Inferring Insertion Times and Optimizing Error Penalties in Time-Decaying Bloom Filters

JONATHAN L. DAUTRICH JR., Google, USA
CHINYA V. RAVISHANKAR, University of California, Riverside, USA

Current Bloom Filters tend to ignore Bayesian priors as well as a great deal of useful information they hold, compromising the accuracy of their responses. Incorrect responses cause users to incur penalties that are both application- and item-specific, but current Bloom Filters are typically tuned only for static penalties. Such shortcomings are problematic for all Bloom Filter variants, but especially so for Time-Decaying Bloom Filters, in which the memory of older items decays over time, causing both false positives and false negatives.

We address these issues by introducing *inferential* filters, which integrate Bayesian priors and information latent in filters to make penalty-optimal, query-specific decisions. We also show how to properly infer insertion times in such filters. Our methods are general, but here we illustrate their application to *inferential time-decaying filters* to support novel query types and sliding window queries with dynamic error penalties.

We present inferential versions of the Timing Bloom Filter and Generalized Bloom Filter. Our experiments on real and synthetic datasets show that our methods reduce penalties for incorrect responses to sliding-window queries in these filters by up to 70% when penalties are dynamic.

CCS Concepts: • **Information systems → Probabilistic retrieval models**; *Stream management*;

ACM Reference Format:
Jonathan L. Dautrich Jr. and Chinya V. Ravishankar. 2019. Inferring Insertion Times and Optimizing Error Penalties in Time-Decaying Bloom Filters. 1, 1 (January 2019), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Bloom Filters are probabilistic data structures used for set membership queries [34]. Although this data structure derives its name from [2], the method is equivalent to the Zatocoding technique described in 1951 by Mooers [25] for encoding information onto punched cards. Bloom Filters have been widely applied in areas where a concise but approximate representation of sets is required. They have, for example, been used to estimate join sizes and to speed up joins [18, 22, 26, 27]. Oracle releases 10.2.0.x and later use Bloom Filters to reduce traffic between parallel query slaves during join processing. Bloom Filters have natural applications in stream processing and duplicate detection [24]. Other applications include maintaining differential files [12] and for spell checking [10, 23]. For surveys of variant Bloom Filter designs and comparisons, see [3, 20].

A Bloom Filter $\mathcal{F}$ comprises an array of m cells and k hash functions $h_1, \ldots, h_k$. An item x is *inserted* into $\mathcal{F}$ by updating the contents of the cells at indices $h_1(x), \ldots, h_k(x)$. The contents of $\mathcal{F}$'s cells define its state $\widehat{\mathcal{F}}$. The set of items inserted into $\mathcal{F}$ is denoted $\{\mathcal{F}\}$.

Authors' addresses: Jonathan L. Dautrich Jr., Google, 19510 Jamboree Road, Irvine, CA, 92612, USA, jjldjr@gmail.com; Chinya V. Ravishankar, University of California, Riverside, 454 Winston Chung Hall, Riverside, CA, 92521, USA, ravi@cs.ucr.edu.

XXXX-XXXX/2019/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The *Classical Bloom Filter* [2] tests if an item $x \in \{\mathcal{F}\}$, returning Pos if $x \in \{\mathcal{F}\}$ and Neg if $x \notin \{\mathcal{F}\}$. Inserted items are never deleted, so $\mathcal{F}$ may become saturated, leading to *false positive* errors, returning Pos even when $x \notin \{\mathcal{F}\}$.

A *Time-Decaying Bloom Filter* [8, 15, 16, 38, 39], in contrast, supports queries that ask how recently x was inserted. New insertions obscure information from older ones, so the memory of old items decays with time, limiting saturation even for continuous streams of item insertions.

Definition 1. *The* insertion age I_x *of item x is a random variable denoting the number of items inserted since x was last inserted. If x was never inserted, we define $I_x = \perp$. Different I_x values represent mutually exclusive events.*

Time-decaying filters answer *retrospective queries*, whose predicates reference insertion ages. A typical retrospective query is the *sliding window* query, which asks whether x was one of the last w items inserted ($I_x < w$). Insertion history is only approximated by $\widehat{\mathcal{F}}$, so we may commit false positive errors, returning Pos when $I_x \geq w$, or false negative errors, returning Neg when $I_x < w$. Such errors incur penalties ultimately borne by the application using $\mathcal{F}$.

Current time-decaying filters waste much of the useful information in $\widehat{\mathcal{F}}$. For example, in [8, 33], cell counters are decremented at each insertion, and hence embed information about insertion age. Yet, these filters check only whether these counters are zero, discarding the more detailed information available. Even filters that *do* consider exact counts [15] do not provide a clear framework for using counter values.

More importantly, filters typically operate using "forward" probabilities, ignoring Bayesian priors. Our work is motivated by the observation that ignoring priors is fundamentally incorrect, and often leads to worse results than using no filter at all. A similar result was reported in [31].

1.1 Inferential Time-Decaying Filters

We present *inferential* time-decaying filters to address these issues. Inferential filters combine latent information in $\widehat{\mathcal{F}}$ with Bayesian priors to infer *posterior* probabilities.

Definition 2. $P(I_x = i|\widehat{\mathcal{F}})$ *is the posterior probability that item x has insertion age i, given the filter state $\widehat{\mathcal{F}}$.*

A *standard* time-decaying filter uses limited information from $\widehat{\mathcal{F}}$ to respond Pos or Neg to sliding window queries. An *inferential* time-decaying filter uses $P(I_x = i|\widehat{\mathcal{F}})$ to achieve greater flexibility and accuracy in answering queries.

False positives/negatives incur application-dependent error penalties. Standard filters may be tuned to minimize static penalties that are fixed at filter design time. In reality, however, penalties vary by queried item, time, and context. A wrong decision on a high-value item costs more than one on a low-value item. Scenarios with query-specific penalties include duplicate detection for items with different values [1], distributed caches with item-specific access times [32], and web crawler caches when pages vary in importance [28].

Optimally, each membership decision should be made dynamically, query-by-query, and minimize expected penalty. Inferential time-decaying filters infer the *sliding window posterior* probability $P(I_x < w|\widehat{\mathcal{F}}) = \sum_{i=0}^{w-1} P(I_x = i|\widehat{\mathcal{F}})$ for each sliding window query. They then use this posterior to compute expected penalties of Pos and Neg responses and make minimum-penalty decisions for each query.

Inferential filters also support novel retrospective queries, beyond enabling minimum-cost decisions. For instance, $P(I_x = i|\widehat{\mathcal{F}})$ can be used to find the most likely insertion age for x. Aggregating over all i gives the *expected* insertion age. As far as we know, our work is the first to support such queries using Bloom Filters.

1.2 Contributions

As noted above, filters have typically operated using "forward" probabilities, ignoring Bayesian priors. Ignoring priors, however, is fundamentally incorrect. We show how to turn existing *standard* filters into *inferential* ones, using Bayesian priors and latent information in $\widehat{\mathcal{F}}$. Section 2 outlines our inferential filter framework. We focus primarily on time-decaying filters, but our framework also immediately yields a more accurate version of the Classical Bloom Filter (see Section 2.4).

We show details of how to develop an inferential version of *Timing Bloom Filters* (TBF) [38], and use it for sliding window queries. We also develop standard and inferential versions of a space-efficient TBF variant called the Block TBF (BTBF), conceptualized in [38]. We discuss standard and inferential BTBFs in Sections 3 and 4, respectively.

We also develop an inferential version of the Generalized Bloom Filter (GBF) [16], in which each cell is a single bit. As new items are inserted, memory of old items steadily decays. The standard GBF has no built-in notion of a sliding window, so the GBF is less accurate than the BTBF when the window width w is fixed, but the inferential GBF can support windows of different w for each query. We discuss standard and inferential GBFs in Sections 5 and 5.3, respectively.

In Section 6, we compare the net penalty incurred by the standard and inferential BTBF and GBF on real and synthetic data streams, randomly varying error penalties for sliding window queries. We also compare them against two baselines, the first of which uses only on prior probabilities. The second is a simple buffer that stores hashes of items in the window. The buffer upper bounds the accuracy of sliding window techniques that require stored items, including those using Counting Bloom Filters [13, 35, 36]. Our results show that the inferential filters improve substantially upon the standard filters, reducing penalties when Bayesian priors are known. We discuss related work in Section 7.

2 INFERENTIAL FILTER FRAMEWORK

Bloom Filter variants commonly consist of an array of m cells and k independent hash functions $h_1, \ldots, h_k$, where hash h_i maps an item x to a cell $h_i(x)$ in the filter. Notation from Sections 1 and 2 is summarized in Table 1.

Definition 3. *The set R_x of cells* touched *by item x is given by $R_x = \{h_1(x), \ldots, h_k(x)\}$.*

To insert an item x into filter $\mathcal{F}$, we update each cell in R_x according to the rules of $\mathcal{F}$. To query for x, we inspect each cell in R_x, and return Pos or Neg as appropriate. Let n be the number of past insertions.

2.1 The Classical Bloom Filter

The Classical Bloom Filter [2] represents the set $\{\mathcal{F}\}$ of all items inserted into the filter ($n = |\{\mathcal{F}\}|$). Each cell is a single bit initialized to 0. To insert x, each cell in R_x is set to 1. Some cells may be touched by multiple items. A query for x returns Pos if and only if all cells in R_x are 1.

Figure 1 shows inserts and possible query outcomes for a Classical Bloom filter. Cells are never reset to 0, so all cells in R_x remain 1 if $x \in \{\mathcal{F}\}$. There are no false negatives, but a false positive occurs if $x \notin \{\mathcal{F}\}$ but every cell in R_x has been touched by some item, as for x_3.

Let $r_x = |R_x|$. The probability that a given cell is *not* touched by a given insertion is $(1 - 1/m)^k$. Thus, the probability that a given cell *is* touched by at least one of the n items in $\{\mathcal{F}\}$ is $(1 - (1 - 1/m)^{kn})$. When $x \notin \{\mathcal{F}\}$, the *false positive* probability that all r_x cells in R_x are set to 1 (touched) by at least one item in $\{\mathcal{F}\}$ is:

$$P_{FP} \approx \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{r_x} \tag{1}$$

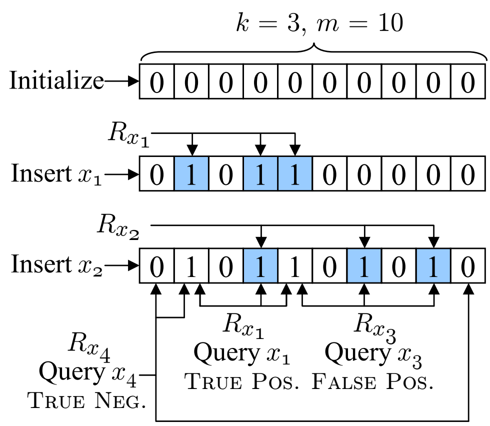


Fig. 1. Inserts and queries on a Classical Bloom filter holding $\{x_1, x_2\}$.

2.2 Analytical Approximations

P_{FP} in Equation 1 is often approximated by replacing r_x with k, as collisions are rare if $m \gg k$. Equation 1 assumes that cell touches in R_x independent events, which is strictly incorrect [6]. Such approximations usually have little impact on accuracy [6], but greatly simplify analysis. We make similar assumptions in our paper when computing posteriors. Our experiments show that the posteriors are generally accurate enough to substantially reduce error penalties.

