Streaming Quotient Filter: A Near Optimal Approximate Duplicate Detection Approach for Data Streams

Sourav Dutta*
Max Planck Institute for Informatics, Germany
sdutta@mpi-inf.mpg.de

Ankur Narang
IBM Research, India
annarang@in.ibm.com

Suman K. Bera
IBM Research, India
sumanber@in.ibm.com

ABSTRACT

The unparalleled growth and popularity of the Internet coupled with the advent of diverse modern applications such as search engines, on-line transactions, climate warning systems, etc., has catered to an unprecedented expanse in the volume of data stored world-wide. Efficient storage, management, and processing of such massively exponential amount of data has emerged as a central theme of research in this direction. Detection and removal of redundancies and duplicates in real-time from such multi-trillion record-set to bolster resource and compute efficiency constitutes a challenging area of study. The infeasibility of storing the entire data from potentially unbounded data streams, with the need for precise elimination of duplicates calls for intelligent approximate duplicate detection algorithms. The literature hosts numerous works based on the well-known probabilistic bitmap structure, Bloom Filter and its variants.

In this paper we propose a novel data structure, *Streaming Quotient Filter*, (SQF) for efficient detection and removal of duplicates in data streams. SQF intelligently stores the *signatures* of elements arriving on a data stream, and along with an eviction policy provides *near zero* false positive and false negative rates. We show that the near optimal performance of SQF is achieved with a very low memory requirement, making it ideal for real-time memory-efficient de-duplication applications having an extremely low false positive and false negative tolerance rates. We present detailed theoretical analysis of the working of SQF, providing a guarantee on its performance. Empirically, we compare SQF to alternate methods and show that the proposed method is superior in terms of memory and accuracy compared to the existing solutions. We also discuss *Dynamic SQF* for evolving streams and the parallel implementation of SQF.

1. INTRODUCTION AND MOTIVATION

The tremendous spurt in the amount of data generated across varied application domains such as information retrieval, tele-communications, on-line transaction records, online climate prediction systems, and virus databases to name a few, has evolved data intensive computing methods into a major area of study both in the industry as well as in the research community. Managing and processing such enormous data constitutes a Herculean task, and is further compounded by the presence of duplicates or redundant data leading to wastage of precious storage space and compute efficiency. Removing duplicates from such data sources, to help improve performance of the applications, with real-time processing capacity of 1 Gbps or higher provides an interesting problem in this context. This paper deals with real-time detection and elimination of duplicates in streaming environments to alleviate the computational prohibitiveness of processing such huge data sources. A record arriving on the data stream is deemed redundant or duplicate if it has already occurred previously on the stream. Formally, this is referred to as the *data de-duplication*, *data redundancy removal*, (DRR) or the *intelligent compression* problem and the terms have been used interchangeably in this paper.

Eliminating duplicates forms an important operation in traditional query processing and Data Stream Management Systems [3], and several algorithms [21] in this context such as approximate frequency moments [2], element classification [24], and correlated aggregate queries [23] have been studied. The real-time nature of the de-duplication problem demands efficient in-memory algorithms, but the inability to store the whole stream (possibly infinite) makes exact duplicate detection infeasible in streaming scenarios. Hence, in most cases a fast approach at the expense of accuracy, but with a tolerable error rate, is acceptable. In this paper, we propose a novel and efficient algorithm to tackle the problem of approximate duplicate detection in data streams.

Consider a large tele-communication network generating call data records, (CDR), capturing vital information such as the caller number, callee number, duration of call etc. Errors in the CDR generation mechanisms might lead to duplicate data being generated. For downstream compute efficiency, before the storage of the CDRs in the central repository, de-duplication needs to be performed periodically over around 5 billion multi-dimensional record-sets at real-time. In such cases, solutions involving classical database access are prohibitively slow and typical Bloom Filter [7] based approaches [22] are extremely resource intensive.

Network monitoring and accounting provides an analysis of the network users and their usage behavioral patterns

*The work was done while the author was at IBM Research, India

Proceedings of the VLDB Endowment, Vol. 6, No. 8

which are widely used for recommender systems and personalized web search. Classification of users as new or existing and updating their profile information provides an interesting application of the DRR problem. Duplicate detection in click streams [33] also help to prevent fraudulent activities in the Web advertisement domain, wherein the site publisher fakes clicks to garner more profit from the advertiser. The detection of duplicate user ID, IP, etc., by the advertiser commission helps reduce such fraudulent activities.

Search engines periodically crawl the Web to update their corpus of extracted URLs. Given a list of newly extracted URLs, the search engine must probe its database archive to determine if the URL is already present in its database, or the contents need to be fetched [25]. With the exponential number of web pages, data de-duplication becomes indispensable in such scenarios. Imprecise duplicate detection may lead to already present pages being visited again (false negative, FN) thereby degrading performance, or may lead to new web pages being ignored (false positive, FP) making the corpus stale. Hence, a tolerable error rate on both fronts has to be maintained for a healthy performance of the application [12]. This poses a strong necessity of de-duplication algorithms that work in-memory, support real-time performance, as well as exhibit reasonably low FP and FN rates.

1.1 Contributions

In this paper, we propose *Streaming Quotient Filter*, (SQF) which extends and generalizes the Quotient Filter structure [5], and accordingly a novel algorithm for detecting duplicates in data streams. We further discuss the implementation of SQF on *parallel architectures* as well as on the *Map-Reduce* framework. We also propose *Dynamic SQF* for the case of evolving streams, wherein SQF adapts itself accordingly to provide a near-optimal performance. We analyze and prove bounds for the error rates of SQF and show that they are fast decaying and become *nearly zero* for large streams, thereby making SQF far superior than the existing methods. We also present the parameter setting issues along with empirical results on huge datasets, both real and synthetic, show-casing the efficiency of SQF in terms of convergence, low error rates and memory requirements.

1.2 Roadmap

Section 2 presents the precise problem statement along with a background study of the existing structures and approaches. The working principle of SQF is then presented in Section 3. Section 4 provides the detailed theoretical analysis on the error bounds of SQF. Section 5 provides a glimpse of the Dynamic SQF for evolving data streams, and the implementation of SQF on parallel architectures and on Map-Reduce framework. Experimental evaluation of the performance of SQF along with parameter settings are reported in Section 6. Section 7 finally concludes the paper.

2. PRELIMINARIES AND RELATED WORK

Consider the input data stream to be a sequence of elements, $S = e_1, e_2, \ldots, e_i, \ldots, e_N$, where N is the size of the stream which can potentially be unbounded. We assume that the elements of the stream are drawn uniformly from a finite alphabet set Γ with cardinality U, $|\Gamma| = U$. Each element of the stream can then be transformed into a number using hashing or *fingerprinting* method [37], and we later use this approach in our experimental set-up. Formally, the problem of de-duplication can be stated as: *given a data stream, S and a fixed amount of memory, M, report whether each element e_i in S has already appeared in $e_1, e_2, \ldots, e_{i-1}$ or not.* Since the storage of all the data arriving on the stream is infeasible, an approximate estimate minimizing the error is required.

Naïve approaches involving database and archive querying or pair-wise string comparison are prohibitively slow and may involve disk accesses defeating the real-time characteristic of the problem. Straightforward caching and buffering methods [21] involve populating a fixed-sized buffer with elements from the stream and checking the presence of each new element within the buffer. After the buffer becomes full, a new element may be stored by evicting an element from the buffer. Several replacement policies have been proposed in [21]. However, it can be observed that the performance of the buffering technique depends largely on the eviction policy adopted and the behavior of the stream. Hence, fuzzy duplicate detection mechanisms [6, 43] were proposed.

The problem of *bit shaving* to address fraudulent advertiser traffic was investigated in [38]. Approximate duplicate detection for search engines and Web applications were proposed in [10, 11, 31]. [1, 16, 42] uses file-level hashing in storage systems to detect duplicates, but suffers from a low compression ratio. Secure hashes were studied for fixed-sized data blocks in [36].

