$ are identical to those in FM algorithm. But since each bitmap has seen only $1/m$ of the total distinct values, we multiply by m. PCSA has several advantages over FM: it reduces the cost of FM by using a single hash function and increases the estimation accuracy to an expected standard error of $0.78/\sqrt{m}$.

3.2 Leading 0s algorithms

This algorithm family uses the number of leading 0s in the bit pattern observable's bitmap as the core method to estimate F_0. But, the algorithms of this family do not maintain an actual bitmap. Instead they keep only the maximum observable value which equals to the number of leading 0s in the observable's bitmap.

3.2.1 Alon, Martias and Szegedy (AMS)

Alon et al. provided the first theoretic definition and discussion of the frequent-moments statistics for approximate counting [3]. In their work, they revise the FM algorithm as a randomized algorithm for estimating F_0 and adapt it into the AMS algorithm. First, they argue that FM was designed assuming an explicit family of ideal random hash functions that could be unrealistic. In consequence, they proposed to use linear hash functions instead. Second, AMS keeps using the same observable $\rho(y)$. But it uses R, the position of most significant bit flipped to 1 in the bitmap B, as an indicator of $log_2(F_0)$. In other words, R is the number of leading 0s in B. The number of distinct values is likely to be 2^r, if $\rho(y) = r$. Thus, after scanning the entire dataset, one of the observable values hits $\rho(y) \geq log_2 F_0$. The maximum value of $\rho(y)$, namely R, is a good estimation of $log_2(F_0)$. AMS estimates F_0 by:

$$\hat{F}_0^{AMS} = 2^R \tag{5}$$

Finally, AMS does not use a bitmap to record the observable values. Instead, it keeps track of only the maximum observable value R. The authors proved that AMS guarantees a ratio error of at most c with a probability of at least $1 - \frac{2}{c}$ for any $c > 2$. Using m hash functions can further improve the accuracy with the trade-off of increasing the space and time linearly.

FM is still more accurate than AMS [19]. The least significant 0-bit in B (R in FM and PCSA) is more accurate than the most-significant 1-bit (R in AMS) to estimate F_0. The reason is that the bit that represents R in AMS can be set by a single outlier hash value. As a result, AMS overestimates F_0 as 2^R, especially when the bits preceding the most significant bit are zeros. AMS and FM share the drawback of performing m hashes for every element.

3.2.2 Bar-yossef, Jayram, Kumar, Sivakumar and Trevisan (BJKST)

Three theoretical algorithms for approximating F_0 are presented in [4]. We focus on the third algorithm, because it is the most used and, in a sense, the best one. It is known as the BJKST algorithm, an acronym of the authors last names. This algorithm is an improvement of the work in [5] based on AMS, unified with the Coordinated Sampling algorithm presented by Gibbons and Tirthapura [20].

In essence, BJKST is based on AMS. It uses the same function $\rho(y)$, but does not simply keep track of the maximum value of $\rho(y)$. Instead, it resembles the Coordinated Sampling algorithm and uses a pairwise independent universal hash function h and a buffer B to estimate F_0. The hash function h guarantees that the probability of $\rho(h(x)) \geq r$ is precisely $1/2^r$ for any $r \geq 0$ as stated in [3]. Thus, items $\{x_0, x_2, \ldots, x_n\}$ of the dataset can be assigned to a level according to their $\rho(h(x_i))$ values as following: half of the items have a level equal to 1, a quarter of them have a level equal to 2, and $1/2^r$ have a level equal to r.

The buffer B initially stores all the elements scanned so far and their level is at least $Z = 0$. Whenever the buffer size is larger than a predefined threshold θ, the level Z is increased by one and all the elements in B with a level of less than Z are removed, and so on. Unlike the Coordinated Sampling algorithm, BJKST stores pairs $(g(x_i), \rho(h(x_i)))$ instead of keeping the actual value of the element x_i in order to further improve the efficiency of the buffer. g is another uniform pairwise independent hash function. These pairs are stored in array of binary search trees where the j-th entry contains all the pairs in level j.

After one pass over the dataset, BJKST finds the minimum level Z for which the buffer size does not exceed a specific threshold θ. Also, it expects $F_0/2^r$ elements to be in the level Z, i.e., $|B| = F_0/2^Z \leq \theta$. Therefore, $\hat{F}_0$ is

$$\hat{F}_0^{BJKST} = |B| * 2^Z \tag{6}$$

BJKST can provide an (ε, δ) approximation scheme of F_0, when the output is the median of running $\mathcal{O}(log(1/\delta))$ parallel copies of the algorithm. Then, $\theta = 576/\varepsilon^2$ for any $\varepsilon > 0$ and $0 < \delta \leq \frac{1}{3}$.

Both BJKST and Coordinated Sampling have the advantages of keeping samples of the data that can be used later. But BJKST improves the efficiency of the buffer, both in space and processing time, as explained above.

3.2.3 LogLog

Durand and Flajolet introduced another AMS-based estimator for F_0, which uses only $\log_2 \log_2(N_{max})$ of memory to estimate cardinalities in range of millions with a relatively high accuracy [11]. This LogLog algorithm uses PCSA's intuition to overcome the overestimation problem in AMS. It improves space usage over PCSA by trading off the accuracy.

The algorithm uses m buckets $B_1, \ldots, B_m$ to distribute the dataset items over them. Then, LogLog uses the AMS approach and maintains R_i for each bucket B_i. Each bucket is responsible for about F_0/m of the distinct elements. Thus, the arithmetic mean $\overline{R}$ of $R_1, \ldots, R_m$ is a good approximation of $log_2(F_0/m)$. The LogLog estimator returns $\hat{F}_0$ with a standard error $\approx 1.3/\sqrt{m}$, as the following:

$$\hat{F}_0^{LogLog} = \alpha_m * m * 2^{\overline{R}} \qquad with \qquad \overline{R} = \frac{1}{m} * \sum_{i=1}^{m} R_i \tag{7}$$

In a practical implementation, the correction factor $\alpha_m = 0.39701$ as soon as $m \geq 64$.

Whenever a new element x_j is scanned, the algorithm uses the first $k = log_2(m)$ bits of the binary representation of $h(x_j)$ to map the element x_j to a bucket B_i. Then, it updates R_i after comparing its value with $\rho(h(x_j))$, after ignoring the first k bits. Like AMS, LogLog maintains only the value of the maximum R_i, and not a bit vector.

