Duplicate Detection in Click Streams [*]

Ahmed Metwally [†] Divyakant Agrawal Amr El Abbadi

Department of Computer Science
University of California, Santa Barbara
{metwally, agrawal, amr}@cs.ucsb.edu

Abstract

We consider the problem of finding duplicates in data streams. Duplicate detection in data streams is utilized in various applications including fraud detection. We develop a solution based on Bloom Filters [9], and discuss the space and time requirements for running the proposed algorithm in both the contexts of sliding, and landmark stream windows. We run a comprehensive set of experiments, using both real and synthetic click streams, to evaluate the performance of the proposed solution. The results demonstrate that the proposed solution yields extremely low error rates.

1 Introduction

Recently, online monitoring of data streams has emerged as an important data management problem. This research topic has its foundations and applications in many domains, including databases, data mining, algorithms, networking, theory, and statistics. However, new challenges have emerged. Due to their vast sizes, some stream types should be mined fast before being deleted forever. In general, the alphabet is too large to keep exact information for all elements. Conventional database, and mining techniques, though effective with stored data, are deemed impractical in this setting.

There is a growing need to develop new techniques to cope with high-speed streams, and answer online queries. Currently, data stream management systems are used for monitoring click streams [37], stock tickers [13, 58], sensor readings [10], telephone call records [19], network packet traces [21], auction bidding patterns [4], traffic management [5], network-aware clustering [16], and security against DoS [16]. Golab and Ozsu review the literature in [33].

[*]This work was supported in part by NSF under grants EIA 00-80134, NSF 02-09112, and CNF 04-23336.

[†]Part of this work was done while the first author was at ValueClick, Inc.

1.1 Motivating Application

This work is primarily motivated by the setting of Internet advertising commissioners, who represent the middle persons between Internet publishers, and Internet advertisers. In a standard setting, an advertiser provides the publishers with its advertisements, and they agree on a commission for each user action, e.g., clicking an advertisement, filling out a form, bidding on an item, or making a purchase. The publisher, motivated by the commission paid by the advertiser, displays advertisements, text links, or product links on its page; and uses a form of tracking code for these advertisements on their sites to keep logs of the traffic it drives to each advertiser's site. On the other end, the advertiser keeps track of the traffic that is generated by each of its publishers. Inconsistencies between the size of the driven traffic as measured by the advertiser and the publisher are resolved by a third tracking entity, the advertising commissioner. Whenever a customer uses a link, the customer is referred from the publisher's Web site to the servers of the advertising commissioner, who logs the click and *clicks-through* the customer to the Web site of the advertiser. The model is Illustrated in Figure 1.

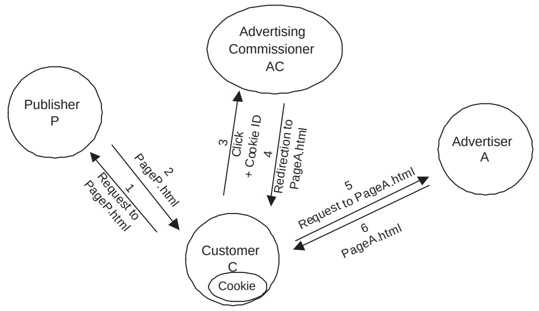

Figure 1: The Advertising Networks Model.

Since the publishers earn revenue on the traffic they drive to the advertisers' Web sites, there is an incentive for them to falsely increase the number of clicks their sites generate. This process is referred to, in [3], as *click inflation*.

One of the advertising commissioner's roles is to detect any fraud taking place on either the publisher's side or the advertiser's side. Thus, the advertising commissioner should be able to tell whether the clicks generated at the publisher's side are authentic, or are generated by a script running on some machines on the publisher's end, to claim more traffic, and thus, more revenue. In order to do that, the advertising commissioner should be able to track each click by the advertisement ID, and the customer ID. The advertising commissioner tracks individual customers, by setting cookies. Duplicate clicks within a short period of time, a day for example, raise suspicion on the commissioner's side.

Classically, advertising commissioners run queries towards the end of the month, when calculating the publishers' commissions, to capture duplicate clicks within a short period of time. Due to the large size of click streams, such queries degrade the performance of their databases considerably. Hence, due to monthly accounting constraints, this problem has to be solved on a real-time basis.

From the above example we are motivated to solve the problem of finding duplicates occurring in a very fast data channel, the click stream. To the best of our knowledge, this problem has not been addressed before.

A similar problem, the frequent elements [48], has been proposed to search for all elements which have occurred more than a user specified ratio of the stream. However, the stream properties that are assumed in the context of finding frequent elements are different from those that are assumed for duplicate detection. In the context of duplicate detection, the majority of the elements occurring in the data stream are supposed to be distinct. That is, the skew in the data stream is very weak. On the other hand, in the context of finding frequent elements, the underlying assumption is that a few elements, the frequent ones, are supposed to have much higher frequencies than the other elements. Thus, the number of distinct elements observed in the data stream is much smaller than the size of the stream; and the data is supposed to be more skewed than in the case of streams queried for duplicates. The theoretical space bound for finding frequent elements is inversely proportional to the error, which is one tenth to one hundredth of the user required ratio [11]. Thus, using these algorithms for duplicate detection is costly and inefficient.

The problem could be partially solved using the algorithms proposed for estimating distinct values [26, 39, 56]. However, The commissioner will have no way to prove which clicks were fraudulent. Only through identifying duplicate clicks and storing them for future auditing the commissioner can be standing strong on solid ground.

We develop an efficient solution based on Bloom Filters [9], and consider both types of stream windows, the sliding, and the landmark windows. We analyze the space and time requirements, and run a set of experiments to evaluate the performance of our proposed solution.

1.2 Roadmap

The rest of the paper is organized as follows. Section 2 highlights the related work. In Section 3, we formalize the problem, followed by a discussion of the preliminary solutions in Section 4. The more efficient, and approximate solution, is presented in Section 5. We report the results of our experiments in Section 6, and conclude in Section 7.

2 Background and Related Work

This work touches on three main domains: *management of data stream*, *approximate duplicate detection*, and *the security of web advertising schemes.* We briefly outline the literature in these three domains.