In order to address this computational challenge, Bloom Filters [7] are typically used in such applications [8, 9, 41, 14, 26, 30, 32, 15]. Bloom Filter provides a space-efficient probabilistic synopsis structure to approximately answer *membership queries* in sets. A Bloom Filter comprises a bit vector of m elements initially set to 0. Typical Bloom Filter approach involves comparison of k selected bits (by k hash functions) from the vector to decide if it is distinct or duplicate. Insertion of an element e_i also involves the setting of k bit positions of the Bloom Filter computed by k independent hash functions, $h_1(e_i), h_2(e_i), \ldots, h_k(e_i)$. However, the memory and compute efficiency is achieved at the expense of a small *false positive* rate, FPR, wherein a distinct element is falsely reported as duplicate. The probability of a false positive rate for a standard Bloom Filter is given by [34],

$$FPR \approx \left(1 - e^{-kn/m}\right)^k$$

where m is the number of bits in the Bloom Filter and n is the number of distinct elements seen so far. Bloom Filters were first used by the TAPER system [28]. Disk-based Bloom Filters have also been proposed [35], but suffer from overall performance degradation.

To support the situation where elements are continually inserted and deleted from the Bloom Filter structure, [18] introduces counting Bloom Filters. This approach replaces the bits of the filter by small counters maintaining the number of elements hashed to a particular bit position. The deletion of elements now gives rise to *false negative* rate, FNR, wherein a duplicate element is wrongly reported as unique. Further variants of the Bloom Filter model have been proposed to suit the various needs of modern applications. These include compressed Bloom Filter [34], space-code Bloom Filters [30], Decaying Bloom Filter [40], and spectral Bloom Filters [39] to name a few. Window model of Bloom Filters such as landmark window, jumping win-

dow [33], and sliding window [40] operating with a definite amount of stream history to drawn conclusions for the future elements of the streams have also been studied.

Bloom Filters have also been applied to network-related applications such as heavy flows for stochastic fair blue queue management [19] to assist routing , packet classification [4], per-flow state management and longest prefix matching [14]. For representation of multi-attribute items with a low false positive rate, a variant of the Bloom Filter structure was proposed by [27]. [26] extends *Bloomjoin* for distributed joins to minimize network usage in database statistic query execution. [32] discusses the use of Bloom Filters to speed-up the name-to-location resolution process in large distributed systems. Parallel versions of Bloom Filters were also proposed for multi-core applications [13, 22, 27]. A related problem of computing the number of distinct elements in a data stream was studied in [20].

Recently an interesting Bloom Filter structure, *Stable Bloom Filter*, (SBF) [12] provides a guarantee regarding the performance of the structure. It continuously evicts elements from the structure and provides a constant upper bound on the FPR and FNR. This constant performance and stability provides huge improvements in the real-time efficiency of de-duplication applications. However, SBF suffers from a high FNR and theoretically the convergence to stability is attained at infinite stream length. To alleviate the issues faced in SBF, [17] proposed *Reservoir Sampling based Bloom Filter*, (RSBF), a novel combination of reservoir sampling technique with the Bloom Filter structure.

The inherent problem of high FNR can be attributed to the fact that deletion of a bit in the Bloom Filter may result in the logical deletion of more than one element, i.e., all the elements mapped to that bit position. *Quotient Filter* [5] eliminated this problem using *quotienting technique* [29], allowing a Bloom Filter to be "deletion-friendly". However, it does not naturally support data stream queries.

In this paper, we propose the *Streaming Quotient Filter*, (SQF) for the de-duplication problem, providing near zero FPR and FNR with extremely low memory requirements as compared to the existing structures. We establish theoretical bound for the near-optimal performance of SQF, and also discuss the dynamic variant of SQF for evolving streams. Empirical results supporting the accuracy and efficiency of SQF are also presented in this paper. To the best of our knowledge, SQF provides the lowest FPR, FNR and memory constraints as compared to the prior approaches.

3. STREAMING QUOTIENT FILTER

In this section, we describe the *Streaming Quotient Filter*, (SQF) a memory-efficient data structure for data duplicate removal with extremely low error rates. The main working principle of SQF is the storage of multi-set of elements (in some data structure, F) by intelligent representation of the p-bit fingerprint of each element arriving on the data stream. We assume that each element of the stream, S is uniformly drawn from a finite universe, Γ of cardinality U and is hashed to a p-bit fingerprint using Rabin's method [37]. The fingerprint of the elements are provided as input to SQF. Conceptually, $F = h(S) = \{h(e) \mid e \in S\}$, where $h : \Gamma \rightarrow \{0, 1, \ldots, 2^p - 1\}$. Insertion of an element, e involves the insertion of $h(e)$ into F, and similarly, deletion of e involves the deletion of $h(e)$ from F. The membership query of e then reduces to checking whether $h(e) \in F$.

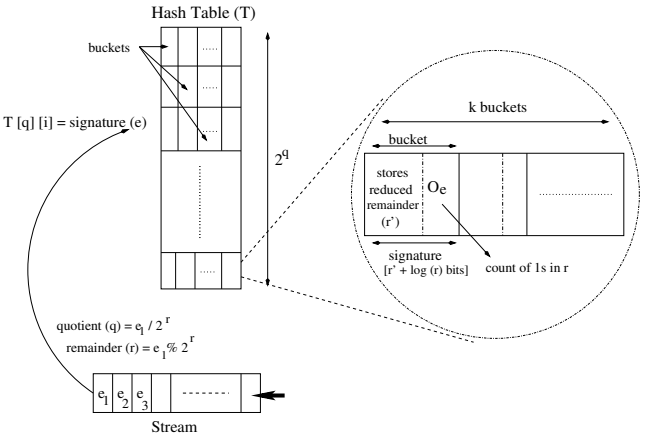


Figure 1: The structure of SQF.

We also compute and show that the amount of memory required by the SQF structure is extremely low making it an in-memory de-duplication structure.

3.1 Structure

SQF stores the p-bit fingerprint of each element in T, an open hash table with the number of rows, $R_T = 2^q$ (where $q < p$), by the *quotienting* technique suggested in [29]. In this method, the input fingerprint of an element e, f_e, is partitioned into its r least-significant bits, $f_e^r = f_e \ mod \ 2^r$ (remainder), and its $q = p - r$ most-significant bits, $f_e^q = \lfloor f_e / 2^r \rfloor$ (quotient). Insertion of the element e in SQF involves storing certain characteristics of f_e^r in the hash table T at row f_e^q; i.e., $T[f_e^q] = \sigma(f_e^r)$.

Each row of the hash table, T is further subdivided into k buckets, each storing the characteristics of the remainder of the element hashed into the particular row. For an element e, SQF stores two characteristics of the remainder of e.
(i) O_e, denoting the number of bits set to 1 in f_e^r, and
(ii) Assume Ω to be a function, $\Omega : \mathbb{R}^r \rightarrow \mathbb{R}^{r'}$, selecting $r'(< r)$ bit positions of f_e^r. The number formed by the r' chosen bits, $f_e^{r'}$ is termed as *reduced remainder*.

O_e along with the *reduced remainder*, $f_e^{r'}$ is henceforth referred to as the *signature* of the element e, σ_e. SQF stores the signatures of the input elements in the hash table, T. The structure of SQF is depicted in Figure 1.

Example: For example, assume the fingerprint ($p = 8$) of an element e, f_e, to be $(10100111)_2$ and $r = 4$. Hence, $q = p - r = 4$ with the hash table T containing $R_T = 2^q = 16$ rows, and let each row of T to consist of $k = 1$ buckets. Binary representation of decimal numbers are explicitly stated with their radix in this example. The quotient of f_e, $f_e^q = (1010)_2 = 10$ and the remainder, $f_e^r = (0111)_2$. Let Ω be a simple function selecting $r' = 2$ most-significant bits of f_e^r. Hence, the *reduced remainder*, $f_e^{r'}$ becomes $(01)_2$. The number of ones in f_e^r, $O_e = 3 = (11)_2$. The *signature* of the element e is $\sigma_e = 0111$, computed by appending O_e to $f_e^{r'}$ (both in their binary representation). Considering element e to be unique, σ_e is stored in a bucket in row $T[f_e^q] = T[10]$.

3.2 Algorithm

As discussed above, SQF stores the *signatures* from the remainders of the input element fingerprints arriving on the data stream in the buckets (Lines 19-23 of Algorithm 1) of the corresponding rows (obtained from the quotient of the fingerprints) in the hash table, T (Line 7 in Algorithm 1). The membership query for an element of the stream, thus, simply involves checking the presence of the signature of the element in the buckets of the corresponding row of the hash table (obtained as described above) (Lines 13-18 of Algorithm 1), providing an elegant yet efficient algorithm for duplicate detection.

