Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters

Fan Deng
University of Alberta
fandeng@cs.ualberta.ca

Davood Rafiei
University of Alberta
drafiei@cs.ualberta.ca

ABSTRACT

Traditional duplicate elimination techniques are not applicable to many data stream applications. In general, precisely eliminating duplicates in an unbounded data stream is not feasible in many streaming scenarios. Therefore, we target at approximately eliminating duplicates in streaming environments given a limited space. Based on a well-known bitmap sketch, we introduce a data structure, Stable Bloom Filter, and a novel and simple algorithm. The basic idea is as follows: since there is no way to store the whole history of the stream, SBF continuously evicts the stale information so that SBF has room for those more recent elements. After finding some properties of SBF analytically, we show that a tight upper bound of false positive rates is guaranteed. In our empirical study, we compare SBF to alternative methods. The results show that our method is superior in terms of both accuracy and time efficiency when a fixed small space and an acceptable false positive rate are given.

1. INTRODUCTION

Eliminating duplicates is an important operation in traditional query processing, and many algorithms have been developed [20]. A common characteristic of these algorithms is the underlying assumption that the whole data set is stored and can be accessed if needed. Thus, multiple passes over data are possible, which is the case in a traditional database scenario. However, this assumption does not hold in a new class of applications, often referred to as data stream systems [3], which are becoming increasingly important. Consequently, detecting duplicates precisely is not always possible. Instead, it may suffice to identify duplicates with some errors.

While it is useful to have duplicate elimination in a Data Stream Management System (DSMS)[3], some new properties of these systems make the duplicate detection problem more challenging and to some degree different from the one in a traditional DBMS. First, the timely response property of data stream applications requires the system to respond in real-time. There is no choice but to store the data in limited main memory rather than in huge secondary storage. Sometimes even main memory is not fast enough. For example, for network traffic measurement and accounting, ordinary memory (DRAM) is too slow to process each IP packet in time, and fast memory (on-chip SRAM) is small and expensive [17, 5].

Second, the potentially unbounded property of data streams indicates that it is not possible to store the whole stream in a limited space. As a result, exact duplicate detection is infeasible in such data stream applications.

On the other hand, there are cases where efficiency is more important than accuracy, and therefore a quick answer with an allowable error rate is better than a precise one that is slow. Sometimes there is no way to have a precise answer at all. Therefore, load shedding is an important topic in data stream system research [28, 4]. Next, we provide some motivating examples.

1.1 Motivating Examples

URL Crawling. Search engines regularly crawl the Web to enlarge their collections of Web pages. Given the URL of a page, which is often extracted from the content of a crawled page, a search engine must probe its archive to find out if the URL is in the engine collection and if the fetching of the URL can be avoided [8, 21].

One way to solve the problem is to store all crawled URLs in main memory and search for a newly encountered URL in it. However, the set of URLs can be too large to fit in memory. Partially storing URLs in the secondary storage is also not viable because of the large volume of searches that is expected to be performed within limited time.

In practice, detecting duplicates precisely may not be indispensable. The consequence of an imprecise duplicate detection is that some already-crawled pages will be crawled again, or some new URLs which should be crawled are missed. The first kind of errors may lead the crawler to do some redundant crawling. This may not have a great influence on performance as long as the error rate is acceptable. For the second kind of errors, since a search engine can archive only a small portion of the entire web, a small miss rate is usually acceptable. In addition, if a missed URL refers to a high quality page, it is quite likely that the URL will be listed in the content of more than one crawled page, and there is

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...$5.00.

less chance of repeatedly missing it. The solution adopted by the Internet Archive crawler introduces the second kind of errors [21].

Selecting distinct IP addresses. In network monitoring and accounting, it is often important to understand the traffic and to identify the users on the network [22]. The following two queries, for example, may be interesting to network monitors: who are the users on the network within the past hour? Where do they go? The queries may be written as:

Select distinct Source/Destination IP
from IP_PacketsStream
Within Past 1 Hour

The result could be helpful for further analyzing the user profiles, interests and the network traffic. Because of the current high throughput of Internet routers and limited amount of fast memory, it is hard to capture per package information precisely. Often, sampling or buffering is used as a compromise, which is discussed in this paper.

Duplicate detection in click streams. Recently, Metwally et al. propose another application for approximate duplicate detection in a streaming environment [24]. In a Web advertising scenario, advertisers pay web site publishers for clicks on their advertisements. For the sake of profit, it is possible that a publisher fakes some clicks (using scripts), hence a third party, the advertising commissioner, has to detect those false clicks by monitoring duplicate user IDs. We discuss more about their work in the related work section.

1.2 Our Contributions

In this paper, we propose *Stable Bloom Filter(SBF)*, which extends and generalizes the regular Bloom filter, and accordingly, a novel algorithm which dynamically updates the sketch to represent recent data. We find and prove the stable properties of an SBF including stability, exponential convergence rate and monotonicity, based on which we show that using constant space, the chance of a false positive can be bounded to a constant independent of the stream size, and this constant is explicitly derived. Furthermore, we show that the processing time of SBF for each element in the stream is also constant independent of the stream size. To make our algorithm readily applicable in practice, we provide detailed discussions of the parameter setting issues both in theory and in experiments. And we compare our method to alternative methods using both real and synthetic data. The results show that our method is superior in terms of both accuracy and time efficiency when a fixed small space and an acceptable false positive rate are given.

1.3 Roadmap

In Section 2, we present the problem statement and some background on existing approaches. Our solution is presented and discussed in Section 3. In Section 4, we discuss how our algorithm can be used in practice. In Section 5, we verify our theoretical findings, experimentally evaluate our method and report the results of comparisons with alternative methods. Related work is reviewed in Section 6, and conclusions are given in Section 7.

2. PRELIMINARIES

This section presents the problem statement in a data stream model and some possible solutions.

2.1 Problem Statement

We consider a data stream as a sequence of numbers, denoted by $S_N = x_1, \ldots, x_i, \ldots, x_N$, where N is the size of the stream. The value of N can be infinite, which means that the stream is not bounded. In general, a stream can be a sequence of records, but it is not hard to transform each record to a number (e.g., using hashing or fingerprinting) and use this stream model.

Our problem can be stated as follows: given a data stream S_N and a certain amount of space, M, estimate whether each element x_i in S_N appears in $x_1, \ldots, x_{i-1}$ or not. Since our assumption is that M is not large enough to store all distinct elements in $x_1, \ldots, x_{i-1}$, there is no way to solve the problem precisely. Our goal is to approximate the answer and minimize the number of errors, including both false positives and false negatives , where a false positive is a distinct element wrongly reported as duplicate, and a false negative is a duplicate element wrongly reported as distinct.

To address this problem, we examine two techniques that have been previously used in different contexts.

2.2 The Buffering Method

A straightforward solution is to allocate a buffer and fill the buffer with enough elements of the stream. For each new element, the buffer can be checked, and the element may be identified as distinct if it is not found in the buffer, and duplicate otherwise. When the buffer is full, a newly arrived element may evict another element out of the buffer before it is stored. There are many replacement policies for choosing an element to be dropped out (e.g. [20]). Clearly, buffering introduces no false positives. We will use this method in our experiments and compare its performance to that of our method.

2.3 Bloom Filters

Bloom [7] proposes a synopsis structure, known as the Bloom filter, to approximately answer membership queries. A Bloom filter, BF, is a bit array of size m, all of which are initially set to 0. For each element, K bits in BF are set to 1 by a set of hash functions $\{h_1(x), \ldots, h_K(x)\}$, all of which are assumed to be uniformly independent. It is possible that one bit in BF is set multiple times, while only the first setting operation changes 0 into 1, and the rest have no effect on that bit. To know whether a newly arrived element x_i has been seen before, we can check the bits $\{h_1(x_i), \ldots, h_K(x_i)\}$. If any one of these bits is zero, with 100% confidence we know x_i is a distinct element. Otherwise, it is regarded as a duplicate with a certain probability of error. An error may occur because it is likely that the cells $\{h_1(x_i), \ldots, h_K(x_i)\}$ are set before by elements other than x_i.