2.3 Probability Functions

The posterior $P(I_x = i|\widehat{\mathcal{F}})$ is conditioned on $\widehat{\mathcal{F}}$, the filter state, which includes all cells in $\mathcal{F}$. However, most information relevant to x is present in the cells R_x.

DEFINITION 4. *$P(I_x = i|R_x)$, also denoted $P(i|R_x)$, is the posterior probability that exactly i insertions occurred since x was last inserted, given the current contents of cells R_x.*

To turn a standard filter into an inferential one, we must compute $P(i|R_x)$, based on the filter's contents and on the prior probability that $I_x = i$. From Bayes' theorem,

$$P(i|R_x) = \frac{P(i)P(R_x|i)}{P(R_x)}, \qquad \text{where} \tag{2}$$

$P(I_x = i)$ or $P(i)$ is the prior probability that exactly i insertions occurred since x was last inserted, $P(R_x)$ is the prior probability that cells R_x have their current contents, and $P(R_x|I_x = i)$ or $P(R_x|i)$ is the conditional probability that cells R_x have their current contents, given that exactly i insertions occurred since x was last inserted.

2.3.1 *Computing Prior Probability $P(i)$.*

DEFINITION 5. *The* sample probability *mass function p_x is the probability that x is the next item to be inserted.*

Let U be the universe of all items that may be inserted or queried. For any two items $x \neq y$, p_x and p_y may differ, but we assume that p_x itself is time-invariant, giving:

$$P(i) = \begin{cases} p_x(1-p_x)^i & \text{if } i \neq \perp \ (0 \leq i < n) \\ (1-p_x)^n & \text{if } i = \perp \end{cases} \tag{3}$$

Table 1. General Notation

$\mathcal{F}$	A Bloom Filter
$\widehat{\mathcal{F}}$	The state of Bloom Filter $\mathcal{F}$
$\{\mathcal{F}\}$	Set of all items inserted into a filter
x	Item to be inserted or queried
$n = \|\{\mathcal{F}\}\|$	Total number of items inserted
w	Width of sliding window
U	Universe of items inserted/queried
p_x	Sample probability of x
m	Number of cells in filter
k	Number of hash functions used in filter
$h, h_1(x)$	Hash function, cell touched by h_1 on x
I_x, i	Number of insertions since x last inserted
$\perp$	$I_x = \perp$ means x was never inserted
R_x	Cells touched by k hashes applied to x
r_x	$\|R_x\|$
c_x	For standard filter, number of 1-bits in R_x
P_{FP}	False positive probability of standard filter
$P(i)$	Prior prob. i insertions since x last inserted
$P(R_x\|i)$	Conditional probability of R_x given $I_x = i$
$P(i\|R_x)$	Posterior prob. of $I_x = i$ insertions since x last inserted, given contents of cells in R_x
$P(I_x < w\|R_x)$	Posterior prob. x one of last w insertions
$D(j)$	Expected num. distinct items in j inserts

$P(\perp)$ is the probability that x was not inserted thus far, and $P(i), i \neq \perp$ is the probability that x was inserted, followed by i items other than x. We say the data stream is continuous when the number of past insertions n goes to infinity, giving:

$$\lim_{n\to\infty} P(i) = \begin{cases} p_x(1-p_x)^i & \text{if } i \neq \perp \\ 0 & \text{if } i = \perp \end{cases} \tag{4}$$

Often, we need $P(\alpha \leq I_x < \beta)$ for $0 \leq \alpha < \beta$:

$$\begin{aligned} P(\alpha \leq I_x < \beta) &= \sum_{i=\alpha}^{\beta-1} P(i) = p_x \sum_{i=\alpha}^{\beta-1} (1-p_x)^i \\ &= (1-p_x)^\alpha - (1-p_x)^\beta \\ &= (1-p_x)^\alpha (1-(1-p_x)^{\beta-\alpha}). \end{aligned} \tag{5}$$

2.3.2 Computing Posterior $P(i|R_x)$. $P(R_x)$ in Equation 2 can be written as a marginal sum over $P(i)P(R_x|i)$, giving:

$$P(i|R_x) = \frac{P(i)P(R_x|i)}{P(\perp)P(R_x|\perp) + \sum_{i'=0}^{n-1} P(i')P(R_x|i')}. \tag{6}$$

We still need $P(R_x|i)$ and $\sum_{i'=0}^{n-1} P(i')P(R_x|i')$. Both challenges are filter-specific, so we address them for the BTBF and GBF in Sections 4 and 5.3, respectively.

2.3.3 Retrospective Queries. Inferential time-decaying filters use $P(i|R_x)$ in responding to retrospective queries. $\max_i P(i|R_x)$ gives the highest probability choice i for I_x, which is when x was most likely last inserted.

For a continuous stream ($n \to \infty$), we get

$$\lim_{n\to\infty} P(i|R_x) = \frac{P(i)P(R_x|i)}{\sum_{i'=0}^{\infty} P(i')P(R_x|i')}. \tag{7}$$

The expected number of insertions after x's last insertion is

$$E[I_x|R_x] = \lim_{n\to\infty} \sum_{i=0}^{n} i \cdot P(i|R_x). \tag{8}$$

We can also derive the *sliding window posterior*

$$\begin{aligned}
\lim_{n\to\infty} P(I_x < w|R_x) &= \sum_{i=0}^{w-1} \lim_{n\to\infty} P(i|R_x) \\
&= \sum_{i=0}^{w-1} \frac{P(i)P(R_x|i)}{\sum_{i'=0}^{\infty} P(i')P(R_x|i')} = \frac{\sum_{i=0}^{w-1} P(i)P(R_x|i)}{\sum_{i=0}^{\infty} P(i)P(R_x|i)} \qquad (9)\\
&= 1 - \frac{\sum_{i=w}^{\infty} P(i)P(R_x|i)}{\sum_{i=0}^{\infty} P(i)P(R_x|i)}. \qquad (10)
\end{aligned}$$

2.4 Example: Classical Bloom Filters

As warm-up, we develop an inferential version of Classical Bloom filters (Section 2.1), by computing the posterior $P(x \in \{\mathcal{F}\}|R_x)$. Since this filter is not time-decaying, we do not need the full power of our approach, but we show our results to be consistent with the simpler derivation in [31]. We also show how to obtain optimal responses from the inferential Classical Bloom Filter, given item-specific prior probabilities p_x and query-specific error penalties. Since $n = |\{\mathcal{F}\}|$,

$$\begin{aligned}
P(x \in \{\mathcal{F}\}|R_x) = P(I_x < n|R_x) &= \textstyle\sum_{i=0}^{n-1} P(i|R_x) \\
&= \frac{\sum_{i=0}^{n-1} P(i)P(R_x|i)}{P(\perp)P(R_x|\perp) + \sum_{i=0}^{n-1} P(i)P(R_x|i)}. \qquad (11)
\end{aligned}$$

Let $r_x = |R_x|$. Let c_x cells (bits) in R_x be set to 1.

$$P(R_x|i) = \begin{cases} 1 & \text{if } c_x = r_x \text{ and } 0 \le i < n \\ 0 & \text{if } c_x \ne r_x \text{ and } 0 \le i < n \\ \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{c_x} \left(\left(1 - \frac{1}{m}\right)^{kn}\right)^{r_x - c_x} & \text{if } i = \perp \end{cases} \tag{12}$$

If x was inserted ($0 \le i < n$), then all cells in R_x must be 1 ($c_x = r_x$). If x was *not* inserted ($i = \perp$), then every one of the c_x 1-cells in R_x must have been touched (set) by some combination of the n insertions, while the remaining $r_x - c_x$ 0-cells in R_x must not have been touched by any insertion.

THEOREM 1. *The posterior probability that x was inserted into the Classical Bloom Filter is given by:*

$$P(I_x < n|R_x) = \begin{cases} 0 & \text{if } c_x \ne r_x \\ \dfrac{1}{1 + \dfrac{(1-p_x)^n P_{FP}}{1-(1-p_x)^n}} & \text{if } c_x = r_x \end{cases} \tag{13}$$

where P_{FP} is as in Equation 1.

Proof: $P(I_x < n|R_x)$ is given by Equation 11, $P(i)$ by Equation 3, and $P(R_x|i)$ by Equation 12.

Case $c_x \neq r_x$:

$$P(I_x < n|R_x) = \frac{\sum_{i=0}^{n-1} P(i) \cdot 0}{P(\perp)P(R_x|\perp) + \sum_{i=0}^{n-1} P(i) \cdot 0} = 0.$$

Case $c_x = r_x$:

$$\begin{aligned} P(I_x < n|R_x) &= \\ &\frac{\sum_{i=0}^{n-1} P(i) \cdot 1}{P(\perp)\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{r_x} + \sum_{i=0}^{n-1} P(i) \cdot 1} \\ &= \frac{(1 - P(\perp))}{P(\perp) \cdot P_{FP} + (1 - P(\perp))} \\ &= \frac{1}{1 + \frac{P(\perp) \cdot P_{FP}}{1 - P(\perp)}} = \frac{1}{1 + \frac{(1-p_x)^n \cdot P_{FP}}{1-(1-p_x)^n}}. \quad \square \end{aligned}$$

For the Classical Bloom Filter it is common to assume that $r_x = k$, $k \approx (m/n)\ln 2$ and that $P_{FP} \approx (1 - e^{-kn/m})^k$, as in [16, 31, 33, 38]. Doing so gives $P_{FP} \approx (1/2)^{(m/n)\ln 2}$, so when $c_x = r_x$, rearranging Theorem 1 and substituting $P(I_x < n) = 1 - (1 - p_x)^n$ gives

$$P(I_x < n|R_x) \approx \frac{P(I_x < n)}{P(\perp)\left(\frac{1}{2}\right)^{(m/n)\ln 2} + P(I_x < n)}, \tag{14}$$

which is consistent with the probability expressions in [31].

2.5 Expected Number of Distinct Items

The accuracy of our posteriors can be improved if we know the expected number of *distinct* items inserted during j insertions, which we label $D(j)$. $D(j)$ depends on the distribution of p_x for $x \in U$. Applying linearity of expectation, we can find $D(j)$ by summing, over all $x \in U$, the probability that x is inserted at least once, given by

$$D(j) = \sum_{x \in U} \left(1 - (1 - p_x)^j\right). \tag{15}$$

When items are sampled from the uniform distribution, we have $p_x = 1/|U|$ for all $x \in U$, and Equation 15 becomes

$$D(j) = |U|\left(1 - \left(1 - \frac{1}{|U|}\right)^j\right). \tag{16}$$

If the item probabilities follow a Zipf-like discrete power law $p_x = 1/(H_{|U|} \cdot x)$, then it is shown in [37] that

$$D(j) \approx \frac{j}{H_{|U|}}\left(1 - \gamma + \ln \frac{|U| \cdot H_{|U|}}{j}\right), \tag{17}$$

where $H_{|U|} = \sum_{i=1}^{|U|} 1/i$ is the $|U|^{\text{th}}$ harmonic number and $\gamma = 0.57721566\ldots$ is Euler's constant.