3.2.4 SuperLogLog

SuperLogLog is an optimization of LogLog algorithm [11]; Durand and Flajolet suggest two improvements. The first one decreases the variance of the $\hat{F}_0$ around the mean, while the second improves the space cost by bounding the size of each R_i. To implement the improvements, SuperLogLog uses two rules: the truncation rule and the restriction rule.

The *truncation rule* refers to discarding the largest 30% of the estimates when averaging R_i to produce the final estimate. In other words, SuperLogLog retains only the

$m_0 = \lfloor 0.7 * m \rfloor$ smallest values to compute the truncated sum $\sum^* R_i$. Thus, SuperLogLog estimates F_0 by:

$$\hat{F}_0^{SuperLogLog} = \tilde{\alpha}_m * m_0 * 2^{\overline{R}} \quad with \quad \overline{R} = \frac{1}{m_0} * \sum{}^* R_i \tag{8}$$

The modified statistical correction factor $\tilde{\alpha}_m = 1.09295$ minimizes the bias [27]. Empirically, this truncation increases the accuracy and bounds the standard error of order $1.05/\sqrt{m}$.

The *restriction rule* says that each R_i can be represented using only $\lceil log_2 \lceil log_2 * (\frac{N_{max}}{m}) + 3 \rceil \rceil$ bits.

3.2.5 HyperLogLog

Flajolet et al. introduced HyperLogLog as a near-optimal successor to LogLog [14]. HyperLogLog uses the same observable, $\rho(y)$, as LogLog and also maintains the maximums R_i. But, it reduces the estimation's variance using harmonic means to estimate F_0 from the maximums R_i. Based on the same intuition behind LogLog, the harmonic mean $\overline{R}$ of $2^{R_1}, \dots, 2^{R_m}$ is close to F_0/m. Therefore, HyperLogLog returns an estimation of F_0 as a normalized bias corrected harmonic mean:

$$\hat{F}_0^{HyperLogLog} = \alpha_m * m * \overline{R} \tag{9}$$

with

$$\overline{R} = \frac{m}{\frac{1}{2^{R_1}} + \dots + \frac{1}{2^{R_m}}}$$

and α_m is a bias correction factor where $\alpha_{16} = 0.673, \alpha_{32} = 0.697, \alpha_{64} = 0.709$, and $\alpha_m = 0.7213/(1 + 1.079/m)$ for $m \geq 128$.

The algorithm archives a standard error in the order of $1.04/\sqrt{m}$. The authors' practical results analysis shows that the estimation of F_0 maintains the theoretical standard error in the range $]\frac{5}{2} * m, \frac{2^{32}}{30}]$ for any $m \in \{2^4, \dots, 2^{16}\}$.

Two corrections are introduced to deal with the $\hat{F}_0^{HyperLogLog}$ values that fall outside the specified range, either in the *small range* (i.e., $\leq \frac{5}{2} * m$) or in the *large range* (i.e., $> \frac{2^{32}}{30}$). The problem in small range is the presence of nonlinear distortions. The source of the bias is the high number V of $R_i = 0$ in the harmonic mean when n is small compared to m. The small range correction uses LC to estimate F_0 from the maximums R_i when $V > 0$ as:

$$\hat{F}_0^{HyperLogLog*} = m * log(m/V) \tag{10}$$

In the large range the cardinality is reaching 2^{32}, which causes an increase in the hash collisions due to 32-bit hash function used by HyperLogLog. So, the algorithm applies a correction to the estimation and returns:

$$\hat{F}_0^{HyperLogLog*} = -2^{32} log(1 - \hat{F}_0^{HyperLogLog}/2^{32}) \tag{11}$$

Hence, HyperLogLog is a bit-pattern observable algorithm. Yet, it can also be viewed as order statistics observable algorithm, because $1/2^{R_i}$ is an estimation of $min(B_i)$ up to a factor at most 2. The authors argue that HyperLogLog is near optimal, because its estimation standard error is near $1/\sqrt{m}$, the lower bound for accuracy achievable by order statistics algorithms.

3.2.6 HyperLogLog++

HyperLogLog++ is a revision of the HyperLogLog algorithm [24]. The authors suggest a series of changes to improve the original algorithm's estimation accuracy and reduce the space cost. The development of this algorithm was driven by the need to accurately estimate cardinalities well beyond 10^9, as well as small cardinalities and to efficiently adapt memory usage to the cardinality. The authors present three improvements, which can be applied together or independently to fit the need of the application.

First, HyperLogLog++ uses a 64-bit hash function as a replacement to the high range correction in the original algorithm with low additional cost in memory. This increases the size of each R_i by only one bit, but it enables to estimate cardinalities approaching 2^{64} before the hash collisions start to increase.

Second, the authors experimentally found a bias correction method that works effectively up to $n = 5 * m$. They estimate the bias of F_0 from $\hat{F_0}$ using k-nearest neighbours interpolation with the empirically determined values. They further combine this bias correction of the estimation with LC. LC is used to correct estimations that are lower than θ, an empirically determined threshold.

Third, HyperLogLog++ develops a *sparse representation* to avoid the cases where $n \ll m$ and most of the R_i's are never used. This representation is identical to the one used in BJKST, where the algorithm stores pairs $(idx, \rho(y))$. But, HyperLogLog++ switches back to the original *dense representation* whenever maintaining this list consumes more memory than the original memory consumption. As a result, the memory consumption is reduced for small cardinalities with small runtime overhead to maintain the new representation. For more details on the concrete implementation of the sparse representation, refer to [24].

3.3 K-th minimum value algorithms

This algorithm family uses order statistics as their observable, specifically k-th minimum value.

3.3.1 MinCount

The MinCount algorithm was introduced by Giroire in [21] as a generalization of the first algorithm presented in [4], which is where also BJKST is introduced. Like the original algorithm, MinCount is an interval-based algorithm that uses the *k-th minimum value order statistics observable* (KMV) to estimate the density of the interval, which is in turn used to estimate F_0. In other words, MinCount considers the hashed values as a set of independent uniformly distributed real numbers in the interval $[0, 1]$ with repetitions, i.e., an *ideal multi-set* $\acute{E}$.