The probability of a false positive (*false positive rate*) $FP = (1-p)^K$, where $p = (1-1/m)^{Kn}$ is the probability that a particular cell is still zero after seeing n distinct elements. It is shown that when the number of hash functions $K = ln(2)(m/n)$, this probability will be minimized to $(1/2)^{ln(2)(m/n)}$, where m is the number of bits in BF and n is the number of distinct elements seen so far [25].

3. STABLE BLOOM FILTERS

The Bloom filter is shown to be useful for representing the presence of a set of elements and answering membership queries, provided that a proper amount of space is allocated according to the number of distinct elements in the set.

3.1 The Challenge to Bloom Filters

However, in many data stream applications, the allocated space is rather small compared to the size of the stream. When more and more elements arrive, the fraction of zeros in the Bloom filter will decrease continuously, and the false positive rate will increase accordingly, finally reaching the limit, 1, where every distinct element will be reported as a duplicate, indicating that the Bloom filter is useless.

Our general solution is to avoid the state where the Bloom filter is full by evicting some information from it before the error rate reaches a predefined threshold. This is similar to the replacement operation in the buffering method, in which there are several possible policies for choosing a past element to drop. In many real world data stream applications, often the recent data is more important than the older data [13, 26]. However, for the regular Bloom filter, there is no way to distinguish the recent elements from the past ones, since no time information is kept. Accordingly, we add a random deletion operation into the Bloom filter so that it does not exceed its capacity in a data stream scenario.

3.2 Our Approach

To solve this problem, we introduce the *Stable Bloom Filter*, an extension of the regular Bloom filter.

Definition 1 (Stable Bloom Filter (SBF)). *An SBF is defined as an array of integer $SBF[1], \dots, SBF[m]$ whose minimum value is 0 and maximum value is Max. The update process follows Algorithm 1. Each element of the array is allocated d bits; the relation between Max and d is then $Max = 2^d - 1$. Compared to bits in a regular Bloom filter, each element of the SBF is called a* **cell**.

Concretely speaking, we change bits in the regular Bloom filter into cells, each consisting of one or more bits. The initial value of the cells is still zero. Each newly arrived element in the stream is mapped to K cells by some uniform and independent hash functions. As in a regular Bloom filter, we can check if a new element is duplicate or not by probing whether all the cells the element is hashed to are non-zero. This is the duplicate detection process.

After that, we need to update the SBF. We first randomly decrement P cells by 1 so as to make room for fresh elements; we then set the same K cells as in the detection process to Max. Our symbol list is shown in Table 1, and the detailed algorithm is described in Algorithm 1.

3.3 The Stable Property

Based on the algorithm, we find an important property of SBF both in theory and in experiments: after a number of iterations, the fraction of zeros in the SBF will become fixed no matter what parameters we set at the beginning.

Algorithm 1: Approximately Detect Duplicates using SBF

```
Data: A sequence of numbers $S = x_1, \dots, x_i, \dots, x_N$.
Result: A sequence of "Yes/No" corresponding to each
        input number.
begin
    initialize SBF[1]... SBF[m] = 0
    for each $x_i \in S$ do
        Probe $K$ cells $SBF[h_1(x_i)] \dots SBF[h_K(x_i)]$
        if none of the above $K$ cells is $0$ then
            DuplicateFlag = "Yes"
        else
            DuplicateFlag = "No"
        Select $P$ different cells uniformly at random
        $SBF[j_1] \dots SBF[j_P], P \in \{1, \dots, m\}$
        for each cell $SBF[j] \in \{SBF[j_1], \dots, SBF[j_P]\}$ do
            if $SBF[j] \geq 1$ then
                $SBF[j] = SBF[j] - 1$
        for each cell $\in \{SBF[h_1(x_i)], \dots, SBF[h_K(x_i)\}$ do
            $SBF[h(x_i)] = Max$
        Output DuplicateFlag
end
```

Table 1: The Symbol List

Symbols	Meanings
N	Number of elements in the input stream
M	Total space available in bits
m	Number of cells in the SBF
Max	The value a cell is set to
d	Number of bits allocated per cell
K	Number of hash functions
k	The probability that a cell is set in each iteration
P	Number of cells we pick to decrement by 1 in each iteration
p	The probability that a cell is picked to be decremented by 1 in each iteration
h_i	The *ith* hash function

We call this the *stable property* of SBF and deem it important to our problem because the false positive rate is dependent on the fraction of zeros in SBF.

Theorem 1. *Given an SBF with m cells, if in each iteration, a cell is decremented by 1 with a probability p and set to Max with a probability k, the probability that the cells becomes zero after N iterations is a constant , provided that N is large enough, i.e.*

$$\lim_{N \to \infty} Pr(SBF_N = 0)$$

exists, where SBF_N is the value of the cell at the end of iteration N .

In our formal discussion, we assume that the underlying distribution of the input data does not change over time.

Our experiments on the real world data show that this is not a very strong assumption, and the experimental results verify our theory.

Proof. Within each iteration, there are three operations: detecting duplicates, decreasing cell values and setting cells to Max. Since the first operation does not change the values of the cells, we just focus on the other two operations.

Within the process of iterations from 1 to N, the cell could be set $0, ..., (N-1)$ or even N times. Since the newest setting operation clears the impact of any previous operations, we can just focus on the process after the newest setting operation.

Let A_l denote the event that within the N iterations the most recent setting operation applied to the cell occurs at iteration $N-l$, which means that no setting happened within the most recent l iterations (i.e. from iteration $N-l+1$ to iteration N, $l < N$), and let $\bar{A}_N$ denotes the event that the cell has never been set within the whole N iterations. Hence, the probability that the cell is zero after N iterations is as follows:

$$\begin{aligned} Pr(SBF_N = 0) = &\sum_{l=Max}^{N-1} [Pr(SBF_N = 0 \mid A_l) Pr(A_l)] \\ &+ Pr(SBF_N = 0 \mid \bar{A}_N) Pr(\bar{A}_N), \end{aligned} \tag{1}$$

where

$$Pr(SBF_N = 0 \mid A_l) = \sum_{j=Max}^{l} \binom{l}{j} p^j \, (1-p)^{l-j} \tag{2}$$

$$Pr(A_l) = (1-k)^l k \tag{3}$$

$$Pr(SBF_N = 0 \mid \bar{A}_N) = 1 \tag{4}$$

$$Pr(\bar{A}_N) = (1-k)^N. \tag{5}$$

We have Eq. 2 because during those l iterations, there is no setting operation, and the cell becomes zero if and only if it is decremented by 1 no less than Max times. Clearly when $l < Max$, the cell is impossible to be decreased to 0, and $l = N$ means $\bar{A}_N$ happens, so we just consider the cases of $Max \leq l \leq (N-1)$ in Eq. 2. When $\bar{A}_N$ happens, the cell is 0 with a probability 1 because the initial value of the cell is 0 and it has never been set, therefore we have Eq. 4. Having the above equations, we can prove that $\lim_{N\to\infty} Pr(SBF_N = 0)$ exists. Due to the page limit, we put the detailed proof in an extended version of this paper. □

Having Theorem 1, now we can prove our stable property statement.

Corollary 1 (Stable Property). *The expected fraction of zeros in an SBF after N iterations is a constant, provided that N is large enough.*

Proof. In each iteration, each cell of the SBF has a certain probability of being set to Max by the element hashed to that cell. Since the underlying distribution of the input data does not change, the probability that a particular element appears in each iteration is fixed. Therefore, the probability of each cell being set is fixed.

