Persistent Bloom Filter: Membership Testing for the Entire History

Yanqing Peng[§], Jinwei Guo[†], Feifei Li[§], Weining Qian[†], Aoying Zhou[†]
[§]University of Utah [†]East China Normal University
{ypeng, lifeifei}@cs.utah.edu, guojinwei@stu.ecnu.edu.cn, {wnqian, ayzhou}@dase.ecnu.edu.cn

ABSTRACT

Membership testing is the problem of testing whether an element is in a set of elements. Performing the test exactly is expensive space-wise, requiring the storage of all elements in a set. In many applications, an approximate testing that can be done quickly using small space is often desired. Bloom filter (BF) was designed and has witnessed great success across numerous application domains. But there is no compact structure that supports set membership testing for temporal queries, e.g., has person A visited a web server between 9:30am and 9:40am? And has the same person visited the web server again between 9:45am and 9:50am? It is possible to support such "temporal membership testing" using a BF, but we will show that this is fairly expensive. To that end, this paper designs *persistent bloom filter (PBF)*, a novel data structure for temporal membership testing with compact space.

CCS CONCEPTS

• **Theory of computation → Bloom filters and hashing**; **Sketching and sampling**; • **Information systems → Query optimization**;

KEYWORDS

Bloom filter; persistent bloom filter; persistent data structure

ACM Reference Format:
Yanqing Peng, Jinwei Guo, Feifei Li, Weining Qian, and Aoying Zhou. 2018. Persistent Bloom Filter: Membership Testing for the Entire History. In *SIGMOD/PODS '18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3183737

1 INTRODUCTION

In the era of big data, it is impractical to store and access a large amount of raw data while hoping to achieve "interactive query processing", since the data size is often too big to be stored in its entirety without incurring high (storage and processing) costs. To that end, probabilistic data structures that store raw data using small space and answer queries approximately become an important machinery in addressing many of the big data challenges.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183737

A great example is the famous *bloom filter* designed for set membership testing. Very simply, the goal is to check/test if an element is in a given set of elements, without storing the entire content of the set; i.e., is $x \in A$ for a query element x and a set A? Bloom filter [4] is a compact, probabilistic data structure that supports membership testing by maintaining a fix-sized bitmap, with a tunable small false positive rate and zero false negative rate. Bloom filters are found useful in numerous applications [6].

The same challenge is especially true for big temporal data sets, where the same element may appear many times at different timestamps. Consider the following web server log example:

```
09:30am, 155.95.78.223
09:30am, 170.22.23.36
09:30am, 155.95.78.223
...
09:47am, 155.95.78.223
09:48am, 223.12.251.22
09:50am, 223.12.251.22
...
10:00am, 87.125.33.64
```

Note that the same IP address `155.95.78.223` has appeared three times in this excerpt, and very likely appeared numerous number of times elsewhere in this log.

For analytical purposes, an administrator may ask questions like *"has IP address 155.95.78.223 ever visited my web server?"*, which is the problem of *Membership Testing*. For a small universe, bitmap is the optimal solution. When exact answers are required, one may use compressed bitmap, binary search tree or the exact membership testers proposed by Carter et al. [8]. But exact solutions may consume too much space or become computationally expensive unless the characteristics of the input data follow certain strong assumptions. In practice, small error rates are often acceptable; in which case, the most widely-used solution is bloom filter. A bloom filter provides membership testing with constant space and time with small errors, which is often an ideal solution in practice.

But given such *temporal data*, applications often ask "temporal queries" : various queries that are further conditioned on the timestamp values in raw data. This naturally leads to the temporal membership testing queries. For example, *has* `155.95.78.223` *visited my web server between 9:30am and 9:40am? And has the same IP address visited again between 9:45am and 9:50am?* The answer is yes to both questions with respect to the above example. But we need to store and access the raw log data to answer such queries. An interesting challenge is to understand if we can do so probabilistically with small errors using a compact structure in small space?

Definition 1.1 (Temporal membership testing). Given an upper bound T on the time dimension, and a universe U where elements are drawn from, a *temporal set* $\mathcal{A}$ consists of (element, timestamp)

pairs up to time T (assuming discrete timestamps). Hence, any $(a, t) \in \mathcal{A}$ satisfies $a \in U$ and $t \in [1, T]$, and $\mathcal{A}$ is a *multi-set*.

Two timestamps s and e, where $1 \le s \le e \le T$, define a temporal range $[s, e]$, and $\mathcal{A}[s, e]$ is the subset of pairs in $\mathcal{A}$ such that their timestamps fall into $[s, e]$, i.e., $\mathcal{A}[s, e] = \{(a, t) \in \mathcal{A} | t \in [s, e]\}$.

Let A be the set of *distinct elements (without timestamp)* in $\mathcal{A}$, i.e., $A = \{a | (a, t) \in \mathcal{A}\}$, and $A[s, e]$ refers to the subset of distinct elements in $\mathcal{A}[s, e]$ (henceforth, $A = \mathcal{A}[1, T]$), a temporal membership testing query (tmt-query in short) $q(x, [s, e])$ asks if $x \in A[s, e]$ for any query element x chosen from U (or equivalently, if $(x, t) \in \mathcal{A}$ for at least one time-instance value $t \in [s, e]$). The query length $|q|$ is defined by the length $(e - s + 1)$ of the query temporal range.

A *temporal set* $\mathcal{A}$ may contain duplicate pairs, e.g., two copies of the pair `(155.95.78.223, 9:30)` in $\mathcal{A}$ imply that the IP address `155.95.78.223` has showed up twice at 9:30.

Example 1. The temporal set $\mathcal{A}$ of the log excerpt above is:

```
{(155.95.78.223, 9:30), (170.22.23.36, 9:30),
 (155.95.78.223, 9:30), (155.95.78.223, 9:47),
 (223.12.251.22, 9:48), (223.12.251.22, 9:50),
 (87.125.33.64, 10:00)}.
```

The corresponding element set A is:

`{155.95.78.223, 170.22.23.36, 223.12.251.22, 87.125.33.64}`;

and A`[9:45,9:50]={155.95.78.223, 223.12.251.22}`.

Given an element x and a time range $[s, e] \subseteq [1, T]$, we want to query if there exists an element x that arrives within time range $[s, e]$. As an example, a system administrator for a website would like to know if a given IP address has visited the website, or whether a page is accessed, within a time range of interest. Many other use cases for membership testing can easily find similar applications for temporal membership testing. For example, membership testing using bloom filter has been applied in distributed storage to reduce disk lookups for non-existent rows (columns) [21]. Assuming that this is for a multi-version database [3] and we want to find rows (columns) with certain values that were inserted/updated during a time range, then temporal membership testing is required.

Since membership testing is a special case of temporal membership testing, as discussed above concerning the limitations of exact solutions for membership testing, there will be similar limitations of designing exact solutions. Thus, a probabilistic data structure that produces an approximate answer with high accuracy using small space and time is an appealing solution.

To the best of our knowledge, there is no prior study on designing an efficient probabilistic data structure with compact space to support tmt-queries. To address this problem, we propose a new data structure called Persistent Bloom Filter (PBF). In a nutshell, a PBF is a set of carefully constructed bloom filters. Each bloom filter is responsible for a carefully selected subset of elements. PBF processes a tmt-query by decomposing the query to a number of different *classic membership testing* queries, and sends each membership testing query (mt-query in short) into a different bloom filter. PBFs have the following properties:

- Restricted to one-sided error: there is no false negatives. That is, if a PBF answers no for a tmt-query $q(x, [s, e])$, then $x \notin A[s, e]$, i.e, $(x, t) \notin \mathcal{A}$ for any $t \in [s, e]$.
- Efficient space usage: PBF uses m bits in total, where m is a user parameter.
- Fast to update and query: Inserting a new pair (a, t) into a PBF is done via a few insertions to different bloom filters. Answering a tmt-query takes only sublinear time of the temporal range length (i.e., $O(\log(|e - s|))$ for $q(x, [s, e])$.
- High accuracy: false positive rate is adjustable by setting the total number of bits m and the number of bits for each bloom filter within a PBF.

2 PRELIMINARIES

2.1 An Overview of Bloom Filter

A standard bloom filter[4] is used to represent a set of elements $S = \{a_1, a_2, \ldots, a_n\}$ from a big universe U. The number of elements in S (called members), $n = |S|$, is usually much smaller than the size of the universe, $M = |U|$.

A bloom filter b is an m-bit array with a hash family $H = \{h_1, \ldots, h_k\}$ using k independent, uniformly random hash functions, where each bit is initialized to 0. We use $b[i]$ to denote the ith bit in the bloom filter b. Each hash function maps an element from U to the range $\{1, 2, ..., m\}$, i.e., $h_i : U \to [m]$ for any $h_i \in H$.

An element $a \in S$ sets $h_1(a)$-th, . . ., $h_k(a)$-th bits in b to 1. To answer a mt-query $q(x)$, i.e., check if $x \in S$ for an element $x \in U$:

$$q(x) = \begin{cases} 1, \text{ if } b[h_1(x)] \wedge b[h_2(x)] \wedge \cdots \wedge b[h_k(x)] = 1 \\ 0, \text{ otherwise.} \end{cases}$$

It is easy to see that the answer to $q(x)$ using b will never return a false negative, but may return a false positive (when $x \notin S$, but the k bits tested for x are set jointly by more than one element in S). After inserting all n members of S into b, the probability that a specific bit in b remains '0' is simply:

$$\Pr[b[i] = 0] = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}}, \text{ for any } i \in [1, m]. \quad (1)$$

Let $z = \Pr[b[i] = 0]$, the probability of a false positive is:

$$p(b) = \Pr[x \notin S \wedge q(x) = 1] = (1 - z)^k \approx \left(1 - e^{\frac{-kn}{m}}\right)^k. \quad (2)$$

One can tune the false positive rate of b by setting m and k properly. In typical scenarios, m is given by memory constraint and desirable space usage. We can choose a k value that minimizes the false positive probability by setting the derivative of $p(b)$ to zero:

$$k = \frac{m}{n} \ln 2, \text{ and } p(b) = \left(\frac{1}{2}\right)^k = 2^{-\frac{m}{n} \ln 2} \approx 0.6185^{\frac{m}{n}}. \quad (3)$$

Lastly, bloom filter works for a multi-set (i.e., S having duplicates) as well, since inserting the same element multiple times always sets the same k bits to 1. The above analysis stays the same except that n represents the *number of distinct elements* in S,

2.2 Frequently Used Notations

We use $N(\mathcal{A})$ to denote the total number of pairs in a temporal set $\mathcal{A}$, and $n(\mathcal{A})$ as the number of distinct pairs in $\mathcal{A}$. Similarly, we use $N'(A)$ to denote the number of elements in an element set A, and $n'(A)$ to denote the number of distinct elements in A. Note that by definition 1.1, $N'(A) = n'(A)$ for any element set A. Hence, we only use $n'(A)$ in the rest of this paper. In most cases $N(\mathcal{A}) > n(\mathcal{A})$ and $n(\mathcal{A}) \gg n'(A)$ for A's corresponding temporal set $\mathcal{A}$.

When context is clear, we simply use N, n, n' to represent $N(\mathcal{A})$, $n(\mathcal{A})$, $n'(A)$ respectively. For example, $N = 7$, $n = 6$, and $n' = 4$ in Example 1 from Section 1. We also use $N(s, e)$, $n(s, e)$, and $n'(s, e)$ to denote $N(\mathcal{A}[s, e])$, $n(\mathcal{A}[s, e])$ and $n'(A[s, e])$ respectively. Table 1 summarizes the frequently used notations in this paper.

U	Universe of elements.
T	Upper bound on the discrete time range.
M	Size of the universe, $M = \|U\|$.
a	An element from U.
t	A discrete timestamp from $[1, T]$.
$[s, e]$	A temporal range from timestamp s to e.
$\mathcal{A}$	Temporal set of pairs (a, t) (a multi-set).
A	(Distinct) Element set of $\mathcal{A}$: $A = \{a \mid \exists(a, t) \in \mathcal{A}\}$.
$\mathcal{A}[s, e]$	Pairs in $\mathcal{A}$ with timestamps in time range $[s, e]$.
$A[s, e]$	Element set of $\mathcal{A}[s, e]$.
$N(\mathcal{A})$ $(N(s, e))$	Number of pairs in $\mathcal{A}$ ($\mathcal{A}[s, e]$).
$n(\mathcal{A})$ $(n(s, e))$	Number of distinct pairs in $\mathcal{A}$ ($\mathcal{A}[s, e]$).
$n'(A)$ $(n'(s, e))$	Number of elements in A ($A[s, e]$).
m	Total number of bits in a bloom filter or PBF.
m_i	Number of bits in the ith bloom filter in a PBF.
k	Number of hash functions in a hash family.
$b(S)$	Bloom filter for multi-set S (or simply b).
$b[i]$	The ith bit in a bloom filter b.
$q(x)$	A standard membership testing query (mt-query).
$q(x, b)$	Answering $q(x)$ using a bloom filter b.
$\beta(\mathcal{A})$ (β)	Persistent bloom filter of temporal set $\mathcal{A}$.
b_i	The ith bloom filter in a PBF β.
$L(\beta)$	Total number of levels in a PBF β.
$u(\beta)$	Total number of bloom filters in a PBF β.
g	A time granularity.
$q(x, [s, e])$	A temporal membership testing query (tmt-query).
$q(x, [s, e], \beta)$	Answering $q(x, [s, e])$ using a PBF β.
$p(b)$ $(p(\beta))$	False positive rate of a bloom filter b (PBF β).

Table 1: Notation used in the paper.