2.1 Management of Data Streams

Recently, data streams management has emerged as an active research area. There have been several works for analyzing data streams. Problems studied in this context include approximate frequency moments [1], differences [25], distinct values estimation [26, 39, 56], bit counting [20], approximate quantiles [34, 44, 49], histograms [36, 35], wavelet based aggregate queries [30, 50], correlated aggregate queries [28], elements classification [38], frequent elements [11, 17, 18, 21, 22, 24, 31, 41, 42, 48, 51], and top-k queries [7, 12, 21, 29, 51].

2.2 The Security of Web Advertising Schemes

The problem of *hit shaving* has been studied in [54]. Hit shaving is another type of fraud performed by advertisers, who do not pay commission on some of the traffic received from publishers. Reiter *et al.* [54] presented a new scheme to click-through the customers from the publisher to the advertiser, so that the publisher can keep a more accurate record of the traffic driven to the advertiser.

Anupam *et al.* [3] proposed a click inflation attack on the *pay-per-click* scheme described above. The attack requires the cooperation of more than one publisher to fraud an advertiser. The attack is relatively difficult to detect, though it leads to a relatively high *click-through-rate*[1], which should still raise the suspicion of the advertising commissioners.

2.3 Duplicate Detection

The problem of finding approximate duplicate items has been researched before in both the contexts of data management, and Web applications. The work done in

[1]The click-through-rate of a publisher is the number of users who click advertisements on the publisher's Web site, as a ratio of all the visitors to the Web site.

[2, 8, 27, 40, 46, 52, 53, 55] focused on detecting fuzzy duplicate records in databases. These works were primarily motivated by data cleaning in the process of loading data into a data warehouse. On the other end of the spectrum, [14, 15, 43, 45, 47, 57] were interested in detecting duplicate documents and Web pages. This is directly applicable to eliminating duplicate pages reported by search engines.

A work that is very close to ours was done in 1979 [6]. The work focused on using specialized hardware to implement projection and join operators in the context of relational databases. Duplicate elimination, a crucial intermediate step in data processing, was implemented using a technique very similar to ours.

3 Formalizing the Problem

The problem of duplicate detection has two basic variations, depending on the way the stream is handled. The *Duplicate Detection on a Sliding Window* asks for duplicate elements that have occurred in the last N elements. An example of this query asks for the duplicate clicks that occurred in the last 1,000,000 clicks. The *Duplicate Detection on a Landmark Window* asks for duplicate elements that have occurred since the occurrence of a specific landmark. For instance, a query could ask for the duplicate clicks that have occurred since the beginning of the current day or week.

3.1 Duplicate Detection on Sliding Windows

The model for sliding window processing in data streams was first introduced in [20]. The goal is to make the query answer relevant to the last observed part of the stream. Applying the sliding window concept to our motivating application, N could be set large enough so that even if duplicates occur on intervals which are more than N clicks apart, they would have negligible effect on the commission paid to the publisher. In addition, it is still acceptable, from a practical perspective, that a user might click the same advertisement, a few number of times in the span of a long time period.

From a practical point of view, in order to slide the window, for every new entry arriving, an old entry has to be evicted. It is necessary to keep the latest N clicks in a circular queue, in order to know which entry to evict [32]. In addition, the entire window has to be summarized in a data structure that supports fast search, incorporation, and deletion of entries.

3.2 Duplicate Detection on Landmark Windows

Processing a stream based on landmark windows requires handling disjoint portions of the streams, which are separated by landmarks. Landmarks can be defined either in terms of time, e.g., on daily or weekly basis, or in terms of the number of elements observed so far since the previous landmark, e.g., every 1,000,000 elements.

In this model, it is not necessary to keep the entire window, rather a summary may be sufficient, since none of the elements will be deleted. On the other hand, at the beginning of every landmark window, all the structures that hold the clicks have to be re-initialized. In addition, the window size is not fixed, in contrast to the sliding window model, since the window grows as more clicks are observed.

The classical landmark window model facilitates the computation of duplicates at the expense of the effectiveness of the query. Although the landmark window is expected to require less space, since it does not store the entire window, but only a structure that can be queried for duplicate clicks, it could always happen that two duplicate clicks, that are very close in time, are missed, because they did not occur in the same window. This can happen if one of them is towards the end of a window, and the second click is towards the beginning of the next window. This is the reason landmark windows are usually set to grow in size slightly more than that of sliding windows.

3.3 Duplicate Detection on Jumping Windows

The jumping window model [58] is a compromise between the landmark and the sliding window models. The basic idea is to slide the window in jumps rather than in individual elements. The window is divided into smaller disjoint sub-windows of size n, and a summary of each sub-window is stored in a separate data structure. When the data structure of the latest sub-window is populated, its results are combined with the results of the jumping window; and the data structure of the eldest sub-window is deleted, and its results are deleted from the jumping window.

The basic advantage of the jumping window model is that it guarantees the freshness of the results, since the queried portion of the stream is of fixed length N, and is at most $(n-1)$ elements behind the most recent observed element in the stream. In addition, this model does not require storing the whole window. However, it entails using data structures whose results can be combined, and subtracted efficiently.

4 Preliminary Algorithms

To find duplicates in a click stream, the obvious solution is to store all the clicks, and when observing a new click, just compare it to all the entries already stored. If the new click is found to be a duplicate, then do not count it towards the corresponding publisher's commission.

The basic disadvantage of this solution is the need to scan the entire window to tell whether a newly observed click is a duplicate, or not. Thus processing a stream/window of size N requires $O(N^2)$ comparisons to detect duplicates. Building an index on the elements, would reduce the search cost to $O(\log(N))$ per element, but on the other hand, would increase the element insertion cost to $O(\log(N))$, instead of just appending an element in constant time.

The straightforward solution described above solves the exact problem, but suffers from slow processing of the data stream. The best estimate of the complexity is $O(N \log(N))$, which will not be practical for large streams. We propose another efficient technique, which has a constant amortized cost, does not need more space, detects all duplicates, and produces very few false positive errors. By false positive errors, we refer to non-duplicate elements erroneously identified by the algorithm as duplicates. We will start by proposing simpler algorithms that will lead to the development of the efficient solution. We will limit the discussion to handling landmark windows, and then will extend our solution to sliding and jumping windows in Section 5.2.