The algorithm's main idea is that the first minimum of $\acute{E}$ is an indication of $\frac{1}{F_0+1}$. However, the inverse of this minimum has an infinite expectation. MinCount avoids this by using two new aspects: It combines $M^{(k)}(k \geq 2)$ the k-th minimum of $\acute{E}$ and a sub-linear function of $1/M^{(k)}$, such as its logarithm or square root, as a replacement of the first minimum and its direct inverse alone.

Furthermore, MinCount reduces the standard error using the stochastic averaging like in PCSA. So, the hashed values interval is divided into m buckets. A hash value $h(x)$ is mapped to the bucket i if $\frac{i-1}{m} \leq h(x) < \frac{i}{m}$. For each bucket, the values $M_i^{(k)}$ are maintained for $i = 1, \dots, m$. MinCount's best estimate of F_0 is using $k = 3$ and the logarithm function as follows:

$$\hat{F}_0^{MinCount} = m * \left(\frac{\Gamma(k - \frac{1}{m})}{\Gamma(k)}\right)^{-m} * e^{\overline{R}} \tag{12}$$

$$with \qquad \overline{R} = -\frac{1}{m} * \sum_{i=1}^{m} lnM_i^{(k)}$$

where Γ is the Euler Gamma function. The standard error of this estimation up to $1/\sqrt{M}$ using $M = k * m$ units of storage.

3.3.2 AKMV

Beyer et al. also revised the first algorithm proposed in [4] to introduce their unbiased version of F_0 estimator based on KMV order statistics [6, 7]. The authors provided several estimators of F_0 in two scenarios: (1) when the dataset consists of only one partition, which is what we study in this survey, and (2) when the dataset is split into partitions and the estimation is obtained in parallel with the presence of multi-set operations.

The original algorithm uses the k-th smallest hash value to estimate $\frac{K}{F_0}$. Beyer et al. showed that the original estimator overestimates F_0 and it is biased upwards towards F_0 [6]. To lower this bias, the AKMV estimator is given by:

$$\hat{F}_0^{AKMV} = (k-1)/M^{(k)} \tag{13}$$

where $F_0 > k$, otherwise the algorithm finds the exact F_0. If F_0 is expected to be large, the suitable synopsis size k can be determined based on the error bounds. AKMV's relative error is bounded to $\sqrt{\frac{2}{\pi(k-2)}}$.

For the second scenario, i.e., to support the multi-set operations among the synopses of the partitions, the authors introduce the AKMV synopsis and a corresponding F_0 estimator, to estimate F_0 of each partition as well as of the whole dataset. In addition to the k minimum hash values, AKMV maintains k counters. Each counter contains the multiplicity of the corresponding element in the k minimum hash values set. The $\hat{F}_0^{AKMV}$ estimator was generalized to estimate F_0 for compound partition created from disjoint partitions by multi-set operations.

3.4 Linear synopses based estimators

The core method for estimating F_0 for this algorithm family is how packed or empty its linear synopsis is.

3.4.1 Linear Counting (LC)

Whang et al. present a probabilistic algorithm for estimating the number of distinct values in a dataset called Linear Counting (LC) [32]. This algorithm is a straightforward application of the bitmap of hash values approach. LC maintains a bitmap B of size b, in which all entries are initialised to 0.

LC is neither a logarithmic hashing algorithm nor a logarithmic counting algorithm. In contrast, it is a linear counting algorithm that applies a uniform hash function h to each item from the dataset x. Then, $h(x)$ maps the item *uniformly* to a bucket in the bitmap and sets it to 1, i.e., $B[h(x)] = 1$ and the bucket is hit.

After hashing the entire dataset, if there were no collisions the number of 1-bits in B would be F_0. But F_0 can be estimated based on the probability that a bucket is empty. V_n denotes the fraction of empty buckets in the bitmap and is a good estimation of this probability. The expected probability of a bucket being empty is given by $e^{-n/b}$. As a result, F_0 is estimated using maximum likelihood estimator:

$$\hat{F}_0^{LC} = -b * ln(V_n) \tag{14}$$

The size b of the bitmap is defined in terms of a constant called load factor t as $b = F_0/t$. Whang et al.'s analysis reveals that using $t \leq 12$ provides $\hat{F}_0$ with 1% standard error. This fact reduces the synopsis by a factor of t. That leads to the main limitation of LC: it needs some prior knowledge of F_0 to determine the size of B. Practically, the upper bound of cardinality n_{max} is used when creating B instead of F_0. Thus, a linear space of order $\mathcal{O}(n_{max})$ is the main drawback of LC, when we have limited memory or datasets of high cardinalities. Nevertheless, LC is a simple algorithm that can provide a highly accurate estimation $\hat{F}_0$, if one chooses the right load factor. The standard error of $\hat{F}_0^{LC}$ is $\sqrt{(e^t - t - 1)/(t * n_{max})}$.

3.4.2 Bloom filter (BF)

The main source of the algorithm LC's estimation error are hash collisions in the bitmap. Bloom filters can reduce collisions using m independent hash functions. Unlike LC, each element is mapped to a fixed number of bits $\leq m$, i.e., $B[h_i(x)] = 1$. The standard Bloom filter is designed to maintain the membership information rather than a statistical information about the underlying dataset. But one can count the distinct elements in a multiset by combining a Bloom filter with a counter, which is incremented whenever an element is not in the filter. The value of the counter can never be larger than the exact cardinality due to the Bloom filter's nature, but hash collisions can cause it to underestimate F_0.

Bloom filters have been used effectively for cardinality estimation. Like LC, Swamidass and Baldi introduced an estimator of the population of Bloom filter using X as the number of bits set to 1 in the filter [31]. Intuitively, after inserting all the elements of the dataset into a Bloom filter, the number of elements in a Bloom filter is in fact F_0 of the represented dataset. Given a Bloom filter of the size b with m hash functions, F_0 can be estimated by:

$$\hat{F}_0^{BF1} = -\frac{b}{m} * ln(1 - X/b) \tag{15}$$

Papapetrou et al. proposed a probabilistic approach to estimate F_0 of a dataset from its standard Bloom filter representation [28]. This approach estimates the number of elements in a Bloom filter based on its density and requires only X and the configuration of the Bloom filter (b and m). They estimate F_0 by the maximum likelihood value for the number of hashed elements:

$$\hat{F}_0^{BF2} = \frac{ln(1 - X/b)}{m * ln(1 - 1/b)} \tag{16}$$

Their evaluation results prove that the Bloom filter configuration affects the estimation accuracy – larger Bloom filter provides higher estimation accuracy. Bloom filters with fewer hash functions exhibit a more accurate cardinality estimation. Bloom filter shares with LC the same limitation of the need of a prior knowledge of the maximum cardinality in order to choose the suitable size of the filter.