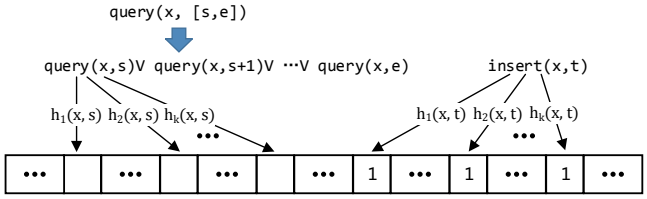


Figure 1: SBF: **insert $\mathcal{A}$ as a multi-set to a single BF.**

3 BASELINE

3.1 Naive Approach Using a Single Bloom Filter

A naive approach to answer tmt-queries is to construct a standard bloom filter for the temporal set $\mathcal{A}$, by simply treating $\mathcal{A}$ as a standard *multi-set*. More specifically, we construct a family $\mathcal{H}$ of hash functions, such that each hash function $h_i \in \mathcal{H}$ maps a pair (a, t) to $\{1, \ldots, m\}$, where $e \in U$ and $t \in [1, T]$; i.e., $h_i : U \times T \rightarrow [m]$. Each (distinct) pair $(a, t) \in \mathcal{A}$ is treated as a unique element and inserted into a bloom filter $b(\mathcal{A})$.

To answer a tmt-query $q(x, [s, e])$, we ask $(e - s + 1)$ number of mt-queries using $b(\mathcal{A})$. More specifically, the following mt-queries are posed against $b(\mathcal{A})$:

$$q_1 = q((x, s)), q_2 = q((x, s + 1)), \ldots, q_{e-s+1} = q((x, e)).$$

It is easy to see we have:

$$q(x, [s, e]) = \begin{cases} 1, \text{if } q_1 = 1 \vee q_2 = 1 \vee \cdots \vee q_{e-s+1} = 1. \\ 0, \text{if } q_1 = 0 \wedge q_2 = 0 \wedge \cdots \wedge q_{e-s+1} = 0. \end{cases}$$

We dub this baseline method SBF (single bloom filter), and use $\beta_0(\mathcal{A})$ to denote the resulting PBF (or simply β_0), and $p(\beta_0)$ as its false positive rate with respect to $q(x, [s, e])$. We have:

$$p(\beta_0) = 1 - (1 - p(b))^{e-s+1}. \quad (4)$$

Since each mt-query into $b(\mathcal{A})$ never returns a false negative, $\beta_0(\mathcal{A})$ returns no false negative. It is a standard bloom filter over the universe $U' = U \times T$, which can be easily maintained in a streaming fashion. Figure 1 illustrates the idea of SBF.

Performance analysis. The query cost of this baseline method is linear to the size of the query time range, i.e., $O(e - s) = O(T)$. Its false positive rate also depends on the value of $(e - s)$ as shown above. As the query time interval enlarges, the accuracy drops exponentially and the query cost becomes very expensive too.

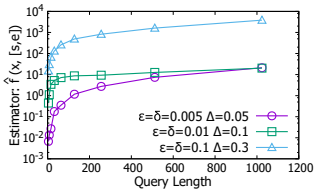


Figure 2: Estimation of PCM Sketch when $f(x, [s, e]) = 0$.

3.2 Using a Persistent Sketch

A Count-Min (CM) sketch [12] is a probabilistic data structure that returns an approximation $\widehat{f}(x)$ for the frequency of x $(f(x))$ in a multi-set S, using small space. A persistent Count-Min (PCM) sketch [37] (aka persistent sketch) extends it to support temporal frequency queries over a temporal set, i.e., what's the frequency of x in $\mathcal{A}[s, e]$ for a query temporal range $[s, e]$ (denoted as $f(x, [s, e])$).

One may use a PCM sketch to get an estimation $\widehat{f}(x, [s, e])$ for $f(x, [s, e])$. To answer a tmt-query $q(x, [s, e])$, we ask for $\widehat{f}(x, [s, e])$ using the PCM sketch. Then we can estimate:

$$q(x, [s, e]) = \begin{cases} 1, \text{if } \widehat{f}(x, [s, e]) \geq 1. \\ 0, \text{if } \widehat{f}(x, [s, e]) < 1. \end{cases}$$

As a sketching algorithm, PCM can be maintained in a streaming fashion. Note that unlike a CM sketch, which never underestimates (most likely it will overestimate due to collision) an item's frequency, a PCM sketch [37] may actually either underestimate or overestimate $f(x, [s, e])$ because it uses a piece-wise linear curve for its estimation. Its estimation is made with respect to the gradient of the piece-wise linear curve. Hence, using a PCM sketch to answer a tmt-query may lead to both false positives and negatives.

Performance analysis. PCM sketch cannot approximate low frequency values accurately enough for it to be useful to answer tmt-query. In our case, when $f(x, [s, e]) = 1$ or $f(x, [s, e]) = 0$, we need $\widehat{f}(x, [s, e])$ to be able to distinguish if $f(x, [s, e]) = 1$ or $f(x, [s, e]) = 0$, and the false positive rate can be arbitrarily large since PCM sketch only provides an additive error bound (defined with respect to the total frequency of the stream, i.e., $\Pr[|\widehat{f}(x, [s, e]) - f(x, [s, e])| < \varepsilon N + \Delta] > 1 - \delta$), rather than a relative error bound (relative to $f(x, [s, e])$). As a result, the above PCM method is very likely to produce high false positive rates.

To verify this, we did a test where $x \notin A[s, e]$, i.e., $f(x, [s, e]) = 0$, with a dataset that N is about 5 million. We varied the query range length $(e - s)$ to see the values of the estimator $\widehat{f}(x, [s, e])$ as shown in Figure 2. No matter what parameter values were set for a PCM sketch, $\widehat{f}(x, [s, e])$ keeps increasing when the query length enlarges. That is because when the query length increases, there are more collisions on the cells at each level of a PCM sketch (where each cell maintains a piece-wise linear curve), which result in higher degrees of inaccuracy. Even through $f(x, [s, e]) = 0$, when there is a collision on majority of the cells corresponding to x (i.e., where x is hashed to in each level), using $\widehat{f}(x, [s, e])$ to answer a tmt-query is likely to produce a high rate of false positives.

3.3 Other Baselines

There are other possible alternatives, such as using a persistent binary search tree, or exploring the possiblity of building a multiversion bloom filter. But none of these methods is effective in answering the tmt-queries. In the interest of space, please refer to Appendix A for more details.

4 PERSISTENT BLOOM FILTER

A major inefficiency of SBF is resulted from the way we decompose a query: if a tmt-query has a query length ℓ, then it will be decomposed into ℓ mt-queries to a standard Bloom filter. It not only consumes linear time, but also raises the probability of false positive to the ℓ-th power.

In what follows, we will describe two different PBFs: PBF-1 and PBF-2. We focus on their constructions in this section, and present the detailed analysis to their performance and optimizing their configurations in next section. For the dynamic case, we assume only insertions and do not support deletions.

4.1 PBF-1

Intuition. To reduce the number of mt-queries, we can decompose a temporal query range using dyadic ranges. In particular, we maintain a binary decomposition of $\mathcal{A}$ with respect to the time dimension, and build a bloom filter for the *corresponding element set* of each temporal range resulted from the binary decomposition.

Insertions and query processing are similar to classic segment tree operations. Specifically, in PBF-1, when inserting an element x with timestamp t, we use depth-first search (DFS) over the binary decomposition tree to find a path from the root to the leaf containing t, and insert x to every bloom filter along the path. For a query $q(x, [s, e])$, we use DFS to find the *canonical cover* of the range $[s, e]$ (i.e., a binary decomposition of $[s, e]$), and query each bloom filter of these intervals. We answer YES only if at least one bloom filter returns YES for the membership testing on x.

Lastly, a full binary decomposition can be expensive, leading to many bloom filters to maintain. Thus, we restrict the time-span of the leaf level to be at least g to control the number of intervals in the tree. To handle any operation that needs to query at a time granularity less than g, we also use another bloom filter to maintain a SBF to take over those queries.

For ease of illustration, we present the structure of PBF-1 under the static case first, and then discuss its construction and maintenance in streaming setting.

Static case. For ease of discussion, assume for now that T is a power of 2. Let g denote a time granularity that is a multiple of 2, but less than $\log T$ ($\log_2$ by default). A binary decomposition of $[1, T]$ consists of $L = \lceil \log(\lceil T/g \rceil) \rceil + 1$ levels, indexed by $\ell = 0, \cdots, L-1$. Level 0 is the root level with a single interval $[1, T]$, and level $L-1$ is the leaf level with $\lceil T/g \rceil$ intervals of length g each. Level ℓ for $\ell \in [0, L-1]$ contains 2^ℓ intervals of length $T/2^\ell$ each.

More specifically, level 0 contains $\{[1, T]\}$, level 1 contains $\{[1, \frac{T}{2}], [\frac{T}{2}+1, T]\}$, and level ℓ contains $\{[1, \frac{T}{2^\ell}], [\frac{T}{2^\ell}+1, \frac{2T}{2^\ell}], \ldots, [T-\frac{T}{2^\ell}+1, T]\}$. This leads to a total of $u = (2T/g - 1)$ temporal intervals and their corresponding temporal sets (each of which is a temporal subset of $\mathcal{A}$). These temporal intervals form a binary tree, and we will index them by a breadth-first search fashion (aka level-order),

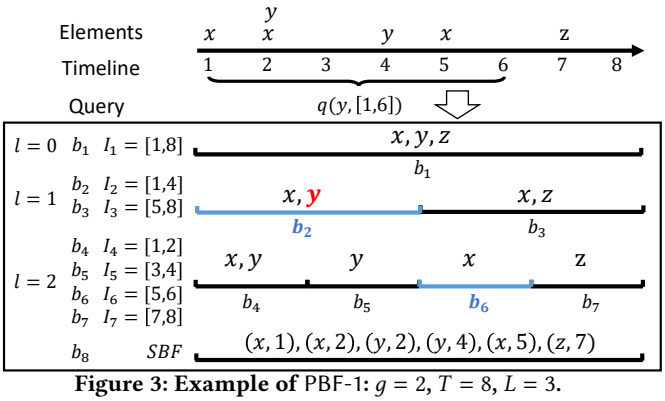


Figure 3: Example of PBF-1: $g = 2$, $T = 8$, $L = 3$.

which leads to an interval set $I = \{I_1, I_2, \ldots, I_u\}$. Level $\ell \in [0, L-1]$ contains the following subset of I: $\{I_{2^\ell}, \ldots, I_{2^{\ell+1}-1}\}$.

We denote the temporal sets associated with these temporal intervals as $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_u$, and their corresponding element sets as $A_1, A_2, \ldots, A_u$. PBF-1 is organized into the same L levels, and consisted of bloom filters $b_1 = b(A_1), b_2 = b(A_2), \ldots, b_u = b(A_u)$, such that level ℓ, for $\ell \in [0, L-1]$, contains bloom filters $\{b_{2^\ell}, \ldots, b_{2^{\ell+1}-1}\}$.

PBF-1 also adds an additional bloom filter b_{u+1} that's simply the SBF from Section 3. In summary, PBF-1, denoted as $\beta_1(\mathcal{A})$ (or simply β_1 when context is clear), is $\{b_1, b_2, \ldots, b_u, b_{u+1}\}$.

We require β_1 to use only m bits *in total*. Suppose the bloom filter b_i in β_1 uses m_i bits, $m_1 + \cdots + m_u + m_{u+1} = m$. By default, each bloom filter in β_1 uses the same number of bits (i.e., $m_1 = \cdots = m_{u+1} = m/(u+1)$. An example of PBF-1 is shown in Figure 3.

Query. We first define the *cover* $C([s, e])$ for an interval $[s, e]$. For any interval $I_i \in I$, define its *parent interval* $P(I_i)$ as the interval in the level above that fully contains I_i. Similarly, we refer to the two intervals in the level below that are fully contained by I_i as its *children intervals*.

$C([s, e])$ contains all intervals from I that are fully contained by $[s, e]$, but whose parent intervals *are not*. Formally, let $C([s, e]) = \{I_{\alpha_1}, \ldots, I_{\alpha_f}\}$, where $\alpha_i \in [1, u]$ are the index values of intervals from I that are contained in the cover. There are two cases.

(1) $C([s, e]) \subset I$ and intervals in $C([s, e])$ are disjoint (i.e., $\alpha_i \neq \alpha_j$ for $i, j \in [1, f]$);

(2) For any $I_i \in I$, if $I_i \subseteq [s, e]$ *and* $P(I_i) \not\subseteq [s, e]$, then $I_i \in C([s, e])$.

For example, with respect to the example in Figure 3, the cover $C([1, 6])$ is $\{I_2, I_6\}$, and the cover $C([1, 5])$ is $\{I_2\}$.

For any cover, its *span interval* $I(C)$ refers to the temporal range spanned by intervals in C. For example, $I(C([1, 6])) = [1, 6]$ and $I(C([1, 5])) = [1, 4]$. The definition of the cover $C([s, e])$ immediately implies that $I(C([s, e])) \subseteq [s, e]$.

It is also easy to show that the number of intervals $f = \lceil \log(e - s + 1) \rceil$ if $g = 1$; in fact, when $g = 1$, the cover of $[s, e]$ is its canonical cover [13]. In the general case, $f \approx \lceil \log(e - s + 1)/g \rceil$.