If two fingerprints f and f' map to the same row of T, i.e., have the same quotient ($f^q = f'^q$), we refer to it as *soft collision* as in [5]. However, since each row of T contains only k buckets, when more than k distinct elements of the stream are mapped to the same row of T, we refer to it as *hard collision*. As such, the $(k+1)^{th}$ colliding element uniformly and randomly chooses an element signature (out of k stored signatures in the buckets) to be evicted and is stored in its place (Lines 21-22 of Algorithm 1). The pseudo-code for the working of SQF is presented in Algorithm 1.

Thus, SQF can easily be implemented with the help of a hash table (with buckets) and operates on bit-operations (quotienting) making it extremely fast, thereby catering to the real-time nature of the concerned problem.

3.3 Memory Requirements

We now consider the amount of memory space required by the SQF structure and show that it can easily be stored in-memory making the algorithm extremely efficient for real-time applications.

Let the fingerprints of the stream elements to be of p bits, q be the number of bits in the quotient, r' the number of bits in the reduced remainder (computed by Ω), and k to be the number of buckets in each row of the hash table T. Hence, the number of rows in T, $R_T = 2^q$ and $r = p - q$. Since each row of T contains k buckets, the total number of buckets in T is $k.R_T = k.2^q$. Each bucket stores the *signature* of the elements computed from the reduced remainder, $f_e^{r'}$ and the number of ones in the remainder, O_e.

$f_e^{r'}$ consists of r' bits selected by Ω, and the number of ones present in the remainder, O_e is upper bounded by the number of bits in the remainder, given by r. Hence, O_e can be represented by $\log r$ bits. Therefore, the signature of an element can be stored in $(r' + \log r)$ bits, which provides the size of each bucket in T. Thus, the total memory taken by SQF (in bits) is given by,

$$M = k.2^q.(r' + \log r) \quad (1)$$

Considering the example provided in Section 3.1, we have $p = 8$, $q = 4$, $r = 4$, $k = 1$, and $r' = 2$. Substituting these values in Equation (1), we observe that the memory consumed by SQF is,

$$M = 1.2^4.(2 + \log 4) = 16.(2 + 2) = 64\ bits$$

or 8 bytes only. Hence, with merely 8 bytes, SQF is able to represent $k.R_T = 1.2^4 = 16$ unique element signatures.

Empirically we later show (in Section 6) that even with such small memory requirements, SQF not only outperforms the existing approaches, but also provides near optimal performance in terms of error rates. This makes SQF an extremely memory-efficient and effective structure for de-duplication scenarios.

Algorithm 1: $SQF(S)$

Require: Stream (S), Number of bits in the fingerprint of elements (p), Memory available in bits (M), and Bit selection function (Γ)

Ensure: Detecting *duplicates* and *distinct* elements in S with low error rates

1: Set the parameters: Number of bits in the remainder ($r(< p)$), Number of buckets in each row of hash table (k), and Size of each bucket in bits (s_b)
2: Bits in quotient, $q \leftarrow p - r$. Create a hash table, T, with $R_T \leftarrow 2^q$ rows.
3: Each row is further divided into k buckets each of size s_b bits. Initialize T to *null* or empty.
4: **for** Each element e of the stream S **do**
5: $Result \leftarrow DISTINCT$
6: Let f_e be the p-bit fingerprint of e.
7: Quotient of f_e, $f_e^q \leftarrow \lfloor f_e / 2^r \rfloor$
8: Remainder of f_e, $f_e^r \leftarrow f_e \% 2^r$
9: $O_e \leftarrow$ number of ones in f_e^r.
10: Select $r'(< r)$ bit positions using function Γ.
11: The *reduced remainder* of e, $f_e^{r'} \leftarrow$ number formed by the r' selected bits.
12: Signature of element e, σ_e comprises O_e and $f_e^{r'}$.
13: **for** Each bucket, b_i in row f_e^q of T **do**
14: **if** Entry at $b_i = \sigma_e$ **then**
15: $Result \leftarrow DUPLICATE$
16: break
17: **end if**
18: **end for**
19: **if** $Result = DISTINCT$ **then**
20: Let b_{empty} be an empty bucket at row f_e^q of T ($T[f_e^q]$).
21: **if** b_{empty} does not exist, i.e., no empty buckets are found at $T[f_e^q]$ **then**
22: $b_{empty} \leftarrow$ a bucket uniformly and randomly selected from the k buckets in $T[f_e^q]$
23: **end if**
24: Store σ_e in bucket b_{empty}
25: **end if**
26: **end for**

4. THEORETICAL FRAMEWORK

This section presents the detailed theoretical analysis for the performance of SQF, and provides bounds on its error rates. The event of *false positive* (FP) occurs when a distinct element is mis-reported as duplicate. *False negative* (FN) occurs when a duplicate element is mistakenly considered as distinct. We show that SQF exhibits extremely low FP and FN rates, making it better than the existing approaches.

4.1 False Positive Rate

Let $\Pr(FP)_{m+1}$ represent the probability of occurrence of a false positive when the $(m+1)^{th}$ element, e_{m+1}, arrives on the stream. Such an event occurs when e_{m+1} is actually unique (had not occurred previously in the stream), but SQF contains an entry with the same *signature* as that of e_{m+1} in the corresponding row of the hash table T, leading to e_{m+1} being wrongly reported as duplicate. Since the elements of the stream are considered to be uniformly drawn from a

finite universe, Γ of cardinality U, the probability of e_{m+1} actually being unique is given by,

$$\Pr(e_{m+1}\ is\ unique) = \left(\frac{U-1}{U}\right)^m \tag{2}$$

We now compute the probability for the event of T containing the same signature as that of e_{m+1} in the corresponding row. e_{m+1} will thus be reported as false positive if all the following three conditions hold for another element, E which arrived earlier in the stream:
(i) E was hashed to the same *row* as that of e_{m+1} (i.e. quotient(E) = quotient(e_{m+1})).
(ii) E and e_{m+1} have the same *reduced remainder*.
(iii) The number of ones in the remainder is the same for both E and e_{m+1}.
Conditions (ii) and (iii) produce the same signatures for E and e_{m+1}.

Since the quotient (obtained as described previously) contains q bits and the number of rows in T, $R_T = 2^q$, the probability associated with condition (i) is given by,

$$P_q = \frac{1}{R_T} = \frac{1}{2^q} \tag{3}$$

Similarly, the reduced remainder obtained from function Ω contains r' bits. Hence, the probability of condition (ii) is,

$$P_{r'} = \frac{1}{2^{r'}} \tag{4}$$

The remainder of an element is represented by r bits. However, a remainder consisting only of 1s or 0s can be associated to only one element. Hence, no two elements can have the same remainder in such cases. We therefore consider the other cases, where an element contains $o \in [1, r-1]$ bits set to 1 in its remainder.

The probability that e_{m+1} has only one bit set to 1 out of its r remainder bits is given by $\binom{r}{1}/2^r$. For condition (iii) to occur, E must also contain exactly one bit set to 1 in its remainder, albeit at a different position to that of e_{m+1} (since $E \neq e_{m+1}$). The probability of this event is given by $\binom{r-1}{1}/2^r$. However, since o can range from $[1, r-1]$, the probability that E has the same number of ones in its remainder as that of e_{m+1} is given by,

$$\begin{aligned} P_{one} &= \sum_{i=1}^{r-1}\left(\frac{\binom{r}{i}}{2^r}.\frac{\binom{r-1}{i}}{2^r}\right) \leq \sum_{i=1}^{r-1}\left(\frac{\binom{r}{i}}{2^r}\right)^2 \\ &\leq \sum_{i=0}^{r}\left(\frac{\binom{r}{i}}{2^r}\right)^2 = \frac{\binom{2r}{r}}{2^{2r}} \end{aligned} \tag{5}$$

The r^{th} Catalan number has a value of,

$$\begin{aligned} C_r &= \frac{1}{r+1}.\binom{2r}{r} \approx \frac{4^r}{p^{3/2}.\sqrt{\pi}} \\ \therefore \binom{2r}{r} &\approx \frac{(r+1).4^r}{r^{3/2}.\sqrt{\pi}} \approx \frac{4^r}{\sqrt{\pi r}} \end{aligned} \tag{6}$$