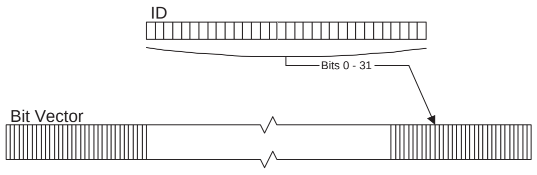

Figure 2: The bit vector solution. The ID is 32 bits long, and the bit vector is of length 2^{32} bits.

The first improvement on the index-based solution for detecting duplicates in a data stream is to use a bit vector. Assume the elements of the data stream, the clicks, are coming from an alphabet A. Then keeping a bit for every element in A is enough to keep track of which elements have been observed in the stream by flagging their corresponding bits to 1. Figure 2, clarifies the solution by giving an example for the case where the ID has 32 bits, which entails keeping a bit vector of length 2^{32} bits. A new element is a duplicate if its bit has been flagged 1, before. The algorithm is simple, exact, and takes $O(1)$ steps and space to insert a new element into the bit vector, or to check it for duplication.

However, this simple scheme cannot be implemented in our case. The alphabet we are dealing with in our application is the domain of IDs of clicks. Each click

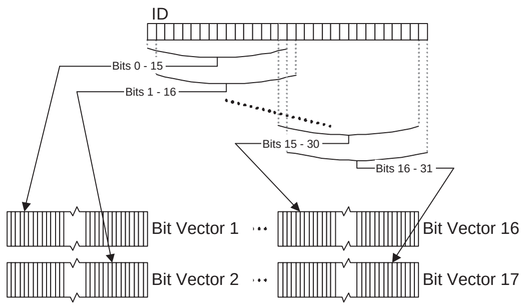

Figure 3: The overlapping bit-substrings solution. The ID is 32 bits long, and 17 bit vectors is of length 2^{16} bits are kept.

ID is represented by a pair of an advertisement ID, and a user cookie ID. Click IDs are represented by 64 characters. Thus, keeping a bit for every ID entails keeping 2^{512} bits $\approx 1.676 * 10^{153}$ bytes, which is infeasible.

The next modification is to keep partial information, rather than all the combinations of the alphabet. Assuming an element is represented by b bits, where b is 512 in our application, we can keep less than 2^b bits, and still get approximate results with a very low error rate. The basic idea is to keep a bit vector of size 2^p for all the combinations of the first p bits, another bit vector for the second p bits, and so forth, where $1 \leq p \leq b$.

Assuming the bits of an element are labeled $0 \ldots (b-1)$, the first bit vector has bits for all the combinations of bits $0 \ldots (p-1)$. The second bit vector has bits for all the combinations of bits $1 \ldots p$. Notice that the first and the second bit vectors correspond to bits that largely overlap in the element ID. In general, the n^{th} bit vector has bits for all the combinations of bits $(n-1) \ldots (n+p-2)$. There are $b-p+1$ bit vectors utilized.

As the stream is observed, for each element, its first p bits will be checked if they have occurred before in the first bit vector, then, the bit corresponding to those first p bits will be flagged 1. The focus will be shifted by 1 bit to the bits $1 \ldots p$, and those bits will be checked against the second bit vector in the same manner, and so on. An element which has at least one substring of p bits that never occurred before is guaranteed not to be a duplicate. On the contrary, there is a high probability that an element is a duplicate if it has collided on all its substrings of length p bits.

If $p = b$, then this is the exact bit vector solution discussed above. Otherwise, the space requirements will drop from 2^b to $2^p(b-p+1)$ bits. An example of the solution where $b = 32$, and $p = 16$ is given in Figure 3. There are $(b-p+1) = (32-16+1) = 17$ bit vectors. The number of bits used in this solution, as a percentage of the number of bits used in the exact solution is $\frac{2^p(b-p+1)}{2^b} * 100 = \frac{2^{16} * 17}{2^{32}} * 100 = 111411200/4294967296 \approx 0.026\%$.

There is a tradeoff here between the rate of false positive errors reported as duplicates and the space usage. The larger the value of p, the smaller the rate of false positive errors, and the larger the space requirements.

Any two elements picked at random have a probability of 2^{-p} that they will collide in any bit vector of length p, given that the bits are distributed uniformly. The underlying idea of the algorithm is that when viewing the i^{th} element, it has a probability of $\left(1-(1-2^{-p})^{i-1}\right)$ that it, by chance, collides with any of the previously observed $i-1$ elements on the first p bits. It is even less probable that an element collides with others on the first and the second p bits. The probability gets smaller as more substrings of length p bits are taken into consideration. Thus, if the algorithm finds out that an element has collided on all its bit-substrings of length p, for a reasonable length of p, then most probably, this element is a duplicate.

Unfortunately, a probabilistic analysis of this scheme is intractable. The main problem is that an element a that collides with another element b in a bit vector n is more likely to collides with the same element b than the rest of the elements in the bit vectors $(n-p+1)$ to $(n+p-1)$. For example, given that the two elements a, and b have the same values in bits $(n-1)$ to $(n+p-2)$, then there is a probability of $\frac{1}{2}$ that the values of the bits starting at n to $(n+p-1)$ will be equal, using the basic conditional probability rule. This probability is much higher than the probability of 2^{-p} given above, which is the general probability of collision between two elements in one bit vector. We modify the algorithm once more to serve both purposes of achieving better results, and facilitating the probabilistic analysis. We will use the same idea of shrinking the size of the bit vector to less than 2^b. However, instead of using overlapping bit-substrings of the IDs, we will use independent hash functions.

Utilizing independent hash functions eliminates the risk of correlated collisions discussed above, and thus yields better results, and facilitates the probabilistic analysis. Interestingly, using independent hash functions makes our solution another development of Bloom Filters [9].

5 The Bloom Filter-Based Solution

In this section we will describe the classical Bloom Filters and their motivating application of testing approximate membership of elements. Then, we will discuss how to utilize Bloom Filters in our context.