Count-Min, not to be confused with MinCount, is a Bloom filter-like sub-linear synopsis, which estimates the dataset item's frequencies [10]. Count-Min is not originally designed to track the number of distinct values, but to solve problems such as determining quantiles and heavy hitters. However, Cormode pointed out that Count-Min sketches could be updated using FM-like synopses to achieve this goal [9].

Table 2: Error-guarantees of the twelve algorithms

Algorithm	**Error**	**Notes**
FM	Std. err. = $(1/\sqrt{m})$	m: Number of hash functions
PCSA	Std. err. = $0.78/\sqrt{m}$	m: Number of bitmaps
AMS	Ratio err. $< c$ with probability $> 1 - \frac{2}{c}$	$c > 2$
BJKST	(ε, δ) Approximation Scheme	Relative err. $\varepsilon > 0$ and $0 < \delta \leq \frac{1}{3}$
LogLog	Std. err. = $1.3/\sqrt{m}$	m: Number of maximums R_i
SuperLogLog	Std. err. = $1.05/\sqrt{m}$	m: Number of maximums R_i
HyperLogLog	Std. err. = $1.04/\sqrt{m}$	m: Number of maximums R_i
HyperLogLog++	Smaller by factor 4 compared to HyperLogLog for cardinalities up to 12,000 [24].	Precision = 14 Sparse representation precision = 25
MinCount	Std. err. = $1/\sqrt{k * m}$	k: order of the used minimum m: Number of buckets
AKMV	Relative err. $\approx \sqrt{\frac{2}{\pi(k-2)}}$	k: order of the used minimum
LC	Std. err. = $\sqrt{(e^t - t - 1)/(t * n_{max})} = 0.01$ for $t \geq 12$	t: load factor n_{max}: Upper bound on n
BF	Relative err. ≤ 0.04 [28]	BF density = 0.9

To summarize, we presented an overview of the state-of-the-art cardinality estimation algorithms. In the following section, we compare the described algorithms and benchmark their accuracy, runtime, and memory consumption. Table 2 summarizes the error guarantees of the algorithms presented in this section.

4. COMPARATIVE EXPERIMENTS

Most of the algorithms presented in this survey are accompanied with a theoretical analysis of how well they estimate F_0 in terms of error and space bounds. Nevertheless, these analyses suffer from some shortcomings. The $\mathcal{O}()$ notation in space bounds hides the actual space used for maintaining hash functions and data structures. Furthermore, there is no unified error metric or hashing assumption among the algorithms. To decide on a suitable algorithm for a given use case one needs more information.

In this section we experimentally compare all twelve F_0 estimation algorithms to analyze and better understand their behavior using a unified error metric, available memory, and hash function. First, we describe the experimental setup and the implementation details. Then,we briefly evaluate a sampling-based algorithm. Then, we compare the algorithms' accuracies among each other and per algorithm family. Next, we study the correlation between runtime, exact F_0, and dataset size. Finally, we measure memory consumption of these algorithms and report minimum memory needed to run each algorithm.

4.1 Experimental setup

Hardware. We performed all experiments on a Dell PowerEdge R620 server running CentOS 6.4. It has two Intel Xeon E5-2650 (2.00 GHz, Octa-Core) processors, 128 GB DDR3-1600 RAM and a 4 TB RAID-5 storage. We implemented all algorithms as single-threaded Java applications using OpenJDK 64-Bit Server VM 1.8.0_111-b15.

Implementations. To guarantee a unified test environment when comparing the twelve F_0 estimation algorithms, we implemented them for the Metanome data profiling framework [29]. Metanome is a standard framework decoupled from the algorithms. It provides basic functionalities, such as input parsing and performance measurement[1]. In addition to various of discovery algorithms for complex metadata, such as keys or functional dependencies, Metanome supports basic statistics algorithms, including F_0 estimation. To further unify our comparison we need to avoid the significant impact of the used hash function on the runtime and the estimation accuracy. Thus, we implemented all algorithms using the same hash function, namely MurmurHash[2]. We chose MurmurHash based on the results in [30], where the authors showed that PCSA, LC, and LogLog yield the fastest and most accurate F_0 estimation when using MurmurHash compared to Jenkins, Modulo congruential hash, SHA-1, and FNV.

The next implementation decision was whether to use a 64-bit or 32-bit version of MurmurHash. Nowadays, in the era of "Big Data", it is important to estimate cardinalities of over 10^8. The algorithms based on an observable of the hash values are limited by the number of bits used to represent these hash values. For linear synopses, using 64-bit hash functions reduces collisions in the case of large datasets. As a result, we implement all algorithms using the 64-bit MurmurHash version. We made an exception of 32-bits for AKMV and MinCount, because they both use the k-minimums of the hashed values and using 64 bits adds an overhead without improving the algorithms' counting ability. We counted the exact value of F_0 using the "JavaHashSet". Unless stated otherwise, all algorithms were configured to produce theoretical (standard/relative) errors of 1% according to Table 2. LC and Bloom filter use the number of tuples in the dataset as n_{max}. Bloom filter is implemented as a standard Bloom filter with four bits per element and three hash functions to minimize the false positive rate and preserve the membership test ability of the filter. For a detailed experimental evaluation of the influence of Bloom filter length, number of hash functions, and number of blocks, on estimation accuracy, refer to [28]. Our reimplementations, all datasets, and results are available on our repeatability page[3].

[1] `www.metanome.de`

[2] `https://sites.google.com/site/murmurhash/`

[3] `https://hpi.de/naumann/projects/repeatability/data-profiling.html`

Datasets. To benchmark the estimation accuracy and runtime of the considered F_0 estimation algorithms, we have run them over real-world datasets as well as synthetic datasets. The 90 synthetic datasets were generated by the Mersenne Twister random number generator [26]. For each specific cardinality, we generated ten independent datasets using different seeds for each of them and report their average runtime and estimation error. The exact cardinalities were made to be the powers of 10, starting with 10 up to 10^9. We always refer by *dataset cardinality* to the number of distinct values in the dataset, while the *dataset size* is the overall number of elements. Table 3 shows our real-world datasets, selected based on tuple count, i.e., how large the dataset size is, and the variety of columns cardinalities as illustrated in Figure 2. NCVoter is a collection of North Carolinas voter registration data[4]. We used the first 25 columns to perform experiments. The Openadresses dataset is a public database connecting the geographical coordinates with their postal addresses[5]. To avoid the possibly miss-leading results caused by NULL semantics, all NULL values are discarded by cardinality estimation algorithms and while determining datasets size and exact cardinality (Figure 2).