Meanwhile, the probability that an arbitrary cell is decremented by 1 is also a constant. According to Theorem 1, the probabilities of all cells in the SBF becoming 0 after N iterations are constants, provided that N is large enough. Therefore, the expected fraction of 0 in an SBF after N iterations is a constant, provided that N is large enough. □

Now we know an SBF converges. In fact this convergence is like the process that a buffer is filled by items continually. SBF is stable means that its maximum capacity is reached, similar to the case that a buffer is full of items. Another important property is the convergence rate.

Corollary 2 (Convergence Rate). *The expected fraction of 0s in the SBF converges at an exponential rate.*

Proof. From Eq. 1, 4 and 5, we can derive

$$\begin{aligned} &Pr(SBF_N = 0) - Pr(SBF_{N-1} = 0) \\ =&Pr(SBF_N = 0 \mid A_{N-1}) Pr(A_{N-1}) \\ &+ Pr(\bar{A}_N) - Pr(\bar{A}_{N-1}) \\ =&k(1-k)^{N-1}(Pr(SBF_N = 0 \mid A_{N-1}) - 1) \end{aligned} \tag{6}$$

Clearly, Eq. 6 exponentially converges to 0. i.e. $Pr(SBF_N[c] = 0)$ converges at an exponential rate, and this is true for all cells in the SBF. Therefore, the expected fraction of 0s in the SBF converges at an exponential rate. □

Lemma 1 (Monotonicity). *The expected fraction of 0s in an SBF is monotonically non-increasing.*

Proof. Since the value of Eq. 6 is always no greater than 0, the probability that a cell becomes zero is always decreasing or remains the same. Similar to the proof of Corollary 2, we can draw the conclusion. □

This lemma will be used to prove our general upper bound of the false positive rate where the number of iterations needs not to be infinity.

3.4 The Stable Point

Currently we know the fraction of 0s in an SBF will be a constant at some point, but we do not know the value of this constant. We call this constant the stable point.

Definition 2 (Stable Point). *The stable point is defined as the limit of the expected fraction of 0s in an SBF when the number of iterations goes to infinity. When this limit is reached, we call SBF stable.*

From Eq. 1, we are unable to obtain the limit directly. However, we can derive it indirectly.

Theorem 2. *Given an SBF with m cells, if a cell is decremented by 1 with a constant probability p and set to Max with a constant probability k in each iteration, and if the probability that the cell becomes 0 at the end of iteration N is denoted by $Pr(SBF_N = 0)$,*

$$\lim_{N\to\infty} Pr(SBF_N = 0) = (\frac{1}{1+\frac{1}{p(1/k-1)}})^{Max} \tag{7}$$

Proof. The basic idea is to make use of the fact that SBF is stable, the expected fraction of 0,1,...,Max in SBF should be all constant. See the full paper for the details. □

The theorem can be verified by replacing the parameters in Eq. 1 with some testing values.

From Theorem 2 we know the probability that a cell becomes 0 when SBF is stable. If all cells have the same probability of being set, we can obtain the stable point easily. However, that requires the data stream to be uniformly distributed. Without this uniform distribution assumption, we have the following statement.

Theorem 3 (SBF Stable Point). *When an SBF is stable, the expected fraction of 0s in the SBF is no less than*

$$(\frac{1}{1+\frac{1}{P(1/K-1/m)}})^{Max},$$

where K is the number of cells being set to Max and P is the number of cells decremented by 1 within each iteration.

Proof. The basic idea is to prove the case of $m = 2$ first, and generalize it to $m \geq 2$. See the full paper for the details. □

3.5 False Positive Rates

In our method, there could be two kinds of errors: false positives (FP) and false negatives (FN). A false positive happens when a distinct element is wrongly reported as duplicate; a false negative happens when a duplicate element is wrongly reported as distinct. We call their probabilities *false positive rates* and *false negative rates*.

Corollary 3 (FP Bound when Stable). *When an SBF is stable, the FP rate is a constant no greater than FPS,*

$$FPS = (1 - (\frac{1}{1+\frac{1}{P(1/K-1/m)}})^{Max})^K. \tag{8}$$

Proof. If $Pr_j(0)$ denotes the probability that the cell $SBF[j] = 0$ when the SBF is stable, the FP rate is

$$(\frac{1}{m}(1 - Pr_1(0)) + \cdots + \frac{1}{m}(1 - Pr_m(0))^K$$
$$= (1 - \frac{1}{m}(Pr_1(0) + \cdots + Pr_m(0)))^K$$

Please note that $\frac{1}{m}(Pr_1(0) + \cdots + Pr_m(0))$ is the expected fraction of 0s in the SBF. According to Theorems 1 and 3, the FP rate is a constant and Eq. 8 is an upper bound of the FP rate. □

This upper bound can be reached when the stream elements are uniformly distributed.

Corollary 4 (The case of reaching the FP Bound). *Given an SBF with m cells, if the stream elements are uniformly distributed when the SBF is stable, the FP rate is FPS (Eq. 8).*

Proof. Because elements in the input data stream are uniformly distributed, each cell in the SBF will have the same probability to be set to Max. According to Theorem 1 and the proof of Theorem 3 we can derive this statement. □

Corollary 5 (General FP Bound). *Given an SBF with m cells, FPS (Eq. 8) is an upper bound for FP rates at all time points, i.e. before and after the SBF becomes stable.*

Proof. This can be easily derived from Lemma 1 and Corollary 3. □

Therefore, the upper bound for FP rates is valid no matter the SBF is stable or not.

From Eq. 8 we can see that m has little impact on FPS, since $1/m$ is negligible compared to $1/K$ ($m \gg K$). This means the amount of space has little impact on the FP bound once the other parameters are fixed. The value of P has a direct impact on FPS: the larger the value of P, the smaller the value of FPS. This can be seen intuitively: the faster the cells are cleared, the more 0s the SBF has, thus the smaller the value of FPS is. Oppositely, increasing the value of Max results in the increase of FPS. In contrast to P and Max, from the formula we can see the impact of the value of K on FPS is twofold: intuitively, using more hash functions increases the distinguishing power for duplicates (decreases FPS), but "fills" the SBF faster (increases FPS).

3.6 False Negative Rates

A false negative(FN) is an error when a duplicate element is wrongly reported as distinct. It is generated only by duplicate elements, and is related to the input data distribution, especially the distribution of gaps. A gap is the number of elements between a duplicate and its nearest predecessor.

Suppose a duplicate element x_i whose nearest predecessor is $x_{i-\delta_i} (x_i = x_{i-\delta_i})$ is hashed to K cells, $SBF[C_{i1}] \ldots SBF[C_{iK}]$. An FN happens if any of those K cells is decremented to 0 during the δ_i iterations when x_i arrives. Let $PR0(\delta_i, k_{ij})$ be the probability that cell C_{ij} ($j = 1 \ldots K$) is decremented to 0 within the δ_i iterations. This probability can be computed as in Eq. 1:

$$\begin{aligned} PR0(\delta_i, k_{ij}) = \sum_{l=Max}^{\delta_i - 1} [Pr(SBF_{\delta_i} = 0 \mid A_l) Pr(A_l)] \\ + Pr(SBF_{\delta_i} = 0 \mid \bar{A}_{\delta_i}) Pr(\bar{A}_{\delta_i}), \end{aligned} \tag{9}$$

where

$$Pr(SBF_{\delta_i} = 0 \mid A_l) = \sum_{j=Max}^{l} \binom{l}{j} p^j (1-p)^{l-j}, \tag{10}$$

$$Pr(A_l) = (1 - k_{ij})^l k_{ij}, \tag{11}$$

$$Pr(SBF_{\delta_i} = 0 \mid \bar{A}_{\delta_i}) = \sum_{j=Max}^{\delta_i} \binom{\delta_i}{j} p^j \, (1-p)^{\delta_i - j} \,, \tag{12}$$

$$Pr(\bar{A}_{\delta_i}) = (1 - k_{ij})^{\delta_i}, \tag{13}$$

and k_{ij} is the probability that cell C_{ij} is set to Max in each iteration. The meanings of the other symbols are the same as those in the proof of Theorem 1. Also, most of above equations are similar, except that Eq. 12 is different from Eq. 4. This is because the initial value of the cell in the case of Theorem 1 is 0, but it is Max here.