5.1 The Classical Bloom Filters

A Bloom Filter [9] is a data structure that was proposed to detect approximate membership of elements. Given two sets, X, and Y, the Bloom Filter algorithm would loop on every element in set X, to check if it belongs to set Y, too. The algorithm is probabilistic, requires $O(|X|)$ operations, and $O(|Y|)$ space. A Bloom Filter can assert that an element in X does not belong to Y, but cannot assert that an element in X belongs to Y. That is, its errors are only false positive, and never false negative. On the whole, the number of elements that are erroneously found to belong to Y, and that actually do not, is very small, and is inversely proportional to the hidden constant in the big-O of the space and time requirements.

An empty Bloom Filter is an array of M cells, with addresses $0 \ldots M-1$, that are initially zeroed. Each element, y, in Y is hashed using d independent hash functions to addresses y_1, y_2, $\ldots$, y_d, which are set to 1, such that $0 \leq y_i \leq M-1, \forall i$. For each element, x, in X, its d hash results, x_1 to x_d, are generated in the same manner, and checked against the Bloom Filter that represents the set Y. If any of the cells x_1 to x_d is not set to 1, then it can be asserted that $x \notin Y$. However, if all the cells x_1 to x_d are set to 1, there is a good probability that $x \in Y$.

The intuition is very simple, and is similar to that discussed in Section 4. The probability of a false positive is inversely proportional to d, the number of hash functions used, given that the space utilized grows proportionally with d. The more hash functions used, the less it is probable that a non-duplicate element will collide with other elements on all its d hash functions.

The interesting thing about Bloom Filters is that they do not store the elements of the set whose membership is tested. This is very useful in cases were the IDs of the elements are huge, like in our case. However, it is not possible to regenerate the original set from its Bloom Filter representation.

We summarize now the original analysis given in [9]. Assuming the d hash functions are distinct, and each almost uniformly distributes the hashed elements, then the probability that a certain cell has not been set by any of the hash functions after inserting one element in Y is $(1-\frac{1}{M})^d$. After inserting all the elements in Y, then the probability that the cell has never been set to 1 is $(1-\frac{1}{M})^{d*|Y|}$. Thus, the probability that it has been set to 1 is $1-(1-\frac{1}{M})^{d*|Y|}$.

When testing whether, or not, an element in X belongs to Y, the probability that it collides on all its d hash functions, without being a duplicate, is $(1-(1-\frac{1}{M})^{d*|Y|})^d \approx (1-e^{-d*|Y|/M})^d$. This probability clarifies the tradeoff. The false positive errors increase as the size of Y increases, and decrease as the number of cells, M, increases. Differentiating this probability with respect to d, for a fixed array size, and a fixed size of Y, yields the number of hash functions to be used to minimize the probability, which is $\frac{M}{|Y|} * \ln 2$. Substituting this value for d back in the probability of false positive errors yields $(2^{\ln 2})^{M/|Y|} \approx (0.619)^{M/|Y|}$, which is a concise form for the probability of false positive errors.

In the next section we will discuss how Bloom Filters can be used to detect duplicates in data streams.

5.2 Using Bloom Filters for Duplicate Detection in Data Streams

Bloom Filters can be used to detect duplicates in a data stream, by assuming that set X is the last observed element, and set Y is the rest of the data stream.

5.2.1 The General Framework

We start by allocating M bits, where M is $O(N)$, and N is the estimated size of the processed window. As illustrated in Figure 4, using $d = 17$ independent hash functions, we test every new element on the Bloom Filter structure of the previously observed elements, and then insert it into the Bloom Filter structure. Before setting any of the d cells to 1, the cell is tested whether it has been set before to 1, or not. The element is not counted as a duplicate if at least 1 bit was switched from 0 to 1, and is considered to be a duplicate otherwise. Both M and d can be determined according to the required error rate, and the expected window size.

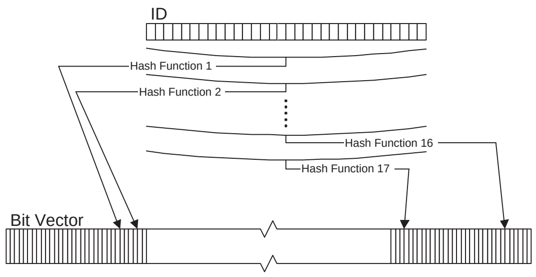

Figure 4: The classical Bloom Filters. The ID is 32 bits long, and 17 hash functions are used.

Bloom Filters are very concise when compared to the basic solution using indexed storage. It does not store the excessively long IDs of the clicks, and its false positive errors are almost negligible. Bloom Filters' errors do not depend on the nature or distribution of the duplicated elements. That is, the number of elements erroneously identified as duplicates is the same whether one element was duplicated a lot of times, or a lot of elements were duplicated a few times.

The probability of outputting false positive errors can be made very low using very limited storage. For instance, based on the previous analysis, using 9.6 bits per element in the data stream reduces the probability of outputting false positive errors to 1%; and using 2 bytes per element reduces this probability to 0.046%, which is a great achievement from a practical point of view. In addition, the IDs that are output as false positive errors can be stored in a separate cache to verify that they are duplicated frequently.

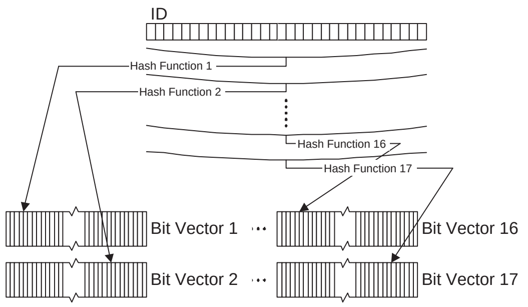

Figure 5: Another Variation of Bloom Filters. The ID is 32 bits long, and 17 hash functions are used.

The original Bloom Filters use one hash space and hashes all the elements onto it. However, when we developed this solution, independently of Bloom Filters, we used separate hash spaces for different hash functions, as sketched in Figure 5. This scheme of using hash functions perfectly replaces the scheme of bit vectors substrings described in Section 4, and has a similar analysis to Bloom Filters.

Assuming d hash functions, the hash space for each hash function will be $\frac{M}{d}$. Assuming each hash function almost uniformly distributes the hashed elements, then the probability that a certain cell has not been set by any of the hash functions after inserting one element is $(1 - \frac{d}{M})$. When inserting the i^{th} element, the probability that the cell has never been set to 1 is $(1-\frac{d}{M})^{(i-1)}$. Thus, the probability that it has been set to 1 is $1-(1-\frac{d}{M})^{(i-1)}$.