To answer a tmt-query $q(x, [s, e])$, there are two cases:

Case 1: $I(C([s, e])) = [s, e]$. In this case, intervals in $C([s, e])$ cover the query interval $[s, e]$ *exactly*. We answer $q(x, [s, e])$ using β_1 as follows, where $q(x, b_i)$ means answering a mt-query $q(x)$ using a bloom filter b_i:

$$q(x, [s, e]) = q(x, b_{\alpha_1}) \vee q(x, b_{\alpha_2}) \cdots \vee q(x, b_{\alpha_f}). \quad (5)$$

For example, consider $q(y, [1, 6])$ in Figure 3, $C([1, 6]) = \{I_2, I_6\}$ where $I_2 = [1, 4]$ and $I_6 = [5, 6]$. This is a case where $C([s, e])$ covers $[s, e]$ *exactly*, since $I(C([1, 6])) = I_2 \cup I_6 = [1, 6]$. So β_1 answers $q(y, [1, 6])$ by probing into b_2 and b_6 respectively. And in this case $q(y, b_2)$ guarantees to return 1, and $q(y, b_6)$ guarantees to return 0. Hence, β_1 will guarantee to return 1 given (5).

Case 2: $I(C([s, e])) \subset [s, e]$. When $g \neq 1$ in the construction of β_1, it is possible that $C([s, e])$ does not cover $[s, e]$ exactly. For example, consider the case in Figure 3, $C([1, 5]) = \{I_2\}$ and $I(C([1, 5])) = [1, 4]$. It is even possible that $C([s, e])$ is an empty set. For example, $C([4, 5])$ is ϕ with respect to Figure 3.

In this case, we find the set of intervals in the leaf level of I that overlaps with $[s, e] - I(C([s, e]))$. It is easy to see that there are at most two such intervals for any $[s, e] - I(C([s, e]))$; otherwise, two of them can be merged into their parent interval which must be fully contained by $[s, e]$, hence included in $C([s, e]$ and they cannot be part of $[s, e] - I(C([s, e]))$.

Suppose it is either I_{o_1} or $\{I_{o_1}, I_{o_2}\}$ who overlaps with $[s, e] - I(C([s, e]))$. We answer $q(x, [s, e])$ with the following routes:

(route 1) If $q(x, b_{\alpha_1}) \vee q(x, b_{\alpha_2}) \cdots \vee q(x, b_{\alpha_f}) \vee q(x, b_{o_1}) \vee q(x, b_{o_2}) = 0$, returns 0.

(route 2) If $q(x, b_{\alpha_1}) \vee q(x, b_{\alpha_2}) \cdots \vee q(x, b_{\alpha_f}) = 1$, returns 1.

(route 3) If $q(x, b_{\alpha_1}) \vee q(x, b_{\alpha_2}) \cdots \vee q(x, b_{\alpha_f}) = 0$, but $q(x, b_{o_i}) = 1$ for $i = 1$ and/or 2, we send a tmt-query $q(x, [s, e] \cap I_{o_i})$ to SBF β_0 contained in β_1 (which is b_{u+1}), and $q(x, [s, e])$ returns 1 if β_0 returns 1 and returns 0 otherwise.

When $[s, e] - I(C([s, e]))$ has only one overlapping interval, the term involving I_{o_2} is simply omitted.

In (route 1), if all bloom filters for intervals that are either fully contained by $[s, e]$ ($C([s, e])$ or overlapping with $[s, e]$ (I_{o_1}, I_{o_2}) assert that their element sets do not contain x, we know for sure that $x \notin A[s, e]$, since bloom filters report no false negative.

In (route 2), if a bloom filter b_{α_i} for any one interval I_{α_i} from $C([s, e])$ reports that its element set A_{α_i} contains x, it is highly likely that $x \in A[s, e]$ since $I_{\alpha_i} \in C([s, e])$ is fully contained by $[s, e]$, which implies that $A_{\alpha_i} \subseteq A[s, e]$. Note that since b_{α_i} may report false positive with a low false positive rate, this step can only assert that $x \in A[s, e]$ with high probability.

Lastly, in (route 3), if all bloom filters for intervals in $C([s, e])$ assert that x does not belong to their element sets, we know with certainty that $x \notin A[I(C([s, e])]$.

But if b_{o_1} for overlapping interval I_{o_1} reports 1, we know with high probability $x \in A[I_{o_1}]$. Hence, it is possible that $x \in A[[s, e] \cap I_{o_1}]$. However, since I_{o_1} only overlaps with $[s, e]$ and is *not* fully contained by $[s, e]$, b_{o_1} cannot tell us if this high probability occurrence of $x \in A[I_{o_1}]$ is from $x \in A[[s, e] \cap I_{o_1}]$ or $x \in A[I_{o_1} - [s, e] \cap I_{o_1}]$. Henceforth, we need to initiate a probe into $\beta_0 = b_{u+1}$, using a tmt-query $q(x, [s, e] \cap I_{o_1})$. Note that the same steps need to be taken for I_{o_2} if it exists and b_{o_2} also reports 1 in this case.

For example, to answer $q(z, [5, 7])$ with respect to the example in Figure 3. We first find the cover $C([5, 7]) = \{I_6\}$, and $I(C([5, 7])) = [5, 6]$ and $I_{o_1} = I_7$. In this case, $q(z, b_6)$ will report 0 with certainty, and $q(z, b_7)$ will guarantee to report 1. Note that $I_{o_1} \cap [s, e] = [7, 7]$ and $I_{o_1} - I_{o_1} \cap [s, e] = [8, 8]$; to find out if it is $z \in [7, 7]$ that leads to $q(z, b_7) = 1$, we send the query $q(z, [7, 7])$ to $b_8 = \beta_0$, which will ask $q((z, 7))$ against the standard bloom filter b it maintains over

Algorithm 1: Query PBF-1: q(x, $[s, e]$, β_1)

```
1  Function query(x, s, e)
2      qRecursion(x, s, e, 1, 1, T);
3  end
4  Function qRecursion(x, s, e, j, s_j, e_j) /* on I_j = [s_j, e_j]  */
5      if s ≤ s_j ∧ e ≥ e_j then
           /* I_{α_i} ∈ C[s, e] is found, where α_i = j            */
6          return q(x, b_j);
7      else if (|[s_j, e_j]| == g) ∧ ([s_j, e_j] ∩ [s, e] ≠ ∅) then
8          return q(x, [s_j, e_j] ∩ [s, e], b_{u+1});
9      else
10         mid = ⌊(s_j + e_j)/2⌋;
11         if [s_j, mid] ∩ [s, e] ≠ ∅ then
               /* I_j's left child interval is I_{2j}              */
12             return qRecursion(x, s, e, 2 · j, s_j, mid);
13         end
14         if [mid + 1, e_j] ∩ [s, e] ≠ ∅ then
               /* I_j's right child interval is I_{2j+1}           */
15             return qRecursion(x, s, e, 2 · j + 1, mid + 1, e_j);
16         end
17         return false;
18     end
19 end
```

$\mathcal{A}$ using the universe $U \times T$. In this case, $q((z, 7), b_8)$ guarantees to return 1, so β_1 will return 1 for $q(z, [5, 7])$.

The query algorithm can be easily implemented through a top-down recursion over the binary decomposition of β_1. The pseudocode of this algorithm is shown in Algorithm 1. The search starts from the root I_1 (Line 2). At any node, if its interval I_i is entirely covered by $[s, e]$, we initiate $q(x, b_i)$ (Line 5-6); or if it is a leaf node and $I_i \cap [s, e]$, we initiate $q(x, [s, e] \cap I_i, b_{u+1})$ if $q(x, b_i) = 1$ (Line 7-8). Otherwise, if its children intervals overlap with $[s, e]$, a recursion starts on the corresponding children node (Line 10-16).

Performance analysis. The query cost of Case 1 in PBF-1 is simply $O(f) = O(\log([e - s]/g)) = O(\log(T/g))$. The query cost of Case 2 in PBF-1 is slightly more involved. It has to check all bloom filters for intervals in $C([s, e])$, and in worst case has to additionally incur $q(x, I_{o_1} \cap [s, e])$ and $q(x, I_{o_2} \cap [s, e])$ into β_0. Since the time granularity of the decomposition that forms I in β_1 is g, this implies that the length of I_{o_1} and I_{o_2} is at most g. Hence, the query cost of $q(x, [s, e] \cap I_{o_1})$ (and $q(x, [s, e] \cap I_{o_2})$) is at most g. Hence, the overall query cost of Case 2 is $O(f + g) = O(\log(T/g) + g)$.

It is easy to verify that β_1 returns no false negative. And given (5) in its query Case 1 and the three possible routes in its query Case 2, its false positive rate is given by the multiplication of the non-false positive rates of its bloom filters being queried:

$$\begin{aligned} p(\beta_1) &= 1 - \prod_{i=\alpha_1,\ldots,\alpha_f} (1 - p(b_i)) \cdot (1 - p(b_{u+1})) \\ &= 1 - (1 - p(b))^{2\log(T/g)} \cdot (1 - p(b))^{O(g)}. \end{aligned} \quad (6)$$

The derivation of second step in (6) is only possible when we assume that each standard bloom filter in β_1 (including the one used in β_0) has the same false positive rate $p(b)$, which may not be the case in practice, since even if they are using the same number of

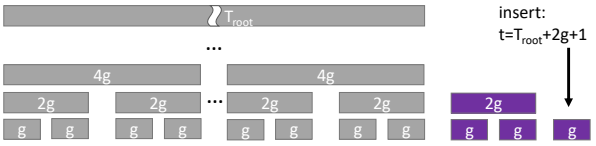


Figure 4: Dynamic maintenance of PBF-1: Inserting a new record with timestamp $T_{\text{root}}+2g+1$ into a full binary tree covering $[1, T_{\text{root}}]$. Purple intervals are new intervals created by the insertion.

bits and hash functions, the distinct number of elements for the sets they correspond to may still be different. Nevertheless, (6) provides a good estimation to PBF-1's false positive rate. Section 5 provides an in-depth analysis of PBF-1's false positive rate.

Dynamic case. When elements arrive into $\mathcal{A}$ in a streaming fashion, we can easily maintain β_0 in β_1 in a streaming fashion since it is just a standard bloom filter for elements from the universe $U' = U \times T$. Next, assuming each remaining bloom filter from β_1 uses the same number of bits, we can maintain the binary decomposition for β_1 in an online fashion.

We start with an interval $[1, g]$ at the leaf level. Every g time instances define a new interval in the leaf level, and every $2g$ time instances a new interval is introduced in the level above by "union" the latest two intervals at the leaf level. In general, suppose $T_{now} = t$, let $L_t = \lceil \log(\lceil t/g \rceil) \rceil + 1$, at level ℓ for $\ell \in [0, L_t - 1]$, every $g^{L_t-1-\ell}$ time instances leads to a new parent interval to level $\ell - 1$ by union the latest two intervals in level ℓ. When a new parent interval is introduced for the current root level (i.e., level 0 where $\ell = 0$), a new root level is introduced and every existing level's index value increases by 1 (e.g., existing level 2 becomes level 3).

The construction of PBF-1 ensures that given an interval I' in level ℓ, and its two children intervals I'_l and I'_r (left and right child intervals), $I' = I'_l \cup I'_r$ and as a result their element sets satisfy $A' = A'_l \cup A'_r$, therefore their bloom filters satisfy $b' = b'_1 \text{ bitor } b'_2$ (bitor stands for bitwise or). Hence, the "union" step described above is easily done in a streaming fashion.

Lastly, we do not maintain the index values for intervals in I and the corresponding bloom filters in β_1, since their index values undergo constant updates as new intervals and levels are introduced. Instead, we maintain I and the corresponding bloom filters by levels (within each level they are sorted by their timestamps), and only update the level indices when needed, as described above. At any time instance $t = T_{now}$, the index value of any particular interval and its associated bloom filter can be easily computed based on their current level index value ℓ and their position in level ℓ. It is easy to see that the insertion cost of one pair (a, t) into β_1 is $O(L) = O(\log(T_{now}/g))$.

Figure 4 illustrates the dynamic insertion procedure described above. In this example, a full binary tree covers the temporal range $[1, T_{\text{root}}]$, and new insertions will lead to the creation of partial binary trees in front, as illustrated in Figure 4.

The algorithm of inserting a record into a full binary tree is shown in Algorithm 2. We omit the details of dynamic growth from this pseudocode as illustrated in Figure 4 for simplicity.

4.2 PBF-2

Intuition. PBF-1 uses the natural idea of decomposition along the time dimension using the dyadic ranges, but it still uses linear number of bloom filters $O((T/g))$. Using too many bloom filters not only leads to space overhead, but also slows down the optimization process for bits allocation (to different bloom filters) as we will present in Section 5 and affects the overall accuracy in the online case as we will discuss in Section 7.

Algorithm 2: Insert an element a with its timestamp t into PBF-1

```
1  Function insert(a, t)
2      b_{u+1}.insert(a, t);
3      insertRecursion(a, t, 0, 1, T);
4  end
5  Function insertRecursion(a, t, l, s, e)
6      Let b_i be the bloom filter at level l containing interval
         [s, e] ;
7      b_i.insert(a);
8      if e − s + 1 > g then
9          mid = ⌊(s + e)/2⌋;
10         if t ≤ mid then
11             insertRecursion(a, t, l + 1, s, mid);
12         else
13             insertRecursion(a, t, l + 1, mid + 1, e);
14         end
15     end
16 end
```

This motivates us to find a solution that uses fewer bloom filters. But if we reduce the number of bloom filters by too much, it will negatively affect the performance (an extreme case will be the naive solution SBF using just one bloom filter). Therefore, we propose PBF-2 that uses *only one* bloom filter per entire level of the decomposition (rather than one bloom filter for each interval at each level of the decomposition). To do so, we *group* elements based on their arrival timestamps and a *level-specific time granularity which doubles at every level*. The insertion and query processing in PBF-2 are similar to those in PBF-1. The only difference is that when we are inserting into/querying any bloom filter at level i in PBF-1, we insert into/query the i-th bloom filter in PBF-2 instead.

