One is Enough: Distributed Filtering for Duplicate Elimination

Georgia Koloniari
Computer Science Dept.
University of Ioannina, Greece
kgeorgia@cs.uoi.gr

Nikos Ntarmos
Computer Science Dept.
University of Ioannina, Greece
ntarmos@cs.uoi.gr

Evaggelia Pitoura
Computer Science Dept.
University of Ioannina, Greece
pitoura@cs.uoi.gr

Dimitris Souravlias
Computer Science Dept.
University of Ioannina, Greece
dsouravl@cs.uoi.gr

ABSTRACT

The growth of online services has created the need for duplicate elimination in high-volume streams of events. The sheer volume of data in applications such as pay-per-click clickstream processing, RSS feed syndication and notification services in social sites such Twitter and Facebook makes traditional centralized solutions hard to scale. In this paper, we propose an approach based on distributed filtering. To this end, we introduce a suite of distributed Bloom filters that exploit different ways of partitioning the event space. To address the continuous nature of event delivery, the filters are extended to support sliding window semantics. Moreover, we examine locality-related tradeoffs and propose a tree-based architecture to allow for duplicate elimination across geographic locations. We cast the design space and present experimental results that demonstrate the pros and cons of our various solutions in different settings.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Information Filtering*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Distributed Bloom Filters, Duplicate Elimination

1. INTRODUCTION

The growth of the Internet and the advent of large-scale computing infrastructures, have given ample ground to researchers and developers to create applications and systems that gather and process huge amounts of data from all over the world, usually in a streaming fashion. In turn, the problem of eliminating duplicates from such streams of events has become very important in a number of different settings. When it comes to large amount of data, duplicate-free event delivery is important from two perspectives (a) from the perspective of the event recipients, since it avoids overwhelming them with large amounts of similar data and (b) from a system perspective, since it eliminates the cost of processing and communicating duplicate data.

A recent example is that of click-fraud avoidance in pay-per-click online ad services. In this scenario, malicious parties generate fraudulent accesses to online ads, leading to an arbitrary inflation of related charges. Relevant large-scale systems (such as, Google AdWords/AdSense and Yahoo! Search Marketing) usually filter-out successive identical accesses as a first (and usually most important) step in fighting back[1]. Another example is the re-syndication of events that occurs in applications such as Twitter and Facebook, where events are re-posted (re-tweeted) by a number of different sources besides their original source resulting in multiple appearances of the same event in users feeds. Related research usually relies on variations of Bloom filters [2] to identify and drop duplicates [14, 13]. A common theme in these works is that the naive solution of using a single large Bloom filter for everything does not scale well, due to problems such as excessive memory footprint, lock contention or reduced parallelism. The authors advocate breaking up the filter into smaller units, while incoming events are "rehashed" and assigned to a number of these filter partitions. With this work, we extend such solutions to function in the widely distributed settings of modern data management infrastructures. Such architectures range for partitioning the Bloom filter among threads on a single host to spreading (groups of) such partitions and the related filtering load across multiple hosts and data centers.

The contribution of our work lies in casting the design space regarding the distribution of Bloom filters and their use for duplicate-free event dissemination. In particular, we propose two fundamental ways of partitioning the filters: a *horizontal* and a *vertical* one. We study the two partitioning methods both theoretically and experimentally in terms of accuracy, efficiency, load balance and fault tolerance. Vertical partitioning achieves in practice better load balance and fault tolerance but induces higher communication costs. We further present timer-based filters to support sliding window semantics. In this case, an event is considered a duplicate, only if it was previously delivered in the same time window. A nice property of our extension is that it allows us to improve accuracy through a heuristic we call "equal timers".

[1] `http://adwords.blogspot.com/2007/02/invalid-clicks-googles-overall-numbers.html`

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

Our structures are dynamic, in the sense that they automatically adjust their size thus being capable of handling bursts of events.

To exploit the geographical or network locality of sources, we propose multiple levels of filtering through Bloom trees. We introduce two variations of Bloom trees, namely the sparse and dense Bloom tree. The *sparse Bloom tree* is built based on a simple "first test for duplicates and then set" principle that results in each event being stored only once in the structure. On the other hand, the *dense Bloom tree* is built based on a simple "test and set" principle that results in each event being maintained many times in the structure. Both structures have the same accuracy but differ in efficiency.

The rest of this paper is structured as follows. Section 2 briefly presents our system model. In Section 3, we introduce our distributed filters. Filters are extended to support sliding windows in Section 4 and bursts of events in Section 5. In Section 6, we introduce the Bloom tree. Section 7 presents our experimental results, while Section 8 includes a comparison with related work. Section 9 offers conclusions.

2. MODEL AND PROBLEM DEFINITION

We consider a system where users are interested in events generated by a number of distributed data sources. The data sources that generate the events are called *producers*, while the users that are interested in the events are called *consumers*. The generated events are delivered (pushed) to the consumers by some notification service or mechanism.

We are interested in delivering distinct events to the consumers. To this end, we add a filtering component. All events generated by the producers are first passed through this component that filters out/discards duplicate events, i.e., events that have reached the system before. The straightforward solution for implementing such a filtering component would be to centrally collect and store all produced events. Any new event is compared against these events and is delivered to the consumers only if found to be distinct. The problem with this approach is that it induces large space, communication and processing overheads. To address these issues, we propose using Bloom filters to summarize the set of events.

A Bloom filter [2] is a compact data structure for representing a set of elements. The idea is to allocate an array *bf* of m bits, initially all set to 0, and then choose k independent hash functions, h_i, $1 \leq i \leq k$, each with range 0 to *m-1*. For each element $a \in A$, the bits at positions $h_i(a)$ in *bf* are set to 1. Given a query for c, the bits at positions $h_i(c)$ are checked. If any of them is 0, then certainly $c \notin A$. Otherwise, we conjecture that c is in the set, although there is a certain probability that we are wrong, called *false positive*. For a Bloom filter with n distinct elements, the false positive probability, i.e., the probability that a distinct event is characterized as a duplicate, is:

$$P_{fp}(m,k,n) = (1 - (1 - \frac{1}{m})^{k \cdot n})^k \quad (1)$$

For large m, Eq. (1) is closely (i.e., within $O(1/m)$) approximated by:

$$P_{fp}(m,k,n) \approx (1 - e^{-k \cdot n/m})^k \quad (2)$$

Using Eq. (2), one can set the parameters of a Bloom filter (i.e., number of hash functions k and size m) so as to minimize the false positive probability. In particular, P_{fp} is minimized for $k = \ln 2 \cdot m/n$, in which case $P_{fp} = 0.6185^{m/n}$.

Filtering works as follows. For each incoming event, the corresponding bits in the filter are checked and if they are set, the event is considered a duplicate; otherwise it is considered distinct. If the event is determined as a duplicate, no further action is required. Otherwise, the event is inserted in the filter by setting the above bits. We call this the *first-test-then-set* primitive. We also define a *test-and-set* primitive, where a non-set bit is set when tested.

Instead of forwarding the actual event to the filtering component, the producers can just send the bit positions corresponding to the event. This way, the communication overhead between the producers and the notification service is reduced. In particular, only $k \cdot \log_2(m)$ bits are required. Note that in this case, the communication cost depends on the number of hash functions and their range. The producer of the event needs to forward the actual event to the notification service, only if the event is found to be distinct.

3. SLICING THE BLOOM FILTER

Using Bloom filters reduces the space and processing overheads for detecting duplicates. However, the Bloom filter constitutes a single point of failure and a potential bottleneck for the notification service. To address these issues, we propose distributing the filter by slicing it.