Substituting Equation (6) in Equation (5), we have

$$P_{one} \approx \frac{4^r}{2^{2r}\sqrt{\pi r}} = \frac{1}{\sqrt{\pi r}} \tag{7}$$

Using Equations (3), (4), and (7), the probability that an element E exists in T at the same row and having the same signature as that of e_{m+1} (false positive event) is given by,

$$P_{dup} \approx \frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}} \tag{8}$$

Assume that E had last occurred on the stream at position l and was inserted into the hash table T. For e_{m+1} to be reported as a FP, E should not be deleted from T by the arrival of elements from $(l+1)^{th}$ to m^{th} position of the stream (denote event by ϕ). For the sake of simplicity, we assume that all the buckets of T are already filled. So for each element arriving after E, it will not be deleted from T (event ϕ) if either of the following three cases hold:
(i) The element maps to different row of T.
(ii) The element maps to the same row as that of E, but is a duplicate (i.e. any of the k buckets contains the same signature), and is hence not inserted.
(iii) The element maps to the same row as that of E and has a different signature than any of the buckets in the row, but randomly selects a bucket other than that containing E for eviction.

Hence the probability of the event ϕ using Equations (3), (4), (7), and (8) is given by,

$$\begin{aligned} P_{ndel} &= \prod_{i=l+1}^{m}\left[\left(1-\frac{1}{2^q}\right) + \frac{k}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}} + \frac{1}{2^q}.\left(1-\frac{1}{2^{r'}\sqrt{\pi r}}\right)^k \right. \\ &\qquad \left. .\left(1-\frac{1}{k}\right)\right] \\ &= \left[1-\frac{1}{2^q}+\frac{k}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}+\frac{1}{2^q}.\left(1-\frac{1}{2^{r'}\sqrt{\pi r}}\right)^k.\left(1-\frac{1}{k}\right)\right]^{m-l} \end{aligned} \tag{9}$$

However, the value of l can vary between $[1, m]$. Hence, the probability that e_{m+1} is reported as a false positive, $\Pr(FP)_{m+1}$ using Equations (2), (8), and (9) is,

$$\begin{aligned} \Pr(FP)_{m+1} &\approx \left(\frac{U-1}{U}\right)^m.\sum_{l=1}^{m}\left\{\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}.\left(1-\frac{1}{2^q}+\right.\right. \\ &\qquad \left.\left.\frac{k}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}+\frac{1}{2^q}.\left(1-\frac{1}{2^{r'}\sqrt{\pi r}}\right)^k.\left(1-\frac{1}{k}\right)\right)^{m-l}\right\} \\ &\approx \left(\frac{U-1}{U}\right)^m.\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}.\left(1-\frac{1}{2^q}+\right. \\ &\qquad \left.\frac{k}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}+\frac{1}{2^q}.\left(1-\frac{1}{2^{r'}\sqrt{\pi r}}\right)^k.\left(1-\frac{1}{k}\right)\right) \end{aligned} \tag{10}$$

(ignoring higher order terms)

From Equation (10) it can be observed that the FPR encountered by SQF is extremely small. With increase in stream length, m, the factor $\left(\frac{U-1}{U}\right)^m$ becomes nearly 0, producing near-optimal results. For smaller values of m, the inverse exponential factors of q and r' dominate making the FPR extremely low.

With increase in the number of bits in the quotient (q) and stream length (m), the FPR of SQF exponentially decreases and provides near optimal results. The effect of the number of buckets (k) on FPR can be considered to be insignificant as it is suppressed by the $2^{q+r'}$ factor (Equation (10)). Later, we provide empirical results showing the FPR of SQF to be nearly 0 with low memory requirements.

4.2 False Negative Rate

In this section, we compute the false negative rate of SQF. A *false negative* occurs when a duplicate element is misreported as distinct. Let $\Pr(FN)_{m+1}$ denote the probability of the $(m+1)^{th}$ element on the stream, e_{m+1} being reported as a false negative. Such an event occurs when the element e_{m+1} had previously arrived on the stream and its signature was stored in the hash table T, but was subsequently deleted by the eviction policy of SQF (due to hard collisions). Also, no other element, E having the same quotient and signature arrived after the deletion of e_{m+1} from T.

Let event ϕ represent the last occurrence of element e_{m+1} on the stream at position l, i.e. $e_l = e_{m+1}$. Assuming that the stream elements are drawn uniformly from a finite universe, Γ with cardinality U, the probability associated with event ϕ is,

$$\Pr(e_{m+1} \text{ at } l) = \frac{1}{U}.\prod_{i=l+1}^{m}\frac{U-1}{U} = \frac{1}{U}.\left(\frac{U-1}{U}\right)^{m-l} \quad (11)$$

Similar to the analysis of FPR, for the sake of simplicity we assume that all the buckets of T are already occupied at the time of event ϕ. The signature of e_{m+1} will be evicted from T by another element, X in the following conditions:
(i) The element, X maps to the same row as that of e_{m+1},
(ii) The signature of X is not present in the buckets of T at that row, and
(iii) The bucket containing the signature of e_{m+1} is selected for deletion.

Using Equations (3) and (8), the probability of the above events occurring simultaneously is given by,

$$P_{del} = \frac{1}{k.2^q}.\left(1-\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}\right)^k \quad (12)$$

Assume e_{m+1} to be deleted by a stream element at position p. For the event of false negative to occur for e_{m+1}, no element E arriving after position p should have the same quotient and signature as that of e_{m+1}. The associated probability using Equation (8) is given by,

$$P_E = \prod_{i=p+1}^{m}\left(1-\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}\right) = \left(1-\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}\right)^{m-p} \quad (13)$$

Hence, the probability that e_{m+1} was deleted at position p of the stream and no element E having the same quotient and signature arrived subsequently can be obtained by combining Equations (12) and (13). However, the value of p can vary between $[l+1, m]$, as e_{m+1} can be deleted by any of the stream elements arriving at these positions. Using Equations (12) and (13), the probability of e_{m+1} being reported as false negative (given its last occurrence was at position l) is,

$$P_{FN} = \sum_{p=l+1}^{m}\left[\frac{1}{k.2^q}\left(1-\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}\right)^k\left(1-\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}\right)^{m-p}\right]$$
$$= \sum_{p=l+1}^{m}\left[\frac{1}{k.2^q}.\left(1-\frac{1}{2^{q+r'}}\frac{1}{\sqrt{\pi r}}\right)^{m-p+k}\right]$$
$$\therefore P_{FN} \approx \frac{1}{k.2^q}.\left(1-\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}\right)^k \quad (14)$$
(ignoring higher order terms)

The last occurrence of e_{m+1} was considered to be at position l, which can again vary from $[1, m]$. Hence, the probability that e_{m+1} is reported as a false negative, $\Pr(FN)_{m+1}$, is given by combining Equations (11) and (14) as,

$$\Pr(FN)_{m+1} = \sum_{l=1}^{m}\left[\frac{1}{U}.\left(\frac{U-1}{U}\right)^{m-l}.P_{FN}\right]$$
$$\approx \sum_{l=1}^{m}\left[\frac{1}{U}.\left(\frac{U-1}{U}\right)^{m-l}.\frac{1}{k.2^q}.\left(1-\frac{1}{2^{q+r'}}.\frac{1}{\sqrt{\pi r}}\right)^k\right] \quad (15)$$

It can be observed from Equation (15) that for large stream lengths (m), the FNR exhibited by SQF is nearly *zero* due to the factor $\left(\frac{U-1}{U}\right)^{m-l}$. Similarly, for small values of m, the factor $\frac{1}{2^q}$ dominates and produces extremely low FNR.

Equation (15) states that FNR increases exponentially as the number of reduced remainder bits (r') increases and the number of bits in the quotient (q) decreases. The effects of r, the number of bits in the remainder of an element, on the FNR of SQF can be considered nearly insignificant. FNR is also seen to decrease with the increase in k, the number of buckets in each row of the hash table T. We later present empirical results to validate our claims.