Table 3: Real-world dataset characteristics

Dataset	# Attributes	# Tuples
NCVoter	25 (of 71)	7,560,886
Openadresses-Europe	11	93,849,474

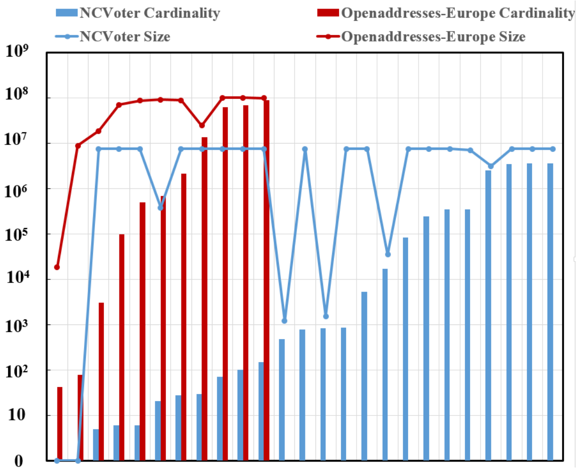


Figure 2: Exact cardinality range of the real-world datasets columns and corresponding column size.

Evaluation metrics. We allocate the same memory capacity to all algorithms and evaluate their performance regarding runtime and estimation accuracy. We use relative error as the measure of estimation accuracy as described in Section 2. The total time taken by an algorithm to process all the data elements and estimate F_0 is considered as its runtime. Runtime and relative error of the real-world datasets are averaged over ten runs using the same dataset. Runtimes and relative errors are averaged among ten synthetic datasets for each specific cardinality.

4.2 Sampling-based experiments

Sampling has inherent difficulty to accurately estimate the number of distinct values (Section 2.2). However, for the sake of completeness, we briefly evaluated *Guaranteed-Error Estimator(GEE)* [8] as an example of sampling-based cardinality estimator. GEE estimates the number of distinct value based on values frequency within values that have been sampled uniformly and randomly from a column or a dataset. We used *Reservoir sampling* without replacement. Table 4 shows the effect of different sampling rates on the average GEE estimation error. In order to produce a cardinality estimation with 1% relative error, GEE needs to sample more than 90% of the dataset. We also observed that when GEE is applied to datasets with duplicated values, the estimation error is less. GEE runtime noticeably increases with the size of the dataset, but only slightly with the sampling rate. The main drawback of GEE is its memory consumption. A GEE synopsis consists of both sampled values and their frequencies. GEE needs a minimum heap size of at least 13 GByte and 35 GByte to guarantee an estimation error below 1% on NCVoter and Openadress-Europe, respectively.

Table 4: GEE average estimation relative error vs. sampling rate

Dataset	Sampling rate				
	20%	40%	60%	80%	100%
Synthetic	0.54	0.43	0.4	0.2	0
NCVoter	0.26	0.19	0.17	0.07	0.00002
Openadresses	0.28	0.2	0.19	0.09	0.00001

4.3 Accuracy experiments

This section compares the accuracy of the twelve algorithms and how they scale with the exact cardinalities of the datasets. We limited the Java Virtual Machine (JVM) to 100 GB and ran each algorithm on each dataset with runtime limit of two hours. Figure 3 illustrates the change of the algorithms' relative error for each exact F_0 of the input dataset for synthetic and real datasets. The runtimes of this experiment set are depicted in Figure 4, and are discussed in the next section.

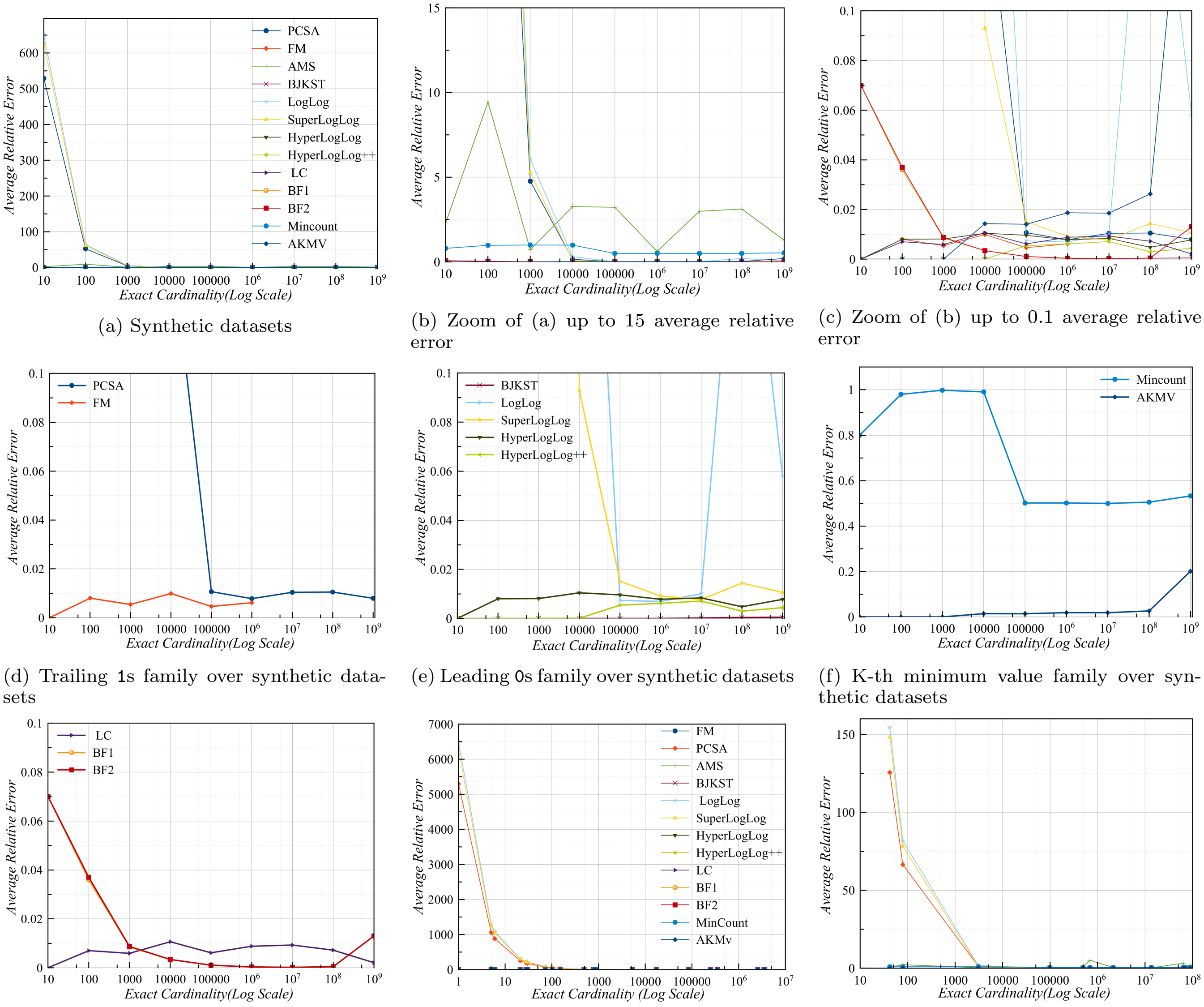
4.3.1 Accuracy comparison among all algorithms