3.1 Horizontal and Vertical Slicing

The Bloom filter is partitioned into disjoint parts, called *slices*, each being placed at a different system site. Clearly, the number of slices determines the degree of distribution. We propose two different approaches for slicing the Bloom filter: horizontally and vertically.

Definition 1. A horizontal Bloom filter is a basic Bloom filter of size m split into S slices, s_i $(1 \leq i \leq S)$, each of which is a basic Bloom filter of size m/S. Each event u is inserted in one of the S slices chosen uniformly at random by an additional hash function H_S with range $[1, S]$.

To test whether an event u belongs to the horizontal Bloom filter, the slice $s_i = H_S(u)$ is selected, and u is looked up against it by applying the k hash functions with range $[0, (m/S) - 1]$. If the event is determined as distinct the corresponding bit positions in s_i are set. The processing cost is $O(k)$, and if the message regarding the event corresponds to bits positions in the respective slice, the communication cost is $k \cdot \log_2(m/S)$ bits.

A *vertical Bloom filter* splits the filter vertically into slices corresponding to a subset of the range of the original filter.

Definition 2. A vertical Bloom filter is a Bloom filter of size m split into S slices, s_i, $1 \leq i \leq S$, such that each slice s_i is assigned bits $((i-1) \cdot m/S)$ to $(i \cdot m/S - 1)$ of the original Bloom filter.

For an incoming event u, the appropriate filter slices that contain the corresponding bits are located by applying the k hash functions to u. In the worst case, all k bits for the event are contained in k different filter slices and a message has to be sent to each of them to set the bits in their filters.

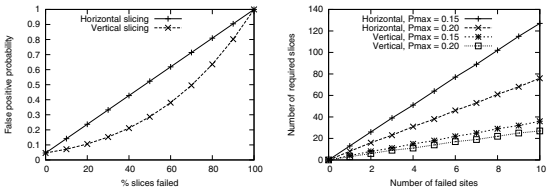
Note that each producer needs to know the location of the slices to communicate with the corresponding sites. This information can be distributed among the sites holding the filter slices, so that each producer knows the location of just one of the sites. This can be achieved by building an overlay network among the sites that maintain the slices. The sites can use this overlay for forwarding the event (or hash functions) to the site(s) holding the appropriate slices.

3.2 Properties

False Positive Probability. Assume that we slice a Bloom filter of size m with k hash functions into S slices. In the case of vertical Bloom filters, all elements are inserted into the same size m filter, thus as in Eq. (1), the false positive probability is equal to:

$$VP_{fp}(m,k,S,n) = (1 - (1 - \frac{1}{m})^{k \cdot n})^k \quad (3)$$

In the case of horizontal Bloom filters, each slice has size m/S. Assuming that H_S distributes the n elements uni-



(a) (b)

Figure 1: Fault tolerance.

formly at random at the S slices, each one of the slices receives approximately n/S elements. Thus, the false positive probability is equal to:

$$HP_{fp}(m,k,S,n) = (1-(1-\frac{S}{m})^{k\cdot n/S})^k. \quad (4)$$

From Eq. (3) and (4), $HP_{fp}(m,k,S,n) > VP_{fp}(m,k,S,n)$, for $S>1$. For small $\frac{S}{m}$, $HP_{fp}(m,k,S,n)$ is closely (within a factor of $O(S/m)$) approximated by $(1-e^{-k\cdot n/m})^k$, which in conjunction with Eq. (2) leads to the following conclusion:

Proposition 1. The false positive probability for both horizontal and vertical Bloom filters is asymptotically the same and equal to that of the non-sliced filter.

Load. Assume an incoming stream of N events taking values in a domain N_D, according to some distribution. For this input, N lookups/filter tests are required in a single Bloom filter. By slicing the filter, this load is distributed among the S slices.

With horizontally sliced filters, each of the N events is inserted/looked-up at only one slice, as $H_S(u)$ partitions N_D uniformly into parts of N_D/S size each (on average), each assigned to a different slice. Thus, the input load is also distributed uniformly across the S slices. Therefore, for a uniform input distribution, the load is balanced among the S slices, with each slice receiving approximately N/S test requests. Whereas, for a skewed distribution, the load at each slice follows this distribution.

Vertical slicing, on the other hand, results in each input event being inserted at k (possibly) different sites, as dictated by the k hash functions. In this case the total load for N events is increased by a factor of k. This added communication overhead comes at the benefit of load distribution; as each of the k functions also spreads its input domain uniformly over its output domain, the final load distribution is the sum of k permutations of the input distribution and thus more balanced than with horizontal slicing. We study this issue in more detail experimentally.

To conclude, while both horizontal and vertical filters distribute uniform input distributions evenly among the slices, vertical filters also manage to balance skewed distributions more evenly in the cost of increasing the total load.

Parallelism & Pipelining. With vertical Bloom filters, k sites need to be contacted. Both the first-test-then-set and the test-and-set primitives can be implemented either in parallel or using pipelining. In the first case, the k sites are contacted in parallel and each site independently tests, sets, or tests-and-sets the corresponding bits. To determine whether the event is distinct, the sites need to inform the producer of the event with their decision. With pipelining, the k sites test, set and test-and-set their bits in turn. In this case, only one site communicates the final outcome; this site is either the first site having a slice with an unset bit, or the last site in the pipeline, if the bits are set in all sites. In addition, the procedure for test stops, once the first unset bit is encountered, thus the response time is improved.

Fault Tolerance. Let us now compute the false positive probability in the case of failures. We assume that all sites have an equal probability to fail.

In the case of horizontal Bloom filters, each event is inserted in just one of the S filters. This filter constitutes a single point of failure for the event. If we treat events that hash to a failed site as duplicates, the false positive probability in the case of f failures is given by:

$$\frac{f}{S} + (1-\frac{f}{S})\cdot(1-e^{-k\cdot n/m})^k \quad (5)$$

Vertical Bloom filters exhibit better fault tolerance, since information for each event is available in more than one site. For each incoming event u, if there is at least one site that has not failed and at this site the corresponding bit is not set, we can safely characterize u as distinct. Thus, additional false positives are introduced only in the case in which, for an event u, the corresponding bits in all the non-failed sites are set.

We assume that the bits that the events set are distributed uniformly among the S slices. Thus, for f failed slices, the probability that a bit is not available is $\frac{f}{S}$. The false positive probability in this case is:

$$(\frac{f}{S} + (1-\frac{f}{S})\cdot(1-e^{-k\cdot n/m}))^k \quad (6)$$

Fig. 1(a) shows the false positive probability of horizontal and vertical filters with increasing percentage of failed sites.

Since the increase in the false positive probability depends on the number of slices S of the filter, we can tune S, so that for a given number of failed sites (filters) f, the false positive probability is not increased above a desired threshold P_{fp}^{max}.

Definition 3. Given a Bloom filter bf with size m and k hash functions with n inserted elements, we define its fault tolerance parameter S_f, as the number of slices we need to split bf into so as to achieve a false positive probability that does not exceed P_{fp}^{max} when f out of the S_f sites fail.