Static case. More precisely, PBF-2 contains $L = \lceil \log T \rceil + 1$ levels, and the time granularity g_ℓ at level $\ell \in [0, L-1]$ is set to $g_\ell = 2^{L-1-\ell}$. Each level maintains a standard bloom filter b_ℓ over the universe $U'_\ell : U \times \lceil T/g_\ell \rceil$. A temporal element pair $(a, t) \in \mathcal{A}$, for any time instance $t \in [1, T]$ and element $a \in U$, is inserted into *all levels* with the following mapping:

$$\text{insert } (a, \mathcal{M}_\ell(t)) \text{ into } b_\ell \text{ where } \mathcal{M}_\ell(t) = \lceil t/g_\ell \rceil \text{ for all } \ell. \quad (7)$$

To be consistent with PBF-1, we call level 0 as the root level whose time granularity $g_0 = 2^{\lceil \log T \rceil}$ (which is T when T is a power of 2 or $T + 1$ otherwise), and level $(L - 1)$ (level $\lceil \log T \rceil$) the leaf level whose time granularity $g_{L-1} = 1$. An example for $T = 8$ is illustrated by Figure 5, where we use x_i to denote a pair (x, i) in level ℓ. For example, $(y, 4) \in \mathcal{A}$ is mapped to $(y, 1)$ in $\mathcal{A}_0$ from level 0, $(y, 1)$ in $\mathcal{A}_1$ from level 1, $(y, 2)$ in $\mathcal{A}_2$ from level 2, and $(y, 4)$ in $\mathcal{A}_3$ from level 3.

Let $\mathcal{A}_\ell$ be the resulting temporal set from the mapping defined in (7). A_ℓ is a temporal set from the universe $U'_\ell = U \times \lceil T/g_\ell \rceil$. For example, $U'_0 = U \times 1$, $U'_1 = U \times 2$, $U'_2 = U \times 4$, and $U'_3 = U \times 8$

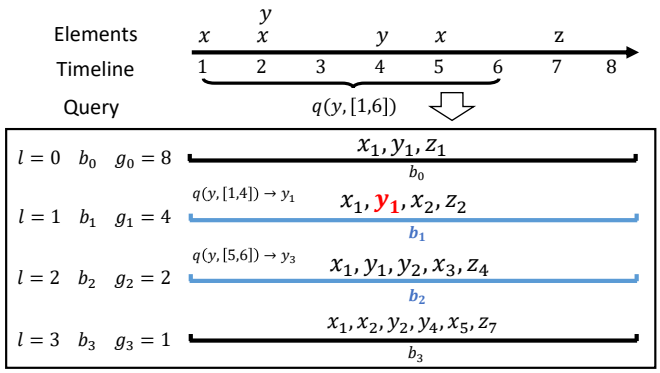


Figure 5: PBF-2 example: $T = 8$; x_i in level ℓ represents a mapping pair (x, i) in level ℓ for $(x, t) \in \mathcal{A}$, where $i = \mathcal{M}_\ell(t)$ as defined in Equation 7.

in Figure 5. A PBF-2 filter β_2 is a collection of L filters, where the filter in level ℓ is built over $\mathcal{A}_\ell$ using a standard bloom filter b_ℓ.

Query. As shown in Algorithm 3, the query algorithm of PBF-2 is in fact very similar to the query algorithm of PBF-1, by leveraging the binary decomposition of the query temporal range $[s, e]$ for a tmt-query $q(x, [s, e])$. An observation is that we can achieve a complete binary decomposition of $[s, e]$ using the combination of different levels in β_2.

More specifically, we find the *cover* $C([s, e])$ from a complete binary decomposition $B(T)$ of $[1, T]$, where a complete binary decomposition refers to a binary decomposition with $g = 1$ (time granularity at the leaf level). Note that $B(T)$ is a *virtual* structure and *never* materialized for any T. Suppose $C([s, e]) = \{I_{\alpha_1}, \ldots, I_{\alpha_f}\}$. First of all, since $B(T)$ is a complete binary decomposition using $g = 1$ at leaf level, the cover $C([s, e])$ is exactly the *canonical cover* [13] of $[s, e]$, and $f = \lceil \log(e - s) \rceil$ in the worst case. From the discussion in Section 4.1, we know that $\alpha_i \in [1, 2T - 1]$, and I_{α_i} is from level ℓ if $2^\ell \le \alpha_i \le 2^{\ell+1} - 1$, for $\ell \in [0, \lceil \log T \rceil]$. It implies that I_{α_i} belongs to level $\ell_i = \lfloor \log \alpha_i \rfloor$ in $B(T)$.

Suppose $I_{\alpha_i} = [s_i, e_i]$. Since $I_{\alpha_i} \in C([s, e])$ is from a complete binary decomposition of $[1, T]$, clearly, $e_i - s_i + 1 = g_{\ell_i}$ (time granularity of level ℓ_i), and $s_i = (c_i - 1) \cdot g_{\ell_i} + 1$ and $e_i = c_i \cdot g_{\ell_i}$ for some constant $c_i \in [1, \lceil T/g_{\ell_i} \rceil]$. In particular, $c_i = \lceil s_i / g_{\ell_i} \rceil$.

Note that I_{α_i}'s timestamps s_i and e_i are both mapped to time instance c_i in level ℓ_i with respect to $\mathcal{A}_{\ell_i}$. That said, in order for x to appear in $[s, e]$ in $\mathcal{A}$, it must be the case that (x, c_i) appears in $\mathcal{A}_{\ell_i}$ for *at least one* α_i from $C([s, e])$. If this is not the case for all α_i's in $C([s, e])$, we know for sure $(x, [s, e]) \notin \mathcal{A}$. This is to check, for all I_{α_i} in $C([s, e])$,

$$q_i = q((x, c_i), b_{\ell_i}), \text{ using } b_{\ell_i} \in \beta_2.$$

Hence, β_2 answers $q(x, [s, e])$ as follows:

$$q(x, [s, e]) = q_1 \vee q_2 \vee \cdots \vee q_f, \tag{8}$$

which is reflected by lines 5-8 in Algorithm 3.

This means that β_2 returns 0 for $q(x, [s, e])$ as soon as there is a level ℓ_i reports 0 for q_i, and reports 1 for $q(x, [s, e])$ iff all involved levels report 1 for $q_1, \ldots, q_f$ respectively. Finally, as noted above, $B(T)$ is never materialized, so does $C([s, e])$. We can easily compute α_i, $[s_i, e_i]$, ℓ_i, and c_i as discussed above for any $I_{\alpha_i} \in C[s, e]$ using $[s, e]$. This can be done via a top-down recursion search over a *conceptual* complete binary decomposition of $[1, T]$ denoted as $B(T)$. This is shown in lines 10-17 in Algorithm 3.

Algorithm 3: Query PBF-2: q(x, $[s, e]$, β_2)

```
1  Function query(x, s, e)
2      qRecursion(x, s, e, 1, 1, T);
3  end
4  Function qRecursion(x, s, e, j, s_j, e_j)
5      if s ≤ s_j ∧ e ≥ e_j then
           /* an I_{α_i} ∈ C[s, e] is found, where α_i = j.   */
6          ℓ = ⌊log j⌋ ;
7          g_ℓ = 2^{L−1−ℓ}; // L = ⌈log T⌉ + 1
8          return q(x, ⌈s_j/g_ℓ⌉, b_ℓ);
9      else
10         mid = ⌊(s_j + e_j)/2⌋;
11         if [s_j, mid] ∩ [s, e] ≠ ∅ then
12             return qRecursion(x, s, e, 2 · j, s_j, mid);
13         end
14         if [mid + 1, e_j] ∩ [s, e] ≠ ∅ then
15             return qRecursion(x, s, e, 2 · j + 1, mid + 1, e_j);
16         end
17         return false;
18     end
19 end
```

For example, consider query $q(y, [1, 6])$ in Figure 5. The cover of $[1, 6]$ is $\{[1, 4], [5, 6]\}$, which are intervals I_2 and I_6 in $B(T)$. Hence, it will be translated into $q_1 = q(y, 1)$ against b_1 in level 1 and $q_2 = q(y, 3)$ against b_2 in level 2. And in this case, clearly, q_1 returns 1 with certainty and q_2 returns 0 with certainty, hence, β_2 guarantees to return 1 for $q(y, [1, 6])$.

Performance analysis. We ask β_2 to use m bits in total, and b_ℓ in level ℓ uses m_ℓ bits. By default, $b_0, \ldots, b_{L-1}$ use the same number of bits: $m/L = m/(\lceil \log T \rceil + 1)$ bits. The query cost of PBF-2 is bounded by the size of the canonical cover of a query interval $[s, e]$, which in worst case is $\lceil \log(e - s) \rceil$. Hence, the query cost is $O(\log(e - s)) = O(\log T)$.

Since each bloom filter in (7) returns no false negative, PBF-2 never returns false negative. And its false positive rate is:

$$p(\beta_2) = 1 - \prod_{i=1}^{f} (1 - p(b_{\ell_i})) = 1 - (1 - p(b))^f \le 1 - (1 - p(b))^{2 \log T} \tag{9}$$

The second step assumes that all bloom filters in β_2 have the same positive rate $p(b)$, which may not be the case in practice. Section 5 provides a more in-depth analysis.

Dynamic case. Maintaining β_2 in an online streaming setting is easy, as described by Algorithm 4. We initialize β_2 with only 1 level, and set $g_0 = 1$. Let L_c be the current number of levels in β_2. We introduce a new root level whenever $t = T_{now}$ increases to a value that leads to $t > 2^{L_c}$. When this happens, we introduce a new level 0 and initialize its bloom filter b_0 simply as the current b_0 from the existing level 0 and its time granularity $g_0 = 2^{L_c+1}$ (Line 6), and set $L_c = L_c + 1$ (Line 7). We then increase the level index value of all bloom filters in β_2 by 1 (e.g, b_0 becomes b_1, b_1 becomes b_2, etc.) and double their time granularity values (Line 3-5). Regardless if a new level is introduced or not, we insert (a, t) into β_2 using the mapping given in (7) (Line 9-12). The insertion cost of one record (a, t) into β_2 is $O(L_c) = O(\log T_{now})$.

Algorithm 4: Insert an element a with its timestamp t into PBF-2

```
Function insert(a, t)
    if t > 2^{L_c} then
        for i = L_c to 0 do
            b_{i+1} = b_i;
        end
        b_0 = b_1.clone() ;
        L_c = L_c + 1;
    end
    for i = 0 to L_c − 1 do
        g_i = 2^{L_c − 1 − i};
        b_i.insert(a, ⌈t/g_i⌉);
    end
end
```

5 ANALYSIS AND OPTIMIZATION

We next analyze the theoretical properties of PBF-1 and PBF-2, and show how to optimize their configurations. Proofs of all theorems are found in Appendix B.

5.1 PBF-1

We first observe some useful properties of PBF-1. The followings are immediate based on Section 4.1:

Theorem 5.1. PBF-1 *only has one-sided error (false positive).*

Theorem 5.2. *The insertion cost (for one pair (a,t)) of* PBF-1 *is* $O(\log(T/g))$ *insertions into a standard bloom filter, and the query cost of a* tmt-query $q(x,[s,e])$ PBF-1 *is* $O(\log((e-s)/g)+g)$ mt-queries *into standard bloom filters.*

In our discussion in Section 4.1, an important and interesting challenge was left unaddressed, which is to optimize the configuration of a PBF-1 β_1. In particular, assuming that all bloom filters use the optimal number of hash functions, given m bits to use for β_1, how do we allocate bits to different bloom filters inside β_1? The default (and the simple) option is for β_1 to use the same number of bits for each bloom filter b_i, i.e., $m_i = m/(u+1)$. However, this is only a good approach if each bloom filter contains roughly the same number of distinct elements.

As seen from (2), the false positive of a bloom filter b is affected by n (number of distinct items inserted), m and k. A larger n value leads to higher false positive rates for the same number of bits used. Naturally, we should allocate more bits to those bloom filters that are associated with an element set with more distinct elements.

Let the binary decomposition using time granularity g for $[1,T]$ be $B(T,g)$. Recall that $B(T,g)$ leads to a set of intervals $I=\{I_1,I_2,\ldots,I_u\}$, a set of associated temporal sets $\{\mathcal{A}_1,\ldots,\mathcal{A}_u\}$ and element sets $\{A_1,\ldots,A_u\}$ for $u=2T/g-1$. A PBF-1 β_1 builds a bloom filter b_i over the *element set* A_i, i.e., $b_i=b(A_i)$ for $i\in[1,u]$.

The number of distinct temporal pairs and distinct elements from these intervals are $\{n(\mathcal{A}_1),\ldots,n(\mathcal{A}_u)\}$, and $\{n'(A_1),\ldots,n'(A_u)\}$ respectively. We simply denote them as $\{n_1,\ldots,n_u\}$, and $\{n'_1,\ldots,n'_u\}$. An important observation is that the values of $\{n_1,\ldots,n_u\}$ are often not the same, so are the values of $\{n'_1,\ldots,n'_u\}$ as well. In fact, it is well possible that even if $\{n_1,\ldots,n_u\}$ are roughly the same, the values of $\{n'_1,\ldots,n'_u\}$ may still be very different.

For example, consider a typical web server log, naturally the n'_i value for an interval in early afternoon would be much larger than an interval (of the same length, i.e., from the same level in $B(T,g)$ from midnight. And even for intervals around the same time period, an interval at a higher level from $B(T,g)$ is likely to have a larger n' value than any one of its children intervals does.

There is also the $(u+1)$th bloom filter b_{u+1} which is the SBF β_0. It is a standard bloom filter built over $\mathcal{A}$ as a multi-set from the universe $U\times T$. The number of distinct temporal pairs and distinct elements are $n_{u+1}=n(\mathcal{A})$ and $n'_{u+1}=n(A)$ respectively.