When real-world distributions are hard to model analytically, we can experimentally determine $D(j)$ for some j values, and interpolate intermediate values. Our experience suggests that piecewise logarithmic interpolation generally yields acceptable results. If we know $D(j_1)$ and $D(j_3)$, we can interpolate $D(j_2)$ for $j_1 < j_2 < j_3$ as follows:

$$\ln D(j_2) = \ln D(j_1) + \frac{\left(\ln \frac{D(j_3)}{D(j_1)}\right)\left(\ln \frac{j_2}{j_1}\right)}{\left(\ln \frac{j_3}{j_1}\right)}. \tag{18}$$

2.6 Minimum-Penalty Decisions

As noted in Section 1, penalties for incorrect responses may be query-specific, but a standard filter can only be tuned to fixed false positive/negative rates. Inferential filters use posteriors to make better-informed, query-specific decisions.

For sliding window queries, inferential filters return the sliding window posterior $P(I_x < w|R_x)$. Let $\$_{FP}$ and $\$_{FN}$ be the penalties for false positive/negative errors, respectively. Correct responses incur no penalty. The expected penalty of Pos is $E_{\text{Pos}} = \$_{FP} \cdot (1 - P(I_x < w|R_x))$, and of Neg is $E_{\text{Neg}} = \$_{FN} \cdot P(I_x < w|R_x)$. We compute both and return Pos if $E_{\text{Pos}} \leq E_{\text{Neg}}$, and Neg otherwise.

3 STANDARD TIMING BLOOM FILTERS

The Timing Bloom Filter (TBF) [38] is designed to answer sliding window queries. Here we describe the *standard* TBF and its extension, the *standard* Block Timing Bloom Filter (BTBF). We will present the *inferential* BTBF in Section 4. Table 2 summarizes the relevant notation.

3.1 Timing Bloom Filters

The TBF consists of k hash functions and an array of m cells, each of which is a *timer* with *bpt* bits. Each timer θ holds a timestamp $\theta.T \in \{0, \ldots, T_\Omega\} \cup \{T_\varepsilon\}$, where T_ε denotes an *expired* timestamp, defined below. The filter maintains a single *current timestamp* T_+, where T_+ cycles through the range $[0, T_\Omega]$ as items are inserted.

3.1.1 TBF: Insert. We insert item x into a TBF as follows:

(1) For each timer $\theta \in R_x$, set $\theta.T \leftarrow T_+$.
(2) Increment T_+: $T_+ \leftarrow (T_+ + 1) \mod (T_\Omega + 1)$.

Definition 6. *The* age $\lambda(\theta.T)$ *of timestamp* $\theta.T$ *is defined as the number of times that* T_+ *was incremented since the last time that we set* $\theta.T$ *to* T_+.

When $\lambda(\theta.T) \geq w + 1$, we say that $\theta.T$ has *expired*, and we set $\theta.T$ to the *expired timestamp* value T_ε. Thus, as soon as a timestamp $\theta.T$ is set to T_+, it has age $\lambda(\theta.T) = 0$, but since increments occur immediately after insertions, $\lambda(\theta.T) \geq 1$ before queries arrive. We define $\lambda(T_\varepsilon) = \infty$.

3.1.2 TBF: Query. When we query the TBF for item x, it should return Pos whenever $I_x < w$, and Neg otherwise. To query, we examine each timer θ in R_x and compute the age of its timestamp $\lambda(\theta.T)$. The TBF returns Neg if any $\theta \in R_x$ has an expired timestamp, and returns Pos otherwise, yielding false positives but no false negatives.

False Negatives: Since all $\theta \in R_x$ are set to T_+ when x is inserted, we know that $I_x \geq \lambda(\theta.T) - 1$, for all $\theta \in R_x$. Therefore, if for any $\theta \in R_x$, $\theta.T$ has expired, we know that $I_x \geq \lambda(\theta.T) - 1 \geq w$. Since we only return Neg when one of the $\theta.T$ has expired, the TBF has no false negatives.

False Positives: A false positive error occurs when no timestamp in R_x has expired, but $I_x \geq w$. The standard TBF only has false positives if all timers in R_x were touched during the last w insertions, none of which inserted x.

Table 2. TBF/BTBF Notation

bpt	Number of bits per timer
$\theta, \theta.T$	Timer, timestamp stored by timer
T_+	Current timestamp
T_Ω	Maximum timestamp value
T_ε	*Expired* timestamp value
T_x	Oldest timestamp in R_x
$\lambda(T)$	Age of timestamp T
λ_x	Age of oldest timestamp: $\lambda_x = \lambda(T_x)$
$\mathcal{P}$	*Padding* size
B	Insertion block size
b	Insertions since last T_+ increment
C_x	Timers in R_x with oldest timestamp T_x
c_x	$\lvert C_x \rvert$
$F(c_x, r_x, j)$	Prob. specific c_x of r_x timers untouched during j inserts, other $r_x - c_x$ touched
$F(\cdot)$	Short for $F(r_x, c_x, (\lambda_x - 1)B + b)$
$G(r_x, c_x, \lambda_x)$	Prob. c_x timers touched by B inserts, same c_x timers not touched by any of subsequent $(\lambda_x - 1)B + b$ inserts, other $r_x - c_x$ touched
$G(\cdot)$	Short for $G(r_x, c_x, \lambda_x)$

3.1.3 TBF: Marking Expired Timestamps. If any timestamp $\theta.T$ expires, we must mark it expired ($\theta.T \leftarrow T_\varepsilon$) before $T_+ = \theta.T$ again. If we do not, $\lambda(\theta.T)$ will cycle back to 0 and we will not know that $\theta.T$ ought to be expired.[1] If $T_\Omega = w$, there are $w + 1$ values for T_+, so it can only be incremented w times without returning it to its current value. Thus, w is the maximum timestamp age, and timestamps never get a chance to expire. Hence, to correctly support a window of width w, we must have $T_\Omega \geq w + 1$. An example of a TBF with $T_\Omega = w + 1$ is given in Figure 2.

If we have the minimum $T_\Omega = w + 1$, then once any timestamp $\theta.T$ expires, we must set $\theta.T \leftarrow T_\varepsilon$ before the next insertion, which would set $T_+ \leftarrow \theta.T$. Thus, to find all newly expired timestamps, we must check all m timers after every insertion, which is too expensive. The solution in [38] is to increase T_Ω by an amount we call *padding*.

Definition 7. *The* padding $\mathcal{P}$ *is the difference between the chosen and minimum values for* T_Ω.

For a standard TBF, $\mathcal{P} = T_\Omega - w$. If $\mathcal{P} = 1$, $T_\Omega = w + 2$, and we can recognize an expired timestamp up to one insertion after it first expires. Thus, we can split up the search for expired timestamps, such that we need only check half of the timers after each insertion. In general, with padding $\mathcal{P}$ we need only check $m/(\mathcal{P} + 1)$ timers after each insertion. The use of padding is demonstrated in Figure 3.

A good rule of thumb is to set $\mathcal{P} \approx m/k$, so we need only check $O(k)$ timers per insertion. Since we already perform $O(k)$ hashes for each insertion, checking $O(k)$ timers is acceptable. As long as $m \approx w$, as is often the case, this choice of $\mathcal{P}$ increases T_Ω by less than w, so we need at most one extra bit per timer to accommodate the larger T_Ω.

[1]If we assume that timers should never have age 0, we can actually let T_+ cycle back to $\theta.T$, but not beyond, and treat its apparent age 0 as age $T_\Omega + 1$. We can thus reduce the minimum T_Ω value by 1, but we do not do so, since this assumption does not hold for Block Timing Bloom Filters.

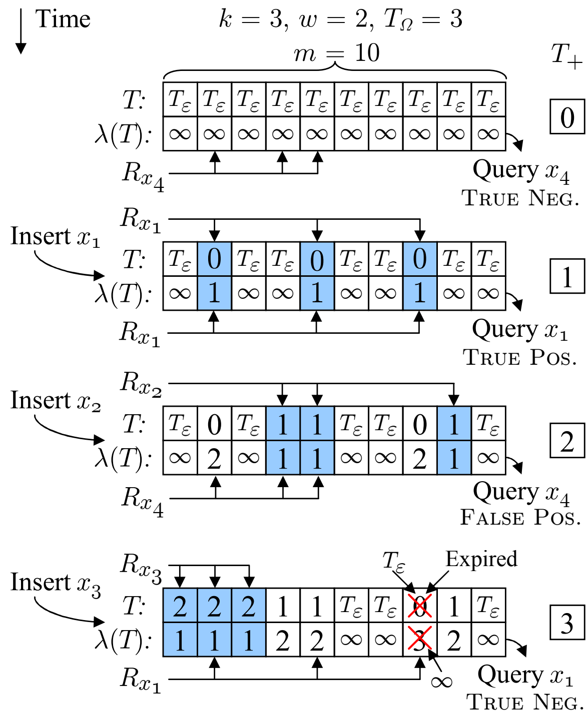


Fig. 2. Timing Bloom Filter inserts and queries. Timestamps touched by each insertion highlighted. Ages are relative to the updated T_+.

3.2 Block Timing Bloom Filters

TBF has the problem that the $T_\Omega + 2$ possible timestamps require it to use $O(\log w)$ bits per timer (*bpt*). For example, the sample TBF in [38] has a window size of $w = 2^{20}$, requiring 21 *bpt*, including 1 bit for padding. It has $m = 15,112,980$ timers, for a total of $21m$ bits, and $21m/w \approx 303$ bits/item in the window of interest, which is excessive. Given 303 bits/item, we could just hash all items in the window into a table using unique hashes. This setup matches TBF performance, and is simpler and more accurate.

We can reduce *bpt* by incrementing T_+ only after every block of $B > 1$ insertions, where B is the *insertion block size*. Using a larger B reduces *bpt*, but uses fewer blocks to cover the window, resulting in a coarser approximation and more false positives (see Section 3.2.2). We call this scheme a *Block Timing Bloom Filter* (BTBF), due to its similarities to the *Block Decaying Bloom Filter* in [33]. The BTBF was alluded to, but not developed, in [38].

3.2.1 BTBF: Insert. Insertions into the BTBF proceed as for the TBF, except that we only increment T_+ once for each block of B insertions. A counter b records the number of insertions since the last time T_+ was incremented. If $B = 1$, as in the standard TBF, then we always have $b = 0$. After each insertion, if $b = B - 1$, we increment T_+ and set $b = 0$. If $b < B - 1$, we increment b and leave T_+ unchanged.

Definition 6 for $\lambda(\theta.T)$ still holds, but our definition of an expired timestamp becomes more general:

DEFINITION 8. *In a BTBF, timestamp $\theta.T$ has* expired *once its age* $\lambda(\theta.T) \geq \left\lceil \frac{w-b}{B} \right\rceil + 1$.