Furthermore, The probability that an FN occurs when x_i arrives can be expressed as follows:

$$Pr(FN_i) = 1 - \prod_{j=1}^{K} (1 - PR0(\delta_i, k_{ij})). \tag{14}$$

When $\delta_i < Max$, $PR0(\delta_i, k_{ij})$ is 0, which means the FN rate is 0. Besides, for distinct elements who have no predecessors, the FN rates are 0. The value of δ_i depends on the input data stream. In the next section, we discuss how to adjust the parameters to minimize the FN rate under the condition that the FP rate is bounded within a user-specified threshold.

4. FROM THEORY TO PRACTICE

In the previous section we propose the SBF method and analytically study some of its properties: stability, convergence rate, monotonicity, stable point, FP rates (upper bound) and FN rates. In this section, we discuss how SBF can be used in practice and how our analytical results can be applied.

4.1 Parameters Setting

Since FP rates can be bounded regardless of the input data but FN rates cannot, given a fix amount of space, we can choose a combination of Max, K and P that minimizes the number of FNs under the condition that the FP rate is within a user-specified threshold. Meanwhile we take into account the time spent on each element, which is crucial in many data stream applications.

Overview of parameters setting. We have 3 parameters to set in our algorithm: Max, K and P. They are related to other parameters: FP rates, FN rates, SBF size m. Among these parameters, we assume that users specify m and the allowable FP rate. Based on the analysis in the previous section, we can obtain a formula computing the value of P from other parameters provided that Max and K have been chosen already. To set Max and K properly, we first derive the relationship between FN rates (the expected number of FNs), other parameters and the input data distribution. We find that the optimal value of K which minimizes the FN rates is independent of the input data. Thus, K can be set by trying different possible values on the formula we derive without considering the input data distribution, assuming Max is known. Last, we show that Max can be set empirically.

The expected number of FNs. Since our goal is to minimize the number of FNs, we can compute the expected number of FNs, $E(\#FN)$, as the sum of FN rates for each duplicate element in the stream: $E(\#FN) = \sum_{i=1}^{\widetilde{N}} Pr(FN_i)$, where $\widetilde{N}$ is the number of duplicates in the stream. Combining it with Eq. 14 we have

$$E(\#FN) = \sum_{i=1}^{\widetilde{N}} [1 - \prod_{j=1}^{K} (1 - PR0(\delta_i, k_{ij}))], \tag{15}$$

where δ_i is the number of elements between x_i and its predecessor, and k_{ij} is the probability that cell C_{ij} is set to Max in each iteration. C_{ij} is the cell element x_i is hashed to by the jth hash function. Since the function $PR0(\delta, k)$ is continuous, for each x_i there must be a $\bar{k}_i$ such that

$$(1 - PR0(\delta_i, \bar{k}_i))^K = \prod_{j=1}^{K} (1 - PR0(\delta_i, k_{ij})).$$

For the same reason, there must be an "average" $\widehat{\delta}$ and an "average" $\widehat{k}$ such that

$$\widetilde{N}[1 - (1 - PR0(\widehat{\delta}, \widehat{k}))^K] = \sum_{i=1}^{\widetilde{N}} [1 - (1 - PR0(\delta_i, \bar{k}_i))^K] = E(\#FN).$$

Let $f(\widehat{\delta}, \widehat{k})$ be the average FN rate, i.e.

$$f(\widehat{\delta}, \widehat{k}) = 1 - (1 - PR0(\widehat{\delta}, \widehat{k}))^K. \tag{16}$$

Our task then becomes setting the parameters to minimize this average FN rate, $f(\widehat{\delta}, \widehat{k})$, while bounding the FP rate within an acceptable threshold.

The setting of P. Suppose users specify a threshold FPS, indicating the acceptable FP rate. This threshold establishes a constraint between the parameters: Max, K, P, m and FPS according to Corollary 5. Thus, users can set P based on the other parameters:

$$P = \frac{1}{(\frac{1}{(1 - FPS^{1/K})^{1/Max}} - 1)(1/K - 1/m)}. \tag{17}$$

Since m is usually much larger than K, $1/m$ is negligible in the above equation, which means that the setting of P is dominated only by FPS, Max, K, and is independent of the amount of space.

The setting of K. Since the FP constraint can be satisfied by properly choosing P , we can set K such that it minimizes the number of FNs. From the above discussions we know the relationship between the expected number of FNs and the probabilities that cells are set to Max. Next, we connect these probabilities with our parameters K, m and the input stream.

Suppose there are N elements in the stream of which n are distinct, and the frequency for each distinct element x_l is f'_l. Clearly $\sum_{l=1}^{n} f'_l = N$. Assuming that the hash functions are uniformly at random, for a cell that element x_i is hashed to, the number of times the cell is set to Max after seeing all N elements is a random variable, $f_i + \sum_{l=1}^{n-1} f'_l I_l$, where f_i is the frequency of x_i in the stream, and each $I_l (l = 1 \ldots n-1)$ is an independent random variable following the Bernoulli

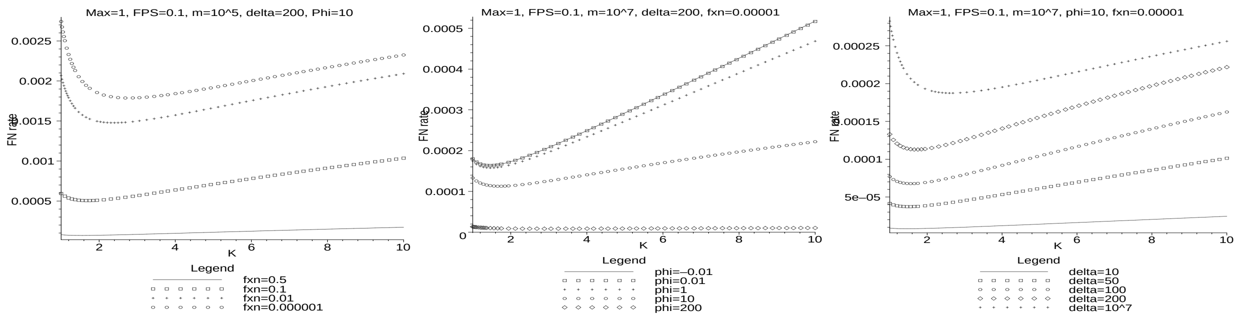


Figure 1: FN rates vs. K

distribution, i.e.

$$I_l = \begin{cases} 1, & Pr(I_l = 1) = \frac{K}{m}, \\ 0, & Pr(I_l = 0) = 1 - \frac{K}{m}. \end{cases}$$