5. GENERALIZED SQF

In this section, we describe a variant of SQF, *Dynamic SQF* for handling evolving data streams. We also discuss the working of SQF on parallel architectures and on the Map-Reduce framework. We argue that with parallel and distributed settings, SQF can efficiently manage petabytes of data with relatively low memory (in the order of hundreds of gigabytes) while keeping the error rate nearly zero.

5.1 Dynamic SQF

Previously, we presented a thorough analysis of SQF assuming a uniform data stream model. However, for certain applications such as news reports, weather systems etc., the stream evolves with time. That is, as the stream advances the distribution of the elements (still considered to be drawn from a finite universe) on the stream changes. To this end, we propose the *Dynamic Streaming Quotient Filter*, DSQF.

DSQF is based on on-demand dynamic memory allocation to handle the changes in the stream behavior. As such, in DSQF the hash table T (of SQF) is replaced by a vector of 2^q pointers, V. DSQF also computes the quotient and the signature of the input element, similar to that of SQF. When an element is to be inserted in row f_e^q (in T), a bucket is dynamically allocated and is attached to the pointer in the corresponding position of the vector V. Deletion of an element involves the removal of the bucket in which the signature of the element was stored and returning the freed space to the global pool of unallocated memory. Membership queries, similar to the procedure of SQF, are handled by checking the buckets chained to the pointer corresponding to the vector position in which the element is mapped.

When the entire available free space is allocated (i.e. no more buckets can be created), the condition of *hard collision* arises and an eviction policy is triggered. Each position of the vector now maintains the time-stamp of its last access. DSQF uses the *least recently used*, LRU eviction policy. It uniformly and randomly selects a bucket from the least recently accessed position of V, allocates it to the pointer corresponding to the quotient of the new element, and stores

the new signature. This enables DSQF to intelligently distribute the buckets among the different positions of V to capture the evolving nature of the stream.

Each bucket in DSQF also contains an addition field to store the memory location of the subsequent bucket, if any. This leads to a slight increase in the memory requirements of DSQF as compared to SQF. However, given the memory-efficiency of SQF, this increase in memory space does not degrade the performance of DSQF compared to the existing methods. As the stream evolves, the allocation of buckets corresponding to the various positions of the vector V dynamically change to reflect the most recent history of the stream. Hence, DSQF efficiently adapts itself to handle evolving streams, exhibiting near-optimal performance as before, albeit with a small increase in memory requirements.

5.2 Parallel Implementation

In this section, we present the parallel working model of SQF for multi-core and multi-node architectures. We show that SQF (and similarly DSQF) is easily parallelizable, making it suitable even for applications with enormous amounts of data. This enables SQF to efficiently handle deduplication of petabytes of data using low memory space (hundreds of gigabytes).

Assuming that the parallel setting consists of P processors, the input data stream is partitioned into blocks of C elements. The elements of a block, b_i are distributed among the P processors along with their position, *pos* in b_i. Hence, each processor receives $\beta = C/P$ elements of block b_i for deduplication. The hash table T storing the signature of the elements is evenly distributed among the P processors, with each node containing R_T/P consecutive rows of T. We consider each processor to internally run two parallel threads, t_1 and t_2, which can easily be generalized to $\alpha(\geq 2)$ threads.

For the C elements of batch b_i, let each node N_j receive element fingerprints $e_{j\beta}, e_{j\beta+1}, \ldots e_{(j+1)\beta-1}$, where $j \in [0, P-1]$. Thread t_1^j in node N_j computes the quotient of the input elements, e_l^j. Depending on the quotient obtained for the element, N_j transfers e_l^j along with its quotient and position *pos* to the node N_q containing the rows of T corresponding to the quotient of e_l^j, i.e. the node in which the particular row of T is stored.

The second thread t_2^q of node N_q now computes the signature of e_l^j. Checking the membership of the element in T involves accessing the buckets of T in the row corresponding to the quotient computed. However, if the membership query proceeds as before, the accuracy of the algorithm may degrade. For example, consider $C = P = 3$ and the elements of a block to be $\{x, y, y^*\}$, where $y = y^*$, (but the arrival time of y^* is greater than that of y), and y has not occurred previously in the stream. In accordance with the problem statement mentioned in Section 2, y should be reported as *distinct* and y^* as *duplicate*. However, in case node N_3 (allocated both y and y^* as $quotient(y) = quotient(y^*)$) containing the corresponding row of T processes y^* before y, it will report y^* as distinct and subsequently y as duplicate.

To alleviate this problem, each bucket of T now contains an extra field, *pos*, capturing the position of the stored element within the block. Consider, node N_q to query the buckets of T for membership of e_l^j. If the signature of e_l^j is absent in T, N_q stores the signature in a bucket, B and sets the *pos* field to $pos_{e_l^q}$. So, when the signature of a later

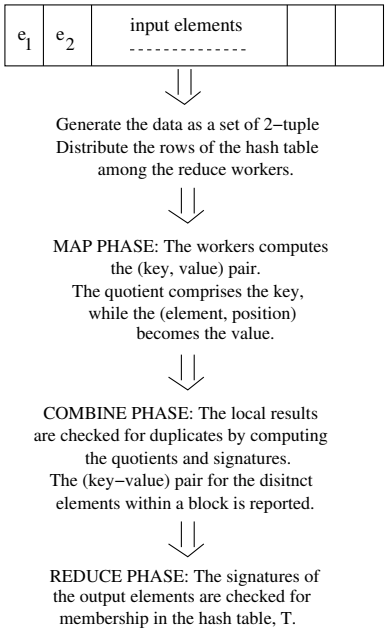


Figure 2: SQF on Map-Reduce Framework

element, $E = e_l^j$ is found in T, N_q checks the *pos* field of the bucket, B. Observe that if $pos_E > B_{pos}$, then E had indeed occurred in the stream after e_l^j and is hence correctly reported as duplicate by N_q. However, $pos_E < B_{pos}$ indicates that E had occurred earlier on the stream but was processed later. N_q in this case reports e_l^j to be duplicate and updates B_{pos} to pos_E. After the entire block is processed, elements corresponding to the *pos* fields of the buckets accessed are reported as distinct, and the next block of elements are fetched.

For the example considered above, assume N_3 processes y^* first. Since its signature is not found in T, N_3 stores it in bucket B and sets B_{pos} to 3, as y^* is the third element of the block. When y is later processed, its signature is found to be present in T. Now, N_3 checks the *pos* field of B, and since $pos_y(= 2) < B_{pos}(= 3)$ infers correctly that y had occurred on the stream before y^* (the stored one). Hence, N_3 reports the third element (since $B_{pos} = 3$) to be duplicate and updates B_{pos} to 2. After all the elements of the block have been processed, the *pos* field of B is set to 2. The algorithm now reports the second element of the block, y (as $B_{pos} = 2$) to be distinct.

5.3 SQF on Map-Reduce Architecture

We now discuss a simple implementation of SQF (and similarly DSQF) on the popular *Map-Reduce* framework.

Assume that the Map-Reduce framework contains m map workers and r reduce workers. As discussed in the previous section, the input data stream is partitioned into blocks of C elements each. Data generation for the map phase involves the construction of a set of 2-tuples consisting of the input element fingerprint (f) and its position in the block (*pos*), $D = \{(f_1, pos_{f_1}), (f_2, pos_{f_2}), \ldots, (f_C, pos_{f_C})\}$. The input data D is then distributed evenly among the m map workers by the *splitter* function.

In the *map* phase, each worker computes the quotient of

its input elements (as described previously), and generates a *key-value* pair for each element. The quotient computed is considered as the key, while the element along with its position constitutes the value.

The *combine* phase of the framework, consumes the local data produced by each map worker. It calculates the signatures of the elements (from the value field) sharing the same quotient (key field). For elements with the same signature and quotient, the key-value pair of the element having the lowest position value (obtained from the value field) is emitted as the result, while the other elements with a higher position value are reported as duplicate.

Similar to the implementation discussed in Section 5.2, the rows of the hash table T are distributed among the reduce workers. In the *reduce* phase, each reduce worker consumes the key-value pair produced from the combine phase and computes the signature of the elements.