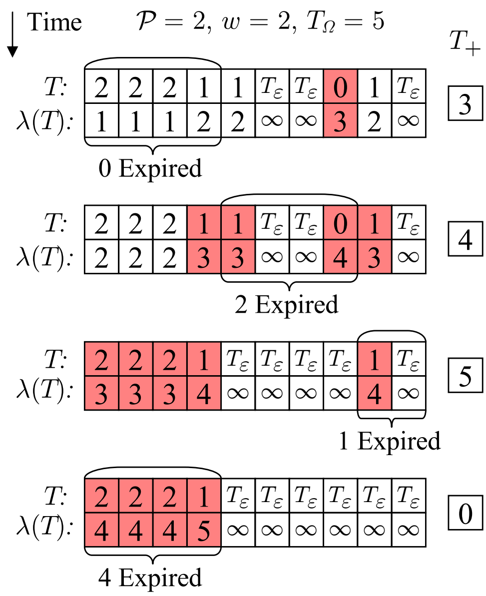


Fig. 3. Padding $\mathcal{P}$ in a Timing Bloom Filter, new insertions omitted for clarity. Newly expired timestamps (highlighted) can remain expired for $\mathcal{P}+1$ insertions before T_+ cycles, so we need only check 4 timers/insertion.

3.2.2 BTBF: Query. Like the TBF, the BTBF returns NEG if and only if some timestamp in R_x has expired. Thus, it has false positives but no false negatives.

False Negatives: If $\lambda(x) = 1$, we know $I_x \geq b$. If $\lambda(x) = 2$, we know that $I_x \geq B + b$. In general, we have

$$I_x \geq \begin{cases} (\lambda(\theta.T) - 1)B + b & \text{if } \lambda(\theta.T) > 0 \\ 0 & \text{if } \lambda(\theta.T) = 0 \end{cases}$$

Therefore, if for any $\theta \in R_x$, $\theta.T$ has expired, we know that

$$\begin{aligned} I_x \geq (\lambda(\theta.T) - 1)B + b &= \left(\left\lceil \frac{w-b}{B} \right\rceil + 1 - 1\right) B + b \\ &\geq \left(\frac{w-b}{B}\right) B + b = w. \end{aligned}$$

That is, if any timestamp in R_x has expired, x is not in the window. NEG is returned only if at least one timestamp in R_x has expired, so the BTBF has no false negatives.

False Positives: In a BTBF, false positives can occur in two ways. As for standard TBFs, they can occur if all timers in R_x are touched by other recent inserts. False positives also occur if x is one of the first items in a block, but only the latter items in the block are in the window. Such false positives are described below and illustrated in Figure 4.

Let $B > 1$, and let x_1 and x_B be the first and last items inserted during a given insertion block. If $I_{x_B} = w - 1$, then $I_{x_1} = w + B - 2$. Since the filter has no false negatives, a query for x_B returns POS. However, since x_1 and x_B are part of the same insertion block, they use the same timestamp and are indistinguishable to the filter, so a query for x_1 must also return POS. Since $I_{x_1} \geq w$, the response is a false positive. At any point, queries for an average of $B/2$ items yield such false positives, so a larger B gives a coarser sliding window approximation with more false positives.

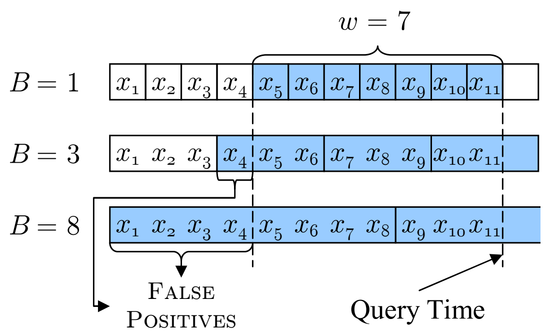


Fig. 4. For the BTBF to treat x_5 as in the window, it must treat all items in x_5's block as in the window. Larger blocks yield more false positives.

3.2.3 BTBF: Marking Expired Timestamps. Marking expired timestamps and the use of padding are the same for the BTBF as for the TBF. However, the minimum T_Ω value is lower for BTBFs, allowing us to reduce *bpt*. To support a window of width w, we now need $T_\Omega \geq \lceil \frac{w}{B} \rceil + 1$. We also now need only check $m/(B(\mathcal{P}+1))$ timers after each insertion, so we can choose $\mathcal{P} \approx m/(kB)$. An example of a BTBF with $\mathcal{P} = 0, B = 3, w = 6$ is given in Figure 5.

4 INFERENTIAL BTBF

We now develop the *inferential* BTBF, which returns the sliding window posterior $\lim_{n\to\infty} P(I_x < w|R_x)$, instead of just a binary Pos or Neg, in response to queries. We derive $\lim_{n\to\infty} P(I_x < w|R_x)$ directly using Equations 9 and 10. For the sake of brevity, we omit the limit notation in the rest of the paper.

Definition 9. *T_x, the oldest timestamp in R_x has age*

$$\lambda_x = \text{Max}\{\lambda(\theta.T) \mid \theta \in R_x\}. \tag{19}$$

If any timestamp in R_x has expired, $\lambda_x = \infty$.

If x had been inserted since the last time $T_+ = T_x$, all timers in R_x would have been set to a more recent timestamp. Thus, if any of the timers in R_x still have the timestamp they were given the last time x was inserted, it is only those timers with timestamp T_x and age λ_x.

Definition 10. *Let C_x be the subset of timers in R_x that have timestamps with age λ_x. That is,*

$$C_x = \{\theta \mid \theta \in R_x \wedge \lambda(\theta.T) = \lambda_x\}. \tag{20}$$

Let $r_x = |R_x|$ and $c_x = |C_x|$. The timers in $R_x \setminus C_x$ must have timestamps set by items other than x, so only the timers in C_x could have been last touched by x, so that only timers in C_x provide worthwhile information about when x was last inserted (I_x). Since all c_x timers in C_x have the same timestamp, with age λ_x, we can accurately compute posteriors given only r_x, c_x, and λ_x. That is, when we refer to $P(R_x|i)$, we are interested in the probability that r_x, c_x, and λ_x have the values we observe, given that $I_x = i$.

The prior $P(i)$ is given by Equation 4. To get posteriors, we must sum over the conditional probability $P(R_x|i)$, which varies depending on the relationship between i and λ_x, so we must handle different ranges of λ_x separately. Figure 6 shows the different expressions for $P(R_x|i)$ derived below for each λ_x case. We need the following function.

Definition 11. *Let $F(r_x, c_x, j)$ be the probability that a specific subset of c_x out of r_x timers* are not *touched during j insertions, and that the remaining $r_x - c_x$ timers* are *touched during the j insertions.*

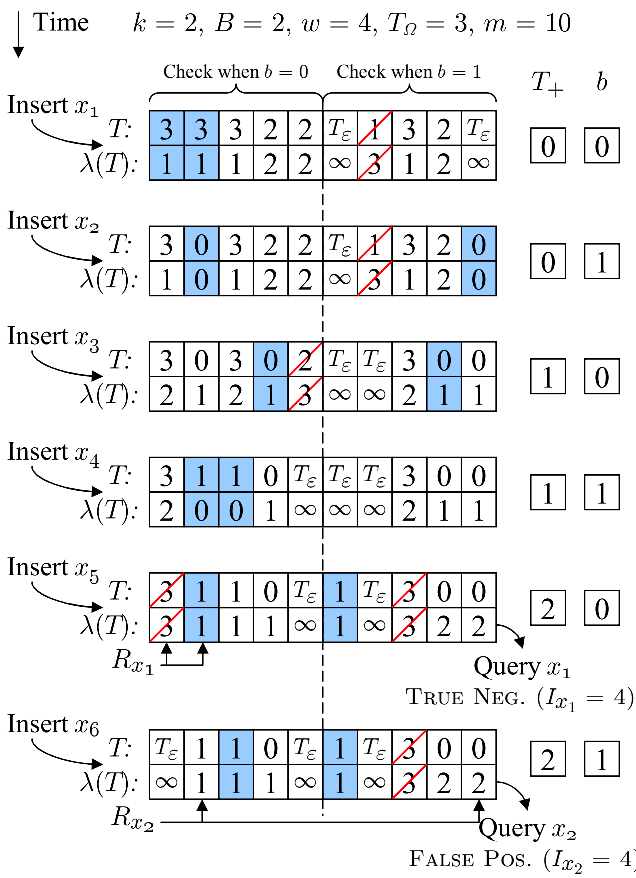


Fig. 5. A BTBF with no padding. Timers touched by the last insertion are highlighted. Timestamp T expires once $\lambda(T) \geq 3$, and is shown with a slash until it is changed to T_ε. Since $B = 2$, we need only check half the timers for expiration after each insertion.

We approximate it as

$$F(r_x, c_x, j) \approx \left(\left(1 - \frac{1}{m}\right)^{kD(j)}\right)^{c_x} \left(1 - \left(1 - \frac{1}{m}\right)^{kD(j)}\right)^{r_x - c_x} \tag{21}$$

The probability that a given timer is not touched during a given insertion is $(1 - 1/m)^k$. If we take $(1 - 1/m)^{kj}$ to be the probability that a timer is not touched during j insertions, we ignore dependencies that arise when the same item is inserted more than once. We account for such dependencies by replacing j with $D(j)$, where $D(j)$ gives the expected number of distinct items among j insertions (see Section 2.5). Raising a probability to an expectation is not entirely valid, but it is an efficient and adequate approximation here, as our approximation error results show (Section 6.1.4).

4.1 Case $\lambda_x > \lceil \frac{w-b}{B} \rceil$

Theorem 2. *If $\lambda_x > \lceil \frac{w-b}{B} \rceil$, then $P(I_x < w | R_x) = 0$.*

Proof: If $\lambda_x > \lceil \frac{w-b}{B} \rceil$, then at least one timestamp in R_x has expired, so we know for certain that $I_x \geq w$, and thus $P(R_x|i) = 0$ for $0 \leq i < w$, giving

$$\begin{aligned} P(I_x < w|R_x) &= \frac{\sum_{i=0}^{w-1} P(i)P(R_x|i)}{\sum_{i=0}^{\infty} P(i)P(R_x|i)} = \frac{\sum_{i=0}^{w-1} P(i) \cdot 0}{\sum_{i=0}^{\infty} P(i)P(R_x|i)} \\ &= 0. \quad \square \end{aligned}$$

$P(I_x < w|R_x) = 0$ when timestamps in R_x have expired because the standard BTBF has no false negatives.

4.2 Case $\lambda_x = 0$

Lemma 4.1. *If $\lambda_x = 0$, then*

$$P(R_x|i) = \begin{cases} 1 & \text{if } i < b \\ F(r_x, 0, b) & \text{if } i \geq b \end{cases} \tag{22}$$

Proof. If $\lambda_x = 0$, then *all* timers in R_x must have timestamp T_+ with age 0, so $c_x = r_x$.

Case $i < b$: If $i < b$, then x would have been inserted since T_+ was last incremented, and all timers in R_x *must* have had their timestamps set to T_+ and could not have been changed since, so $P(R_x|i) = 1$.