For datasets with low number of distinct values (up to 1,000), all the logarithmic hashing algorithms that use stochastic averaging as a method for accuracy boosting, namely **PCSA**, **LogLog**, and **SuperLogLog**, extremely overestimate F_0. This effect is a consequence of stochastic average magnification of estimation by m. From Table 2 we can tell that as m increases, the upper bound of the standard error of the over-all algorithm decreases. The accuracy of PCSA, LogLog, and SuperLogLog increases up to this bound for larger cardinalities. Still, the bias of PCSA is slightly less than that of the others due to the use of trailing 1s of the hash value binary strings as observable. Figure 3a clearly shows this observation.

Overall, we see that all algorithms perform similarly well for larger cardinalities. However, taking a closer look, Figure 3b makes it clear that **AMS** and **MinCount** perform worse than the rest, even for large cardinalities. We are aware that AMS has a high variance and is presented as

[4] https://www.ncsbe.gov/data-statistics

[5] https://openaddresses.io/



(a) Synthetic datasets

(b) Zoom of (a) up to 15 average relative error

(c) Zoom of (b) up to 0.1 average relative error

(d) Trailing 1s family over synthetic datasets

(e) Leading 0s family over synthetic datasets

(f) K-th minimum value family over synthetic datasets

(g) Linear synopses family over synthetic datasets

(h) NCvoter

(i) Openaddresses-Europe

Figure 3: Average relative error of the twelve F_0 estimation algorithms and their families over 90 synthetic datasets and real-world datasets

a theoretical algorithm. Our measurements show this variance in practice and show how the LogLog algorithm solved this problem, at least for large cardinalities, using stochastic averaging.

As is clear from Figure 3c, **BJKST** outperformed all the other algorithms. The error guarantee of BJKST was even better than the theoretical lower bounds (i.e., relative error was always far less than 1%). The second best algorithm after BJKST was **Bloom filter** with error measures close to or equal to zero for most of the cardinality range. Noticeably, the relative error of Bloom filter is inversely proportional to the exact cardinality of the dataset. In other words, when the ratio of Bloom filter size over the dataset cardinality is relatively low, the estimation accuracy is improved until the point when the Bloom filter is full and the error rises again. Obviously, this accuracy comes at a cost: we analyze runtime in Section 4.4 and memory consumption in Section 4.5.

In the third place was **HyperLogLog++**. It maintained a good estimation accuracy with relative error below 0.008, regardless how many distinct values the dataset has. The bias correction implemented in HyperLogLog++ caused a tangible enhancement in estimating the cardinalities of datasets with small F_0. **FM** is a strong competitor of HyperLogLog++, but it exceeded the two hour runtime limit for datasets with $F_0 > 10^6$.

According to experiments in [27], **LC** was the most accurate F_0 algorithm, beating LogLog, SuperLogLog, FM, PCSA, MinCount, and BJKST. The authors studied the accuracy change with only much smaller datasets and with differing space usage, which is a different setting than in our experiments, which uses different cardinalities. In fact, in our setting, LC also beat LogLog, SuperLogLog, PCSA, and MinCount.

HyperLogLog provided a very good, and in particular stable estimation. Its use of LC to estimate small cardinalities explains the similarity of the behavior of HyperLogLog and LC for $F_0 < 10^6$. **AKMV**'s accuracy went down steadily with the increase of the dataset cardinality. In [7], the authors experimentally showed that AKMV is significantly more accurate than SuperLogLog. Their measurements went up to cardinalities of 10^7. In our experiments we observe that AKMV loses this advantage for cardinalities over 10^5 (but it remains to be more efficient than SuperLogLog). In contrast, PCSA, LogLog, and SuperLogLog performed better for high cardinalities than for lower ones.

We also tested the accuracy of the twelve F_0 estimation algorithms on real-world datasets, as shown in Figures 3h and 3i. A very poor performance can be observed for PCSA, LogLog, and SuperLogLog for columns with small cardinalities on both datasets. More than half of the columns of the NCVoter dataset, as well as only two of the Openaddresses-Europe dataset have cardinalities below 1,000 (Figure 2). So, overall we can draw the same conclusion as with our experiments on synthetic datasets.

For the remaining columns of NCVoter, MinCount had the highest average error around 0.5. All the other algorithms provided a very good accuracy with average relative error less than 0.02. The same observation is also valid on Openaddresses-Europe.

4.3.2 Accuracy comparison per algorithm family

Because accuracy covered a very wide range of values, it was not easy to compare all the algorithms together. To extend our discussion to another perspective, this section validates what we know about each family, discusses the advantages and disadvantages of its algorithms, and identifies the most accurate candidate for each algorithm family. Figures 3d, 3e, 3f, and 3g show our findings.

As shown in Fig. 3d, in the **trailing 1s** family, FM is more resilient to dataset cardinality than PCSA, but it is impractical respecting runtime as we show in the next section.