For elements having the same quotient and signature, their position values are again compared to find the element with the lowest position value, E_{low}. The other elements with the same signature are reported as duplicates, and E_{low} is checked if present in T or not. If the signature of E_{low} is found, it is reported as duplicate, else is categorized as distinct and stored in T. The next block of elements re-executes the framework on the updated T. The flowchart for implementation of the parallel version of SQF on MapReduce is shown in Figure 2.

6. EXPERIMENTS AND OBSERVATIONS

In this section, we experimentally evaluate the performance of *SQF* as compared to the prior approaches. The *Stable Bloom Filter* (SBF) exhibits the lowest false positive rate (FPR), while the *Reservoir Sampling based Bloom Filter* (RSBF) attains the lowest false negative rate (FNR) and fastest convergence to stability. Hence, it is against these two state-of-art structures that we compared the error rates of *Streaming Quotient Filter* (SQF), using both real as well as synthetic datasets.

The real dataset of *clickstream data*[1] contains more than 3 million records, while the synthetic datasets were generated using Zipfian as well as uniform distribution having 1 billion entries. We also vary the percent of distinct elements in the synthetic datasets to capture varied streaming scenarios.

Initially, we discuss the setting of the parameters r, r' and k for enhanced performance of SQF. Using these settings, we then perform experiments to capture variations of FPR and FNR with:

- Number of input records,
- Percentage of distinct elements, and
- Memory requirements.

We also measure the query execution time of the algorithms.

6.1 Setting of Parameters

Here, we empirically explore the optimal setting of the parameters for SQF, namely:
(a) Number of buckets in each row of the hash table T (k),
(b) Number of bits in the remainder (r), and
(c) Number of bits in the reduced remainder (r')

[1]obtained from `http://www.sigkdd.org/kddcup/index.php?section=2000&method=data`

A uniformly randomly generated synthetic dataset of 1 billion entries with 15% distinct records has been used.

Figure 3(a) depicts the effect of the number of buckets (k) in each row of the hash table, T on the FPR and FNR of SQF, with the number of bits in the remainder, r set to 2. FPR is observed to be nearly independent of k, as shown by Equation (10) in Section 4.1. However, we find that FNR decreases sharply with increase in k (Equation (15)) and approaches 0 for $k = 4$. When k decreases, the number of *hard collisions* increases and more elements are evicted from T, leading to a rise in the false negative events.

Figure 3(b) shows the variation of FPR and FNR in SQF, with sufficient memory, when the number of bits in the remainder (r) of the input element is changed. FNR is seen to remain nearly constant with change in r, as discussed in Section 4.2. However, FPR increases as r is increased. This can be attributed to the fact that with increase in r, the number of bits in the quotient (q) decreases, as $q = p - r$, leading to an increase in the *soft collisions* in T. This decreases the spread of data in the hash table and also increases the probability that two elements have the same number of ones in their remainder, both leading to a higher FPR.

Figure 3(c) portrays the effect of the number of bits in the reduced remainder (r') on the performance of SQF when memory is kept constant at 64 MB and $r = 7$. We find that as r' increases the FPR decreases, since the probability of occurrence of the same signature for two or more elements decreases. However, with increase in r', FNR increases (shown by Equation (15)) as the bucket size increases, and thus the number of buckets decreases leading to more elements being evicted from the hash table T due to *hard collisions*. Hence, to minimize both FPR and FNR, r' should be set to $r/2$.

From the above observations, we find that SQF performs best when the parameters are set as: $r = 2$, $k = 4$ and $r' = r/2 = 1$. In the rest of the section, we use these parametric values for our experimental set-up. Without loss of generality, we assume the function Ω to select the most-significant bit of the remainder as the reduced remainder for obtaining the signature of an element.

6.2 Detailed Analysis

We now present the detailed results of the SQF algorithm as compared to that of the competing methods, for both real and synthetic datasets.

6.2.1 Real Datasets

Table 1 tabulates the results for the real clickstream datasets. We use two such datasets, $R1$ and $R2$, with nearly 3 million and 300,000 records respectively. We vary the amount of memory space provided to the de-duplication structures, and find that with increase in memory, the performance of all the structures improve.

For the $R1$ dataset, initially, at an extremely low memory of 64 bytes only, SQF provides comparable FNR as that of SBF (RSBF exhibits the lowest), but has 2x more FPR than SBF. However, as the memory is increase to 512 bytes, SQF exhibits near-zero) error rates, with FPR and FNR both in the order of 10^{-4}. SBF and RSBF with 512 bytes memory attains an FPR of 1.03% and 1.69% respectively, and an FNR of 29.76% and 21.16% respectively. SQF thus converges quickly to produce near *zero* FPR and FNR even at low memory, providing more than 10000× improvement over SBF and RSBF, for FPR and FNR both.

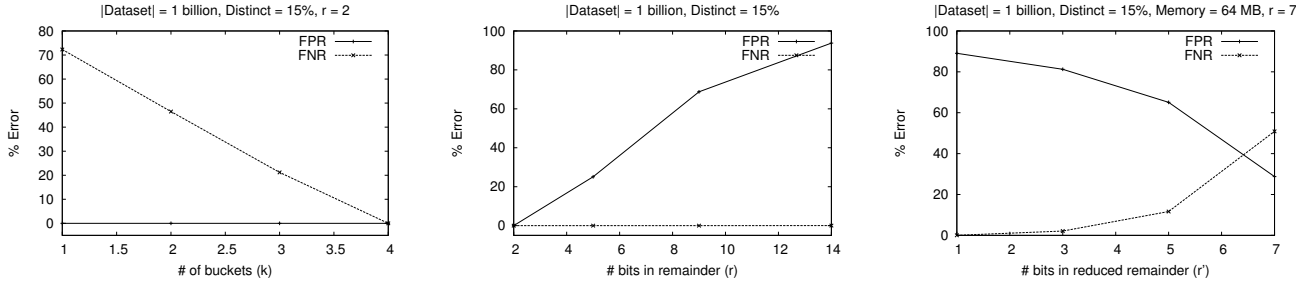


Figure 3: FPR and FNR versus parameter (a) k (b) r (c) r'

Table 1: Performance Comparisons on Real Clickstream Datasets

Datasets	Memory	FPR (%)			FNR (%)		
		SBF	*RSBF*	*SQF*	*SBF*	*RSBF*	*SQF*
|R1| = 3367020	64 B	6.88	8.75	12.72	61.36	47.06	60.54
	256 B	2.11	3.56	8.22	40.16	30.36	40.70
	512 B	1.03	1.69	0.0001	29.76	21.16	0.0005
	1024 B	0.22	0.38	0.00006	19.01	13.82	0.0001
|R2| = 358278	64 KB	1.23	2.01	4.19	13.25	8.86	12.71
	128 KB	0.41	1.15	0.0001	7.92	4.73	0.0001
	256 KB	0.02	0.32	0.00004	3.37	1.46	0.00005

For the dataset $R2$ having more than 300,000 records, we observe a similar behaviour of SQF and the other structures. SQF converges to FPR and FNR in the order of 10^{-4} at 128 KB memory, while SBF and RSBF exhibits 0.41% and 1.15% respectively in FPR and 7.92% and 4.73% in FNR respectively. For $R2$ also, SQF provides nearly 1000× performance improvements over the competing methods. However in $R2$, the convergence to near optimality for SQF occurs at a much higher memory requirement of 128 KB (as compared to $R1$), as the dataset contains larger click counts consolidated over a longer period of time.

Hence, we observe that for varying real datasets, SQF far outperforms the other structures and exhibit near *zero* FPR and FNR with low memory requirements.

6.2.2 Synthetic Datasets

In this section, we benchmark the performance of SQF against huge datasets generated randomly using uniform and Zipfian distributions. To simulate the observed deviations in real streams, we vary the percentage of distinct elements present in the datasets. Table 2 reports the performance of SQF and the other algorithms on the uniform datasets with varying memory, while Table 3 shows the results for the Zipfian dataset. In general, for all the datasets we observe that initially the FPR of SQF is high (2x) compared to SBF, with comparable FNR, at very low memory space. But with a slight increase in memory, SQF converges quickly to produce near optimal (near-zero) error rates.

For the 1 billion dataset with 15% distinct element, at only 64 MB memory SQF exhibits an FPR of 0.0031%, while SBF has an FPR of 2.92%. Hence, SQF provides nearly 1000× performance improvement. Similarly, with 64 MB memory, SQF produces 0.0042% FNR, which is nearly 10000× better than that of RSBF, which attains 43.17% FNR.