Case $i \geq b$: If $i \geq b$, then x would have been most recently inserted before T_+ was last incremented. Thus, for all the timers in R_x to have timestamp T_+, every one of the r_x timers must have been touched through some combination of the last b items inserted, none of which were x. The probability that this event occurs is $F(r_x, 0, b)$. $\square$

Theorem 3. *If $\lambda_x = 0$, then*

$$P(I_x < w|R_x) = 1 - \frac{(1-p_x)^w}{\frac{1-(1-p_x)^b}{F(r_x,0,b)} + (1-p_x)^b}. \tag{23}$$

Proof: Taking $P(R_x|i)$ from Equation 22, we get

$$\begin{aligned} P(I_x < w|R_x) &= 1 - \frac{\sum_{i=w}^{\infty} P(i)P(R_x|i)}{\sum_{i=0}^{\infty} P(i)P(R_x|i)} \\ &= 1 - \frac{F(r_x,0,b) \cdot p_x \sum_{i=w}^{\infty}(1-p_x)^i}{p_x \sum_{i=0}^{b-1}(1-p_x)^i + F(r_x,0,b) \cdot p_x \sum_{i=b}^{\infty}(1-p_x)^i} \\ &= 1 - \frac{F(r_x,0,b)(1-p_x)^w}{(1-(1-p_x)^b) + F(r_x,0,b)(1-p_x)^b} \\ &= 1 - \frac{(1-p_x)^w}{\frac{1-(1-p_x)^b}{F(r_x,0,b)} + (1-p_x)^b}. \quad \square \end{aligned}$$

4.3 Case $0 < \lambda_x \leq \lceil \frac{w-b}{B} \rceil$

Definition 12. *Let $G(r_x, c_x, \lambda_x)$ be the probability that a specific subset of c_x out of r_x timers* are *touched by B inserts, and that the same c_x timers* are not *touched by any of the subsequent $(\lambda_x - 1)B + b$ inserts, while the remaining $r_x - c_x$ timers* are *touched by those subsequent inserts.*

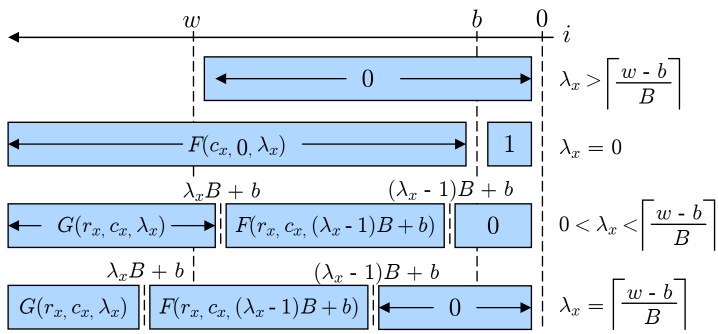


Fig. 6. $P(R_x|i)$ for different values of λ_x and i.

Lemma 4.2. *If* $0 < \lambda_x \leq \lceil \frac{w-b}{B} \rceil$, *then*

$$P(R_x|i) \approx \begin{cases} 0 & \textit{if } i < (\lambda_x - 1)B + b \\ G(r_x, c_x, \lambda_x) & \textit{if } i \geq \lambda_x B + b \\ F(r_x, c_x, (\lambda_x - 1)B + b) & \textit{otherwise} \end{cases} \tag{24}$$

Proof. We know that exactly $(\lambda_x - 1)B + b$ insertions occurred since T_+ changed from timestamp T_x with age λ_x.

Case $i < (\lambda_x - 1)B + b$: In this case, x would have been inserted since T_+ changed from T_x, so all timers in R_x would have been assigned a timestamp more recent than T_x. If so, λ_x would be less than its observed value, which is a contradiction. Thus, $P(R_x|i) = 0$.

Case $i \geq \lambda_x B + b$: In this case, x would have been most recently inserted before $T_+ = T_x$. Thus, the observed c_x, r_x, and λ_x values must have resulted as follows:

(1) All c_x timers in C_x were touched by one of the B insertions during which $T_+ = T_x$.
(2) The same c_x timers *were not* touched during the $(\lambda_x - 1)B + b$ insertions since $T_+ = T_x$, but the remaining $r_x - c_x$ timers *were* touched during those insertions.

The joint probability of these events is exactly $G(r_x, c_x, \lambda_x)$, so we have $P(R_x|i) = G(r_x, c_x, \lambda_x)$.

Case $(\lambda_x - 1)B + b \leq i < \lambda_x B + b$: In this case, x would have been most recently inserted while $T_+ = T_x$, so all timers in R_x must have been set to T_x. Thus, $P(R_x|i)$ is just the probability $F(r_x, c_x, (\lambda_x - 1)B + b)$ that the c_x timers that we observe as still having timestamp T_x *would not* have been overwritten during the last $(\lambda_x - 1)B + b$ insertions, and that the remaining $r_x - c_x$ timers that differ from T_x *would* have been overwritten. □

We obtain $G(r_x, c_x, \lambda_x)$ by finding the probability of each of its constituent events. First, the probability that a particular set of c_x timers were touched by one of B insertions is given by $F(c_x, 0, B)$. Second, the probability that the same c_x timers were not touched by any of $(\lambda_x - 1)B + b$ insertions, while the remaining $r_x - c_x$ timers were, is given by $F(r_x, c_x, (\lambda_x - 1)B + b))$. These two events are largely independent for common BTBF parameters, so we can approximate $G(r_x, c_x, \lambda_x)$ by multiplying their probabilities:

$$G(r_x, c_x, \lambda_x) \approx F(c_x, 0, B) \cdot F(r_x, c_x, (\lambda_x - 1)B + b). \tag{25}$$

Computing $P(I_x < w|R_x)$ is different for $\lambda_x = \lceil \frac{w-b}{B} \rceil$ and $0 < \lambda_x < \lceil \frac{w-b}{B} \rceil$, so we handle each separately. In both cases, $P(R_x|i)$ is defined as in Equation 24. To shrink equations, we substitute $F(\cdot)$ for $F(r_x, c_x, (\lambda_x - 1)B + b)$ and $G(\cdot)$ for $G(r_x, c_x, \lambda_x)$. Since $F(\cdot)$ is a term in our approximation for $G(\cdot)$, $G(\cdot)/F(\cdot)$ simplifies to $F(c_x, 0, B)$.

4.3.1 *Case* $0 < \lambda_x < \lceil \frac{w-b}{B} \rceil$.

THEOREM 4. *If* $0 < \lambda_x < \lceil \frac{w-b}{B} \rceil$, *then*

$$P(I_x < w|R_x) = 1 - \frac{(1-p_x)^{w-(\lambda_x-1)B-b}}{\dfrac{1-(1-p_x)^B}{F(c_x,0,B)} + (1-p_x)^B}. \tag{26}$$

PROOF: If $\lambda_x < \lceil \frac{w-b}{B} \rceil$, then $\lambda_x \le \frac{w-b}{B}$, and $\lambda_x B + b \le w$. Thus, we can construct the posterior sum as follows:

$$\begin{aligned}
P(I_x < w|R_x) &= 1 - \frac{\sum_{i=w}^{\infty} P(i)P(R_x|i)}{\sum_{i=0}^{\infty} P(i)P(R_x|i)} \\
&= 1 - \frac{G(\cdot)\sum_{i=w}^{\infty}(1-p_x)^i}{F(\cdot)\displaystyle\sum_{i=(\lambda_x-1)B+b}^{\lambda_x B+b-1}(1-p_x)^i \;+\; G(\cdot)\sum_{i=\lambda_x B+b}^{\infty}(1-p_x)^i} \\
&= 1 - \frac{(1-p_x)^w}{\dfrac{F(\cdot)}{G(\cdot)}(1-p_x)^{(\lambda_x-1)B+b}(1-(1-p_x)^B) + (1-p_x)^{\lambda_x B+b}} \\
&= 1 - \frac{(1-p_x)^{w-(\lambda_x-1)B-b}}{\dfrac{F(\cdot)}{G(\cdot)}(1-(1-p_x)^B) + (1-p_x)^B} \\
&= 1 - \frac{(1-p_x)^{w-(\lambda_x-1)B-b}}{\dfrac{1-(1-p_x)^B}{F(c_x,0,B)} + (1-p_x)^B}. \quad \square
\end{aligned}$$

4.3.2 *Case* $\lambda_x = \lceil \frac{w-b}{B} \rceil$.

THEOREM 5. *If* $\lambda_x = \lceil \frac{w-b}{B} \rceil$, *then*

$$P(I_x < w|R_x) = \frac{1-(1-p_x)^{w-(\lambda_x-1)B-b}}{1-(1-p_x)^B(1-F(c_x,0,B))}. \tag{27}$$

PROOF: If $\lambda_x = \lceil \frac{w-b}{B} \rceil$, then $(\lambda_x - 1)B + b < w \le \lambda_x B + b$. Thus, we can construct the posterior sum as follows:

$$P(I_x < w|R_x) = \frac{\sum_{i=0}^{w-1} P(i)P(R_x|i)}{\sum_{i=0}^{\infty} P(i)P(R_x|i)}$$

$$= \frac{F(\cdot)\sum_{i=(\lambda_x-1)B+b}^{w-1}(1-p_x)^i}{F(\cdot)\sum_{i=(\lambda_x-1)B+b}^{\lambda_x B+b-1}(1-p_x)^i + G(\cdot)\sum_{i=\lambda_x B+b}^{\infty}(1-p_x)^i}$$

$$= \frac{(1-p_x)^{(\lambda_x-1)B+b}(1-(1-p_x)^{w-(\lambda_x-1)B-b})}{(1-p_x)^{(\lambda_x-1)B+b}(1-(1-p_x)^B) + \frac{G(\cdot)}{F(\cdot)}(1-p_x)^{\lambda_x B+b}}$$

$$= \frac{1-(1-p_x)^{w-(\lambda_x-1)B-b}}{1-(1-p_x)^B(1-F(c_x,0,B))}. \quad \square$$

4.4 Computing Probabilities Efficiently

Consider the inferential BTBF's posteriors given by Equations 23, 26, and 27. Since m and k are fixed, $(1-1/m)^k$ can be precomputed for Equation 21. Equation 23 requires $O(\log_2(w \cdot r_x \cdot D(b))) \leq O(\log_2(w \cdot k \cdot B))$ multiplications to compute exponents. The cost of computing $D(b)$, depends on the distribution (Section 2.5). Equation 23 is the most expensive, but is needed only in the rare case when $\lambda_x = 0$.

Equation 26 needs $O(\log_2 w)$ multiplications, and the cost to compute $F(c_x, 0, B)$. As c_x takes $O(k)$ values and B is fixed, all $O(k)$ values of $F(c_x, 0, B)$ can be pre-computed and cached. These cached values are used for Equation 27, which requires only $O(\log_2 B)$ more multiplications, as $\lambda(x) = \lceil \frac{w-b}{B} \rceil$. Using these techniques, we spent less time computing probabilities than managing the filter itself.

5 GENERALIZED BLOOM FILTER

The Generalized Bloom Filter (GBF) was used in [16] for static set membership queries. As new items are added to the GBF, its memory of older items decays, so it is also well-suited to queries on continuous data streams. Unlike the BTBF, the standard GBF is not built for a particular window width w. Over a stream, however, we can still view GBF as responding to sliding window queries. We use it to show how our analysis can be applied to different filter types.

The GBF consists of $k_0 + k_1$ hash functions and an array of m 1-bit cells. GBF notation is summarized in Table 3.

5.1 Insert

To insert item x into the GBF, we do the following:

(1) Set each cell mapped to by the k_1 hashes to 1
(2) Set each cell mapped to by the k_0 hashes to 0

If a k_0-hash and a k_1-hash collide, the cell is set to 0. Hence, $R_x = R_{x,0} \cup R_{x,1}$, with cells $R_{x,0} = \{h_1(x), \ldots, h_{k_0}(x)\}$ set to 0, and cells $R_{x,1} = \{h_{k_0+1}(x), \ldots, h_{k_0+k_1}(x)\} \setminus R_{x,0}$ set to 1. Future insertions may set cells in $R_{x,0}$ to 1, and cells in $R_{x,1}$ to 0, so the filter loses its memory of x.