For horizontal Bloom filters, by solving Eq. (5) for S_f we get: $S_f \geq \frac{f\cdot(1-P_{fp})}{P_{fp}^{max}-P_{fp}}$. Similarly using Eq. (6), we get for vertical filters: $S_f \geq \frac{f\cdot(1-\sqrt[k]{P_{fp}})}{\sqrt[k]{P_{fp}^{max}}-\sqrt[k]{P_{fp}}}$. Fig. 1(b) illustrates how the fault tolerance parameter S_f varies for different number of failed sites and maximum false positive thresholds. Vertical filters show better fault tolerance than horizontal ones. For instance, for the same maximum false positive probability of 0.15 and f=4 failed sites, vertical filters require less than 20 slices, while horizontal filters require more than 30.

A symmetric approach is also possible, where instead of treating events that hash at failed sites as duplicates, we treat them as distinct. By doing so, we do not increase false positives, but instead introduce false negatives. This approach leads to an alternative definition of S_f such that the false negative probability does not exceed a given P_{fn}^{max} threshold and to the following hybrid approach:

VERTICAL FAULT TOLERANCE HEURISTIC. An event is considered to be a duplicate, if there are at least x bits in non-failed sites with their bits set, and distinct, otherwise.

To have a false negative, it means that one or more of the bits corresponding to the failed sites would have not been set. The probability of a false negative for horizontal Bloom filters is $\frac{f}{S}$, as any event corresponding to a failed site will be considered as distinct. In this case, the false positive probability is reduced to $(1-\frac{f}{S})\cdot(1-e^{-k\cdot n/m})^k$.
For vertical filters, the false negative probability is:

$$\sum_{i=1}^{k}\left(\frac{f}{S}\right)^i\cdot\left(1-\frac{f}{S}\right)^{k-i}$$

while the false positive one becomes:$((1-\frac{f}{S})\cdot(1-e^{-k\cdot n/m}))^k$.

Now consider the hybrid approach for vertical Bloom filters, where an event is considered to be a duplicate, if there are at least x bits in non-failed sites with their bits set, and distinct, otherwise. Using this heuristic, we get:

$$VP_{fp} = ((1-\frac{f}{S})\cdot(1-e^{-k\cdot n/m}))^x$$

and

$$VP_{fn} = \sum_{i=k-x+1}^{k} \left(\frac{f}{S}\right)^i \cdot \left(1 - \frac{f}{S}\right)^{k-i}.$$

3.3 Other Types of Slicing

Let us first examine whether there is a better way to slice a Bloom filter in terms of improving the false positive probability. The most general way to split a Bloom filter into S slices is by using two sets of hash functions. For inserting an event into the filter, one set of k_1 hash functions is used to select k_1 of the S slices. Then, for each of the selected slices, a second set of k_2 hash functions is used to set the corresponding k_2 bits. Note that for horizontal Bloom filters, $k_1 = 1$ and $k_2 = k$, whereas for vertical Bloom filters, $k_1 = k$ and $k_2 = 1$. It can be shown that:

Proposition 2. The false positive probability cannot be improved regardless of how we choose k_1 and k_2, given that we maintain the total size and number of hash functions fixed.

Proof. We split a Bloom filter into S slices using two sets of hash functions. For inserting an event into the filter, the first set of hash functions (k_1 hash functions) is used to select k_1 of the S slices. Then, for each of the selected slices a second set of k_2 hash functions is used to set to 1 the corresponding k_2 bits. Therefore, to insert each event we apply $k_1 * k_2$ hash functions. To maintain the same number of hash functions as for the single Bloom filter, we set $k_1 \cdot k_2 = k$. Let us assume n inserted events into the filter and slices of the same size m/S. Then each slice has $n \cdot k_1/S$ elements on average, making the per slice s_i false positive probability:

$$(1 - (1 - S/m)^{k_2 \cdot k_1 \cdot n/S})^{k_2} \approx (1 - e^{k_1 \cdot k_2 \cdot n/m})^{k_2}$$

Since for each event we check all k_1 slices indicated by the first set of hash functions, the total probability for a false positive to occur is: $(1 - e^{k_1 \cdot k_2 \cdot n/m})^{k_2 \cdot k_1}$. Since $k_1 \cdot k_2 = k$, we conclude that the false positive probability cannot be improved regardless of how we choose to slice a Bloom filter, given that we want to maintain the total size and number of hash functions fixed.

Let us now consider combinations of vertical and horizontal Bloom filters, called *diced Bloom filters*. There are two possible combinations: *h-v diced* Bloom filters are horizontal Bloom filters with k hash functions and S slices of size m/S, with each slice being a vertical Bloom filter of S' slices of size $m/(S \cdot S')$; *v-h diced* Bloom filters are vertical Bloom filters with k hash functions and S slices of size m/S, each being a horizontal Bloom filter of S' slices of size $m/(S \cdot S')$.

Assuming all mappings and hash functions are known, the communication and processing cost for both diced Bloom filters is O(k). For the h-v diced Bloom filter, we first consider the horizontal Bloom filter consisting of S slices and ignore the vertical partitioning. The false positive probability is:

$$\left(1 - (1 - \tfrac{S}{m})^{\frac{k \cdot n}{S}}\right)^k$$

The vertical partitioning does not influence the false positive probability, and thus, the false positve for the h-v diced Bloom filter is equal to the false positive probability of a horizontal Bloom filter with S slices and k hash functions. For the v-h diced Bloom filter, the vertical partitioning again does not influence the false positive probability. Thus, the total false positive probability is influenced by the horizontal partitioning and given by:

$$\left(1 - (1 - \tfrac{S \cdot S'}{m})^{\frac{k \cdot n}{S \cdot S'}}\right)^k$$

Again, for small $\frac{S \cdot S'}{m}$, we conclude that slicing the Bloom filter does not influence its false positive probability.

With respect to load balance, both communication and processing cost is split equally among the $S \cdot S'$ sites that maintain the corresponding filter slices. As for fault tolerance, the diced Bloom filter inherits the fault tolerance properties of the vertical Bloom filters as information about each event is distributed among multiple sites. In particular, the diced Bloom filter behave similarly to a vertical Bloom filter but with $S \cdot S'$ slices. For instance, the false positive probability if we consider that any bit corresponding to a failed site is set, is given by:

$$(\frac{f}{S \cdot S'} + (1 - \frac{f}{S}) \cdot (1 - e^{-k \cdot n/m}))^k \quad (7)$$

The false positives in case of failures are the same for both diced Bloom filters if we also consider that both their initial false positive probability is $(1 - e^{-k \cdot n/m})^k$ by using the e approximation. If instead we use the accurate formulas then the measure differs for the two structures as we need to replace in Eq. (7) the corresponding false positive for each variation as we have evaluated them above.

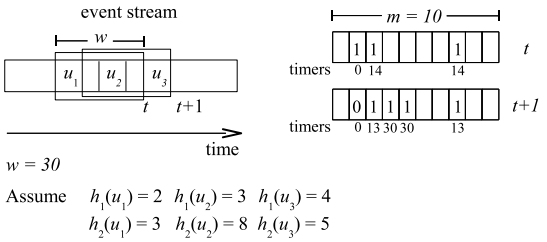


Figure 2: Event stream and the corresponding sliding Bloom filter at time t and $t + 1$.

4. SLIDING WINDOWS

In many applications where duplicate-free delivery is desired, events are only considered during specific time frames (windows). For instance, in a pay-per-click online ad service, an access made one day should not forbid the payment for an access made the next day, i.e., the second access should not be considered a duplicate.

Definition 4. An event u is considered distinct in a *sliding window* of size w, if it has not been delivered during the previous w time units.