SQF requires 128 MB to converge for the dataset with 695 million records and 60% distinct elements as shown in Table 2. SQF exhibits 0.004% FPR, while SBF and RSBF attains 4.31% and 6.73% FPR respectively at 128 MB. However, for even lower memory space, SQF obtains an FPR 2x worse than that of SBF and FNR comparable to RSBF.

The third synthetic dataset contains 100 million records with 90% distinct elements (Table 2). We observe similar performance of SQF as discussed above. When 32 MB of memory is allocated, SQF has an FPR and FNR of around 0.003%, while SBF (and RSBF) suffers from 4.13% (and 6.04%) FPR and 48.88% (and 30.29%) FNR. Thus, SQF performs superior to SBF and RSBF both in terms of FPR and FNR, giving around 1000× and 10000× performance improvements respectively, providing near *optimal* results. Further, the membership query execution time of SQF is better or comparable to the existing approached, making it extremely efficient for real-time applications. The query time slightly increases with memory for all the algorithms.

SQF exhibits similar behavior for the Zipfian distributed synthetic datasets as shown in Table 3.. We observe that for the 1 billion dataset with 15% distinct elements, SQF performs better than both SBF and RSBF in terms of FPR, FNR and query time. At 1 MB memory, the performance of SQF is comparable to that of SBF for FPR and to that of RSBF in terms of FNR. However, as the memory is increased to 8 MB, SQF quickly converges to near-zero performance. At 8 MB, SQF has an FPR and FNR of 0.008% both, while SBF has 0.91% FPR and 1.41% FNR and RSBF exhibits 1.29% and 0.31% FPR and FNR respectively. Hence, SQF provides a performance improvement of around 100× in FPR and around 15× as compared to the other algorithms.

Similarly, for the 695 million dataset with 60% distinct elements, SQF attains near-zero error rates at around 8 MB

memory, while SBF shows 1.03% FPR and 17.08% FNR and RSBF produces 2.16% FPR and 10.52% FPR and FNR respectively. The query execution time for SQF is also better than that of SBF and RSBF for varying memory sizes. It can thus be seen that SQF has an extremely quick convergence rate to produce nearly *zero* FPR and FNR with low memory requirements for various synthetic datasets. This makes SQF a far better de-duplication structure than the existing ones in terms of FPR, FNR, convergence to stability, and also memory space at the same time catering to low real-time processing time demands.

Figure 4 studies the effect of stream length (or dataset size) on the performance of SQF. Figure 4(a) depicts the FPR obtained by SBF, RSBF, and SQF for varying dataset sizes with 60% distinct element and 256 MB available memory. We observe that the FPR of SBF and RSBF increases with stream length. However, the performance of SQF remains stable producing near *zero* FPR. This validates the near-optimal performance of SQF for large data streams.

Further, Figure 4(b) portrays the variation of FNR with stream length for 15% distinctness and 64 MB memory space. We find that SQF again outperforms SBF and RSBF, providing nearly *zero* FNR. The FNR obtained in SBF and RSBF is much higher ($> 40\%$) as compared to that of SQF. This makes SQF an extremely efficient structure for de-duplication exhibiting near *zero* FPR and FNR. Similar results are observed for the Zipfian datasets.

To study the memory requirement of SQF better and compare it with the other methods, we simulate an application with threshold FPR and FNR of 2% and 10% respectively. A uniform synthetic dataset of 695 million records with varying percentage of distinct elements is used to portray the memory space required to honor the error thresholds of the application. The findings are reported in Figure 5.

From Figures 5(a) and 5(b) we observe that for 30% distinct elements, SQF uses 50 MB memory for obtaining the threshold FPR while SBF and RSBF require more than 100 MB space. Similarly for attaining the threshold FNR, SQF requires 75 MB while SBF and RSBF both use more than 400 MB memory space.

With increase in the percentage of distinct elements, the gap in the memory requirement for SQF and the other structures increases. For 90% distinctness, we find that SBF and RSBF require around 450 MB to obtain the threshold FPR, while SQF requires only 250 MB (Figure 5(a) and 5(b)). SQF thus consumes around $2\times$ less memory compared to the prior approaches, making it an extremely memory-efficient.

7. CONCLUSIONS

Real-time in-memory data de-duplication in streaming scenarios poses a very challenging problem given the vast amounts of data stored from varied applications. In this paper we have presented a novel algorithm based on *Streaming Quotient Filter*, SQF structure catering to all the demands of such applications. SQF using simple hash table structure and bit operations provides improved FPR and FNR compared to the existing approaches. In fact, SQF exhibits *near optimal* error rates. We also theoretically prove the bounds on the error rates guaranteed by SQF.

We also presented *Dynamic SQF*, DSQF for evolving streams and provided a basic parallel framework for the implementation of SQF to cater to applications working in distributed environment. Empirical results depict SQF to be far superior to the competing methods both in terms of error rates as well as in memory requirement. SQF demonstrates near *zero* FPR and FNR with far less memory requirements for huge datasets of 1 billions records. SQF exhibits nearly $1000\times$ and $10000\times$ performance improvements in terms of FPR and FNR respectively, while using $2\times$ less memory space. This makes SQF an extremely effective, attractive and memory-efficient structure in the domain of duplicate detection. To the best of our knowledge SQF is the first such structure to exhibit *near zero* FPR and FNR.

Further study and empirical analysis of DSQF along with a complete architectural design and implementation for parallelizing SQF and DSQF provides an exciting direction of future work, leading to advancements in de-duplication.

8. ACKNOWLEDGMENTS

The authors would like to thank Abhinav Srivastav of IBM Research, India for his valuable suggestions pertaining to the parallel and Map-Reduce implementation of SQF.

9. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, R. J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, pages 1–14, 2002.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, pages 20–29, 1996.

[3] B. Babcock, S. Singh, and G. Varghese. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

[4] F. Baboescu and G. Varghese. Scalable packet classification. In *SIGCOMM*, pages 199–210, 2001.

[5] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *VLDB*, 5(11):1627–1637, 2012.

[6] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, pages 39–48, 2003.

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

[8] A. Z. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[9] Y. Chen, A. Kumar, and J. Xu. A new design of bloom filter for packet inspection speedup. In *GLOBECOMM*, pages 1–5, 2007.

[10] 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.

[11] J. Conrad, X. Guo, and C. Schriber. Online duplicate document detection: Signature reliability in a dynamic retrieval environment. In *CIKM*, pages 443–452, 2003.

[12] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *SIGMOD*, pages 25–36, 2006.

Table 2: Performance Comparisons on Synthetic Datasets (Uniform Random)

Datasets (% Distinct)	Memory	FPR (%)			FNR (%)			Query Time (μsec)		
		SBF	*RSBF*	*SQF*	*SBF*	*RSBF*	*SQF*	*SBF*	*RSBF*	*SQF*
1 billion (15 %)	8 MB	9.06	18.69	11.74	84.38	69.76	84.45	1.07	0.80	0.76
	32 MB	5.22	8.93	11.96	72.63	55.81	61.08	1.07	0.81	0.77
	64 MB	2.92	4.29	0.0031	60.89	43.17	0.0042	1.08	0.83	0.79
	128 MB	1.26	1.61	0.0006	45.96	29.55	0.001	1.10	0.86	0.81
	512 MB	0.13	0.14	0.0001	17.13	10.20	0.0003	1.12	0.87	0.86
695 million (60 %)	32 MB	8.64	17.26	13.91	79.16	62.82	72.70	1.10	0.84	0.80
	64 MB	6.77	12.24	12.00	70.83	52.89	53.53	1.12	0.85	0.80
	128 MB	4.31	6.73	0.004	57.97	39.27	0.003	1.14	0.87	0.81
	256 MB	2.09	2.77	0.001	42.08	25.76	0.001	1.15	0.90	0.85
	512 MB	0.82	0.92	0.0003	24.29	15.20	0.0004	1.15	0.92	0.88
100 million (90 %)	8 MB	8.83	17.30	13.92	75.02	57.25	70.70	1.07	0.83	0.72
	16 MB	6.83	11.92	12.05	64.03	44.82	49.76	1.08	0.84	0.72
	32 MB	4.13	6.04	0.003	48.88	30.29	0.0032	1.08	0.89	0.76
	64 MB	2.05	3.17	0.0006	29.61	20.38	0.0007	1.09	0.91	0.77
	128 MB	0.64	1.12	0.0001	16.20	11.45	0.0001	1.09	0.92	0.80