5.2 Query

To query a standard GBF for item x, we do the following:

(1) Identify $R_{x,0}$ and $R_{x,1}$
(2) Return Pos if and only if every cell in $R_{x,0}$ is set to 0 and every cell in $R_{x,1}$ is set to 1

Table 3. GBF Notation

k_0, k_1	Num. hashes that set cells to 0, 1 resp.
$R_{x,0}, R_{x,1}$	Cells set by hashes to 0, 1 resp.
$r_{x,0}, r_{x,1}$	$r_{x,0} = \|R_{x,0}\|$, $r_{x,1} = \|R_{x,1}\|$
$C_{x,0}, C_{x,1}$	Cells in $R_{x,0}$ set 0 and in $R_{x,1}$ set 1, resp.
$c_{x,0}, c_{x,1}$	$c_{x,0} = \|C_{x,0}\|$, $c_{x,1} = \|C_{x,1}\|$
q_0, q_1, q_ϵ	Prob. cell set to 0, 1, or not touched, resp.
$f_0(\phi_0, j)$	Prob. cell left 0 after j inserts if initially zero with prob. ϕ_0
$C^s_{d,e}$	Coefficient in efficient form of $P(I_x < w \| R_x)$

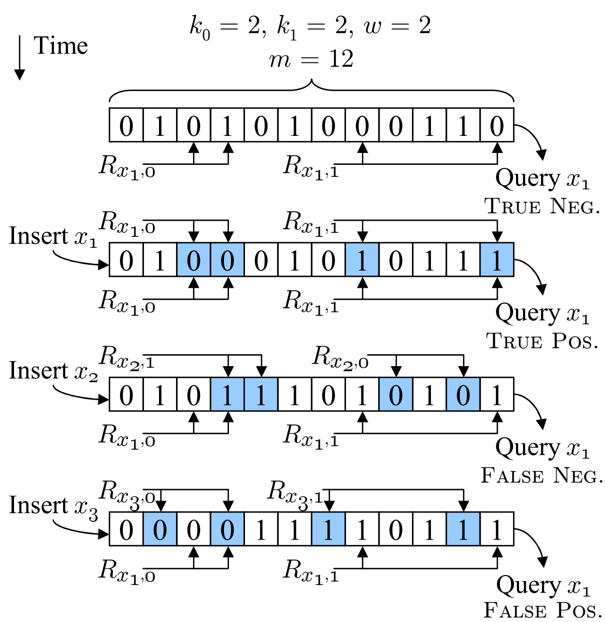


Fig. 7. Operation of a standard GBF. Cells touched by insertions shaded.

False positives and false negatives are both possible. Say we use the GBF for sliding window queries with window width w. A false *negative* occurs when $I_x < w$, but a cell in $R_{x,0}$ is 1, or a cell in $R_{x,1}$ is 0. Such false negatives occur if an item y inserted after x changes one of the cells in R_x, and that cell is not changed back by a subsequent insert.

A false *positive* occurs when $I_x \geq w$, but all cells in $R_{x,0}$ are 0 and all in $R_{x,1}$ are 1. This happens if (1) later inserts happen to set all cells in R_x appropriately, (2) none of the w or more inserts since x change any of the cells in R_x, leaving them unchanged since x's last insertion. Combinations of these cases may also generate false positives. For example, a subsequent insert may change a single cell in R_x, which is later changed back by yet another insert.

Increasing k_0 or k_1 increases false negatives, as cells are changed sooner, but reduces false positives, as more cells must be correctly set in order to respond Pos. Figure 7 demonstrates the standard GBF's operation.

5.3 GBF Analysis

Instead of just a binary Pos or Neg, the *inferential* Generalized Bloom Filter (GBF) returns the sliding window posterior $\lim_{n\to\infty} P(I_x < w|R_x)$ in response to queries. We now derive $P(I_x = j|R_x)$ for the GBF using techniques from Section 2. From this posterior, we derive an expression for the sliding window posterior using Equation 9.

Definition 13. *Let $C_{x,0}$ be the subset of cells in $R_{x,0}$ that are set to* 0*, and $C_{x,1}$ the subset of $R_{x,1}$ set to* 1.

Let $r_{x,0} = |R_{x,0}|$, $r_{x,1} = |R_{x,1}|$, $c_{x,0} = |C_{x,0}|$, $c_{x,1} = |C_{x,1}|$. The values $c_{x,0}$ and $c_{x,1}$ indicate how many of the $r_{x,0}$ and $r_{x,1}$, respectively, are set as they would be if x had just been inserted. Intuitively, the larger $c_{x,0}$ and $c_{x,1}$, the higher the probability that x was recently inserted. Notation for the GBF analysis is summarized in Table 3.

5.3.1 Computing Probabilities. We want $\lim_{n\to\infty} P(I_x < w|R_x)$, given $\lim_{n\to\infty} P(j)$ from Equation 4. Again, we omit the limit notation for brevity. We first find an expression for $P(R_x|j)$, which can be used in Equation 9 or 10 to find $P(I_x < w|R_x)$.

Let q_0 be the probability that at least one of the k_0 hashes sets a given cell to 0.

$$q_0 = 1 - \left(1 - \frac{1}{m}\right)^{k_0} \tag{28}$$

Similarly, let q_1 be the probability that at least one of the k_1 hashes, but none of the k_0 hashes, mapped to a given cell.

$$q_1 = \left(1 - \left(1 - \frac{1}{m}\right)^{k_1}\right)\left(1 - \frac{1}{m}\right)^{k_0} \tag{29}$$

Let q_ϵ be the probability that no hash maps to a given cell.

$$q_\epsilon = 1 - q_0 - q_1 = \left(1 - \frac{1}{m}\right)^{k_0+k_1} \tag{30}$$

Theorem 6. *Let $f_0(\phi_0, i)$ denote the proability that a cell contains a* 0 *after i insertions, given that it was initially* 0 *with probability ϕ_0. $f_0(\phi_0, i)$ is given by:*

$$f_0(\phi_0, i) = \phi_0 q_\epsilon^i + \frac{q_0}{q_0 + q_1}(1 - q_\epsilon^i) \tag{31}$$

Proof: A cell can contain a 0 after i insertions in two cases. First, if it contained a 0 before the insertions and was not touched during any of the i inserts, with probability $\phi_0 q_\epsilon^i$. Second, if the cell was set to 0 by an insert, then not touched by any subsequent inserts, with probability $\sum_{\ell=0}^{i-1} q_0 q_\epsilon^\ell$. Since these events are independent,

$$\begin{aligned} f_0(\phi_0, i) &= \phi_0 q_\epsilon^i + \sum_{\ell=0}^{i-1} q_0 q_\epsilon^\ell = \phi_0 q_\epsilon^i + q_0 \frac{1 - q_\epsilon^i}{1 - q_\epsilon} \\ &= \phi_0 q_\epsilon^i + \frac{q_0}{q_0 + q_1}(1 - q_\epsilon^i) \quad \square \end{aligned}$$

A parallel analysis appears in [16] for false positive and negative rates. We now use it to approximate $P(R_x|i)$. When x was inserted, all $r_{x,0}$ cells in $R_{x,0}$ were set to 0, and all $r_{x,1}$ cells in $R_{x,1}$ set to 1, so we know each cell's *starting* value. $P(R_x|i)$ is the probability that all cells in $R_{x,0}$ and $R_{x,1}$ are set as we observe them, given that i items have been inserted since x was last inserted. Our observations imply that the following events occurred over the i insertions:

Events	Prob. Each
$c_{x,0}$ cells that started 0 remained 0	$f_0(1,i)$
$r_{x,0} - c_{x,0}$ cells that started as 0 changed to 1	$(1 - f_0(1,i))$
$c_{x,1}$ cells that started as 1 remained 1	$(1 - f_0(0,i))$
$r_{x,1} - c_{x,1}$ cells that started as 1 changed to 0	$f_0(0,i)$

We can approximate $P(R_x|i)$ by treating these events as independent and taking the product of their probabilities:

$$\begin{aligned} P(R_x|i) \approx\ & f_0(1,i)^{c_{x,0}} \times (1 - f_0(1,i))^{r_{x,0}-c_{x,0}} \times \\ & (1 - f_0(0,i))^{c_{x,1}} \times f_0(0,i)^{r_{x,1}-c_{x,1}} \end{aligned} \tag{32}$$

We can use $P(R_x|i)$ from Equation 32 and $P(i)$ from Equation 4 in Equation 9 to get $P(I_x < w|R_x)$. The resulting infinite sum having no closed form, we use approximations.

5.3.2 Computing Probabilities Efficiently. To utilize space efficiently, the numbers of 0-cells and 1-cells must be balanced, so $k_0 = k_1 = k$. Now,

$$\begin{aligned} \frac{q_0}{q_0 + q_1} &= \frac{1}{1 + \frac{q_1}{q_0}} \approx \frac{1}{1 + \frac{\left(1-\left(1-\frac{1}{m}\right)^k\right)\left(1-\frac{1}{m}\right)^k}{\left(1-\left(1-\frac{1}{m}\right)^k\right)}} \\ &= \frac{1}{1 + \left(1 - \frac{1}{m}\right)^k} \end{aligned}$$

When m is large, as is common, we get $\left(1 - \frac{1}{m}\right)^k \approx 1$, and $\frac{q_0}{q_0+q_1} \approx \frac{1}{2}$. Thus, Equation 31 reduces to:

$$f_0(\phi_0, i) \approx \phi_0 q_\epsilon^i + \frac{1}{2}(1 - q_\epsilon^i) = q_\epsilon^i \left(\phi_0 - \frac{1}{2}\right) + \frac{1}{2}$$

Thus, Equation 32 reduces as follows:

$$\begin{aligned} P(R_x|i) &\approx f_0(1,i)^{c_{x,0}} \times (1 - f_0(1,i))^{r_{x,0}-c_{x,0}} \times \\ &\quad (1 - f_0(0,i))^{c_{x,1}} \times f_0(0,i)^{r_{x,1}-c_{x,1}} \\ &= \left(\frac{1 + q_\epsilon^i}{2}\right)^{c_{x,0}} \times \left(\frac{1 - q_\epsilon^i}{2}\right)^{r_{x,0}-c_{x,0}} \times \\ &\quad \left(\frac{1 + q_\epsilon^i}{2}\right)^{c_{x,1}} \times \left(\frac{1 - q_\epsilon^i}{2}\right)^{r_{x,1}-c_{x,1}} \\ &= \left(\frac{1 - q_\epsilon^i}{2}\right)^{r_{x,0}+r_{x,1}} \times \left(\frac{1 + q_\epsilon^i}{1 - q_\epsilon^i}\right)^{c_{x,0}+c_{x,1}} \end{aligned}$$

Now we apply Equation 9 to get:

$P(I_x < w|R_x) = \frac{\sum_{i=0}^{w-1} P(i)P(R_x|i)}{\sum_{i=0}^{\infty} P(i)P(R_x|i)}$

$$= \frac{\sum_{i=0}^{w-1} p_x(1-p_x)^i \left(\frac{1-q_\epsilon^i}{2}\right)^{r_{x,0}+r_{x,1}} \times \left(\frac{1+q_\epsilon^i}{1-q_\epsilon^i}\right)^{c_{x,0}+c_{x,1}}}{\sum_{i=0}^{\infty} p_x(1-p_x)^i \left(\frac{1-q_\epsilon^i}{2}\right)^{r_{x,0}+r_{x,1}} \times \left(\frac{1+q_\epsilon^i}{1-q_\epsilon^i}\right)^{c_{x,0}+c_{x,1}}}$$

$$= \frac{\sum_{i=0}^{w-1}(1-p_x)^i \left(1-q_\epsilon^i\right)^{r_{x,0}+r_{x,1}} \times \left(\frac{1+q_\epsilon^i}{1-q_\epsilon^i}\right)^{c_{x,0}+c_{x,1}}}{\sum_{i=0}^{\infty}(1-p_x)^i \left(1-q_\epsilon^i\right)^{r_{x,0}+r_{x,1}} \times \left(\frac{1+q_\epsilon^i}{1-q_\epsilon^i}\right)^{c_{x,0}+c_{x,1}}} \tag{33}$$

We turn to Equation 33. Let us substitute $d = c_{x,0} + c_{x,1}$ and $e = r_{x,0} + r_{x,1} - d$. In general, we want the sum:

$$\Psi_\eta = \sum_{i=0}^{\eta-1}(1-p_x)^i(1+q_\epsilon^i)^d(1-q_\epsilon^i)^e$$

Using the binomial theorem, we can re-write:

$$(1+q_\epsilon^i)^d(1-q_\epsilon^i)^e = \sum_{s=0}^{d+e} C_{d,e}^s (q_\epsilon^i)^s$$

where $C_{d,e}^s$ is the coefficient of $(q_\epsilon^i)^s$ from the convolution

$$C_{d,e}^s = \sum_{\ell=0}^{s}\binom{d}{\ell}(-1)^{s-\ell}\binom{e}{s-\ell} = \sum_{\ell=0}^{s}\binom{d}{s-\ell}(-1)^{\ell}\binom{e}{\ell}$$

where $\binom{a}{b} = 0$ if $b > a$. Now we can re-write Ψ_η as:

$$\begin{aligned}
\Psi_\eta &= \sum_{i=0}^{\eta-1}(1-p_x)^i\left(\sum_{s=0}^{d+e} C_{d,e}^s (q_\epsilon^i)^s\right) \\
&= \sum_{s=0}^{d+e} C_{d,e}^s \sum_{i=0}^{\eta-1}((1-p_x)q_\epsilon^s)^i \\
&= \sum_{s=0}^{d+e} C_{d,e}^s \frac{1-((1-p_x)q_\epsilon^s)^\eta}{1-(1-p_x)q_\epsilon^s}
\end{aligned}$$

Applying to Equation 33 we get:

$$\begin{aligned}
P(I_x < w|R_x) &= \frac{\Psi_w}{\lim_{\eta\to\infty}\Psi_\eta} \\
&= \frac{\sum_{s=0}^{d+e} C_{d,e}^s \frac{1-((1-p_x)q_\epsilon^s)^w}{1-(1-p_x)q_\epsilon^s}}{\sum_{s=0}^{d+e} C_{d,e}^s \frac{1}{1-(1-p_x)q_\epsilon^s}}
\end{aligned} \tag{34}$$

This is easy to evaluate. First, $d + e = r_{x,0} + r_{x,1} \le 2k$, so we need to sum at most $2k$ terms, with k being usually small. We also have $d, e, s \le 2k$, so there are at most $8k^3$ distinct coefficients $C_{d,e}^s$, which we can easily pre-compute. It is also unlikely that $r_{x,0} + r_{x,1}$ is much less than $2k$, so only $O(k)$ combinations of d, e occur in practice, reducing the number of precomputed coefficients to $O(k^2)$. In our experiments we precompute at most $16k^2$ coefficients, which is manageable.

Equation 34 is accurate only for relatively small values of k. As k grows, the coefficients $C_{d,e}^s$ and the corresponding terms can have large absolute values, though their sum must lie between 0 and 1. If precision is limited and k is large, roundoff error can occur. Using double-precision floating point numbers, such roundoff error leads to posterior probabilities outside the $[0, 1]$ range for approximately $k \geq 30$.

An advantage of the inferential GBF over the inferential BTBF is that we can vary w at any time, whereas w for the BTBF must be fixed up-front. We could improve the accuracy of the inferential GBF by replacing i in Equation 31 with $D(i)$ (Equation 15), as we did for the inferential BTBF. However, doing so would prevent us from constructing the efficiently computable expression in Equation 34.

6 EXPERIMENTS

6.1 Experimental Setup

We examined four approaches for sliding window queries: the standard GBF and BTBF, which return Pos or Neg, and the inferential GBF and BTBF, which return the sliding window posterior $P(I_x < w|R_x)$. We now test if using posteriors reduces overall penalties, when penalties for false positives and negatives vary across queries. Our experiments use a real-world data stream and two synthetic data streams.

6.1.1 Queries and Error Penalties. We use the same data stream for queries and inserts. As each new item x arrives, we always query for x and then insert x. This model might be used for an expensive multi-level LRU cache, where we only want to do an expensive check of a large cache level if we are likely to find the item. This model also resembles duplicate detection as used for mitigating click fraud [15, 38], although duplicates would not be inserted in that case.

Let $\$_{FP}$ and $\$_{FN}$ be the penalties incurred if the filter makes false positive/negative errors, respectively. We choose $\$_{FP}$ and $\$_{FN}$ independently and uniformly at random from the range $[1.0, 10.0)$ for each query. The inferential BTBF uses the minimum expected penalty strategy described in Section 2.6 for deciding whether to return Pos or Neg.

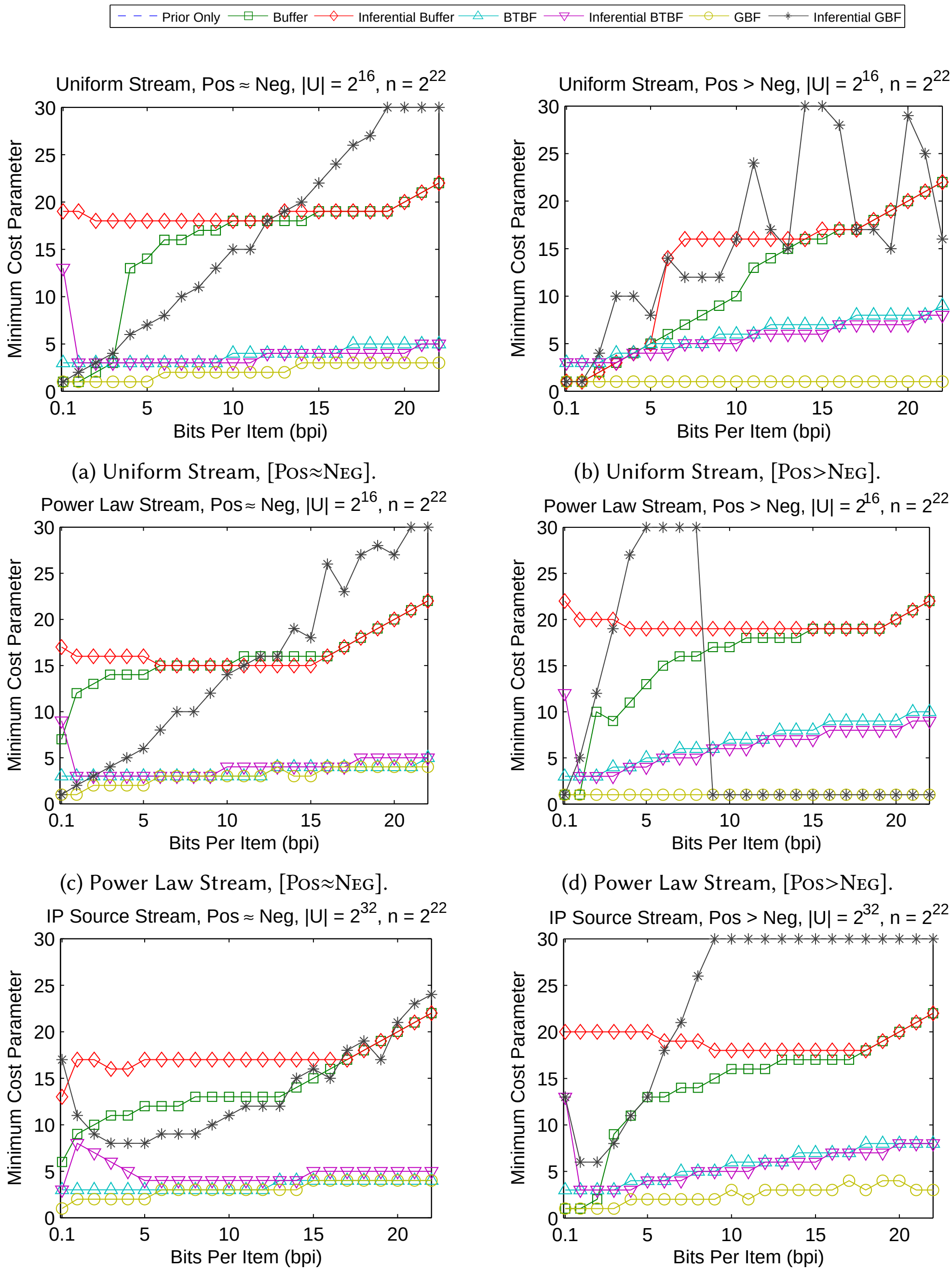
6.1.2 Parameter Selection. Poor choices for filter parameters lead to more errors. However, there is no consensus on how to choose parameters for the BTBF, though [16] and [33] provide limited guidance. For the GBF, we fix $k_0 = k_1 = k$. For the BTBF, we fix $bpt = k$ as in [33]. If $k < 3$, we set the minimum $bpt = 3$ needed by the BTBF. Given k, we choose the smallest $\mathcal{P}$ that allows us to check at most k timers per insertion.

Our focus is not on predicting optimal parameters, so we tried all k for $1 \leq k \leq 30$ for each trial, chosing k to minimize total penalty (see Figure 8). Thus, penalties measured for each filter are independent of the parameter selection mechanism. This way, we are able to ensure that a sub-optimal parameter selection strategy does not adversely impact any filter's reported performance.

In the GBF, bits set by one of the k_0 and k_1 hashes are set to 0, so the GBF would benefit from separately optimizing k_0 and k_1, allowing for a k_0 slightly less than k_1 [16]. For simplicity, we do not optimize them separately here. The standard GBF is prone to false negatives if k is large, so its optimal k are small. However, optimal k are larger for the inferential GBF, which uses the added information available with large k.

6.1.3 Measuring Performance for Each Filter. Each filter is given bpi bits per item in the window, so the total space is $w \cdot bpi$ bits. Each experimental *trial* measures the total penalty incurred by a given filter for a specific data stream, choice of w, and choice of bpi. Each trial over a given stream uses the same sequence of $n = 2^{22}$ item inserts/queries, and the same sequence of penalties $\$_{FP}$ and $\$_{FN}$, ensuring comparable results. Before each trial, all cells in the BTBF are set to T_ε. We then insert 2^{20} items *without* issuing queries, initializing the filters with "past" items from the data stream.