So far, our Bloom filters are append-only: events are inserted in the filters but never deleted. However, to support sliding-window semantics, an event must be deleted from the filter, once it exits the current window. To allow deletions, we extend each Bloom filter array bf to an array bf^T of the same size whose elements are integers called *timers* initially set to zero. To insert an event u, instead of setting the bits $h_j(u)$ of bf to one, we set the timers at the same positions of bf^T equal to w. At each time unit, all non-zero timers are decremented by one. Consequently, a non-zero value t of the timer in position j in bf^T indicates that the last event that has set the corresponding timer j in bf^T was delivered $w - t$ time units ago. A position is considered to be zero, when the corresponding timer becomes equal to zero, since this means that the last event that has set the corresponding timer was delivered w time units ago, thus no longer belongs to the current window. Consequently, the above technique ensures that Definition 4 holds. An example is shown in Fig. 2. The maximum value that any timer can take is w, thus timers need to be $\lceil log_2(w) \rceil$ bits long.

The above approach assumes that all k positions associated with an event are set at the same time. This may not be possible when these positions belong to slices stored at different sites, as it may be the case with vertical Bloom filters. To address possible network delays, the producer may assign to each event u a timestamp t_u equal to the time instant of its delivery to the filtering service. Let t_i be the current time at a slice. When u is inserted, we set the timers at the corresponding positions at this slice to $w - (t_i - t_u)$.

The timers are set only if an event is distinct and not for duplicate events. Thus, using the test-and-set primitive is not possible, since, we cannot determine whether to reset any given timer without testing all associated positions first. Instead, with first-test-then-set, we can safely reset the timers that belong to events already tested as distinct.

Note that our sliding windows are time-based. Alternatively, an event-based sliding window would require that the same event is not among the last w events previously delivered. Our approach can be extended to this case as well, but such semantics are hard to achieve in a distributing setting, since some form of counting of the events is required.

Equal Timers Heuristic. The timers of the sliding Bloom filters can be used to improve the false positive probability. The intuition is that, if during the lookup for an event all k timers are non-zero, and in addition all the corresponding k timers are equal, then the timers have been most probably set by the same event. Thus, the event is a true duplicate with high probability.

We can evaluate the probability for a false positive if all the k corresponding timers are equal as follows. For a false positive, that is for an element that is not maintained in the filter to correspond to k equal timers, and if we assume that two events cannot be inserted in the same time unit, a different element that has been previously inserted in the filter needs to have set the exact same k timers. That is, the k hash functions must have identical outputs for two different elements. The false positive probability in this case may be evaluated as the false positive probability for a lookup when a single element is inserted in a Bloom filter, that is: $(1 - e^{-k/m})^k$. Generalizing the above idea, the more the equal timers among the k ones, the smaller the probability for a false positive. Based on this observation, we introduce the following heuristic:

EQUAL TIMERS HEURISTIC: A sliding Bloom filter with equal timers indicates an event as a duplicate, iff all the corresponding k timers are non-zero and at least k' of the k corresponding timers are equal; else, the event is considered distinct.

Exploiting the timers induces an overhead of k comparisons for each look-up.

5. DYNAMIC SIZE

Given an estimate of the number of events to be inserted in each Bloom filter, we can pre-allocate a maximum size per filter so that the false positive probability is kept below a certain threshold. However, this may either result in unnecessarily large filters if the actual event arrival rate is smaller than the assumed value, or in a higher false positive rate if the event arrival rate is larger. Instead, we can further enhance the flexibility of our Bloom filters, by letting the size of their slices grow dynamically as more events are inserted, so that the false positive probability is kept below a certain threshold.

Specifically, for a given filter configuration (i.e., size m and k hash functions), solving Eq. (1) for n, we get: $n = -\frac{m}{k} \cdot \ln\left(1 - \sqrt[k]{P_{fp}}\right)$. Now, substituting for a target P_{fp} we can derive a *cardinality-based* threshold. If the number n is not known, we can instead use for controlling the false positive probability the *density* of the filter, defined as the ratio z/m where z is the number of bits set to 1 in the filter. It holds: $P_{fp} = (\frac{z}{m})^k$. By solving for z, we get $z = m \cdot \sqrt[k]{P_{fp}}$ and substituting for the target P_{fp} we can derive a *density-based* threshold.

Let m_0 be the filter size initially assigned to each slice. When the density or cardinality of a slice exceeds the specified threshold, a new empty filter of the same characteristics (size and hash functions) is created. From this point on, events are checked against both filters. For an event to be delivered, both filters must indicate the event as distinct. New events are inserted only in the newly created filter. This process is repeatedly applied every time the newest filter reaches the threshold. Clearly, there is a trade-off involved in starting with small versus large initial filters. Small initial filters offer good space utilization; however, they result in many small filters and thus incur additional look-ups and a slightly worse total false positive rate.

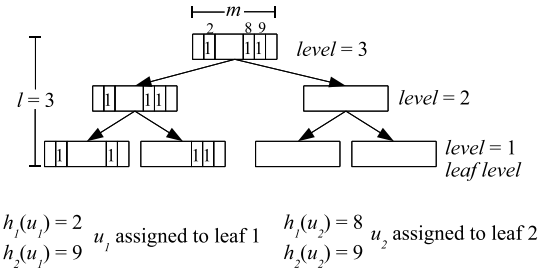


Figure 3: A Bloom tree with filters of size 10 and 2 hash functions, after the insertion of two events.

In the sliding window case, as the window slides ahead, the timers of older filters will eventually be reset to 0. Any empty filter is then discarded. When cardinality is used, n (i.e., the number of items) is incremented, each time an item is inserted in the filter. Since it is not possible to accurately detect when an item exits the window, we cannot decrement n; the old "full" filter is left to live, until it becomes empty. The density-based approach uses the actual load of the filter. Thus, the last (or only) filter may live past w time units, as window sliding gradually resets some of its timers to 0; if the event arrival rate is smaller than the rate by which the filter slides, the last filter will never need to grow. However, when growing based on cardinality, the filter will fluctuate between one and two filters if the arrival rate of events is less than the window sliding rate.

The growing and shrinking of filters is applied per slice, i.e., each slice may grow depending on the number of events it receives, independently of the other slices. This allows better space utilization when dealing with non-uniform inputs. Further, the number of alive filters depends on the rate of the arriving events versus the size of the filter, the target probability of false positive, and the sliding rate of the window. Thus, besides addressing the problem of pre-allocating an appropriate size, using dynamic sizes allows filters to efficiently handle bursts of events, by dynamically growing and shrinking their size over time.

6. MULTI-LEVEL FILTERING

In this section, we focus on multi-level filtering to explore locality so that the communication cost is reduced. The simplest form of multi-level filtering is achieved by assigning a *local Bloom filter* at each producer, in addition to the main Bloom filter. Each producer first tests for the events it produces in its local filter. If the local filter indicates a match, the event is a duplicate of an event that the producer has generated before. Otherwise, the producer forwards the event to the main filtering component.

Locality can be exploited further by multi-level filtering through a tree of Bloom filters (i.e., a *Bloom tree*) that places filters in the proximity of the producers. Events are assigned to the same tree node based on the geographical or network proximity of their producers. Thus, events are first tested against filters nearby in the physical network and are forwarded to remote parts of the network only in case of misses. Besides location criteria, other criteria such as the similarity of events that are usually looked up together or generated together may also be used to build the Bloom tree.