That said, let the *number of distinct items* inserted into bloom filters $\{b_1,\ldots,b_u,b_{u+1}\}$ in β_1 be $\{d_1,\ldots,d_u,d_{u+1}\}$. They are simply $\{n'_1,\ldots,n'_u,n'_{u+1}\}$ respectively.

Another factor we need to consider is the *query frequency* of any particular interval from I. Given a query workload $\mathcal{W}$ of $|\mathcal{W}|$ number of tmt-queries, we define the *expected query frequency* f_i for an interval $I_i\in I$ as the number of times a mt-query query is called by b_i, normalized to the total number of tmt-queries in $\mathcal{W}$:

$$f_i=|\{q\in\mathcal{W}\wedge \text{ a mt-query is sent to } b_i \text{ by } q\}|/|\mathcal{W}|.$$

In other words, f_i indicates the expected number of times *a single* tmt-query $q(x,[t,e])\in\mathcal{W}$ will result into a decomposition that either fully contains I_i in its cover $C([s,e])$ or overlaps with I_i if I_i is at the leaf level of $B(T,g)$, so that a mt-query is checked against the associated bloom filter b_i. Since the decomposition of each tmt-query contains a subset of distinct intervals from I, f_i is at most 1 for any $I_i\in I$ ($i\in[1,u]$), and it is most likely less than 1 on expectation with respect to a query workload $\mathcal{W}$, i.e., $0\le f_i\le 1$ for $i\in[1,u]$.

Note that b_{u+1} in PBF-1 is a special case, as it is the PBF β_0. Whenever it is involved in answering a tmt-query from $\mathcal{W}$, up to g mt-queries may be issued against b_{u+1}. Hence, $0\le f_{u+1}\le g$.

From the query algorithm for a PBF-1 β_1, it is easy to see that for any b_i is involved in answering a tmt-query q, if b_i has returned a false positive, β_1 will return a false positive for q. Given the f_i's and d_i's, we want to minimize the false positive rate for q, under the constrain that $\sum_{i=1}^{u} m_i = m$.

Let bloom filter $b_i\in\beta_1$ use $k_i=\frac{m_i}{d_i}\ln 2$ hash functions, to minimize the false positive rate, we need to maximize the probability that none of bloom filters involved in answering q returns a false positive, according to (3), this is

$$\prod_{i=1}^{u+1}(1-2^{-(\ln 2)m_i/d_i})^{f_i}, \text{subject to } \sum_{i=1}^{u+1} m_i = m.$$

Take the natural logarithm function, we get:

$$\sum_{i=1}^{u+1} f_i \ln(1-2^{-(\ln 2)m_i/d_i}), \text{subject to } \sum_{i=1}^{u+1} m_i = m.$$

Using the Lagrange multiplier, we get:

$$\mathcal{L}=\sum_{i=1}^{u+1} f_i \ln(1-2^{-(\ln 2)m_i/d_i})+\lambda_1(\sum_{i=1}^{u+1} m_i - m).$$

We first take derivative of $\mathcal{L}$ with respect to each m_i:

$$\frac{\partial\mathcal{L}}{\partial m_i}=\frac{f_i}{1-2^{-(\ln 2)\frac{m_i}{d_i}}}\cdot(-2^{-(\ln 2)\frac{m_i}{d_i}}\cdot\frac{-\ln^2 2}{d_i})+\lambda_1=0$$

$$\Longrightarrow \quad m_i=d_i\cdot\frac{1}{\ln^2 2}\cdot\ln(1-\frac{f_i\ln^2 2}{d_i\lambda_1}). \tag{10}$$

We next take derivative of $\mathcal{L}$ with respect to λ_1:

$$\frac{\partial \mathcal{L}}{\partial \lambda_1} = 0 \Longrightarrow \sum_{i=1}^{u+1} m_i - m = 0 \tag{11}$$

If we substitute (10) into (11), we get:

$$\sum_{i=1}^{u+1} d_i \ln(1 - \frac{f_i \ln^2 2}{\lambda_1 d_i}) = m \ln^2 2 \tag{12}$$

Then we can use numerical methods to solve (12) for λ_1, and then use the λ_1 value obtained to solve m_i's using (10).

According to (10), m_i grows with d_i and f_i. This is consistent with our intuition that a bloom filter b_i in β_1 with more distinct items and higher chance of being accessed should be assigned more bits. In another extreme case, if either $f_i = 0$ or $d_i = 0$, $m_i = 0$ by (10). Intuitively, this bloom filter is useless towards helping β_1 to answer a tmt-query from $\mathcal{W}$ and should not be assigned any bits.

5.2 PBF-2

The following results are immediately followed from the discussion in Section 4.2.

Theorem 5.3. PBF-2 *only has one-sided error (false positive).*

Theorem 5.4. *The insertion cost (for one pair (a, t)) of* PBF-2 *is $O(\log T)$ insertions into a standard bloom filter, and the query cost of a* tmt-query *$q(x, [s, e])$* PBF-2 *is $O(\log(e - s))$* mt-queries *into standard bloom filters.*

Another immediate observation is that different levels in PBF-2 generally contain *increasing number of distinct elements*. More specifically, Let d_ℓ be the number of distinct items inserted into the bloom filter b_ℓ in level ℓ from a PBF-2 β_2. Clearly, $d_\ell = n(\mathcal{A}_\ell)$ (recall that $\mathcal{A}_\ell$ is a temporal set obtained by mapping every pair $(a, t) \in \mathcal{A}$ using (7)). For example, $n_0 = n(\mathcal{A}_0) = n'(A)$, and $n_{L-1} = n(\mathcal{A})$, where $L = \lceil \log T \rceil + 1$.

Next, we define the expected query frequency f_ℓ for level ℓ in β_2 using the same concept defined for the analysis of PBF-1, where $\ell \in [0, L-1]$. Since a canonical cover for any interval $[s, e]$ with respect to a complete binary decomposition $B(T)$ of $[1, T]$ consists of *at most two intervals per level from $B(T)$* [13], each tmt-query q may access a bloom filter b_ℓ in β_2 at most twice. Hence, with respect to any query workload $\mathcal{W}$, $0 \le f_\ell \le 2$ for any level ℓ.

Once d_ℓ and f_ℓ are defined as above for all levels $\ell \in [0, L-1]$, the same analysis in Section 5.1 can be carried out, and we get:

$$\sum_{i=0}^{L-1} d_i \ln(1 - \frac{f_i \ln^2 2}{\lambda_2 d_i}) = m \ln^2 2, \tag{13}$$

and (10) still holds for β_2. We can solve (13) for λ_2 and use (10) to obtain m_ℓ's for β_2.

5.3 Space-accuracy tradeoff

In practice, we are also interested in knowing in order to keep the false positive rate less than a desirable threshold p, how many bits do we need? We also want to know the impacts of time granularity and time upper bound to the space cost in order to maintain a stable false positive rate. The following theorem summarizes the performance of optimized PBF-1 and PBF-2 with respect to these issues, and Appendix C provides the detailed analysis.

Theorem 5.5. *For any optimized* PBF-1 *with $g = 1$ or* PBF-2 *instance $\mathcal{P}$, assume that the time upper bound is T, and the number of distinct pairs in the stream $\mathcal{A}$ is $n = n(\mathcal{A})$. We need m bits to ensure that $\mathcal{P}$ answers a* tmt-query *q with a false positive rate p, where*

$$m = n \log T \ln\left(\left(1 - (1-p)^{1/(2\log|q|)}\right)^{-1}\right) \Big/ \ln^2 2. \tag{14}$$

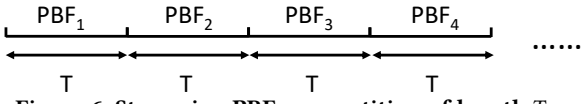


Figure 6: Streaming PBFs on partition of length T.

6 EXTENSION

To optimize the configuration of a PBF-1 β_1 or a PBF-2 β_2 using the analysis from Section 5, we need to know the distributions of distinct elements and query frequency with respect to different decomposed intervals in a binary decomposition of $[1, T]$ (either $B(T, g)$ or $B(T, 1)$). This is easy to do in the offline case, but is not possible in a streaming, online setting.

To address this challenge, we adopt a learning approach, where partitions of length T is introduced along the temporal dimension; see Figure 6. Within each partition, a PBF is built and maintained using the streaming algorithm as discussed in Sections 4.1 and 4.2. The values of d_i's and f_i's for the latest partition are learned from the distributions of previous partitions. Many learning algorithms and methodologies are possible, and which strategy is most effective depends on the properties of the data sets and query workloads. Since what learning strategy to use is not the main focus of our work and it is orthogonal to our study, in this work we just adopt a simple model. We assume that the values of d_i's and f_i's are similar to those from its preceding partition. Therefore, we can just use the d_i's and f_i's values learned from the previous partition to configure the PBF for the current partition that is streaming in.

Note that a new learning method can be easily used in place of the simple strategy above, which may incorporate knowledge from multiple prior partitions. Also note that the partition-based approach still supports *arbitrary* tmt-queries on any query temporal range $[s, e]$ by decomposing the query into 1 or more tmt-queries (with smaller query temporal ranges) to the respective partition(s).

We still need efficient and effective methods to maintain d_i's and f_i's over a streaming partition.

Essentially, we need extra tools to estimate the number of distinct elements inserted into a bloom filter, as well as the query frequency of each bloom filter. Both problems are well-studied. For simplicity, we use the most commonly used algorithms for the two problems respectively: Flajolet-Martin (FM) sketch [17] and Count-Min (CM) sketch [12]. Note that there are recent works that improve these basic algorithms such as [20]. These results are orthogonal to our study and can be easily adopted by our design.

The simple strategy. That said, using $O(T/g)$ (or $O(\log T)$) FM sketches, we can approximate the distinct number of items inserted into each standard bloom filter in β_1 (or β_2) for any partition, i.e., we can approximate the values of d_i's used in the analysis of Section 5 for a partition in a streaming fashion.

Similarly, we use a single CM sketch to approximate the count for the number of queries sent to each standard bloom filters in β_1 (or β_2) for any partition, i.e., we approximate the values of f_i's used in the analysis of Section 5 for a partition in a streaming fashion.

Given d_i's and f_i's for any historical partitions up to now, we can use various learning strategy to estimate the values of d_i's and f_i for the new, latest partition (the simplest method is to just use the values from the last partition, denoted as the SIMPLE strategy), optimize the configuration of its PBF according to the discussion in Section 5, and build and maintain its PBF in a streaming fashion using the dynamic update algorithms in Section 4.

Table 2: Different PBF methods.

method	set up
β_0	SBF
β_1-opt	PBF-1: optimal bits allocation through an offline pass
β_1-online	PBF-1: learning-based bits allocation using SIMPLE
β_2-opt	PBF-2: optimal bits allocation through an offline pass
β_2-online	PBF-2: learning-based bits allocation using SIMPLE

Assume each PBF uses m bits, as time grows, the proposed approach will use $\lceil T_{now}/T \rceil \cdot m$ bits, and we will use more bits to store these PBFs. If there is a specified space budget to support a very large time range, we may have to merge PBFs from consecutive partitions periodically to meet the space budget. It is easy to merge two PBFs from two consecutive partitions when their corresponding bloom filters use the same number of bits and hash functions: two bloom filters are merged into one through the bitwise OR operator. When this is not the case, the problem boils down to merging two standard bloom filters with different number of bits into one (assuming they use the same number of hash functions). Suppose the two filters are b' and b with m' and m bits respectively, and $m^* = gcd(m, m')$. The following method merges b' and b into a bloom filter b^* with m^* bits. For simplicity, assume for now that $k = 1$ for both b and b' (i.e., they use only 1 hash function). We ask b and b' to use the same meta-hash function h that maps $U \rightarrow [v]$ for some value $v > m, m'$. Hash function h_b for b is defined as: $h(x) \mod m$, and hash function for b' is defined as $h(x) \mod m'$. We can verify that the following merge process will successfully merge b and b': for every $b[i] = 1$ or $b'[i] = 1$ for $i \in [1, m]$ or $i \in [1, m']$, set $b^*[i \mod m^*] = 1$. We can easily generalize this approach to the general case by using k meta-hash functions. Lastly, since each PBF uses small number of bits (e.g., in our experiment a PBF uses only 6M for one day of a large data set to achieve a false positive rate of only 1%), we expect merging PBFs is rarely needed in most application scenarios.

7 EXPERIMENT

All experiments were executed on a single server with an Intel Xeon E5-2670 @ 2.6GHz CPU (a total of 8 physical cores) and 32GB RAM while running Windows 7 64bit. The hash function family we used in our implementation is MurmurHash3.

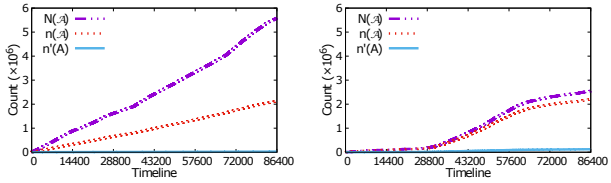
We have implemented different PBF constructions SBF, PBF-1, and PBF-2 with various instantiation, as shown in Table 2. Note that β_1-opt and β_2-opt refer to first making an offline pass over $\mathcal{A}$ and a query workload $\mathcal{W}$ to figure out the values of d_i's and f_i's for all bloom filters in PBF-1 and PBF-2 respectively, then configuring bits allocation for PBF-1 and PBF-2 based on the discussion in Section 5, and testing the resulting PBFs through $\mathcal{A}$ and the same query workload $\mathcal{W}$ in a streaming setting.