Thus, $k_i = \frac{1}{N} f_i + \frac{1}{N}\sum_{l=1}^{n-1} f'_l I_l$ is also a random variable. For the K cells an element x_i is hashed to, the probabilities that those cells are set to Max in each iteration can be considered as K trials of k_i. Since the mean and the variance of each I_l are $\mu_{I_l} = \frac{K}{m}$ and $\sigma^2_{I_l} = \frac{K}{m}(1 - \frac{K}{m})$ respectively, it is not hard to derive that the mean and variance of k_i: $\mu_{k_i} = \frac{1}{N} f_i + \frac{1}{N}\frac{K}{m}\sum_{l=1}^{n-1} f'_l = \frac{1}{N} f_i + \frac{K}{m}(1 - \frac{1}{N} f_i)$ and $\sigma^2_{k_i} = \frac{1}{N^2}\frac{K}{m}(1 - \frac{K}{m})\sum_{l=1}^{n-1} f'^2_l$. Let

$$\phi_i = \frac{k_i - \mu_{k_i}}{\sqrt{\frac{K}{m}(1 - \frac{K}{m})}} = \frac{k_i - \frac{1}{N} f_i - \frac{K}{m}(1 - \frac{1}{N} f_i)}{\sqrt{\frac{K}{m}(1 - \frac{K}{m})}} \qquad (18)$$

be a transformation on k_i. Then $\phi_i \in [-\frac{\frac{1}{N} f_i + \frac{K}{m}(1 - \frac{1}{N} f_i)}{\sqrt{\frac{K}{m}(1 - \frac{K}{m})}}, \frac{1 - \frac{1}{N} f_i - \frac{K}{m}(1 - \frac{1}{N} f_i)}{\sqrt{\frac{K}{m}(1 - \frac{K}{m})}}]$ is a random variable whose mean and variance are: $\mu_{\phi_i} = 0$ and $\sigma^2_{\phi_i} = \frac{1}{N^2}\sum_{l=1}^{n-1} f'^2_l$. Note that $\sigma^2_{\phi_i} \leq \frac{1}{N^2}\sum_{l=1}^{n-1}(f'_l f'_{max}) < \frac{1}{N^2}\sum_{l=1}^{n}(f'_l f'_{max}) = \frac{f'_{max}}{N}$, where f'_{max} is the frequency of the most frequent element in the stream. Since the mean and the variance of the random variable ϕ_i are independent of K and m, we may consider ϕ_i independent of K and m in practice. In other words, ϕ_i can be seen as a property of the input stream. Similar to $\bar{k}_i$ we can obtain a $\bar{\phi}_i$ such that

$$1-(1-PR0(\delta_i, \bar{\phi}_i))^K = 1-\prod_{j=1}^{K}(1-PR0(\delta_i, \phi_{ij})) = Pr(FN_i) \qquad (19)$$

where $\bar{\phi}_i \in [Min(\phi_{ij}), Max(\phi_{ij})]$, and ϕ_{ij} are K trials of $\phi_i (j = 1 \ldots K)$. Since the standard deviation of ϕ_i is very small compared to the range of its possible values, and ϕ_i is considered independent of K and m, $\bar{\phi}_i$ can be approximately considered independent of K and m as well. For example, when $\frac{f'_{max}}{N} = 0.01$, $\frac{f_i}{N} = \frac{1}{10^6}$, $\frac{m}{K} = 10^6$, the value range of ϕ_i is approximately $[0, 1000]$, while $\sigma_{\phi_i} \leq 0.1$.

To set K, keeping all other parameters fixed we vary the values of K and compute the FN rate based on Eq. 19, 9, 17 and 18. By trying different combinations of parameter settings ($Max = 1, 3, 7, 15$, $FPS = 0.2, 0.1, 0.01, 0.001$, $m = 1000, 10^5, 10^7, 10^9$, $\delta_i = 10, 100, 1000, 10^5, 10^7, 10^9$, $fxn = \frac{f_i}{N} = 0.5, 0.1, 0.01, 0.0001, 0.000001$ and $\bar{\phi}_i = 0.001, 0.1, 1, 10, 100, 1000, \ldots$), we find that once the values of FPS and Max are fixed, the value of the optimal or near optimal K is independent of the values of δ_i, f_i/N, $\bar{\phi}_i$ and m.

OBSERVATION 1. *The value of the optimal or near optimal K is dominated by Max and FPS. The input data and the amount of the space have little impact on it. Furthermore, the value is small (<10 in all of our testing).*

For example, when $FPS = 0.2$ and $Max = 1$, the value of the optimal or near optimal K is always between 1 and 2; when $FPS = 0.1$ and $Max = 3$, it is always between 2 and 3; when $FPS = 0.01$ and $Max = 3$, it is always between 4 and 5. Therefore, without considering the input data stream we can pre-compute the FN rates for different values of K based on Max and FPS and choose the optimal one. Our experimental results reported in the next section are consistent with this observation.

Figure 1 shows an example of how the FN rates change with different values of K under different parameter settings based on Eq. 19 and Eq. 9. From the figure we can see that in the case of $Max = 1$ and $FPS = 0.1$, we can set K to 2 regardless of the input stream and the amount of space. Therefore, in practice we can set ϕ_i, δ_i and f_i/N to some testing values (e.g. $0, 200, 0.00001$ respectively) and find the optimal or near optimal K using the formulas.

The setting of Max. Based on the above discussion, we can set K regardless of the input data, but to choose a proper value of Max, we need to consider the input. More specifically, to minimize the expected number of FNs, we need to know the distributions of gaps in the stream to try different possible values of Max on Eq. 16 and 9. Since the expected value of ϕ_i is 0 and its standard deviation is very small compared to its value domain, we set $\widehat{\phi}$ to 0 in the formulas.

To effectively use the space we only set Max to $2^d - 1$ (d is the number of bits/cell), otherwise the bits allocated for each cells are just wasted. Furthermore, in terms of the time cost, Max should be set as small as possible, because the larger Max is set, the larger P will be (see Eq.17, assuming K is a constant). For example, when $Max = 1$, $FPS = 0.01$ and $K = 3$(the optimal K), the computed value of P is 10; while $Max = 15$, $FPS = 0.01$, and $K = 6$ (the optimal K), the value of P computed is 141 (the value of P is not sensitive to m). In practice, we limit our choice of Max to 1, 3 and 7 (if higher time cost can be tolerated, larger values of Max can be tried similarly).

To choose a Max from these candidates, we try each value on Eq. 16 and Eq. 9, and find the one minimizing the average FN rate.

Figure 2 depicts the difference of average FN rates between $Max = 3$ and $Max = 1$ based on Eq.16 and Eq.9. We set $\frac{f_i}{N} = 0$ because we are considering the entire stream rather than a particular element in this case. The figure shows that if the values of gaps($\widehat{\delta}$) are smaller than a threshold, $Max = 3$ is a better choice. When the gaps become larger, $Max = 1$ is better. If the gaps are large enough, there is not much difference between the two options. The figure shows the cases under different settings of $\widehat{\phi}$, space sizes and acceptable FP rates.

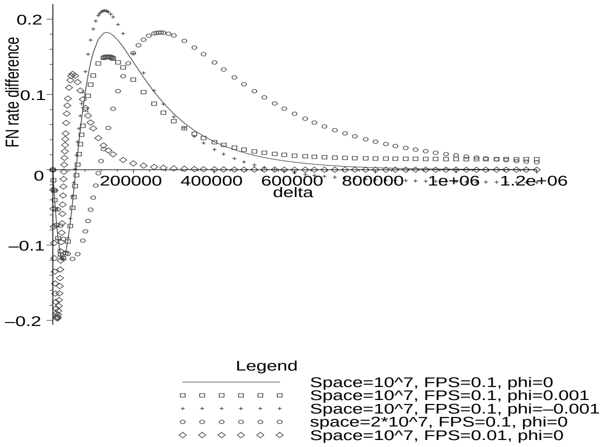


Figure 2: FN rates difference between $Max = 3$ and $Max = 1$ ($Max3 - Max1$) vs. gaps. K is set to the optimal value respectively under different settings.