Algorithm 1 Test-and-Set Algorithm

Input: An incoming event u, a Bloom tree T
Output: Whether u is duplicate or not.

```
1: i = the leaf node associated with the producer of u
2: duplicate = false;
3: while i NOT NULL do
4:    if u ∈ bf_i then
5:       duplicate = true;
6:       return duplicate;
7:    else
8:       set u in bf_i
9:       i = parent node of i in T
10:   end if
11: end while
12: return duplicate;
```

Definition 5. A Bloom tree for a set of events is a tree where each node is a Bloom filter such that:

- each internal node maintains the union of the events that are maintained by its children,
- each leaf node maintains a subset of the total events inserted in the tree, and
- each event is inserted in at least one leaf.

An example Bloom tree is shown in Fig. 3.

To exploit locality, we try to filter each event as close to its producer as possible, through a bottom-up traversal. An event is first tested against the filter of the leaf associated with its producer and if not found there, the process forwards the request to nodes higher in the tree. Definition 5 ensures that if the same event has appeared, reaching the root ensures that this is detected. Moreover, sites located nearby the producer are examined first and remote parts of the network are contacted later if necessary.

We study two types of trees. A *sparse Bloom tree* is a Bloom tree, where the leaf nodes maintain disjoint subsets of the total events inserted into the Bloom tree, while in a *dense Bloom tree* the sets of events maintained by its leaf nodes may overlap. With a dense Bloom tree, we aim at moving the events closer to the producers that generate them so that more requests can be resolved locally. To implement the dense Bloom trees, the test-and-set primitive is used, whereas for the sparse Bloom trees, we use the first-test-then-set primitive.

TEST-AND-SET FOR DENSE BLOOM TREES. Consider an event u generated at leaf i of a dense Bloom tree T. The event is first tested against the filter bf_i of node i. If bf_i indicates a hit, u is considered a duplicate and the process stops. Otherwise, the bits corresponding to u are set in bf_i. This miss does not ensure that u is not a duplicate, since it may have been inserted previously at some other leaf of T. Thus, the process continues up the tree by forwarding a test-and-set request for u to the parent node of i. The process continues until either the event is found, or the root is reached, in which case we determine that u is distinct (Alg. 1).

FIRST-TEST-THEN-SET FOR SPARSE BLOOM TREES. The first-test-then-set primitive starts similarly to the test-and-set one by testing for event u in bf_i. Again, if u is found in bf_i, the event is considered a duplicate and the process stops. The difference lies in the case that bf_i indicates a miss. Then, a test request for u is again forwarded to the parent node of i, but without setting any bits in bf_i. The process continues, until either we have a hit or we reach the root. If the root is reached and u is determined as distinct, the testing stops, and setting is initiated. The same path the event has followed to the root is traversed now backwards and the bits corresponding to the event are set at each node in the path (Alg. 2).

Using a bottom-up traversal for testing and setting events in the Bloom tree alleviates some of the load at the upper levels in the case of duplicates. However, the nodes in the upper levels still receive a considerable amount of load. We can address this problem by slicing the root Bloom filter or the filter of any other overloaded node in the tree.

Algorithm 2 First-Test-Then-Set Algorithm

Input: An incoming event u, a Bloom tree T
Output: Whether u is duplicate or not.

```
1: i = the leaf node associated with the producer of u
2: duplicate = false;
3: S = NULL {stack storing the path from i to the root of T}
4: while i NOT NULL do
5:    if u ∈ bf_i then
6:       duplicate = true;
7:       return duplicate;
8:    else
9:       push i into stack S
10:      i = parent node of i in T
11:   end if
12: end while
13: i = pop(S)
14: while i NOT NULL do
15:    set u in bf_i
16:    i = pop(S)
17: end while
18: return duplicate;
```

Bloom trees can also be enhanced with timers to support sliding windows. However, as with vertical Bloom filters, the test-and-set primitive cannot be used. Instead, the first-test-then-set primitive can be used both for sparse Bloom trees and for implementing the dense Bloom trees.

Top-down testing. Since filters in higher levels maintain more events, in general, the higher in the tree we locate a match for an event, the higher the probability of a false positive. In this case, for both dense and sparse Bloom trees, we may improve the false positive probability for the testing process by using the following heuristic.

TOP-DOWN TESTING HEURISTIC. Let us assume that a match for u has occurred at a node i at level j. If we consider level j as a level with high false positive probability, instead of terminating the process, we continue by exploring the subtree rooted at i. In particular, given a threshold level d, testing proceeds as follows:

- If $j < d$, a test request is forwarded to all children of i except the one i received the initial test request for u from.
- If all the children indicate a miss, then u is considered distinct, the match at i is determined as a false positive and the process stops.
- For all children of i that indicate a match for u, the test request is forwarded similarly to the children of these nodes.
- This process continues until we reach level d.

Note that even in dense Bloom trees, the top-down process consists of tests only, i.e., no bits are set.

False Positives. Let us assume a Bloom tree with a single filter configuration of size m and k hash functions used by all of its nodes. To evaluate the false positive probability we follow a bottom up approach. Assume n distinct inserted elements in the Bloom tree filter in total. We denote as $bf_{(j,i)}$, a filter bf maintained by the i-th node at the j-th level of the tree. Thus, for each leaf filter i with n_i inserted elements, the false positive probability is given by:

$$P_{fp_{(l,i)}} = (1 - e^{-k \cdot n_i/m})^k,$$

where the sum of the n_is of all the leaves in the tree are equal to n for sparse Bloom trees, and greater or equal to n for the dense ones.

Let us consider now a filter corresponding to the i-th internal node at the j-th level of the hierarchy. Its filter $bf_{(i,j)}$ maintains the union of the events maintained by its children. Let us denote as $\mathcal{C}_r$ the set of distinct events inserted in the r-th child of i which has d children in total. Then, the false

positive probability of this filter is 1 if any of its children has indicated a match, and given that none of its children has indicated a match it is:

$$P_{fp_{(i,j)}} = (1 - e^{-k \cdot |\mathcal{C}_1 \bigcup \cdots \bigcup \mathcal{C}_d|/m})^k,$$

where $|\mathcal{C}_1 \bigcup \cdots \bigcup \mathcal{C}_d|$ denotes the cardinality of the union of all events inserted in any of the d children of i. For sparse Bloom trees, this is equal to the sum of events inserted in all the d children as there are no double insertions, while for dense ones it may be smaller since one event may be inserted in more than one child.

Proposition 3. For the same input data and tree configuration, the false positive probability of the sparse and the dense Bloom trees is the same.

Proof. To determine whether an event is distinct in the worst case the bottom-up testing will reach the root in both trees. However, the root nodes of the two trees are identical for the same input data. Even if the distinct events inserted may not be the same, the set bits are. The only difference may be caused by a false positive occurring in the dense Bloom filter that may prevent the insertion of an event, but that means that the corresponding bits are already set.

Proposition 4. If an event is determined as distinct by a single Bloom filter, then it is also determined as distinct by any Bloom tree in which all nodes use the same filter configuration with the single Bloom filter.

Proof. If an event is determined as distinct by the single Bloom filter, it is also determined as distinct by the tree.

We can equivalently show that, if an event is determined as a duplicate by the Bloom tree, then it is determined as a duplicate by the Bloom filter. Thus, if the Bloom tree indicates a false positive for an event, then this event would cause a false positive in the single Bloom filter.