For both β_1-online and β_2-online, they use the SIMPLE strategy presented in Section 6 to approximate the values of d_i's and f_i's for all bloom filters from the preceding partition $\mathcal{A}'$ (length T) in streaming fashion, use these values to configure the PBFs for the current partition, and maintain the PBFs using the dynamic update algorithms from Section 4 for the temporal set of the current partition $\mathcal{A}$ (length T) in streaming fashion. The query workload $\mathcal{W}'$ for the preceding partition and the query workload $\mathcal{W}$ for testing the current partition *contain a different set of randomly generated* queries of *same query length.*

Datasets. We used two real datasets in our experiments. They exhibit different characteristics. The first data set is the EDGAR log file data set [1], which records user access statistics for the SEC.gov website. We extracted the IP and timestamp fields to form a temporal set. The test dataset used is from Jan 1, 2014.

The granularity of timestamp is one second. Therefore, T is bounded by the total seconds in a day, 86, 400. Figure 7(a) shows the characteristics of this data set on Jan 1, 2014, using the CDF (cumulative count) over time on $N(\mathcal{A})$, $n(\mathcal{A})$, and $n'(A)$, which implies the total number of log entries, the total number of distinct temporal pairs (visits by the same client at the same timestamp are considered as duplicates; visits by the same client at different timestamps are considered as unique pairs), and the total number of distinct IPs. At the end of the day, there are 5, 582, 073 log entries from 25, 497 different clients, while 2, 127, 749 of them are distinct visits. Since people from all over the world visited the website throughout the 24-hour period, the CDF of $N(\mathcal{A})$, $n(\mathcal{A})$, and $n'(A)$ witnessed steady increase throughout the day.

The second dataset is a temporal set from the network trace going into a local network. It contains a day's worth of network trace coming into this network. We use source IP and timestamp (in second) to construct temporal pairs, and denote this dataset as Network. The statistics of this data set is shown in Figure 7(b). Clearly, its characteristics is very different from EDGAR. Note that the timestamp starts around midnight; naturally, we see very little activities initially until they pick up during morning hours, and slow down again when it gets close to evening time. At the end of the day, there are 2, 546, 560 log entries from 124, 427 different clients, while 2, 199, 573 of them are distinct visits.



(a) EDGAR. (b) Network.

Figure 7: CDF for $N(\mathcal{A})$, $n(\mathcal{A})$, and $n'(A)$ over T.

Set up. All queries in a testing query workload $\mathcal{W}$ are of *same query length*, denoted as $|q|$, in order to show the effect of query length. The number of queries in $\mathcal{W}$ is denoted as $|\mathcal{W}|$. For all PBF methods, and any tmt-query $q(x, [s, e])$, if $\exists(x, t) \in \mathcal{A}$ for at least one value $t \in [s, e]$, they will definitely return 1, i.e., all PBFs return no false negative. Hence, we focus on the non-trivial cases when $(x, t) \notin \mathcal{A}$ for any $t \in [s, e]$. For a given pair of values for $|q|$ and $|\mathcal{W}|$, we randomly generate $|\mathcal{W}|$ number of such queries, where each query's query temporal range $(e - s + 1)$ is $|q|$. Since we observe that $|\mathcal{W}|$ has little impact towards the average false positive rate, we fix $|\mathcal{W}|$ to be 10, 000 in all experiments. In the online case, since we assume that the workload patterns are stable, we generate the training data simply by adding a random noise into the testing data.

Once the number of bits, m_i, is determined for a bloom filter b_i in a PBF, we use $k_i = \lceil \frac{m_i}{d_i} \ln 2 \rceil$ number of hash function for b_i. In practice, the number of hash functions, k, affects directly the operation performance in a standard bloom filter. In order to limit the overhead of a operation, we use k_{max} to denote the upper

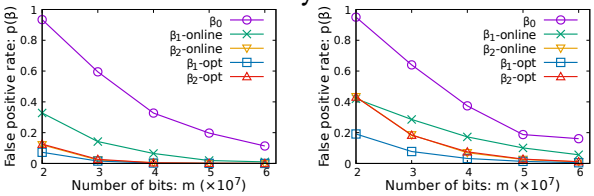
Table 3: Parameters and tested values.

parameter	tested values
m, total bits	$2 \cdot 10^7$, $3 \cdot 10^7$, $4 \cdot 10^7$, $\mathbf{5 \cdot 10^7}$, $6 \cdot 10^7$
$\|q\|$, query length	16, 32, 64, **128**, 256, 512, 1024, 2048, 4096, 8192
T, # of timestamps	14400, 28800, 57600, **86400**

threshold for k. In this experiment, we set $k_{max} = 16$. We have already discussed how the value d_i is defined and calculated for β_1-opt, β_2-opt, β_1-online, and β_2-online methods. For both β_1-opt and β_1-online, the default value of g is 4.

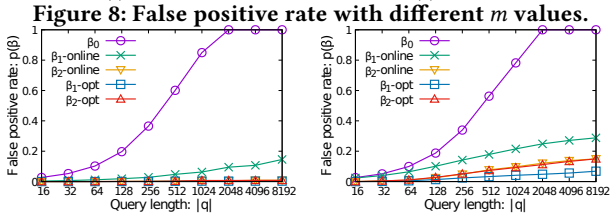
All PBFs use the *same number of bits*, denoted as m. Table 3 summarizes the key parameters and their values tested in our experiments, where the default value of a parameter is shown in bold. In each experiment, we vary the values of one parameter, while using the default values for all other parameters, to understand its effect. For all experiments concerning false positive rate and query efficiency, we show the *average* of one query from $\mathcal{W}$.

7.1 Effectiveness Analysis



(a) EDGAR. (b) Network.

Figure 8: False positive rate with different m values.



(a) EDGAR. (b) Network.

Figure 9: False positive rate with different $|q|$ values.

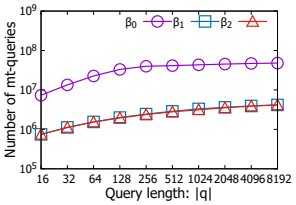


Figure 10: Query efficiency: vary $|q|$, EDGAR.

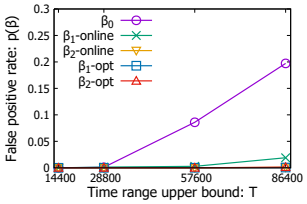


Figure 11: Impact of T values: EDGAR.

Since the basic versions of PBF-1 and PBF-2 without using the optimization for bits allocation (β_1 and β_2 respectively where they allocate bits uniformly to all bloom filters) are always strictly dominated by the improved versions with optimized bits allocation schemes presented in Section 5, in terms of their false positive rates. We omitted β_1 and β_2 from the following results on effectiveness.

Impact of m. As we increase the space budget, i.e., m, all methods gain better performance. Figure 8 illustrates this trend. Clearly, the baseline approach SBF (β_0) has very high false positive rate (FP rate) initially when given insufficient space, since it has to perform 128 mt-queries (the default value of $|q|$ is 128). As m increases, it gains better performance, but still performs much worse compared to PBF-1 and PBF-2. On the other hand, the performance of PBF-1 and PBF-2 is significantly better; β_1-opt has the lowest FP rate and β_2-opt shows a similar performance with much less number of bloom filters. Using a very small space for maintaining the CM and FM sketches, β_2-online shows an almost as good performance as that of β_2-opt with a FP rate as low as approximately only 1%, whereas β_1-online's FP rate does have a notable gap from that of β_1-opt due to the fact that PBF-1 has more bloom filters to deal with in figuring out a good bit allocation strategy, but still β_1-online also demonstrates good performance.

All methods exhibit slightly higher FP rates on Network compared to EDGAR, due to the fact that Network has more fluctuation in its data arrival patterns over time as shown in Figure 7.

Impact of $|q|$. The FP rate of β_0 grows exponentially with the query length. Therefore, it cannot handle queries of longer query intervals, as clearly shown in Figure 9. Its FP rate becomes higher than 80% when $|q|$ is 1024 seconds (approximately only 17 minutes). PBF-1 and PBF-2 also demonstrate a deteriorating performance as $|q|$ increases, but the FP rates of PBF-1 and PBF-2 increase only slightly as $|q|$ increases exponentially and stay as low as less than 5% even when $|q|$ is over 1000 seconds on both datasets, with the exception of β_1-online which has reached 20% FP rate when $|q| = 1024$. β_2-online once again shows excellent performance and matches almost the same performance provided by β_1-opt and β_2-opt.

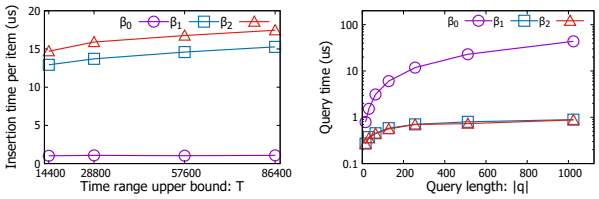
Figure 10 shows that the query processing cost grows dramatically in the case of β_0 for small $|q|$. For large $|q|$, the FP rate of β_0 is very high, and the short-circuit evaluation on β_0 slows down the growth of cost of β_0. The cost grows slightly in β_1 and β_2. Note that β_1 (β_2), β_1-opt (β_2-opt), and β_1-online (β_2-online) have the same query costs. They are both more than one order of magnitude efficient than SBF β_0. The trend for Network is similar.

Impact of T. Next, we vary the upper bound of time limit T to observe its impact on the FP rate. Figure 11 illustrates this result. A larger T value leads to more distinct elements, hence, all methods will have higher FP rates for one single mt-query. For β_0, since its FP rate is the FP rate of single mt-query raised to the power of $|q|$ (default value is 128), its FP rate increases sharply. For PBF-1 and PBF-2, the FP rate of one single mt-query increases quickly as T increases, so they are also deteriorating. Nevertheless, we can see that β_1-opt, β_2-opt and their online versions have shown significantly better performance than β_0. In particular, β_2-online demonstrates superior performance and matches well against the two offline, optimal versions. The result on Network is similarly, and hence omitted for the interest of space. Note that, a standard bloom filter b also suffers from deteriorating FP rates as T grows, which leads to more distinct elements inserted into b.

7.2 Efficiency Analysis

To investigate the efficiency of different methods, we measured the insertion time and query time for different constructions. For this experiment, we used a randomly generated data set with 50,000 items, since the insertion cost is not related to the distribution of values, rather, is only determined by the structure of the underlying PBF. We used the default number of bits for all PBFs ($m = 5 \times 10^7$). The structure of SBF is fixed, but the structures of PBF-1 and PBF-2 are affected by the maximum timestamp value T. Hence, we investigate the insertion cost by examining the amortized cost of inserting one item when we vary the value of T.

We first investigate the basic PBF-1 and PBF-2 without online optimization, i.e., allocating bits uniformly to all bloom filters. We

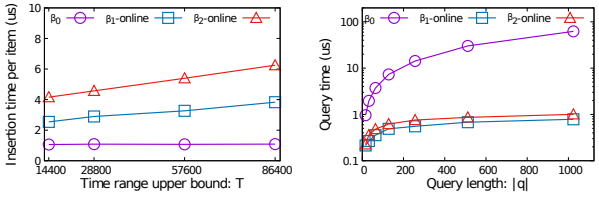


(a) Insertion. (b) Query.

Figure 12: Efficiency of β_0, β_1, β_2.

compared them against SBF β_0. Figure 12(a) shows that even though the amortized insertion costs per item for β_1 and β_2 are higher than that of β_0, both of them still enjoy excellent efficiency for insertion: their insertion costs per item are less than 20 μs in all cases. Another observation is that as T grows, the insertion costs of β_1 and β_2 slightly increase due to the fact that more bloom filters are to be maintained due to their constructions.

In terms of query time, we investigate their performance with different query lengths. To avoid the effect of false positives on query execution time, we turn off the short circuit evaluation, i.e., SBF and PBFs execute all mt-queries regardless of the result of each query. Figure 12(b) shows that the basic versions of PBF-1 and PBF-2 are much more efficient to query than that of SBF. Furthermore, their query costs are less influenced by the increase of query length (a logarithmic growth rate) than that of SBF (a linear growth rate). Both PBF-1 and PBF-2 exhibit excellent query efficiency with less than 1 μs response time.



(a) Insertion. (b) Query.

Figure 13: Efficiency of β_0, β_1-online, β_2-online.

Next, we repeated the same experiment but for the online optimization version of PBF-1 and PBF-2, i.e., β_1-online and β_2-online. The results are shown in Figure 13. The general trends for both insertion costs and query costs of various methods are similar to the results above, except that the insertion costs for both β_1-online and β_2-online are smaller than their basic versions β_1 and β_2 (i.e., they are more efficient). This is due to the fact that with bits allocation optimization certain bloom filters in some levels in PBF-1 and PBF-2 are skipped during an insertion.

7.3 Additional results

Additional results on the scalability for the accuracy-space tradeoff, efficiency under different workloads, and the impact of merging and the performance of the sketching and our simple learning method used by PBF-1 and PBF-2 are available in Appendix D.

8 RELATED WORK

Since the design of bloom filter by Burton Howard Bloom in 1970, it has found numerous applications, and many extensions have been introduced. Weighted bloom filter [7] and data-popularity conscious bloom filter [38] were proposed to reduce the false positive rate in a bloom filter by exploring the skew distribution of query frequency for different elements. The main idea is to customize the number of hash functions used for inserting an element a based on its query frequency. More hash functions are used for a more popular element, to reduce the probability of collision.

Standard bloom filters only support element insertions and membership queries. Counting bloom filters [16] support element deletions by replacing the bits of bloom filters with counters. Guo et al. designed the dynamic bloom filter [19] that also supports deletions in addition to insertions. Spectral bloom filters [10] improve bloom filters by supporting frequency-based queries. Attenuated bloom filters [30] store routing path information using arrays of bloom filters. Bloomier filters [9] generalize bloom filters to support function encoding and evaluation. Exponentially decaying bloom filters [21] encode probabilistic routing tables in a compressed manner that allows for efficient aggregation and propagation of routing information in unstructured peer-to-peer networks. Lastly, a compressed bloom filter can be used to reduce the communication cost of transmitting a bloom filter [25].