The probability that the i^{th} element is identified by mistake as a duplicate, i.e., it collides on all its d hash functions, is $(1 - (1 - \frac{d}{M})^{(i-1)})^d \approx (1 - e^{-d*(i-1)/M})^d$. This is the same as the probability of the original Bloom Filter.

However, having different hashing spaces assigned for different functions results in increased server parallelism, since the memory activation latency is minimized. The memory activation latency, τ, is the idle time between two accesses to the same memory module. Assuming the availability of D memory modules, when using separate hash spaces for different hash functions, the time wasted for memory access latency is $\lceil \frac{d*\tau}{D} \rceil$. However, when using one shared hash space for all the hash functions, the time wasted for memory access latency is between $\lceil \frac{d*\tau}{D} \rceil$, and $d*\tau$, depending on how the hashed-to flags are distributed throughout the hash space.

5.2.2 Using Bloom Filters for Landmark Windows

When used for landmark windows, the Bloom Filter structure is re-initialized at the beginning of each landmark window. Interestingly, Bloom Filters are more effective when used for duplicate detection on landmark windows than when used for approximate membership testing. The basic argument is that superimposing the last observed element on the set X and the rest of data stream window, whose size in growing, on set Y reduces the false positive errors dramatically. For instance, when inserting the first element in the stream into the Bloom Filter, the probability that it is found to be a duplicate is 0. When inserting the second element into the Bloom Filter, the probability that it is found to be a duplicate is $(1 - e^{-d/M})^d$. In general, when inserting the i^{th} element into the Bloom Filter, the probability that it is found to be a duplicate is $(1 - e^{-d*(i-1)/M})^d$. Therefore, the probability that an element is found to be a duplicate is proportional to the number of elements previously inserted into the Bloom Filter. This is because the more elements accommodated in the Bloom Filter, the higher the probability that a hash collision occurs, and the higher the chance that an element is identified as a duplicate.

It is easy to show that the expected number of duplicates found in a landmark window of size N is $\sum_{i=1}^{N}(1 - e^{-d*(i-1)/M})^d$, which is less than $N * (1 - e^{-d*N/M})^d$, the expected error rate of Bloom Filters when testing approximate set membership.

Although we have assumed that the hash functions distribute the elements uniformly, which is a difficult to realize assumption, we expect this will be balanced by the fact that the probability of false positive errors is lower in the early parts of the stream than the estimated bound. The practical performance of the Bloom Filters will be compared to its expected theoretical bound in Section 6.

5.2.3 Using Bloom Filters for Sliding Windows

The main challenge when applying Bloom Filters to sliding window streams is that the structure does not store the IDs of the elements observed so far in the stream, and thus, it is difficult to delete elements which have been already inserted into the Bloom Filter, and is now expiring, i.e., sliding out of the sliding window.

We use a modification of Bloom Filters[2]. The purpose of this modification is to enable Bloom Filters to implement the *delete* operation, without necessarily storing the IDs of the elements inserted into a Bloom Filter structure. The underlying idea is to replace the array of bits with an array of counters, of the same size. For every element that is inserted, increment the d counters to which the element hashes. To delete an element, decrement the d counters to which the element hashes.

An integer in a cell represents the number of elements which hash to this cell. A cell is decremented to 0, only if all the elements that hashed to this cell expire, i.e., slide out of the current sliding window, which is equivalent to deleting all those obsolete elements from the Bloom Filter structure. Therefore, we will call this solution the *counting* Bloom Filters. Thus, it is possible to update the Bloom Filter structure as new elements are added to the sliding window, and as aging elements are deleted.

[2]The same idea was proposed by [23] to modify Bloom Filters for implementing a scalable distributed cache sharing protocol.

5.2.4 Using Bloom Filters for Jumping Windows

Bloom Filters have the following nice property that makes them usable to detect duplicates in jumping window streams. Two Bloom Filter structures are, in abstract, two bit arrays. Thus, it is possible to perform some basic binary operations on them, such as *OR*, given that they use the same hash functions. OR-ing two Bloom Filter structures results in a new one that accommodates all the elements that were inserted into the two original structures.

Similarly, the *counting* Bloom Filter structures proposed in Section 5.2.3 can be *added* and *subtracted*. Adding two Bloom Filter structures, A and B, results in a new one, $(A + B)$, that accommodates all the elements originally inserted into A and B. Since the hash space consists of counters, rather than bits, each counter in A (B) corresponds to all the occurrences of elements that hash to this counter in A (B). Similarly, each counter in the Bloom Filter structure $(A + B)$ corresponds to all the occurrences of elements that hash to this counter in both A and B. Notice that the operation $A - B$ is meaningless, unless all the elements in B have also occurred in A.

Thus, it is possible to represent every sub-windows as a Bloom Filter structure. When a new sub-window is populated, its Bloom Filter representation is combined with the main Bloom Filter structure that represents the entire jumping window, and the eldest window is subtracted from the main Bloom Filter structure. Combining the two Bloom Filter structures is carried out by adding their counters; and deleting the Bloom Filter structure that correspond to the expiring sub-window is performed by subtracting its counters from the main Bloom Filter structure. Thus, the Bloom Filter representation of the jumping window is always correct.

6 Experimental Results

In order to evaluate how effective the proposed solution is, we ran a comprehensive set of experiments. The experiments used a real click stream data collected at Commission Junction, a ValueClick company. In addition we ran a set of experiments on synthetic data. The results are analyzed in this section.

6.1 Experimental Setup

In a preliminary phase of experiments, we used a combination of the user IP address and the advertisement ID, as the click ID. Both components of the click ID were represented as 32-bit integers, and thus, the click ID was represented as 64-bit integers.

The preliminary thinking was that if some clicks on the same advertisement come from the same IP address more than once in a short period, then this reveals some kind of fraud taking place at the publisher's site. However, since Network Address Translation (NAT) boxes can hide hundreds to thousands of computers under the same IP address, we found this scheme to be overly aggressive in terms of detecting fraud. The other option was to use the cookie ID, which is set on the user Internet Browser to be able to track individual users. Hence, individual clicks were identified using 64-byte identifiers.