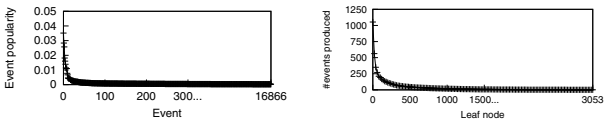
Each node filter maintains a subset of the events inserted in the single Bloom filter, and since the same hash functions are used, the bits set in each node are also a subset of the bits set in the single Bloom filter. Therefore, if a false positive appears at a node filter for an event, a false positive will appear for the same event in the single Bloom filter.

Note that if a node filter indicates a false positive for an event that has not been generated in its rooted subtree but has been inserted by some other path in the tree, then this event is not an actual false positive, since if it would reach the single Bloom filter it would be correctly identified as a duplicate. Thus, even in this case Prop. 4 still holds. However, Prop. 4 does not hold for horizontal Bloom filters. If we configure the node filters similar to the slice of a horizontal Bloom filter, with size m/S and k hash functions, the events in one filter in this case are in general directed to more than one slices in the horizontal Bloom filter. Thus, the slices are not supersets of the node filters. If we configure the node filters for horizontal Bloom filters similarly to the case of vertical Bloom filters, i.e., as a single Bloom filter with size m and k hash functions, while we do not increase the false positive probability as with vertical Bloom filters, still Prop. 4 does not hold.

Let us consider now the case where the main filtering component compared to our tree is a sliced Bloom filter. That is, let us consider that the root of our tree is now a sliced Bloom filter.

Lemma 1. With vertical Bloom filters, Prop. 4 holds if the tree filters are configured as the basic Bloom filter that was split vertically, i.e., with size m and k hash functions.

Concurrency. Inserting and testing for items in the Bloom filter involves accessing a number of bits. We can view the first-test-then-set primitive as a sequence of read(b) followed by write(b) operations, and test-and-set as a sequence of readwrite(b) operations, b being a Bloom filter bit. With concurrent accesses, an interleaving of these operations may lead to false negatives. For instance, assume two producers of the same event u, and that $h_1(u) = i$ and $h_2(u) = j$ for two of the hash functions. Further, assume that both bits i and j are equal to 0. If one producer tests i and then j and the other one j and then i, they both decide that u is distinct. One way to handle this is to implement both primitives as atomic operations or transactions. Implementing concurrency at the bit level is too fine grained; implementing concurrency at the slice level is more reasonable. In the case of vertical filtering, more than one site needs to be contacted for testing/setting the corresponding bits thus some coordination is needed. Note that slicing actually improves concurrency, since we lock smaller units. For trees, it suffices to lock the root, as all first-test-then-set requests will pass through the root node before setting any bits in the tree.



(a) Event popularity (b) Source distribution

Figure 4: Web log data distribution.

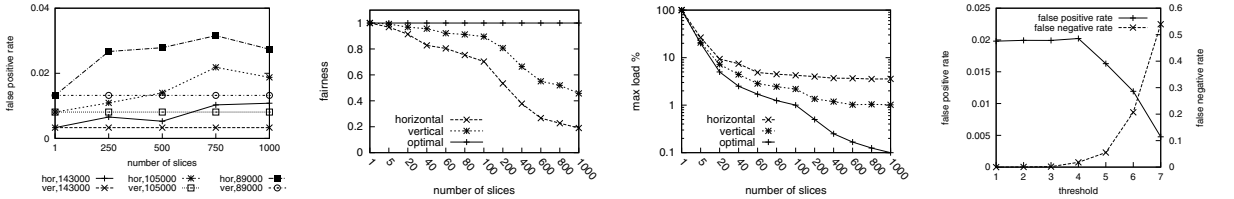
7. EXPERIMENTAL EVALUATION

In this section, we present experimental results regarding the performance of the proposed distributed Bloom filters. We used two real-world datasets - namely, a log of entries of the web server of our department and a network flow log from Yahoo - as well as a set of synthetic datasets. Unless stated otherwise, the presented figures refer to the web log dataset.

For the web log, the initial log file consisted of approximately 4.5 million entries, each including (among others) the IP address of the requesting client, the accessed URL, and a timestamp. We initially pruned all entries with IPs belonging to subnets of our institution, so as to include only remote accesses and avoid biasing the dataset. This pruning step brought the total log entry count down to 2.5 million entries. We then randomly sampled 100000 of these entries to create our input dataset. In this dataset, the client IP is used as the source (i.e., producer) identifier and the URL as the event data. The popularity of accessed URLs (i.e., the percentage of times each URL appears in the input stream) follows a highly skewed Zipf-like distribution as shown in Fig. 4(a). The Yahoo network flow log dataset contains 100000 communication patterns between end-users and Yahoo servers collected from three border routers in October 11 2007. In this dataset, the client IP is treated as the event data to be filtered. Figure 6 illustrates the popularity of each distinct item in this dataset, which also follows a highly skewed Zipf-like distribution. Lastly, for the synthetic datasets, we used both a uniform and a Zipf distribution with parameter α=1.0 for generating the events and further control the number of duplicates by drawing the events from domains having different sizes.

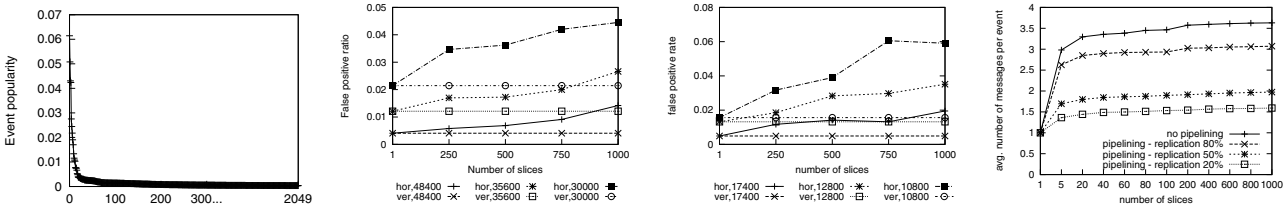
The accuracy of Bloom filters depends on the filter configuration, that is, the filter size and the number of hash functions. Typically, the filter size is set so that, given the number of distinct events, a maximum probability of false positive (P_{fp}) is achieved. Unless stated otherwise, the size of the filter is preset at the optimal size given an estimation of the input size. In particular, we use as default k=4 hash functions, and for example, for the web log dataset that includes $\approx$ 4300 distinct URLs, for $P_{fp} = 0.02$, 0.05, and 0.08, the filter consists of $m\approx$14300, 105000, and 89000 elements (bits/timers) respectively.

Bloom Slicing. The goal of this set of experiments is to examine the performance of Bloom filter slicing and pitch it against the theoretic formulae, using an append-only setting



(a) False positive (b) Fairness (c) Load (d) Fault tolerance heuristic

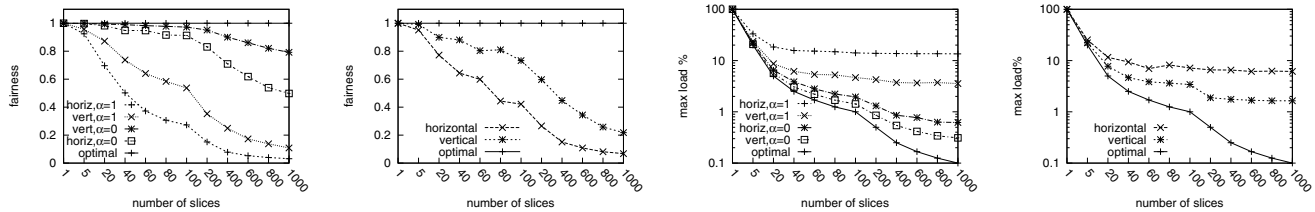
Figure 5: Results for vertical and horizontal Bloom filters for varying number of slices (web log).