We also tested the FN rate difference between $Max = 7$ and $Max = 3$, and observe the same general rule: a larger value of Max is better for smaller gaps, and a smaller FPS suggests a larger setting of Max. Similarly, we find no exceptions under other combinations of settings: $FPS = 0.2, 0.1, 0.01, 0.001$ and $m = 1000, 10^5, 10^7, 10^9$.

Trying different value of Max on Eq. 16 and Eq. 9, we set $\widehat{\phi}$ to 0 and assume that the distribution of the gaps are known. If the assumption cannot be satisfied in practice, we suggest setting Max to 1, because this setting often benefits a larger range of gaps in the stream. And our experiments also show that in most cases setting Max to 1 achieves better improvements in terms of error rates compared to the alternative method, LRU buffering. In fact, buffering performs well when gaps are small, which is similar to the cases that Max is larger. The behavior of our SBF becomes closer to the buffering method when the value of Max is set larger.

Summary of parameters setting. In practice, given an FPS, the amount of available space and the gap distribution of the input data, to set the parameters properly, we first establish a constraint for P, which means P can be computed based on FPS, m, Max and K; then find the optimal values of K for each case of $Max(1, 3, 7)$ by trying limited number(≤ 10) of values of K on the FN rate formulas; Last, we estimate the expected number of FNs for each candidate value of Max using its corresponding optimal K and some prior knowledge of the stream, and thus choose the optimal value of Max. In the case that no prior knowledge of the input data is available, we suggest setting $Max = 1$. The described parameter setting process can be implemented within a few lines of codes.

4.2 Time Complexity

Since our goal is to minimize the error rates given a fixed amount of space and an acceptable FP rate, we do not discuss space complexity, and just focus on time complexity. There are several parameters to be set in our method: K, Max and P. Within each iteration, we firstly need to probe K cells to detect duplicates. After that we pick P cells and decrement 1 from them. Last we set the same K cells as probed in the first step to Max.

Therefore, the time cost of our algorithm for handling each element is dominated by K and P.

Theorem 4 (Time Complexity). *Given that K and Max are constants, processing each data stream element needs O(1) time, independent of the size of the space and the stream.*

Proof. From Eq.17 we know the constraint among K, P, m, Max and FPS(the user-specified upper bound of false positive rates). If K, Max and FPS are constants, the relationship between P and m is inversely proportional, which means m has no impact on the processing time. Since Max, K and FPS are all constants, the time complexity is O(1). □

Based on the discussion of parameter settings, we know that the selection of K is insensitive to m. Furthermore, the value of m and the stream size have little impact on the selection of Max based on our testing on Eq. 16. Therefore, our algorithm needs $O(1)$ time per element, independent of the size of space.

5. EXPERIMENTS

In this section, we first describe our data set and the implementation details of 4 methods: SBF, Bloom Filter(BF), Buffering and FPBuffering (a variation of buffering which can be fairly compared to SBF). We then report some of the results on real data sets. We also ran experiments on some synthetic data sets, but due to the page limit, we can not show the results here. Last, we summarize the comparison between different methods.

5.1 Data Sets

Real World Data. We simulated a web crawling scenario[8] as discussed in Section 1, using a Web crawl data set obtained from the Internet archive[2]. We hashed each URL in this collection to a 64-bit fingerprint using Rabin's method [27], as was done earlier [8]. With this fingerprinting technique, there is a very small chance that two different

URLs are mapped to the same fingerprint. We verified the data set and did not find any collisions between the URLs. In the end, we obtained a 28GB data file that contained about 700 million fingerprints of links, representing a stream of URLs encountered in a Web crawling process.

5.2 Implementation Issues

SBF Implementation. Our algorithm is simple and straightforward to implement: 1) hash each incoming stream element into K numbers, and check the corresponding K cells; 2) generate a random number, decrement the corresponding cell and (P-1) cells adjacent to it by 1; this process is faster than generating P random numbers for each element; although the processes of picking the P cells are not independent, each cell has a probability of P/m for being picked at each iteration. Our analysis still holds. 3) Set those K cells checked in step 1 to Max. One issue we have to deal with is setting the parameters Max, K and P.

According to the discussions of parameters setting in the previous section, we can set Max, K and P for a given FPS without considering the input data sets. For example, for FPS=10%, we set Max=1, K=2 and P=4, which worked well for different data sets in our experiments. To evaluate our work, we implemented 3 alternative methods: Bloom Filters(BF), buffering and FPBuffering.

Bloom Filters Implementation. In our implementation, BF becomes a special case of SBF where Max=1 and P=0. Knowing the number of distinct elements and the amount of space, we can compute the optimal K (see the discussion in Section 2.3).

Buffering Implementation. Implementing buffering needs more work. First, to detect duplicates we need to search the buffer. To speed up the searching process, we used a hash table, as was done by Broder et al. [8]. Second, when the buffer is full, we have to choose a policy to evict an old element and make room for the newly coming one. Broder et al. [8] compared 5 replacement policies for caching Web crawls. They showed that LRU and Clock, the latter of which is used as an approximation of LRU, were the best practical choices for the URL data set (there were some ideal but impractical ones as well); in terms of miss rate (FN rate in our case), there was almost no difference between these two though. We chose LRU in our experiments. Both LRU and clock need a separate data structure for buffering elements, so that we can choose one for eviction [8]. For simplicity of the implementation, we used a double linked list, while Broder et al. chose a heap. This difference should not affect our experimental results since our error rate comparison did not account for the extra space we used in buffering.

FPbuffering Implementation. To fairly and effectively compare our method to buffering method, we introduced a variation of buffering called FPbuffering. There are two reasons for this. First, SBF has both FPs and FNs while buffering has only FNs. In different applications the importance of FPs and FNs may be different. So it is hard to compare SBF to buffering directly. Second, the fraction of duplicates in the data stream is a dominant factor affecting the error rates, because FNs are only generated by duplicates and FPs by distincts. For buffering, a data stream full of duplicates will cause many FNs, while a stream consisting of all distincts cause no errors at all.

FPbuffering works as follows: when a new data stream element arrives, we search it in the buffer. If found, report duplicate as in the original buffering; if not found, we report it as a duplicate with a probability q, and as a distinct with probability $(1 - q)$. In the original buffering, if an element is not found in the buffer, it is always reported as a distinct. This variation can increase the overall error rates of buffering when there are more distincts in the stream, but can decrease the error rates when there are more duplicates in the stream. Clearly, FPbuffering has both FPs and FNs. In fact, q is the FP rate since a distinct element will be reported as duplicate with a probability q. By setting a common FP rate with SBF, we can fairly compare their FN rates, and this comparison will not be affected by the fraction of duplicates in the stream.

In our experiments, we assumed that buffering and FPbuffering required 64 bits per URL fingerprint on the Web data (same as [8]). and 32 bits per element on the synthetic data simulating the size of an IP address. In other words, each element occupies 64 bits for the real data. and 32 bits for the synthetic data.

5.3 Theory Verification

In an experiment to verify some of our theoretical results, we tested the stable properties of our SBF and the convergence rate. The results are shown in Figures 3. From the graph we can see that the fraction of zeros in the SBF decreases until it becomes stable. When the allocated space is small, the convergence rate is higher. This is because when the space is larger, the probability a cell being set is smaller. From Corollary 2 and Eq. 6 we know the convergence rate should be lower in this case. Also, we can see that when the SBF is stable, the fraction of zeros is still fluctuating slightly. This can be caused by the input data stream whose underlying distributions is varying. Furthermore, the fraction of 0s keeps decreasing in general before being stable; at this point, the FP rate should reach its maximum, and our theoretical upper bound for FP rates is also valid before the SBF become stable in this case. Our next experiments show the effectiveness of our theoretical FP bound. When the space is relatively small, the real FP rate is close to the bound.