Pagh et al. [27] proposed a variant of bloom filter that consumes less space than classic bloom filters. Putze et al. [29] proposed several variants that are either faster or use less space than classic bloom filters. Fan et al. [15] proposed cuckoo filter, which is an alternative sketch that not only achieves better space efficiency than bloom filter but also support deletion. These variants and alternatives, however, explore various tradeoffs and assumptions that limit their usage in the general case.

We refer interested readers to an excellent survey and analysis paper on bloom filters by Tarkoma et al. [35].

A closely related area is persistent data structure [14], that always preserves the previous version of itself when it is modified. Many efforts have been made to extend a base structure to a persistent data structure, including Time-Split B-tree [24] and Multiversion B-tree [3, 5, 36]. Persistent data structures are also used in multiversion databases such as Microsoft Immortal DB [22, 23], SNAP [33], Ganymed [28], Skippy [32] and LIVE [31]. Wei et al. [37] introduced *persistent data sketching*, which extends the Count-Min sketch [12] to support temporal frequency queries. But it cannot be used to answer tmt-query effectively, as shown in Section 3.2.

Our study is also related to the topic of sketching, building synopses and other probabilistic succinct data summaries for massive data, and in particular, for streaming data. An excellent review is available by Cormode et al. [11].

Tao et al. studied the problem of finding quantiles for a query temporal range over a temporal set [34]. But the proposed methods are designed for quantile queries and not useful for tmt-queries.

Lastly, a similar topic is the approximate range emptiness problem, whose goal is to determine whether a set of keys does not contain any keys that are part of a specific range. Alexiou et al. [2] proposed ARF to solve approximate range emptiness problem in database applications, which is analyzed in further details by Goswami et al. [18]. However, these studies refer to a range over the element key space, rather than a temporal range.

9 CONCLUSION

This paper presents Persistent Bloom Filter, a probabilistic data structure to support temporal membership testing using compact space. An interesting future work is to extend our design over a temporal set, i.e., support *range membership* testing over a *temporal range*. For example, if $\mathcal{A}[t = 10, t = 16]$ contains any keys from $[140, 200]$. Another interesting direction is to support deletions.

REFERENCES

[1] Edgar log file data set. *https://www.sec.gov/data/edgar-log-file-data-set.*
[2] K. Alexiou, D. Kossmann, and P. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *PVLDB*, 6(14):1714–1725, 2013.
[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.
[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[5] G. S. Brodal, K. Tsakalidis, S. Sioutas, and K. Tsichlas. Fully persistent b-trees. In *SODA*, pages 602–614. SIAM, 2012.
[6] A. Z. Broder and M. Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.
[7] J. Bruck, J. Gao, and A. Jiang. Weighted bloom filter. In *2006 IEEE International Symposium on Information Theory*. IEEE, 2006.
[8] L. Carter, R. W. Floyd, J. Gill, G. Markowsky, and M. N. Wegman. Exact and approximate membership testers. In *STOC*, pages 59–65. ACM, 1978.
[9] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA*, pages 30–39, 2004.
[10] S. Cohen and Y. Matias. Spectral bloom filters. In *SIGMOD Conference*, pages 241–252. ACM, 2003.
[11] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.
[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
[13] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational geometry. In *Computational geometry*. Springer, 2000.
[14] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.
[15] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88. ACM, 2014.
[16] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM TON*, 8(3):281–293, 2000.
[17] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.
[18] M. Goswami, A. G. Jørgensen, K. G. Larsen, and R. Pagh. Approximate range emptiness in constant time and optimal space. In *SODA*, pages 769–775. SIAM, 2015.
[19] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. The dynamic bloom filters. *IEEE Trans. Knowl. Data Eng.*, 22(1):120–133, 2010.
[20] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *PODS*, pages 41–52. ACM, 2010.
[21] A. Kumar, J. J. Xu, and E. W. Zegura. Efficient and scalable query routing for unstructured peer-to-peer networks. In *INFOCOM*, pages 1162–1173. IEEE, 2005.
[22] D. B. Lomet, R. S. Barga, M. F. Mokbel, G. Shegalov, R. Wang, and Y. Zhu. Immortal DB: transaction time support for SQL server. In *SIGMOD Conference*, pages 939–941. ACM, 2005.
[23] D. B. Lomet and F. Li. Improving transaction-time DBMS performance and functionality. In *ICDE*, pages 581–591. IEEE Computer Society, 2009.
[24] D. B. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD Conference*, pages 315–324. ACM Press, 1989.
[25] M. Mitzenmacher. Compressed bloom filters. In *PODC*, pages 144–150. ACM, 2001.
[26] C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.
[27] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *SODA*, pages 823–829. SIAM, 2005.
[28] C. Plattner, A. Wapf, and G. Alonso. Searching in time. In *SIGMOD Conference*, pages 754–756. ACM, 2006.
[29] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.
[30] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *INFOCOM*. IEEE, 2002.
[31] A. D. Sarma, M. Theobald, and J. Widom. LIVE: A lineage-supported versioned DBMS. In *SSDBM*, volume 6187 of *Lecture Notes in Computer Science*, pages 416–433. Springer, 2010.
[32] R. Shaull, L. Shrira, and H. Xu. Skippy: a new snapshot indexing method for time travel in the storage manager. In *SIGMOD Conference*, pages 637–648. ACM, 2008.
[33] L. Shrira and H. Xu. SNAP: efficient snapshots for back-in-time execution. In *ICDE*, pages 434–445. IEEE Computer Society, 2005.
[34] Y. Tao, K. Yi, C. Sheng, J. Pei, and F. Li. Logging every footstep: quantile summaries for the entire history. In *SIGMOD*, pages 639–650, 2010.
[35] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE CST*, 14(1):131–155, 2012.
[36] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Trans. Knowl. Data Eng.*, 9(3):391–409, 1997.
[37] Z. Wei, G. Luo, K. Yi, X. Du, and J. Wen. Persistent data sketching. In *SIGMOD Conference*, pages 795–810. ACM, 2015.
[38] M. Zhong, P. Lu, K. Shen, and J. I. Seiferas. Optimizing data popularity conscious bloom filters. In *PODC*, pages 355–364. ACM, 2008.

ACKNOWLEDGMENTS

Feifei Li and Yanqing Peng are supported in part by NSF grants 1443046 and 1619287. Feifei Li is also supported by NSFC grant 61729202. Jinwei Guo, Weining Qian and Aoying Zhou are supported by National Hightech R&D Program (863 Program) under grant number 2015AA015307, and NSFC under grant numbers 61432006 and 61672232. The authors greatly appreciate the valuable feedback provided by the anonymous SIGMOD reviewers.

A OTHER ALTERNATIVES

A.1 Using a Persistent Binary Search Tree

The classic binary search tree (BST) is widely used to maintain a map, and it can be used to answer frequency counting, in which case it maintains a mapping from elements to their counters.

A persistent binary search tree (PBST) keeps all historical versions after each insertion. Therefore, it can be used to answer a tmt-query by calculating the difference between the frequency values of an element at different timestamps. On a query $q(x, [s, e])$, we simply retrieve the PBST versions at time instances $(s-1)$ and e, and query the counters of element x at both trees. If the difference is non-zero, then x must have appeared in $[s, e]$.

PBST provides exact answers to tmt-queries. It stores the entire input stream and thus uses much more space than approximate solutions. As shown in Figure 14, PBST needs a large number of bits to answer tmt-queries for a dataset with roughly 20 insertions per second, and its space cost increases linearly as time increases, making it impractical for answering tmt-queries in many cases.

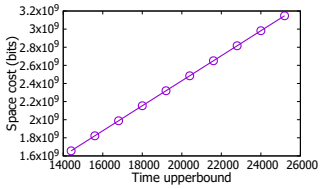


Figure 14: Space cost of PBST for answering tmt-queries.

A.2 Using a Multiversion Bloom Filter

A purely functional implementation of a data structure uses only immutable objects for the data structure. This restriction ensures that the data structure always preserve all previous versions [26], and thus making the data structure persistent. However, preserving all previous versions in a bloom filter is not helpful for answering temporal membership testing.

Assume that we are able to retrieve any historical version of a bloom filter with a purely functional implementation. Then we can retrieve its content at any timestamp t, which represents the set of elements arriving not later than t.

Given a tmt-query $q(x, [s, e])$. If x has NOT appeared before s, but has then appeared between s and e, using the two versions of the bloom filter at $(s-1)$ and e respectively, we will be able to assert the appearance of x in $[s, e]$ with high probability (unless

the bloom filter version at $(s-1)$ returns a false positive, in which case we are not sure if x appeared in $[s,e]$ or x appeared before s).

Similarly, if x has NOT appeared before s, AND has also NOT appeared between s and e, using the two versions of the bloom filter at $(s-1)$ and e respectively, we will be able to assert the non-appearance of x in $[s,e]$ *with high probability* (unless the bloom filter version at either $(s-1)$ or e has returned a false positive).

But this breaks down for the case when x had appeared before timestamp s or the bloom filter version at timestamp $(s-1)$ already reports a false positive when being queried for x. In either of these two cases, whether or not x appears in $[s,e]$, all bits in any subsequent versions of the bloom filter corresponding to x always remain 1 after timestamp s. We are not able to tell if x has appeared in $[s,e]$ or not. Therefore, the idea of answering temporal membership testing by a purely functional bloom filter (or a multi-version bloom filter) doesn't work.

B PROOFS

B.1 Proof of Theorem 5.1

Theorem B.1. PBF-1 *only has one-sided error (false positive).*

Proof. Assume there exists $(a,t) \in \mathcal{A}$. For a tmt-query $(a,[s,e])$ where $t \in [s,e]$, let $C([s,e]) = \{[s_i,e_i]\}$ be the cover of $[s,e]$ for $i \in [1,f]$. Suppose $t \in [s_i,e_i]$. Then a must have been inserted into b_i, and (a,t) must be inserted into the SBF in PBF-1. Since a bloom filter has no false negatives, the algorithm will always detect that $a \in b_i$ and $(a,t) \in$ SBF, so it always returns YES. □

B.2 Proof of Theorem 5.2

Theorem B.2. *The insertion cost (for one pair (a,t)) of* PBF-1 *is $O(\log(T/g))$ insertions into a standard bloom filter, and the query cost of a* tmt-query $q(x,[s,e])$ PBF-1 *is $O(\log((e-s)/g)+g)$* mt-queries *into standard bloom filters.*

Proof. All nodes in the same level are disjoint, and cover the whole time range. So there is exactly one node in each level that contains t. As a result, the element is inserted to each level for once, and the SBF in PBF-1 for once. There are $\log T/g$ levels in the binary tree, so the element is inserted for $1+\log T/g$ times. Because the time range $[s,e]$ is decomposed into at most $2\log((e-s)/g)$ dyadic ranges, each corresponds to a standard bloom filter where the element is checked. There are at most $(2g-2)$ timestamps that are covered by the dyadic range but are not in the query range, so the SBF in PBF-1 will be accessed for at most $(2g-2)$ times. Accordingly, $O(\log((e-s)/g)+g)$ mt-queries needs to be executed. □

B.3 Proof of Theorem 5.3

Theorem B.3. PBF-2 *only has one-sided error (false positive).*

Proof. Assume there exists $(a,t) \in \mathcal{A}$, so $(a,t_\ell) \in \mathcal{A}_\ell$ for $\ell \in [0,L-1]$. For a tmt-query $(a,[s,e])$ where $t \in [s,e]$, let $C([s,e]) = \{[s_i,e_i]\}$ be the cover of $[s,e]$ for $i \in [1,f]$. Suppose $t \in [s_i,e_i]$, so $c_i = t_{\ell_i}$ according to the mapping defined in (7). Then $(a,c_i) \in \mathcal{A}_{\ell_i}$ must have been inserted into b_{ℓ_i}. Since a standard Bloom filter has no false negatives, the algorithm will always detect that $(a,c_i) \in b_{\ell_i}$, so it guarantees to return YES. □

B.4 Proof of Theorem 5.4

Theorem B.4. *The insertion cost (for one pair (a,t)) of* PBF-2 *is $O(\log T)$ insertions into a standard bloom filter, and the query cost of a* tmt-query $q(x,[s,e])$ PBF-2 *is $O(\log(e-s))$* mt-queries *into standard bloom filters.*

Proof. Since PBF-2 contains $L = \lceil \log T \rceil + 1$ levels and an element needs to be inserted to each level for once, the insertion of a pair (a,t) needs $O(\log T)$ standard Bloom filter insertions. Because the time range $[s,e]$ is decomposed into at most $2\log(e-s)$ dyadic ranges, each one where the element is tested can be converted to a standard mt-query, hence, the cost of tmt-query $q(x,[s,e])$ in PBF-2 is $O(\log(e-s))$ mt-queries. □

C SPACE-ACCURACY TRADEOFF

C.1 Space-false positive rate tradeoff

In practice, we are also interested in knowing: in order to keep the false positive rate less than a desirable threshold p, how many bits do we need? In order to derive an upper bound for PBF-1 and PBF-2, we introduce an auxiliary structure whose space cost is easy to analyze, and show that PBF-1 and PBF-2 always outperform it (in terms of accuracy) when given the same number of bits.

Definition C.1. For a PBF-1 (or PBF-2) $\mathcal{P}$, we define a corresponding auxiliary structure $\mathcal{S}$. $\mathcal{S}$ is a single Bloom filter with m bits, where m is the total number of bits used by $\mathcal{P}$. Whenever we insert an element o into the i-th bloom filter of $\mathcal{P}$, we insert $o' = (o,i)$ into $\mathcal{S}$ instead. Whenever we have to query o on the i-th bloom filter of $\mathcal{P}$, we query $o' = (o,i)$ on $\mathcal{S}$ instead.