Table 3: Performance Comparisons on Synthetic Datasets (Zipfian)

Datasets (% Distinct)	Memory	FPR (%)			FNR (%)			Query Time (μsec)		
		SBF	*RSBF*	*SQF*	*SBF*	*RSBF*	*SQF*	*SBF*	*RSBF*	*SQF*
1 billion (15 %)	1 MB	2.38	3.72	1.02	4.19	1.45	1.03	0.30	0.19	0.09
	2 MB	1.98	3.00	0.83	3.75	1.05	0.92	0.32	0.20	0.11
	4 MB	1.29	2.42	0.09	2.56	0.71	0.05	0.35	0.27	0.17
	8 MB	0.91	1.29	0.008	1.41	0.14	0.008	0.40	0.31	0.20
	16 MB	0.38	0.92	0.0009	0.85	0.05	0.001	0.42	0.36	0.15
695 million (60 %)	512 KB	4.13	6.72	4.29	37.41	25.07	23.61	0.46	0.19	0.13
	1 MB	3.75	5.24	3.16	32.81	21.76	19.41	0.49	0.24	0.15
	2 MB	2.99	4.19	0.24	27.31	18.48	0.89	0.49	0.34	0.22
	4 MB	2.18	3.74	0.05	21.56	15.63	0.08	0.52	0.35	0.26
	8 MB	1.03	2.16	0.008	17.08	10.52	0.004	0.57	0.39	0.30

[13] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[14] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using bloom filters. In *ACM SIGCOMM*, pages 201–212, 2003.

[15] P. C. Dillinger and P. Manolios. Bloom filters in probabilistic verification. In *FMCAD*, pages 367–381, 2004.

[16] F. Douglis, J. Lavoie, J. M. Tracey, P. Kulkarni, and P. Kulkarni. Redundancy elimination within large collections of files. In *USENIX*, pages 59–72, 2004.

[17] S. Dutta, S. Bhattacherjee, and A. Narang. Towards "intelligent compression" in streams: A biased reservoir sampling based bloom filter approach. In *EDBT*, pages 228–238, 2012.

[18] L. Fan, P. Cao, J. Almeida, and Z. Broder. Summary cache: a scalable wide area web cache sharing protocol. In *IEEE/ACM Transaction on Networking*, pages 281–293, 2000.

[19] W. Feng, D. Kandlur, D. Sahu, and K. Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *IEEE INFOCOM*, pages 1520–1529, 2001.

[20] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *Computer and System Science*, 31(2):182–209, 1985.

[21] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 1999.

[22] V. K. Garg, A. Narang, and S. Bhattacherjee. Real-time memory efficient data redundancy removal algorithm. In *CIKM*, pages 1259–1268, 2010.

[23] J. Gehrke, F. Korn, and J. Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD*, pages 13–24, 2001.

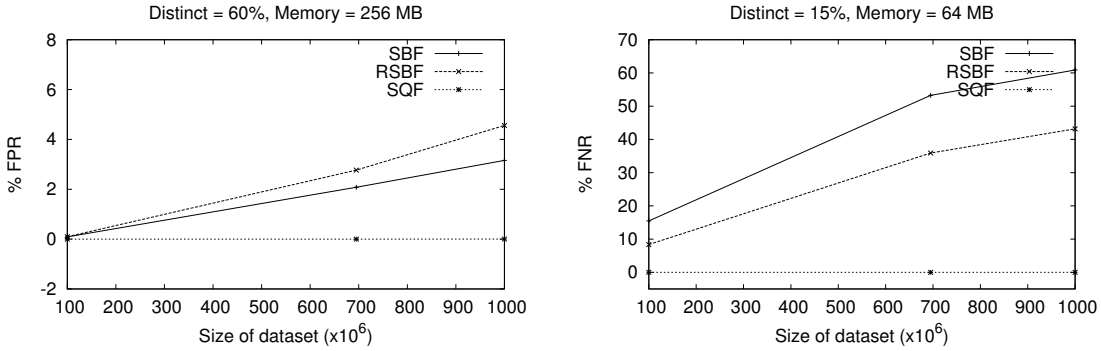
[24] P. Gupta and N. McKeown. Packet classification on multiple fields. In *SIGCOMM*, pages 147–160, 1999.

[25] A. Heydon and M. Najork. Mercator: A scalable, extensive web crawler. In *World Wide Web*, volume 2, pages 219–229, 1999.

[26] T. Hofmann. Optimizing distributed joins using bloom filters. *Distributed Computing and Internet technology (Springer / LNCS)*, 5375:145 – 156, 2009.

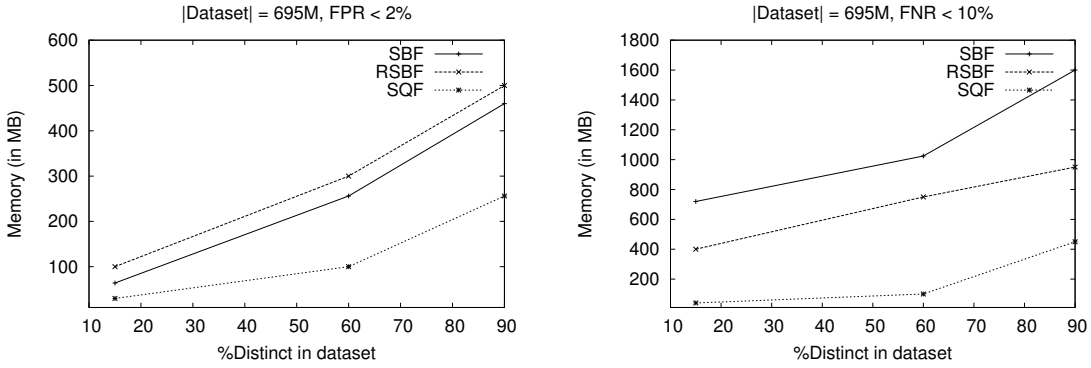
[27] Y. Hua and B. Xiao. A multi-attribute data structure with parallel bloom filters for network services. In *International Conference on High Performance Computing*, pages 277–288, 2006.

[28] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica



(a) (b)

Figure 4: Effect of dataset size on (a) FPR and (b) FNR



(a) (b)

Figure 5: Amount of memory on varying distinct % for constant (a) FPR (b) FNR

synchronization. In *FAST*, pages 281–294, 2005.

[29] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1973.

[30] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code bloom filter for efficient per-flow traffic measurement. In *IEEE INFOCOM*, pages 1762–1773, 2004.

[31] D. Lee and J. Hull. Duplicate detection in symbolically compressed documents. In *ICDAR*, pages 305–308, 1999.

[32] M. Little, N. Speirs, and S. Shrivastava. Using bloom filters to speed-up name lookup in distributed systems. *The Computer Journal (Oxford University Press)*, 45(6):645 – 652, 2002.

[33] A. Metwally, D. Agrawal, and A. E. Abbadi. Duplicate detection in click streams. In *WWW*, pages 12–21, 2005.

[34] M. Mitzenmacher. Compressed bloom filters. In *IEEE/ACM Transaction on Networking*, pages 604–612, 2002.

[35] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14:4–18, 2009.

[36] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, pages 89–101, 2002.

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

[38] M. Reiter, V. Anupam, and A. Mayer. Detecting hit-shaving in click-through payment schemes. In *USENIX*, pages 155–166, 1998.

[39] C. Saar and M. Yossi. Spectral bloom filters. In *ACM SIGMOD*, pages 241–252, 2003.

[40] H. Shen and Y. Zhang. Improved approximate detection of duplicates for data streams over sliding windows. *Journal of Computer Science and Technology*, 23(6):973–987, 2008.

[41] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: An aid to network processing. In *ACM SIGCOMM*, pages 181–192, 2005.

[42] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. C. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *USENIX*, pages 127–140, 2003.

[43] M. Weis and F. Naumann. Dogmatrix tracks down duplicates in xml. In *ACM SIGMOD*, pages 431–442, 2005.