We are interested in the accuracy of Bloom Filters when detecting duplicates. To calculate the accuracy, we measure the *error rate*, the number of non-duplicate clicks identified as duplicates, as a ratio of the number of non-duplicate clicks in the entire stream. We evaluated the effectiveness of Bloom Filters for solving the problem of duplicate detection on data streams with different window models. We carried out the experiments on both landmark and jumping windows. For landmark windows, measuring the error rate is straightforward. However, for jumping windows, we measure the error rate after each jump, and calculate the mean error rate throughout the whole stream.

We did not test the Bloom Filters approach on sliding windows. It is infeasible to measure the accuracy of the solution, since that would require exact identification of all duplicates for every window. For instance, if the stream size is 5,000,000, and the window size is fixed to 1,000,000 clicks, then the number of windows that cover the given stream is 5,000,000 - 1,000,000 + 1 = 4,000,001 windows. Thus, to check the error rate for such solution, the exact solution has to be run 4,000,001 times, and the results should be compared to those of the approximate solution. Thus, we only run experiments on the landmark and jumping windows. For jumping windows, using a small sub-window will result in an overwhelming number of windows. To reduce this problem, we limit the experiments to only a small number of overlapping windows.

6.2 Goals of the Experiments

The underlying idea of using Bloom Filters is to exploit the tradeoff between space and time on one hand, and the error rate on the other hand. The more hash functions used to test for duplicates, the larger the required space, and the smaller the probability of producing false positive errors. However, since the space usage of Bloom Filters is a constant multiple of the number of hash functions used, throughout the experiments, we use the number of hash functions as the independent axis in our graphs, and will report the ratio between the number of hash functions and the size of the hash space used. Hence, the reader can recognize the tradeoff between the space usage and the error rate.

In addition, the experiments validate our arguments in Section 5.2.2. As we mentioned above, we expect that the practical error rate for duplicate detection, in landmark and jumping windows, is close to the theoretical error rate calculated in Section 5.1, if not better. The experiments support this proposition.

6.3 Evaluating the Tradeoff Between the Number of Hash Functions and the Error Rate

In this subsection we sketch and discuss the results of both the real and the synthetic data experiments.

6.3.1 Synthetic Data Results

We conducted experiments using both landmark and jumping windows on streams of synthetic data to illustrate how the theoretical and the practical error rates vary with the number of hash functions. We use a synthetic stream of length 1,000,000 clicks for the landmark window experiments. The clicks are *distinct*, i.e., there are no duplicates in the data. Thus, all the duplicates output by the algorithm are erroneous. For the jumping window experiments, each jumping window consists of 200,000 *distinct* clicks, and each sub-window is of size 50,000 clicks. Therefore, the number of overlapping windows is 8.

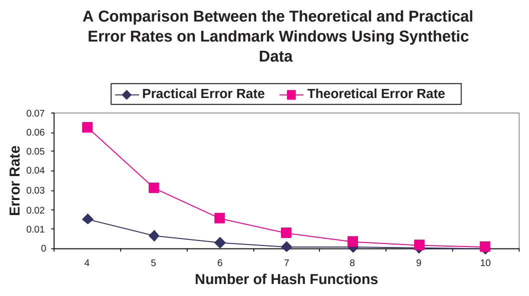

Figure 6: Comparing the Theoretical and the Practical Error Rates of Bloom Filters for Duplicate Detection on Landmark Window Using a Synthetic Click Stream

The experimental results of the jumping window, and the landmark window are sketched in Figures 6 and 7, respectively. For landmark windows, the ratio between the hash function to the number of flags used is 1:1,442,695. For jumping windows, the ratio between the hash function to the number of counters used is 1:288,539. The hash space used is proportional to the size of the window handled.

As is clear from Figure 6, the error rate of Bloom Filters is 4 to 8 times lower than the theoretical error rate. The error rate ratio increased as the number of hash functions increased. However, the tradeoff relation still exists between the number of hash functions used and the error

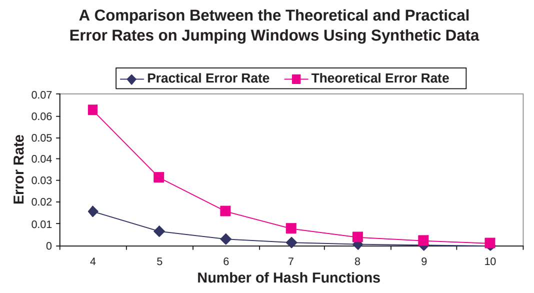

Figure 7: Comparing the Theoretical and the Practical Error Rates of Bloom Filters for Duplicate Detection on Jumping Window Using a Synthetic Click Stream

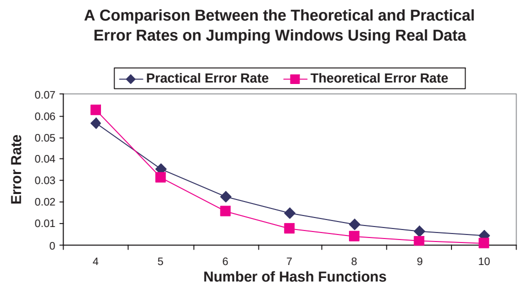

Figure 9: Comparing the Theoretical and the Practical Error Rates of Bloom Filters for Duplicate Detection on Jumping Window Using a Real Click Stream

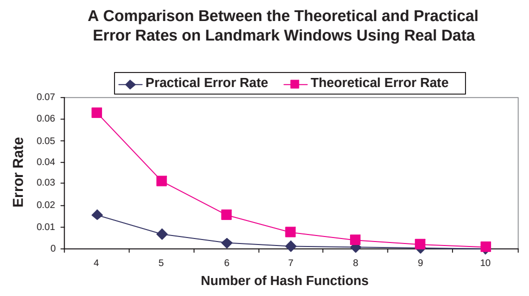

Figure 8: Comparing the Theoretical and the Practical Error Rates of Bloom Filters for Duplicate Detection on Landmark Window Using a Real Click Stream

rate of the Bloom Filters. The same results hold for the jumping window experiments. From Figure 7, the error rate of Bloom Filters is also 4 to 8 times lower than the theoretical error rate.