A remarkable variance concerning their accuracy change among the **leading 0s** family algorithms is illustrated in Figure 3e. The main disadvantage of this algorithm family is that its algorithms are sensitive to hash value outliers (LogLog substantiates this observation). Each algorithm enhances its accuracy by using a dedicated boosting method to combine results from m instances of the algorithm or/and correcting specific bias ranges experimentally. LogLog and SuperLogLog use the stochastic averaging method. HyperLogLog and HyperLogLog++ use harmonic means. SuperLogLog and HyperLogLog++ add also a specific bias correction tuned by experimental observations, which explains why they outperformed LogLog, and HyperLogLog, respectively. Furthermore, we can conclude that using the harmonic means method has an appreciable effect in reducing the impact of hash-value outliers and in boosting the overall estimation accuracy without adding a time overhead (Section 4.4). Interestingly, BJKST is the best member of this family, although it uses only a single instance of the algorithm to overcome the problem of outliers with added cost for maintaining samples of the original dataset.

AKMV from the **K-th minimum value** family is another example of an algorithm using only one instance defeating other algorithms that combine results of a multiple instances. MinCount is designed to use the stochastic averaging method to step-up its accuracy. Still, the fairly simple algorithm AKMV is more accurate than MinCount, as shown in Figure 3f.

As anticipated in the **Linear synopses** family, Bloom filter beats LC due to their adoption of multiple hash functions resulting in a lower hash collision rate (Figure 3g). An expected consequence is that LC is faster than Bloom filter as we show in the next section. Nevertheless, the previous knowledge of maximum F_0 in order to fairly tune these algorithms is the drawback of this family.

4.4 Runtime behavior experiments

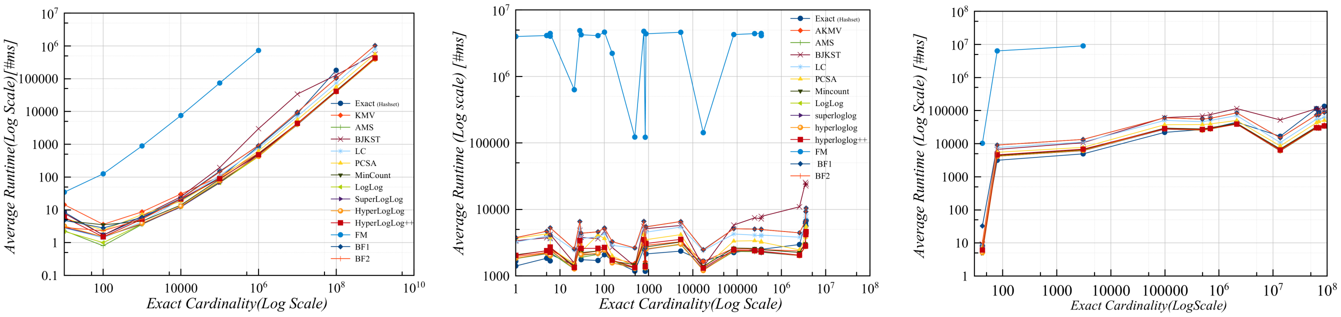
To compare the runtimes of the twelve algorithms, we recorded the runtime of the accuracy experiments in the previous section. We used a runtime limit of two hours. This period was not enough to count exactly the distinct values of the generated datasets with cardinalities above 10^9, nor for the FM algorithm to finish: despite its high accuracy, FM exceeded the time limit for datasets with F_0 over 10^6, both in synthetic and real-world datasets. Figure 4 represents in log scale the runtimes for real-world and synthetic datasets.

General speaking, for the synthetic datasets the runtimes of all algorithms scale quadratically with synthetic dataset size (equals its cardinality here), FM being the slowest. This observation is valid for the real-world datasets. All twelve algorithms apply hashing on every single element of the dataset, and hashing constitutes the majority of each algorithm runtime. Accordingly, the runtime mainly depends on the size of the dataset, not its cardinality. As the synthetic datasets' sizes are identical to their F_0, the correlation between dataset size and runtime is obvious. In contrast, real-world dataset columns consist of duplicated items and have different size within the same dataset due to removal of null values. For example, more than half of the elements in the tenth column of openaddresses-Europe are nulls. Despite the high F_0 value, this column has fewer elements than columns with a lower cardinality, explaining the dip at around 10^7. Figure 4b shows a similar behavior for the NCVoter dataset. Comparing it with NCVoter columns size in Figure 2, we notice the influance of column size of runtime of all algorithms.

In addition to dataset size, two factors have a significant impact on runtime: the number of used hash functions and the maintained data structures (synopsis type). The disadvantage of using m hash functions is noticeable in FM's runtime behavior. It is slower by a factor of two than all other algorithms, regardless of input dataset cardinality, but still follows the same influence of dataset size. The second slowest algorithm is BJKST due to the overhead of keeping samples of the dataset and using a second hash function in order to comprise its synopsis. Our measurements revealed that FM and BJKST were slower than counting the exact F_0 using a hash table, when the JVM heap size was limited to 100 GB (i.e., large memory budget).

LC hashes each element from the dataset once, so it is obviously faster than Bloom filter which uses several hash functions. Both LC and Bloom filter need large synopses, proportional to dataset size. Except FM, PCSA is the only logarithmic hashing algorithm that keeps track of all the observable values (i.e., $\rho(y)$), making it slightly slower.

In summary, all the rest of the algorithms ran in roughly



(a) Synthetic datasets

(b) NCvoter

(c) Openaddresses-Europe

Figure 4: Runtime behavior of the twelve cardinality estimation algorithms on synthetic and real-world datasets

the same time without a critical difference. HyperLogLog++, HyperLogLog, AKMV, and LC are the best algorithms in terms of accuracy and runtime using a large memory budget.

4.5 Memory consumption experiments

In the previous experiments, we discussed the accuracy and runtime behavior using a large memory budget (JVM was limited to 100 GB). It is a key requirement for cardinality estimation to efficiently use all available memory. In some cases, a dataset is so massive that count-distinct queries could not be run within available memory. For example, out-of-memory was the error of a non-negligible part of count-distinct queries in the PowerDrill system [24]. In other cases, such as stream processing, the available memory is typically orders of magnitude smaller than the input [16]. Therefore, we evaluated the memory efficiency of the twelve algorithms. We determined the heap size that was used by each algorithm to estimate F_0 of each dataset as precise and fast as with large memory. We kept a runtime limit of two hours and all algorithms configured to guarantee 1% estimation error. We report the minimum heap size of ten runs for each data point.