5.4 Error Rates Comparison

This experiment compared the error rates between SBF, FPbuffering, buffering and BF on the real data by varying the size of the space. The real data set contained 694,984,445 URL fingerprints, of which 14.75% were distinct. To do the comparison under different fractions of distinct elements, we built two more real data sets by using the first 100,000 and 10 million elements of the original data file. The fractions of distinct elements for these two data set respectively were 75.66% and 48.51%. For SBF, we set the acceptable FP rate (number of FPs/number of distincts), FPS, to 10%, and Max, K,P to 1, 2, 4 respectively. The results under different FPS settings will be shown in the next experiment. For FPbuffering, we set the FP rates to the same number as SBF so that both generated exactly the same number of FPs, and we can just compare their FN rates. Please note that buffering and BF only generate FNs and FPs respec-

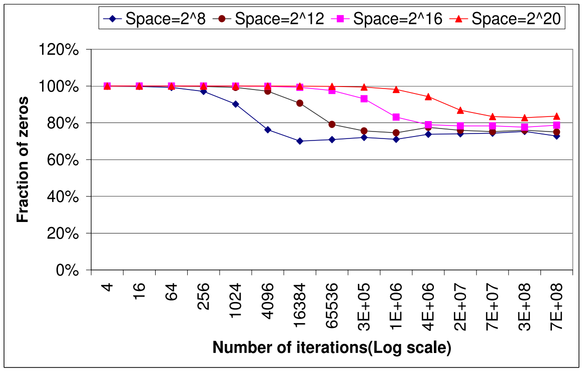
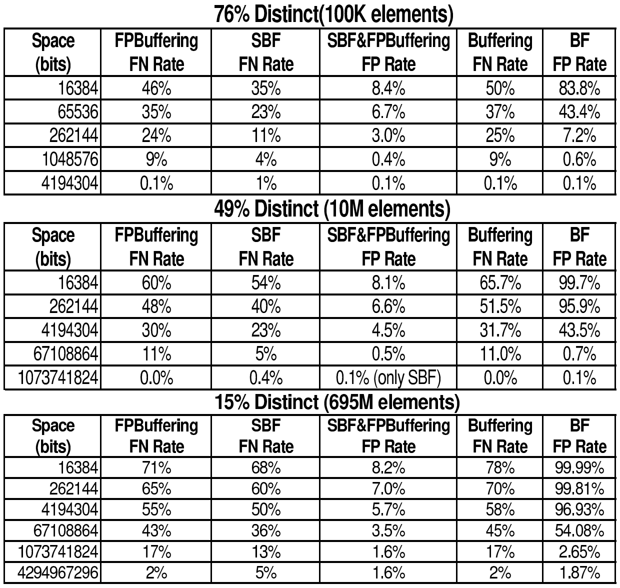


Figure 3: Fraction of zeros changed with time on the whole real data set (Max=1, K=2, P=4, FPS=10%), space unit=64bits

tively, and FPbuffering reduces the FN rates of buffering substantially in most cases by introducing a certain amount of FPs.



76% Distinct(100K elements)

Space (bits)	FPBuffering FN Rate	SBF FN Rate	SBF&FPBuffering FP Rate	Buffering FN Rate	BF FP Rate
16384	46%	35%	8.4%	50%	83.8%
65536	35%	23%	6.7%	37%	43.4%
262144	24%	11%	3.0%	25%	7.2%
1048576	9%	4%	0.4%	9%	0.6%
4194304	0.1%	1%	0.1%	0.1%	0.1%

49% Distinct (10M elements)

Space (bits)	FPBuffering FN Rate	SBF FN Rate	SBF&FPBuffering FP Rate	Buffering FN Rate	BF FP Rate
16384	60%	54%	8.1%	65.7%	99.7%
262144	48%	40%	6.6%	51.5%	95.9%
4194304	30%	23%	4.5%	31.7%	43.5%
67108864	11%	5%	0.5%	11.0%	0.7%
1073741824	0.0%	0.4%	0.1% (only SBF)	0.0%	0.1%

15% Distinct (695M elements)

Space (bits)	FPBuffering FN Rate	SBF FN Rate	SBF&FPBuffering FP Rate	Buffering FN Rate	BF FP Rate
16384	71%	68%	8.2%	78%	99.99%
262144	65%	60%	7.0%	70%	99.81%
4194304	55%	50%	5.7%	58%	96.93%
67108864	43%	36%	3.5%	45%	54.08%
1073741824	17%	13%	1.6%	17%	2.65%
4294967296	2%	5%	1.6%	2%	1.87%

Figure 4: Error rates comparison between SBF, FP-Buffering, Buffering and BF

Comparison between different methods. The tables in Figure 4 show that when the space is relatively small, SBF is better. SBF beats FPbuffering by 3-13% in terms of FN rate on different data sets, when their FP rates are the same. For the problem we are studying, we think this amount of improvement is nontrivial for 2 reasons. First, Broder et al. [8] implemented a theoretically optimal buffering algorithm called MIN, which assumes "the entire sequence of requests is known in advance", and accordingly chooses the best replacement strategy. Even this obviously impractical and ideal algorithm can only reduce the miss rates (FN rates in our case) of the LRU buffering, by no more than 5% in about 2/3 region (different buffer sizes). Second, from the tables we can see that even increasing the amount of the space by a factor of 4, the FN rates for buffering can be decreased by around 10-20%, which means the improvement from SBF may be equivalent of that from doubling the amount of space. The FP rates of BF is much higher than the acceptable FP rates in the first 2-3 rows of each table. Since buffering only generates FNs, it is not comparable to SBF here. But we can see that the FN rates of FPbuffering also decrease by introducing FPs into it.

However, we also notice that when the space is relatively large (the last row of each table), SBF performs not as good as buffering and BF. This is because when the space is large, BF might be able to hold all the distincts and keep a reasonable FP rates. We can directly compute the amount of space required based on the FP rates desired and the number of distincts in the data set according to the formula in Section 2.3. In this case, there is no need to evict information out of the BF, which means SBF is not applicable. If we can afford even more space, which is large enough to hold all the distincts in the data set using a buffer, there will be no errors at all. The last row of the second table shows this scenario. But in many data stream applications, a fast storage is needed to satisfy real time constraints and the size of this storage is typically less than the universe of the stream elements as discussed in Introduction.

Another fact is that in both SBF and buffering, we can refresh the storage and bias it towards recent data; they both evict stale information continuously and keep those fresh elements. While BF is not applicable in this case since BF can be only used to represent a static data set. Thus, it is not useful in many data stream scenarios that require dynamic updates.

Varying acceptable FP rates. Another experiment we ran was to test the effect of changing the acceptable FP rates. The results are shown in Figure 5. In this experiments, we set Max=3, K=4 when acceptable FP rates are set to 0.5% and 1%, and set Max=1, K=2 when acceptable FP rates are set to 10% and 20%. The bar chart depicts the FN rate difference between FPbuffering and SBF. Again, the FP rates of both methods are set to the same number. Clearly, it shows that the more FPs are allow, the better SBF performs.

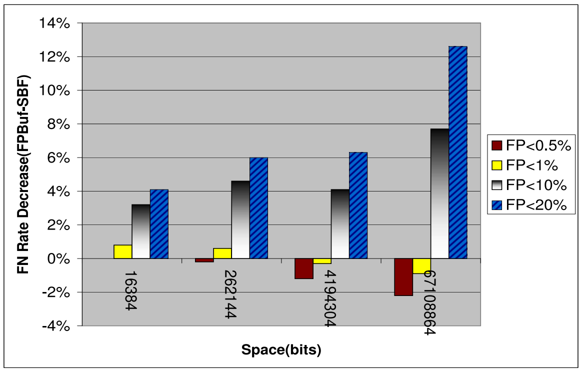


Figure 5: FN rate differences between FPBuffering and SBF varying allowable FP rate(695M elements)

5.5 Time Comparison

As discussed in the implementation section, SBF and BF need O(1) time to process each element. The exact time depends on the the parameter settings. For example, when K=2 and P=4, SBF needs less than 10 operations within each iteration.