6.3.2 Real Data Results

The experiments were run on a clicks stream collected on August 30, 2004. The stream was of size 5,583,301 clicks. Each click had a 64 character identifier, which is composed of an advertisement ID, and a user cookie ID. The stream has 4,093,573 distinct elements, and 1,489,728 duplicates. The duplicates were not uniformly distributed, but rather some elements were duplicated more than others. The most duplicated element occurred 10,781 times, and understandably, was identified as fraudulent. The second most duplicated element had 4,487 occurrences, and was also flagged as being fraudulent.

To evaluate the tradeoff between the number of hash functions and the error rate, we conducted a set of experiments using both landmark and jumping windows. For the landmark window experiments, we use the whole stream as the dataset. For the jumping window experiments, each jumping window is of size 2,000,000 clicks, and each sub-window is of size 500,000 clicks, and we limit the number of windows to 4.

The experimental results of the landmark window are sketched in Figure 8, while those of the jumping window experiments are sketched in Figure 9. For landmark windows, the ratio between the hash function to the number of flags used is 1:5,905,777. For jumping windows, the ratio between the hash function to the number of counters used is 1:2,177,154, on average, depending on the number of distinct elements in the stream window. Notice that the ratio between the hash function to the size of the hash space is proportional to the number of distinct elements in the stream/window.

From Figure 8, it is clear that the error rate of Bloom Filters is much lower than the theoretical error rate, though the tradeoff relation still exists between the number of hash functions used and the error rate.

From Figure 9, it is noteworthy that the practical error rate of landmark window processing is lower than that for handling jumping windows. Most probably this is due to the nature of the real click stream, since that was not the case for synthetic data.

7 Discussion

In this paper, we have introduced the problem of finding duplicates in data streams, to detect fraud in advertisement networks. To the best of our knowledge, no similar work has been done before. A solution based on Bloom Filters [9] was proposed, and was applied to the motivating application of detection fraudulent hit inflation in advertising networks.

Interestingly, the idea of independent hashing employed in Bloom Filters is more effective when used for duplicate detection than when used for approximate membership testing, which is the motivating application for the original Bloom Filter. We ran a set of experiments using both real and synthetic data. The results demonstrated that applying the scheme of independent hashing to duplicate detection in click streams yields practical error rates that are lower than the theoretical error rates. The reason is that the Bloom Filter structure is not fully populated throughout the landmark or jumping windows. Hence, it can be deduced that using the theoretically optimum number of hash functions does not necessarily yield the least error rate. Thus, in the cases of landmark or jumping windows, it could be advisable to use slightly more hash functions than the theoretical optimum.

It could be argued that finding duplicates in click streams in the way discussed above is not effective for detecting the hit inflation attack. The fraudulent publisher could modify the script to delete the cookies every time it simulates a click on the advertisement. Thus, every time the script simulates a click, the advertising commissioner will detect no cookie, and will assign a new cookie ID. Thus the proposed duplicate detection technique will not detect this variation of attack, since the cookie ID is a part of the click ID.

The solution for this attack variation is very simple. If the publisher runs such a modified script, then the advertising commissioner will notice a suspicious number of IDs being assigned to a specific IP address at a fast rate, and for a long period of time. This could not be a normal situation, since the number of computers behind any NAT box is finite. Therefore, the advertising commissioner needs to keep track of the IP addresses that are assigned extremely large numbers of cookie IDs. This is a problem of detecting frequent elements in data streams, which is already a solved problem, as discussed in Section 2.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the 28th ACM STOC Symposium on the Theory of Computing*, pages 20–29, 1996.

[2] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating Fuzzy Duplicates in Data Warehouses. In *Proceedings of the 28th ACM VLDB International Conference on Very Large Databases*, pages 586–597, 2002.

[3] V. Anupam, A. Mayer, K. Nissim, B. Pinkas, and M. Reiter. On the Security of Pay-Per-Click and Other Web Advertising Schemes. In *Proceedings of the 8th International Conference on World Wide Web*, pages 1091–1100, 1999.

[4] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Proceedings of the 9th DBPL International Conference on Data Base and Programming Languages*, pages 1–11, 2003.

[5] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report 2002-67, Stanford University, 2003.

[6] E. Babb. Implementing a Relational Database by Means of Specialized Hardware. *ACM Transactions on Database Systems*, 4(1):1–29, 1979.

[7] B. Babcock and C. Olston. Distributed Top-k Monitoring. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 28–39, 2003.

[8] M. Bilenko and R. Mooney. Adaptive Duplicate Detection Using Learnable String Similarity Measures. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 39–48, 2003.

[9] B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] P. Bonnet, J. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of the 2nd IEEE MDM International Conference on Mobile Data Management*, pages 3–14, 2001.

[11] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Bounds for Frequency Estimation of Packet Streams. In *Proceedings of the 10th SIROCCO International Colloquium on Structural Information and Communication Complexity*, pages 33–42, 2003.

[12] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming*, pages 693–703, 2002.

[13] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[14] A. Chowdhury, O. Frieder, D. Grossman, and M. McCabe. Collection Statistics for Fast Duplicate Document Detection. *ACM Transactions on Information Systems*, 20(2):171–191, 2002.

[15] J. Conrad, X. Guo, and C. Schriber. Online Duplicate Document Detection: Signature Reliability in a Dynamic Retrieval Environment. In *Proceedings of the 12th ACM CIKM International Conference on Information and Knowledge Management*, pages 443–452, 2003.

[16] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Data

Streams. In *Proceedings of the 29th ACM VLDB International Conference on Very Large Data Bases*, pages 464–475, 2003.

[17] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the Rough: Finding Hierarchical Heavy Hitters in Multi-Dimensional Data. In *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, pages 155–166, 2004.

[18] G. Cormode and S. Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. In *Proceedings of the 22nd ACM PODS Symposium on Principles of Database Systems*, pages 296–306, 2003.

[19] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–17, 2000.

[20] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proceedings of the 13th ACM SIAM Symposium on Discrete Algorithms*, pages 635–644, 2002.

[21] E. Demaine, A. Lopez-Ortiz, and J. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *Proceedings of the 10th ESA Annual European Symposium on Algorithms*, pages 348–360, 2002.