(a) False positive (synthetic) (b) False positive (Yahoo) (c) Pipelining

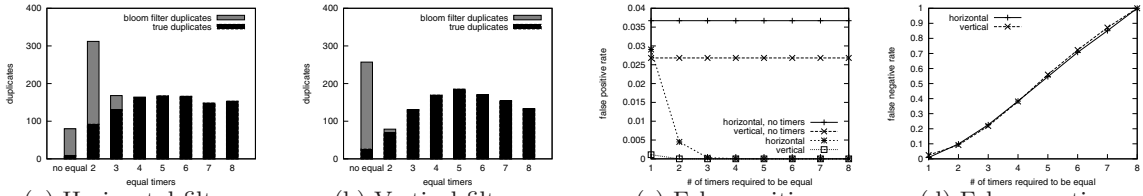
Figure 6: Yahoo dataset event popularity.

Figure 7: False positives for (a) the synthetic and (b) Yahoo datasets, and (c) pipelining for synthetic data.



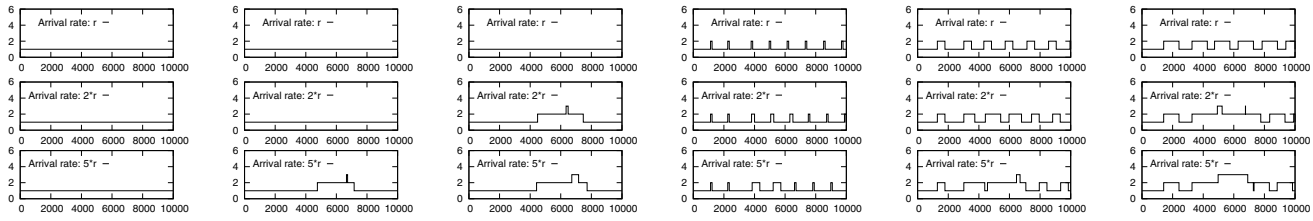
(a) Fairness (synthetic) (b) Fairness (Yahoo) (c) Load (synthetic) (d) Load (Yahoo)

Figure 8: (a,b) Fairness and (c,d) load for the synthetic and Yahoo datasets respectively.



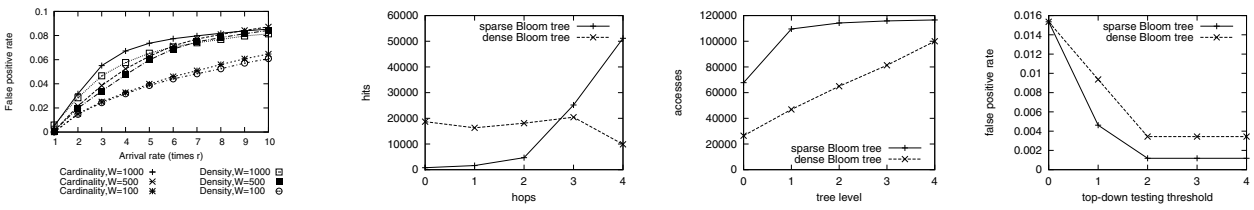
(a) Horizontal filters (b) Vertical filters (c) False positive (d) False negative

Figure 9: Equal timers (web log).



(a) w=100 (b) w=500 (c) w=1000 (d) w=100 (e) w=500 (f) w=1000

Figure 10: Dynamic size – (a,b,c): density-based, (d,e,f): cardinality-based (web log).



(a) Hops until detection (b) Accesses per level (c) Top-down heuristic

Figure 11: Dynamic size – false positive (web log).

Figure 12: Bloom trees (web log).

in which events are gradually inserted in an initially empty filter. Figure 5(a) plots the false positive rate – that is, the ratio $\frac{FP}{FP+TN}$, where FP and TN are the numbers of false positives and true negatives for the whole input stream respectively – for an increasing number of slices. Although theoretically the false positive rate is not (asymptotically) affected by slicing (see Section 3), this only holds for vertical filters; for horizontal filters the more the slices, the more the false positive rate deviates from the theoretic value. This is caused primarily by imbalances in the way items are spread across horizontal partitions and by the range of the k hash functions becoming very small as the size of individual slices decreases.

This load imbalance is also evident in Fig. 5(b) that depicts the fairness [11] of the load distribution – defined as $(\sum_{i=1}^{S} x_i)^2/(S \cdot \sum_{i=1}^{S}(x_i)^2)$, where S is the number of slices and x_i the percentage of load received by the i-th slice, with the optimal fairness value being 1 – with the number of slices. In addition, Fig. 5(c) reports the maximum percentage of the load received by any of the slices, the optimal being $1/S$. Both figures verify the intuition in Section 3, showing vertical filters outperforming horizontal ones for skewed distributions. Note that since the load distribution characteristics are not affected by the filter parameters (m, k), only a single curve is plotted for each case.

The false positives (Fig. 7(a), 7(b)) and load characteristics (Fig. 8) for the synthetic and Yahoo data sets are similar. Note that, since the number of distinct items in these datasets are different from the web log, the filters are configured differently. Specifically, for the synthetic datasets (≈5700 distinct items) and for the same target P_{fp}s of 0.02, 0.05, and 0.08, the filters consist of approximately 48400, 35600, and 30000 elements respectively; whereas, for the Yahoo dataset (≈2049 distinct items), the corresponding sizes are approximately 17400, 12800, and 10800 elements.

We also evaluated our fault-tolerance heuristic (Fig. 5(d)). Here, we use 8 hash functions, to depict how the heuristic works more clearly. In this case, in order to characterize an event as duplicate, we need at least x ($x \in [1, 7]$) of the corresponding bits to be alive and set, else we characterize it as distinct, potentially introducing false negatives. We consider a 20% failure rate (i.e., 8 out of 40 slices failing). As shown in Fig. 5(d), a threshold value of 3 or 4 is the best, since it almost minimizes both the false positive and false negative rate.

Finally, we study how pipelining improves the communication cost for vertical Bloom filters. We use three synthetic datasets that exhibit different degrees of replication. In particular, we consider a scarcely, a medium, and a highly replicated dataset, where the distinct events correspond respectively to about 80%, 50%, and 20% of the input data. Pipelining reduces the average number of messages, i.e., the slices accessed per lookup (Fig. 7(c)) for vertical partitioning. Note that with horizontal partitioning, only one slice is accessed per lookup. The savings are more evident when there are few duplicates, since seeing the first bit equal to 0 suffices to characterize an event as distinct. Note that some hash functions may lead to the same slice, thus the average number of slices accessed without pipelining is at most equal to the number of functions.

In summary, vertical Bloom filters achieve slightly better false positive rates, fault tolerance and load distribution, but induce higher communication costs that can be somewhat reduced using pipelining.

Sliding Bloom Filters and Dynamic Size. In this set of experiments, events are continuously inserted in a timer-based filter, with the timers decaying over time as described in Section 4. We first evaluate the equal timers heuristic in terms of accuracy gains. Again, we use 8 hash functions for better clarity. Figures 9(a) and 9(b) show the distribution of true duplicates over all duplicates when x ($x \in [1, 8]$) out of the 8 timers are equal, for both horizontal and vertical partitioning. Characterizing an event as distinct when all timers are unequal for vertical and at most 2 timers are equal for horizontal seems to strike a good balance between false positives (Fig. 9(c)) and false negatives (Fig. 9(d)).