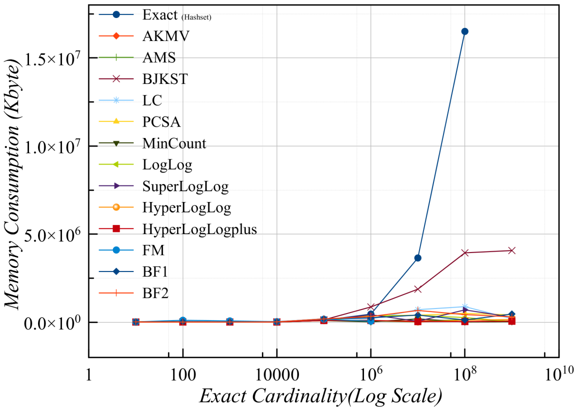


Figure 5: Memory consumption of the twelve cardinality estimation algorithms on real-world datasets.

Among all algorithms, only the hash table required a linear (and thus unacceptable) memory consumption. All others have a similar sub-linear behavior. Among those, BJKST has the highest constant factor. We ran the same experiments over real-world datasets and also observed an increase in the constant factor in correlation with dataset size.

5. CONCLUSION & OUTLOOK

Efficiently estimating the number of distinct values in a column, a dataset, or a stream is a widely studied problem. We reviewed and discussed twelve of the most important algorithms addressing this task. We have confirmed that some preliminary solutions, such as sampling and hash tables, are valid only when one can scale up the available computational resources. Both sampling and hash tables have the disadvantages of linear memory consumption and quadratic runtime with dataset size.

Our work has led us to conclude that none of the twelve estimation algorithms is clearly the best for all datasets and all scenarios. For a given accuracy, dataset size is obviously the main factor, affecting all the algorithms's runtime and memory consumption. We have categorized the algorithms into four families: Count trailing 1s, Count leading 0s, k-th minimum value, and Linear synopses. We showed that: FM, BJKST, AKMV, and Bloom filter are the best among their families, respectively. However, FM needs an extremely high runtime. BJKST and Bloom filter, on the other hand, have a high memory consumption. But AKMV survived for very large cardinalities with low memory consumption and runtime. For datasets with expected small cardinalities, PCSA, LogLog, SuperLogLog are not recommended due to their overestimation problem. Finally, HyperLogLog, AKMV, and LC are efficient over all cardinality ranges by all means.

This study has investigated only single-threaded implementations of the algorithms. However, several algorithms have characteristics that make them ready for parallelization and distributed environments. We can divide them into three categories: (1) Algorithms whose partial results can be easily merged, such as PCSA, AMS, and all their modifications. If the same hash function was used by all threads/nodes, bit-wise OR-operation among their bitmaps can lead to the same final bitmap of a single thread. (2) Algorithms running several copies of the same algorithm or use several hash functions to improve their accuracy, such as FM, MinCount, and Bloom filters. These can be distributed in a straightforward manner. (3) Algorithms allowing set operations like intersections or unions, such as AKMV. In conclusion, there is ample room for future work to evaluate parallel implementations of these algorithms.

6. REFERENCES

[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *VLDB Journal*, 24(4):557–581, 2015.

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

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 20–29, 1996.

[4] Z. Bar-Yossef, T. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.

[5] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[6] K. Beyer, R. Gemulla, P. J. Haas, B. Reinwald, and Y. Sismanis. Distinct-value synopses for multiset operations. *Communications of the ACM*, 52(10):87–95, 2009.

[7] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 199–210, 2007.

[8] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 268–279, 2000.

[9] G. Cormode. Count-Min sketch. In *Encyclopedia of Database Systems*, pages 511–516. Springer, 2009.

[10] 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.

[11] M. Durand and P. Flajolet. LogLog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003.

[12] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 153–166, 2003.

[13] P. Flajolet. On adaptive sampling. *Computing*, 43(4):391–400, 1990.

[14] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics and Theoretical Computer Science (DMTCS) Proceedings*, AH(1):127–146, 2008.

[15] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[16] M. Garofalakis, J. Gehrke, and R. Rastogi. *Data Stream Management: A Brave New World*, pages 1–9. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[17] P. Gibbons, C. Faloutsos, M. Faloutsos, C. Palmer, and G. Siganos. The connectivity and fault tolerance of the internet topology. In *Workshop on Network-Related Data Management; in cooperation with ACM Special Interest Group on Management of Data/Principles of Database Systems*, volume 25, 2001.

[18] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, volume 1, pages 541–550, 2001.

[19] P. B. Gibbons. *Data Stream Management: Processing High-Speed Data Streams*, chapter Distinct-values estimation over data streams. Springer, 2007.

[20] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, 2001.

[21] F. Giroire. Order statistics and estimating cardinalities of massive data sets. *Discrete Applied Mathematics*, 157(2):406–427, 2009.

[22] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, volume 95, pages 311–322, 1995.

[23] P. J. Haas and L. Stokes. Estimating the number of classes in a finite population. *Journal of the American Statistical Association*, 93(444):1475–1487, 1998.

[24] S. Heule, M. Nunkesser, and A. Hall. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 683–692, 2013.

[25] A. Kumar, J. Xu, and J. Wang. Space-code Bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications*, 24(12):2327–2339, 2006.

[26] M. Matsumoto and T. Nishimura. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[27] A. Metwally, D. Agrawal, and A. E. Abbadi. Why go logarithmic if we can go linear? Towards effective distinct counting of search traffic. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 618–629, 2008.

[28] O. Papapetrou, W. Siberski, and W. Nejdl. Cardinality estimation and dynamic length adaptation for Bloom filters. *Distributed and Parallel Databases*, 28(2):119–156, 2010.

[29] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data profiling with Metanome (demo). *PVLDB*, 8(12):1860–1871, 2015.

[30] S. A. Singh and S. Tirthapura. An evaluation of streaming algorithms for distinct counting over a sliding window. *Frontiers in ICT*, 2:23, 2015.

[31] S. J. Swamidass and P. Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3):952–964, 2007.

[32] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[33] K. Youssefi and E. Wong. Query processing in a relational database management system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 409–417, 1979.