[22] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

[23] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[24] M. Fang, S. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing Iceberg Queries Efficiently. In *Proceedings of the 24th ACM VLDB International Conference on Very Large Data Bases*, pages 299–310, 1998.

[25] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An Approximate L1-Difference Algorithm for Massive Data Streams. In *Proceedings of 40th FOCS Annual Symposium on Foundations of Computer Science*, pages 501–511, 1999.

[26] P. Flajolet and G. Martin. Probabilistic Counting Algorithms. *Journal of Computer and System Sciences*, 31:182–209, 1985.

[27] V. Ganti, S. Chaudhuri, and R. Motwani. Robust Identification of Fuzzy Duplicates. In *Proceedings of the 21st IEEE ICDE International Conference on Data Engineering*, 2005.

[28] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates Over Continual Data Streams. In *Proceedings of the 20th ACM SIGMOD International Conference on Management of Data*, pages 13–24, 2001.

[29] P. Gibbons and Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *Proceedings of the 17th ACM SIGMOD International Conference on Management of Data*, pages 331–342, 1998.

[30] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *Proceedings of the 27th ACM VLDB International Conference on Very Large Data Bases*, pages 79–88, 2001.

[31] L. Golab, D. DeHaan, E. Demaine, A. Lopez-Ortiz, and J. Munro. Identifying Frequent Items in Sliding Windows over OnLine Packet Streams. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Conference*, pages 173–178, 2003.

[32] L. Golab, S. Garg, and M. Ozsu. On Indexing Sliding Windows over On-Line Data Streams. In *Proceedings of the 9th EDBT International Conference on Extending Database Technology*, pages 712–729, 2004.

[33] L. Golab and M. Ozsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[34] M. Greenwald and S. Khanna. Space-Efficient Online Computation of Quantile Summaries. In *Proceedings of the 19th ACM SIGMOD International Conference on Management of Data*, pages 58–66, 2001.

[35] S. Guha, P. Indyk, M. Muthukrishnan, and M. Strauss. Histogramming Data Streams with Fast Per-Item Processing. In *Proceedings of the 29th ICALP International Colloquium on Automata, Languages and Programming*, pages 681–692, 2002.

[36] S. Guha, N. Koudas, and K. Shim. Data-Streams and Histograms. In *Proceedings of the 33rd ACM STOC Symposium on the Theory of Computing*, pages 471–475, 2001.

[37] S. Gunduz and M. Ozsu. A Web Page Prediction Model Based on Click-Stream Tree Representation of User Behavior. In *Proceedings of the 9th*

ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 535–540, 2003.

[38] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proceedings of the ACM SIGCOMM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 147–160, 1999.

[39] P. Haas, J. Naughton, S. Sehadri, and L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *Proceedings of the 21st ACM VLDB International Conference on Very Large Data Bases*, pages 311–322, 1995.

[40] X. Jiang and D. Mojon. Filtering Duplicate Publications in Bibliographic Databases. In *Proceedings of the 1st NDDL International Workshop on New Developments in Digital Libraries*, pages 79–88, 2001.

[41] C. Jin, W. Qian, C. Sha, J. Yu, and A. Zhou. Dynamically Maintaining Frequent Items over a Data Stream. In *Proceedings of the 12th ACM CIKM International Conference on Information and Knowledge Management*, pages 287–294, 2003.

[42] R. Karp, S. Shenker, and C. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[43] D. Lee and J. Hull. Duplicate Detection in Symbolically Compressed Documents. In *Proceedings of the 5th ICDAR International Conference on Document Analysis and Recognition*, pages 305–308, 1999.

[44] X. Lin, H. Lu, J. Xu, and J. Yu. Continuously Maintaining Quantile Summaries of the Most Recent N Elements over a Data Stream. In *Proceedings of the 20th IEEE ICDE International Conference on Data Engineering*, pages 362–374, 2004.

[45] D. Lopresti. Models and Algorithms for Duplicate Document Detection. In *Proceedings of the 5th ICDAR International Conference on Document Analysis and Recognition*, pages 297–300, 1999.

[46] W. Low, M. Lee, and T. Ling. A Knowledge-Based Approach for Duplicate Elimination in Data Cleaning. *Information Systems Journal*, 26(8):585–606, 2001.

[47] G. Lucca, M. Penta, and A. Fasolino. An Approach to Identify Duplicated Web Pages. In *Proceedings of 26th COMPSAC International Computer Software and Applications Conference*, pages 481–486, 2002.

[48] G. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th ACM VLDB International Conference on Very Large Data Bases*, pages 346–357, 2002.

[49] G. Manku, S. Rajagopalan, and B. Lindsay. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *Proceedings of the 18th ACM SIGMOD International Conference on Management of Data*, pages 251–262, 1999.

[50] Y. Matias, J. Vitter, and M. Wang. Dynamic Maintenance of Wavelet-Based Histograms. In *Proceedings of the 26th ACM VLDB International Conference on Very Large Data Bases*, pages 101–110, 2000.

[51] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proceedings of the 10th ICDT International Conference on Database Theory*, pages 398–412, 2005.

[52] A. Monge. Matching Algorithms within a Duplicate Detection System. *IEEE Data Engineering Bulletin*, 23(4):14–20, 2000.

[53] A. Monge and C. Elkan. An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records. In *Proceedings of ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 23–29, 1997.

[54] M. Reiter, V. Anupam, and A. Mayer. Detecting Hit-Shaving in Click-Through Payment Schemes. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, pages 155–166, 1998.

[55] Z. Tian, H. Lu, W. Ji, A. Zhou, and Z. Tian. An N-gram-Based Approach for Detecting Approximately Duplicate Database Records. *International Journal on Digital Library*, 3(4):325–331, 2002.

[56] K. Whang, B. Vander-Zanden, and H. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems*, 15:208–229, 1990.

[57] S. Ye, R. Song, J. Wen, and W. Ma. A Query-Dependent Duplicate Detection Approach for Large Scale Search Engines. In *Proceedings of the 6th Asia-Pacific Web Conference*, pages 48–58, 2004.

[58] Y. Zhu and D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of the 28th ACM VLDB International Conference on Very Large Data Bases*, pages 358–369, 2002.