For buffering and FPbuffering, their processing time is the same. It depends on 2 processes: element searching and element evicting. Searching can be quite expensive without an index structure. Both our experiments and those of Broder et al.[8] used a hash table to accelerate the search process. The extra space that is needed for a hash table to keep the search time constant is linear in the number of elements stored. The process of maintaining the LRU replacement policy(finding the least recently used element) is also costly, and extra space is needed to make it faster. This extra space can be quite large for LRU. However, this cost can be reduced to 2 bits per elements by using the Clock approximation of LRU [8].

Therefore, buffering and FPbuffering need extra space linear in the number of buffer entries to reach a similar O(1) processing time. But in our error rate comparison, we did not count this extra space for buffering and FPbuffering.

5.6 Methods Comparison Summary

We compared 4 methods in this section: SBF, BF, FPbuffering and buffering. Among them, BF and buffering have only FPs and FNs respectively, and SBF and FPbuffering have errors of both sides.

BF is a space efficient data structure which has been studied in the past and is widely used. It is good for representing a static set of data provided that the number of distinct elements is known. However, in data stream environments, the data is not static and it keeps changing. Usually it is hard to know the number of distinct elements in advance. Moreover, BF is not applicable in cases where dynamic updates are needed since elements can only be inserted into BF, but cannot be dropped out. Consequently, BF is not suitable for many data stream applications. Another variation of BF, counting BF, allows deletions, but it is still not applicable in the scenario we consider. See the discussion in the first paragraph of the related work section.

SBF, buffering and FPBuffering can be all applied to data stream scenarios. SBF is better in terms of accuracy and time when certain amount of FP rates are acceptable and the space is relatively small, which is the case in many data stream applications due to the real-time constraint. When the space is relatively large or only tiny FP rates are allowed, buffering is better.

6. RELATED WORK

The recent work of Metwally et al.[24]also study the duplicate detection problem in a streaming environment based on Bloom filters(BF) [7]. They consider different window models: Landmark windows, sliding windows and jumping windows. For the landmark window model, which is the scenario we consider, they apply the original Bloom filters without variations to detect duplicates, and thus do not consider the case that the BFs become "full". For the sliding window model, they use counting BFs [18] (change bits into counters) to allow removing old information out of the Bloom filter. However, this can be done only when the element to be removed is known, which is not possible in many streaming cases. For example, if the oldest element needs to be removed, one has to know that which counters are touched by the oldest element, but this information cannot be found in counting BFs, and maintaining this knowledge can be quite expensive. For the jumping window model, they cut a large jumping window into multiple sub-windows, and represent both the jumping window and the sub-windows with counting BFs of the same size. Thus, the jumping window can "jump" forward by adding and removing sub-window BFs.

Another solution for detecting duplicates in a streaming environment is the buffering or the caching method, which has been studied in many areas such as database systems, computer architecture, operating systems, and more recently URL caching in Web crawling [8]. We compare our method with those of Broder et al.[8] in the experiments. The problem of exact duplicate elimination is well studied, and there are many efficient algorithms(e.g. see [20] for details and references). For the problem of approximate membership testing in a non-streaming environment, the Bloom filter has been frequently used and occasionally extended [18, 25]. Cohen and Matias[14] extend the Bloom filter to answer multiplicity queries. Counting distinct elements using the Bloom filter is proposed by Whang et al.[31]. Another branch of duplicate detection focus on fuzzy duplicates [11, 1, 6, 30], where the distinction between elements is not straightforward to see.

A related problem to duplicate detection is counting the number of distinct elements. Flajolet and Martin[19] propose a bitmap sketch to address this problem in a large data set. The same problem is also studied by Cormode et al. [15], and a sketch based on stable random variables is introduced. Besides, the sticky sampling algorithm of Manku and Motwani [23] also randomly increment and decrement counters storing the frequencies of stream elements, but the decrement frequency is varying and not for each incoming element. Their goal is to find the frequent items in a data stream.

As for data stream systems [3, 9, 29, 10, 16, 12], as far as we know, most of them divide the potentially unbounded data stream into windows with limited size and solve the problem precisely within the window. For example, Tucker et al. introduce punctuations into data streams, and thus duplicate eliminations could be implemented within data stream windows using traditional methods[29].

Since there is no way to store the entire history of an infinite data stream using limited space, our SBF essentially represents the most recent information by discarding those stale continuously. This is useful in many scenarios where the recent data is more important and this importance decays over time. A number of such kinds of applications are provided in [13] and [26]. Our motivating example of web crawling also has this property, since it may not matter that much to redundantly fetch a Web page that have been crawled a long time ago compared to fetching a page that have been crawled more recently.

7. CONCLUSIONS

In this paper, we propose the SBF method to approximately detect duplicates for streaming data. When a certain false positive rate is allowed, SBF is superior in terms of both accuracy and time for a fixed amount of space compared to the alternative methods. Extending our work to handle sliding window queries is a future direction.

Acknowledgement

This work is supported by Natural Sciences and Engineering Research Council of Canada. We like to thank the anonymous reviewers for their comments and Ahmed Metwally for his feedback.

8. REFERENCES

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. of VLDB*, 2002.

[2] Internet Archive. `http://www.archive.org/`.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of PODS*, 2002.

[4] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of ICDE*, 2004.

[5] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to cams? In *Proc. of INFOCOMM*, 2003.

[6] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity m easures. In *Proc. of KDD*, 2003.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. In *CACM*, 1970.

[8] A. Z. Broder, M. Najork, and J. L. Wiener. Efficient url caching for world wide web crawling. In *Proc. of WWW*, 2003.

[9] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B.Zdonik. Monitoring streams - a new class of data management applications. In *Proc. of VLDB*, 2002.

[10] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proc. of CIDR*, 2003.

[11] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. of ICDE*, 2005.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. of SIGMOD*, 2000.

[13] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proc. of PODS*, 2003.

[14] S. Cohen and Y. Matias. Spectral bloom filters. In *Proc. of SIGMOD*, 2003.

[15] G. Cormode, M. Datar, P. Indyk, and S.Muthukrishnan. Comparing data streams using hamming norms (how to zero in). *IEEE Trans. Knowl. Data Eng.*, 15(3):529–540, 2003.

[16] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of SIGMOD*, 2003.

[17] C. Estan and G. Varghese. Data streaming in computer networks. In *Proc. of Workshop on Management and Processing of Data Streams(MPDS) in cooperation with SIGMOD/PODS*, 2003.

[18] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache:a scalable wide area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

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

[20] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, 2000.

[21] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4), 1999.

[22] Cisco System Inc. Cisco network accounting services. **`http://www.cisco.com/warp/public/cc/pd/iosw/prodlit/nwact_wp.pdf`**, 2002.

[23] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. of VLDB*, 2002.

[24] A. Metwally, D. Agrawal, and A. E. Abbadi. Duplicate detection in click streams. In *Proc. of WWW*, 2005.

[25] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.

[26] T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *Proc. of ICDE*, 2004.

[27] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[28] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of VLDB*, 2003.

[29] P. A. Tucker, D. Maier, and T. Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Eng. Bull.*, 26(1):33–40, 2003.

[30] M. Weis and F. Naumann. Dogmatix tracks down duplicates in xml. In *Proc. of SIGMOD*, June 2005.

[31] K. Whang, B. T. V. Zenden, and H. M.Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.*, 15(2):208–229, 1990.