We also experimented with the dynamic adjustment of the size of the sliding Bloom filters for handling bursts in the arrival rate of events. We show the growth of the filter size when a burst in the rate of events appears after 1/3 of the stream has passed, lasts for 1/3 and then events continue to arrive with the initial rate. We consider two such bursts, one with events appearing at twice the initial arrival rate and one with events appearing at five times the arrival rate. We further experiment with various window sizes, ranging from 100 to 500 and 1000 time slots. We show the number of Bloom filters that are created with (i) a density-based method (Fig. 10(a), 10(b), and 10(c)) and (ii) a cardinality-based method (Fig. 10(d), 10(e), and 10(f)) and the corresponding false positive rate (Fig. 11). In both cases, additional filters are temporarily created to handle the peaks. As expected, the cardinality-based method exhibits larger fluctuations in the number of additional created filters than the density-based method, as the former has no easy way of estimating the actual number of items in the filter at any time, while the latter knows exactly how many bits are set. We can also see that, the larger the window size, the more extra filters are created during the burst period and the longer they live, as older events are phased out more slowly. We also repeated the sliding window experiments with the Yahoo flow and synthetic datasets with similar results, thus omitted for space conservation reasons.

Table 1: Local filters.

	Hits		Accesses	
	dense	sparse	dense	sparse
L0	34K	77K	50K	111K
L1	49K	5K	100K	116K

Multi-level Filtering. Since locality is clearly application-dependent, to evaluate the benefits of multi-level filters, we used the client IPs from the web log to create the tree hierarchy. Figure 4(b) plots the distribution of events across the leaf level of our tree. Table 1 shows the locality achieved with local filters, that is with 2 levels, where level 0 corresponds to the root and level 1 to the leaves. The number of hits indicates the number of events detected as duplicates at each level. Clearly, for dense trees most duplicates are detected by using only the local filters, whereas for sparse trees, the vast majority of events are detected as duplicates at the root. Thus, dense trees are better in terms of locality, since they avoid contacting the root filter. The number of accesses refers to the number of filters accessed per level. Again, dense filters exploit locality better as expected. They also reduce the number of accesses, since for sparse trees, in the case of non-duplicates, each filter is accessed twice, once during testing and once during setting.

We also built a Bloom tree with 5 levels for the same dataset. Figure 12(a) shows the number of levels each event goes up the tree before detected as a duplicate or not, and Fig. 12(b) the distribution of accesses per level. Again, the dense tree resolves most requests locally (i.e., at lower levels). Figure 12(c) shows the reduction of false positives achieved with the top-down heuristic for various levels. A threshold value of 2 seems the best choice for both trees, since continuing at lower levels practically does not improve the false positive rate any further. Similar results were obtained for the second real and the synthetic datasets as well. In summary, both sparse and dense trees exploited the local-

ity in the dataset. Dense trees are more efficient and should be used, unless sliding-window semantics must be supported.

8. RELATED WORK

In this paper, we have proposed distributed filtering for duplicate-free event delivery. Our solution is based on distributed Bloom filters. The contribution of our work lies in casting the design space regarding the distribution of Bloom filters and their use for duplicate-free event dissemination. Horizontal vs vertical partitioning, sparse and dense trees and their analysis are novel in this paper.

Bloom filters have been used in a variety of applications (see [3] for a survey). For example, Bloom filters are deployed for cache sharing among web proxies; each proxy contains Bloom filters that summarize the cache content of all other participating proxies to quickly determine which of them contains an item of interest [9]. Bloom filters have also been used for computing the union of non-distinct sets residing at distributed sites by providing compact summaries of the content of each site [6]. In addition, Bloom filters have been deployed for query routing in distributed overlays, summarizing the content of a single node or sets of nodes in its neighborhood. In the latter case, multi-level Bloom filters have been proposed, most notably attenuated Bloom filters [16] and filters for hierarchical overlays [12]. An attenuated Bloom filter of depth d is an array of d Bloom filters. Each node in the overlay maintains an attenuated Bloom filter for each of its links such that the i-th filter, $1 \leq i \leq d$, provides a summary of the content of all nodes reachable through that link within at most i-hops. In [12], Bloom filters are placed on the nodes of a hierarchical overlay of nodes sharing XML data, where the filter at each node summarizes the XML content of all descendants in the overlay. In contrary to our approach, these works use Bloom filters to summarize the content of nodes, focusing on using the filters for routing and on lazy filter update methods for changing content.

There has been some previous work regarding streaming and dynamic-size filters. To handle unbounded streams of items, in stable Bloom filters, some randomly chosen bits are reset to 0, as new items are inserted, such that old items that have become stale are deleted from the filter [7], with applications in click-fraud detection [13]. Sliding windows with counters are used in [14] and [18] for a single stream and in [17] for distributed streams but using a centralized filter. The equal-timers heuristic and the distributed system aspects are new in this paper. Dynamic [10] and Scalable [1] Bloom filters start with a small filter and add filters when the current ones get full. Block partitioned Bloom filters [15] use blocks of Bloom filters where each block can grow dynamically. The combination of dynamic size with sliding windows for handling bursts and the distributed issues considerations are novel in this paper.

Finally, counting Bloom filters have been used for frequency estimation [5], finding most frequent items [8] and long duration network flows [4]. Spectral Bloom filters use counters to estimate the multiplicities of items [5], while multistage Bloom filters use multiple filters to reduce false positives for identifying large network flows [8, 4]. In such applications, sampling could also be applied for duplicate elimination. However, sampling suffers from false negatives while inducing much larger space, communication, and computation overheads than Bloom filters.

9. SUMMARY

In this paper, we address the problem of large-scale duplicate-free delivery of events produced by distributed sources. To this end, we have proposed a distributed filtering mechanism based on Bloom filters. We have presented a suite of distributed Bloom filters that exploit different ways of slicing the filter. To address the dynamic nature of event dissemination, we have proposed extensions that provide sliding windows semantics and dynamically adjustable sizes. To exploit locality, sparse and dense Bloom filters have been introduced that support multi-level filtering. We have studied both theoretically and experimentally the properties of the various proposed structures.

10. REFERENCES

[1] P. S. Almeidaa, C. Baqueroa, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

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

[3] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2002.

[4] A. Chen, Y. Jin, J. Cao, and L. E. Li. Tracking long duration flows in network traffic. In *INFOCOM*, 2010.

[5] S. Cohen and Y. Matias. Spectral bloom filters. In *SIGMOD Conference*, 2003.

[6] I. Dar, T. Milo, and E. Verbin. Optimized union of non-disjoint distributed data sets. In *EDBT*, 2009.

[7] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *SIGMOD Conference*, 2006.

[8] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.

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

[10] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. The dynamic bloom filters. *IEEE TKDE*, 22(1):120–133, 2010.

[11] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *DEC Research Report TR-301*, 1984.

[12] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *EDBT*, 2004.

[13] S. Majumdar, D. Kulkarni, and C. Ravishankar. Addressing click fraud in content delivery systems. In *Proc. INFOCOM*, 2007.

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

[15] O. Papapetrou, W. Siberski, and W. Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *Distributed and Parallel Databases*, 28(2,3):119–156, 2010.

[16] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. INFOCOM*, 2002.

[17] X. Wang, Q. Zhang, and Y. Jia. Efficiently filtering duplicates over distributed data streams. In *Proc. CSSE*, 2008.

[18] T. Xia, C. Jin, X. Zhou, and A. Zhou. Filtering duplicate items over distributed data streams. In *WAIM*, 2005.