Lemma C.2. *For any optimized $\mathcal{P}$, its corresponding structure $\mathcal{S}$ always has worse performance than $\mathcal{P}$ in terms of false positive rates.*

Proof. We will show that if we assign the number of bits proportional to the distinct number of elements d_i for each bloom filter b_i in $\mathcal{P}$, then $\mathcal{S}$ and $\mathcal{P}$ will have the same false positive rates. Henceforth, a PBF-1 (PBF-2) instance $\mathcal{P}$ with optimal bits allocation as discussed in Sections 5.1 and 5.2 will always have lower false positive rates than S.

More specifically, let $\theta = m/\sum d_i$ where d_i is the number of distinct elements in the i-th bloom filter of $\mathcal{P}$. In other words, θ is the ratio between the total number of bits and the total number of distinct elements from all bloom filters in $\mathcal{P}$. Now, assume that for each bloom filter b_i in $\mathcal{P}$, we allocate $m_i = \theta \times d_i$ bits to b_i.

In this allocation scheme, all bloom filters in $\mathcal{P}$ have the same false positive rates, which equals to the false positive rate of $\mathcal{S}$. For a query that needs to access t bloom filters in $\mathcal{P}$, it is converted to access $\mathcal{S}$ for t times, so the false positive rates of any query to $\mathcal{P}$ and its corresponding query to $\mathcal{S}$ are the same. In other words, $\mathcal{S}$ and $\mathcal{P}$ have identical false positive rates in this allocation scheme.

Clearly the above allocation scheme is a special case, and the instance of $\mathcal{P}$ with the optimal bit allocation scheme will thus always outperform $\mathcal{S}$ (using the same number of bits). □

Note that this lemma works for both PBF-1 and PBF-2, therefore we can derive the space upper bound, in order to meet a desirable false positive rate, for both PBF-1 and PBF-2 by analyzing $\mathcal{S}$.

Theorem C.3. *For any optimized* PBF-1 *with* $g = 1$ *or* PBF-2 *instance* $\mathcal{P}$*, assume that the time upper bound is* T*, and the number of distinct pairs in the stream* $\mathcal{A}$ *is* $n = n(\mathcal{A})$*. We need* m *bits to ensure that* $\mathcal{P}$ *answers a* tmt-query q *with a false positive rate* p*, where*

$$m = n \log T \ln\left(\left(1 - (1-p)^{1/(2\log|q|)}\right)^{-1}\right) \Big/ \ln^2 2. \quad (14)$$

Here for simplicity we assume $g = 1$ for PBF-1. For $g > 1$, the $2\log|q|$ term becomes $(2\log|q/g| + g - 1)$.

Proof. We construct the auxiliary structure $\mathcal{S}$ as described in Definition C.1. Since $\mathcal{S}$ is actually a classic bloom filter, its false positive rate for a single query can be derived by Equation 3, which is $2^{-\ln 2 \frac{m}{n \log T}}$. Therefore, the false positive rate for $\mathcal{S}$ to answer q is at most (note that a tmt-query q is decomposed into at most $2\log|q|$ queries into PBF-1 or PBF-2):

$$1 - \left(1 - 2^{-\ln 2 \frac{m}{n \log T}}\right)^{2\log|q|} = p. \quad (15)$$

Rearranging this equation will lead to Equation 14. By Lemma C.2, this gives an upper bound for the space cost of $\mathcal{P}$. □

C.2 Impact of time granularity

Time granularity affects the number of time instances that PBF-1 and PBF-2 need to deal with. It is possible that in certain applications, raw timestamps can be as small as a microsecond. But often times, we don't necessarily need such a high resolution for queries. Next, we study the tradeoff between the time granularity and the number of bits needed for a desirable false positive rate.

For a PBF, using 2x granularity (i.e., split each time unit into two) is equivalent to enlarge the time upper bound from T to $2T$, while keeping the same number of insertions. Furthermore, a query $q = (x, [s, e])$ for the original granularity will be mapped to $q' = (x, [2s, 2e])$ under the 2x granularity, and its length doubles.

We can formally analyze the impact to false positive rates of PBF-1 and PBF-2 by Equation 14. Clearly, the number of distinct pairs n is unchanged. The change in the $\log|q|$ term is small (increase by 1 under 2x granularity even in the worst case analysis), so we can ignore its effect for simplicity. As a result, the number of bits m grows at roughly $O(\log T)$. Remember that this value m is an upper bound on number of bits needed to achieve a desirable false positive rate p. Therefore, the space cost grows at most logarithmically when using a finer granularity (e.g., the $\log T$ term in Equation 14 becomes $\log 2T$).

C.3 Impact of time upper bound

We are also interested in how many bits we have to use for a time span T, in order to achieve a desirable false positive rate p.

Again, by looking at Equation 14, the space cost grows at roughly $O(T \log T)$, assuming in the general case that $n = O(T)$ (i.e., there is at least one pair arriving at each time instance so that the distinct number of pairs grows over time). In practice, the time upper bound is unlikely to be extremely large (say larger than 2^{64}). Therefore, this growth rate is just slightly greater than linear growth in most cases. Note that this is an upper bound, and the actual number of bits needed grows only linearly as T (and hence n) grows in order to maintain a stable false positive rate, as shown in Section D.1.

In fact, linear increase of m is required to maintain a good false positive rate while T (and hence n) grows. Assume that there is a structure that only requires a sub-linear growth of m while keeping a stable false positive rate, then the information encoded in each bit (i.e. the compression ratio) grows super-linearly, and finally reaches infinity when T is large enough. That's clearly impossible.

This also implies that in streaming setting where the time upper bound is potentially unbounded, we have to choose an appropriate time upper bound T to maintain a desirable error (false positive) rate with reasonable space cost. But this is true for any other probabilistic data structures designed for streaming settings. For example, if we use a bloom filter with fixed number of bits for an unbounded stream, its performance deteriorates when the stream grows and inserts increasingly more distinct number of elements.

D ADDITIONAL EXPERIMENTAL RESULTS

D.1 Scalability of accuracy-space tradeoff

In our earlier experiments, we've assumed that the time granularity is 1 second and the time span is one day ($T \leq 86400$ in seconds). Next, we will investigate the impacts of the time granularity and the time upper bound. In particular, we want to understand the accuracy-space tradeoff as finer time granularity are used or the time span has increased to very large values.

That said, we experimented with the Network dataset and set the target false positive rate to 5%. Other parameters are set to the default values in Table 3 unless otherwise specified.

We first show what happens when we pick a finer time granularity. In this experiment, we refine the time granularity by randomly mapping a timestamp to a value in the corresponding finer time range (e.g., map $1s$ to a random value in $[1000ms, 2000ms)$ if we use ms instead of s as the time granularity). The finest granularity we use in this experiment is $1ms$.

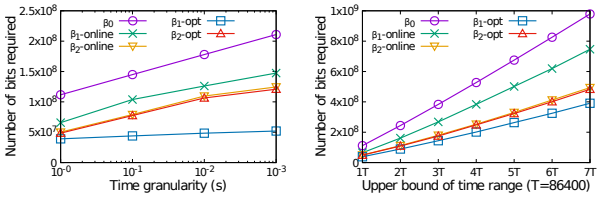
Figure 15(a) shows the impact of time granularity. As shown in Appendix C.2, the bits needed to maintain a desirable, stable false positive rate for all PBFs grow in a sub-logarithmic speed.

Next, we discuss the case for larger time span (time upper bound) T. We enlarge T by duplicating the dataset for up to 7 times, and tested how many bits we have to use in order to maintain roughly the same false positive rate. The trend is shown in Figure 15(b). As discussed in Appendix C.3, we showed that all PBFs will have, in the worst case, at most $T \log T$ growth of bits with T. Figure 15(b) indicates that in practice we only need a roughly linear growth on the number of required bits as T (and hence n) increases.

Lastly, as we have already pointed out above, the accuracy of many probabilistic data structures depends on the length of the stream when used in a streaming setting (or more accurately, depending on certain properties of the stream which often change as the stream grows, e.g., distinct number of elements). More bits are needed if a desirable accuracy is required as the underlying stream grows indefinitely. For example, consider a standard bloom filter. When it is used in a streaming setting, as the stream grows, there will be increasingly more distinct number of elements being inserted into the bloom filter and its false positive rate will start to increase unless more bits are used.

D.2 Efficiency under different workloads

We next study the overall efficiency of PBFs with different workloads. In this experiment, we fix the number of operations (insertion

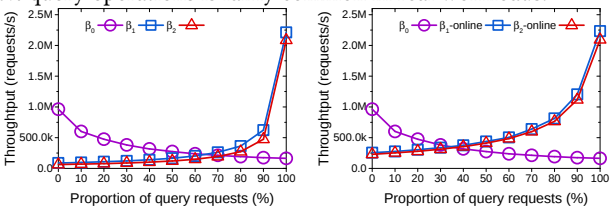


(a) Finer granularity. (b) Larger time span.

Figure 15: Scalability tests for space-accuracy tradeoff.

or query) to be 1 million, and all other parameters (e.g., query length, time upper bound, etc.) to be the default parameter values in Table 3. We then gradually increase the ratio of query operation.

We test the throughput on SBF, basic PBFs and online PBFs. Figure 16 shows that the throughput of PBF-1 and PBF-2 drops, whereas the throughput of SBF increases, when the percentage of insertion (query) operations in a workload increases (drops). The performances of SBF and PBFs reach a tie when the ratio of insertions increase to about 40% for the base case of PBFs, and when the ratio of insertions increase to 70% for online PBFs. In the extreme case that all operations are insertions, the PBFs only have throughput 1/17 of the SBF (because for each insertion in SBF it takes $\lfloor 1 + \log(86400) \rfloor = 17$ insertions for PBF). However, when the ratio of query operations is higher than this point (i.e., more than 60% query operations for the base PBFs and more than 30% query operations for the online PBFs respectively), the throughput of PBFs raises dramatically and outperforms SBF significantly. The online PBFs are more efficient than their base cases, which confirms our earlier analysis. Note that we expect that having more than 30% query operations is fairly common in real workloads.



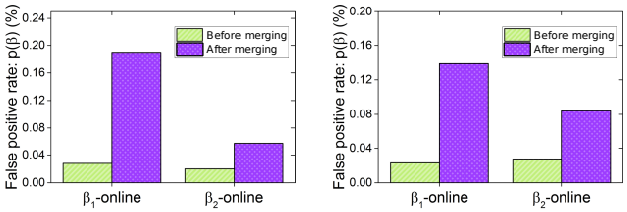
(a) Base case. (b) Online case.

Figure 16: Throughput of β_0, β_1, β_2, β_1-online, β_2-online: ratio of insertion and query operations changes in a workload.

D.3 Impact of merging

As mentioned in Section 6, when being used in a streaming application, as time continues to grow and in the rare case where a tight space budget is to be observed at all time, we might have to merge two PBFs that correspond to two neighboring partitions (on the time dimension; see Figure 6) to stay within a given space budget. This is a rare operation due to the fact that PBFs are highly effective with small number of bits for a relatively large temporal range, thus, maintaining multiple PBFs over temporal partitions is not likely to lead to a large memory footprint. And even if over a long period of time this results in too many PBFs, we can often "retire" PBFs for partitions from remote history, by writing them back to disk, as they are unlikely to be queried for.

In fact, the same argument goes for the classic bloom filter. When a bloom filter is used in a streaming fashion, it rarely needs a merge operation. And the same partition idea can be applied if needed.



(a) EDGAR. (b) Network.

Figure 17: Impacts of merging two neighboring PBFs: $T = 32,768$ for each temporal partition before the merge; $T = 65,536$ for the partition after the merge; $m = 5 \times 10^7$.

We discussed how to merge two PBFs if needed in Section 6, and investigate the impact of merging next. We carried out this experiment with two neighboring temporal partitions, each with a temporal length $T = 32,768$, on both EDGAR and Network data sets. In other words, after the merge, there is one partition left with a temporal range length $T = 65,536$.

We are interested to understand the impact of the merge operation to the effectiveness of the underlying PBFs, i.e., how their false positive rates are influenced by the merge. Figure 17 shows the results of this experiment, where we used the default number of bits ($m = 5 \times 10^7$) for the online optimization versions of PBF-1 and PBF-2 (β_1-online and β_2-online, respectively).

Both PBF-1 and PBF-2 suffer a clear loss of accuracy due to the merge operation as indicated in Figure 17: their false positive rates are clearly higher than that of before merging on both data sets. Nevertheless, the good news is that they are still highly effective even after the merging operation. As shown in Figure 17, their false positive rates are still less than 0.20% and 0.06% for β_1-online and β_2-online respectively on EDGAR, and less than 0.15% and 0.08% for β_1-online and β_2-online respectively on Network.

D.4 Bits allocation cost

Finally, we investigate the efficiency of optimizing PBF-1 and PBF-2, i.e., what is the cost of doing bits allocation for PBF-1 and PBF-2 respectively. As discussed in Section 4.2, a disadvantage of PBF-1 is that it uses $O(T)$ number of bloom filters. As a result, its optimization cost is much more expensive than that of PBF-2.

Figure 18 shows their optimization execution time using the EDGAR dataset with default parameters. While PBF-1 takes about 0.1s to execute the bits allocation, PBF-2 takes only 100μs. Furthermore, the execution time of PBF-1 grows quickly when T increases, while that for PBF-2 remains roughly stable (logarithmic to T).

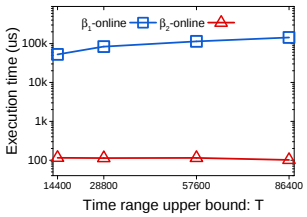


Figure 18: Optimization execution time of β_1-online, β_2-online.