An *experiment* is a group of trials with the same stream and w, and measures penalties incurred by the standard and inferential BTBF for a range of bpi values. For each stream, we used two



(a) Uniform Stream, [Pos≈Neg].

(b) Uniform Stream, [Pos>Neg].

(c) Power Law Stream, [Pos≈Neg].

(d) Power Law Stream, [Pos>Neg].

(e) IP Source Stream, [Pos≈Neg].

(f) IP Source Stream, [Pos>Neg].

Fig. 8. Minimum cost parameters (k for BTBF/GBF, *bph* for Simple Buffer) in each experiment suite.

experimental conditions. Condition [Pos≈Neg] uses a small enough w to make the numbers of

Table 4. Data parameters and characteristics for each experiment/condition. Pos/Neg gives the ratio of queries for items with $I_x < w$ to those with $I_x \geq w$.

Stream	$\|U\|$	Condition	w	Pos/Neg
Uniform	2^{16}	[Pos≈Neg]	2^{16}	1.718
Uniform	2^{16}	[Pos>Neg]	2^{18}	53.547
Power Law	2^{16}	[Pos≈Neg]	2^{11}	1.182
Power Law	2^{16}	[Pos>Neg]	2^{18}	13.451
IP Source	2^{32}	[Pos≈Neg]	2^{8}	1.171
IP Source	2^{32}	[Pos>Neg]	2^{15}	8.893

queries requiring Pos and Neg responses roughly the same. Condition [Pos>Neg] uses a larger w, so Pos queries outnumber Neg ones.

Each experiment is shown as a single curve on a graph. Some graphs show the total penalty over the trial, while others show the *penalty ratio*, which is the ratio of the total penalty of the inferential filter over that of the corresponding standard one. A penalty ratio under 100% indicates that the inferential filter outperformed the standard one.

The choice of w and the Pos/Neg ratios for each experiment are given in Table 4. For simplicity, we chose w to be a power of 2, but our implementation supports arbitrary integral w values.

6.1.4 Errors from Approximations. We made several approximations while deriving the posterior $P(I_x < w|R_x)$, so we evaluate its accuracy for each dataset. In each experiment, we group queries into 20 bins based on the posterior value P returned. The first bin contains queries with $0 \leq P < 0.05$, the second with $0.05 \leq P < 0.1$, on up to the last with $0.95 \leq P \leq 1$. Let η_ℓ be the number of queries in bucket ℓ, and let M_ℓ be the midpoint of bucket ℓ. We let f_ℓ be the fraction of queries in bucket ℓ for which $I_x < w$. Without approximations, we should have $f_\ell \approx M_\ell$ for all ℓ.

Posterior Error, defined as the average absolute difference between a query posterior and its bucket midpoint, equals

$$\frac{1}{n}\sum_{\ell=1}^{20} |f_\ell - M_\ell| \cdot \eta_\ell. \tag{35}$$

Some Posterior Error is unavoidable due to the coarseness of our grouping. Thus, we expect a baseline error of less than half the bucket width (0.025). We graph Posterior Errors for each experiment below.

6.1.5 Implementation. We implemented the filters in Java, running each trial as a single thread on a 2.4GHz processor. The average time to query and insert an item fell between 0.5 and 1.5 microseconds for the standard and inferential BTBF. Table 5 shows the average time to do an insert/query pair for each decision technique, with a fixed $k = 4$, $bpi = 15$, $w = 2^{16}$ for all trials. Times were measured for the Uniform data stream (see Section 6.3) and averaged over all n items; times for other streams and window sizes were comparable.

Increasing bpi primarily increases the number of cells, which should have negligible effect on runtime as long as the filter still fits in RAM. Time varies roughly linearly with k; the GBF evaluates $2k$ hashes for each insert/query, so its times are double those of the BTBF. Hashing is done separately for query and insert. Times can be reduced using optimized or hardware implementations.

The inferential BTBF caches $O(k)$ static floating-point values to speed up computation in common cases (Section 4.4), and the GBF caches $O(k^2)$ static floating-point values (Section 5.3.2). Cache

Table 5. Average time per insert/query on the Uniform [Pos≈Neg] data stream with $bpi = 10$ and $k = 4$.

Technique	Time per Query (μs)
Prior Only	0.33
BTBF (Standard)	0.80
BTBF (Inferential)	1.13
GBF (Standard)	1.64
GBF (Inferential)	1.85

sizes are substantial only for a GBF with small w, small bpi, and large k, so we do not count caches as part of the space consumed by our filters.

6.2 Simple Buffer

One might ask under what circumstances space devoted to filters could be better spent on a list of bph-bit hashes of the last $\delta \leq w$ items inserted. The Simple Buffer is a contrived decision technique that performs queries via a linear scan over such a list. Practical techniques based on such a list may index it using a Counting Bloom Filter [13, 35, 36] or a hash table. If the GBF or BTBF incurs lower costs than (outperforms) the Simple Buffer, then it also outperforms these other techniques, which store the list *and* the index.

Storing all items in the window ($\delta = w$, $bph = bpi$) is impractical when bpi is small, as all 2^{bpi} hash values will be in the buffer with high probability. Thus, for each trial, we tried all bph for $1 \leq bph \leq 30$, and chose the value of bph that minimized total cost, with $\delta = \lfloor w \cdot bpi/bph \rfloor$ (see Figure 8). Once bpi is large enough to uniquely represent each item in U or in the window ($bpi \approx \log_2 Min(w, |U|)$), the Simple Buffer achieves near-perfect accuracy.

We can analyze the Simple Buffer using the framework described in Section 2. Let $R_x = 1$ if the bph-bit hash of x is in the buffer, and $R_x = 0$ otherwise. The posterior is given by:

$$P(I_x < w | R_x) = \begin{cases} 1 - (1 - p_x)^{w-\delta} & R_x = 0 \\ 1 - \dfrac{(1-p_x)^w}{\frac{1-(1-p_x)^\delta}{1-\left(1-\left(\frac{1}{2}\right)^{bph}\right)^{D(\delta)}} + (1-p_x)^\delta} & R_x = 1 \end{cases} \tag{36}$$

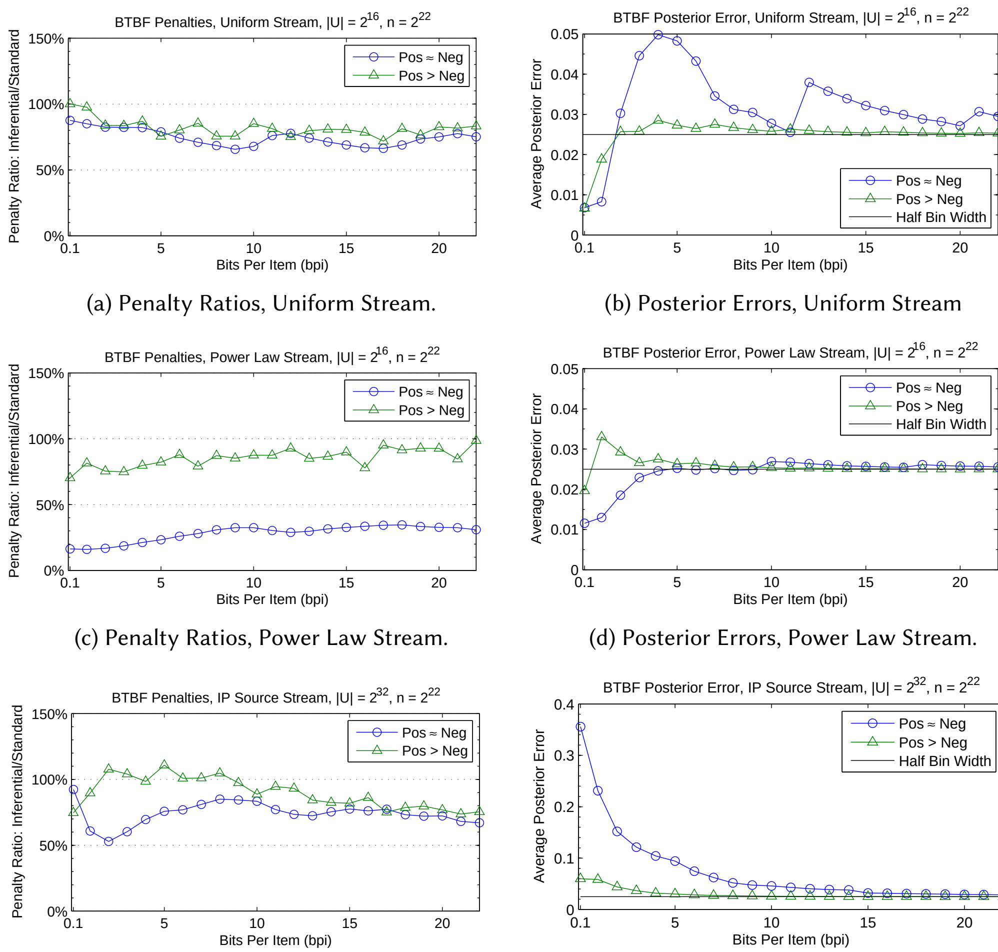
If a technique outperforms the *inferential* Simple Buffer, then it also outperforms *inferential* list-based techniques.

6.3 Uniform Data Stream

The *Uniform* data stream samples uniformly with replacement, from a set U of 2^{16} integers. $p_x = 1/|U|$ forall $x \in U$. $D(j)$ is given by Equation 16. Figure 9a shows Penalty ratios, and Figure 9b shows Posterior Errors. At very low bpi all the filters hold little information, so posteriors depend primarily on the prior $P(I_x = i)$. Since the prior is known exactly, the posterior here is quite accurate.

Penalties for the inferential BTBF are about 80% of those for standard BTBF. For large bpi, so much state information is available that most posteriors are close to 0 or 1. They differ from their corresponding bin centers by half the bin width, hence the convergence to 0.025 for the BTBF. Our approximations produce noteworthy error in the BTBF only for moderate bpi. The Posterior Errors for such bpi remain under 0.05, indicating largely accurate posterior expressions.

The inferential GBF consistently outperforms the standard GBF (see Figure 10a), but both perform poorly overall. For the GBF under [Pos≈Neg], posterior error increases steadily with bpi (see Figure 10b).



(a) Penalty Ratios, Uniform Stream.

(b) Posterior Errors, Uniform Stream

(c) Penalty Ratios, Power Law Stream.

(d) Posterior Errors, Power Law Stream.

(e) Penalty Ratios, IP Source Stream.

(f) Posterior Errors for IP Source stream.

Fig. 9. BTBF performance for various stream types. Lower Y values are better.

Figures 11a and 11b show that in both suites, the inferential BTBF outperforms all other techniques until $bpi \approx 20$, where the Simple Buffer can afford nearly-unique representations of all items in the window. One exception occurs in the [Pos≈Neg] suite at $bpi = 1$, where the inferential GBF beats the BTBF by a small margin. For the [Pos≈Neg] suite, k reaches our maximum value 30 at $bpi = 19$ (see Figure 8), indicating that larger k, though inefficient, may yield better results for higher bpi.

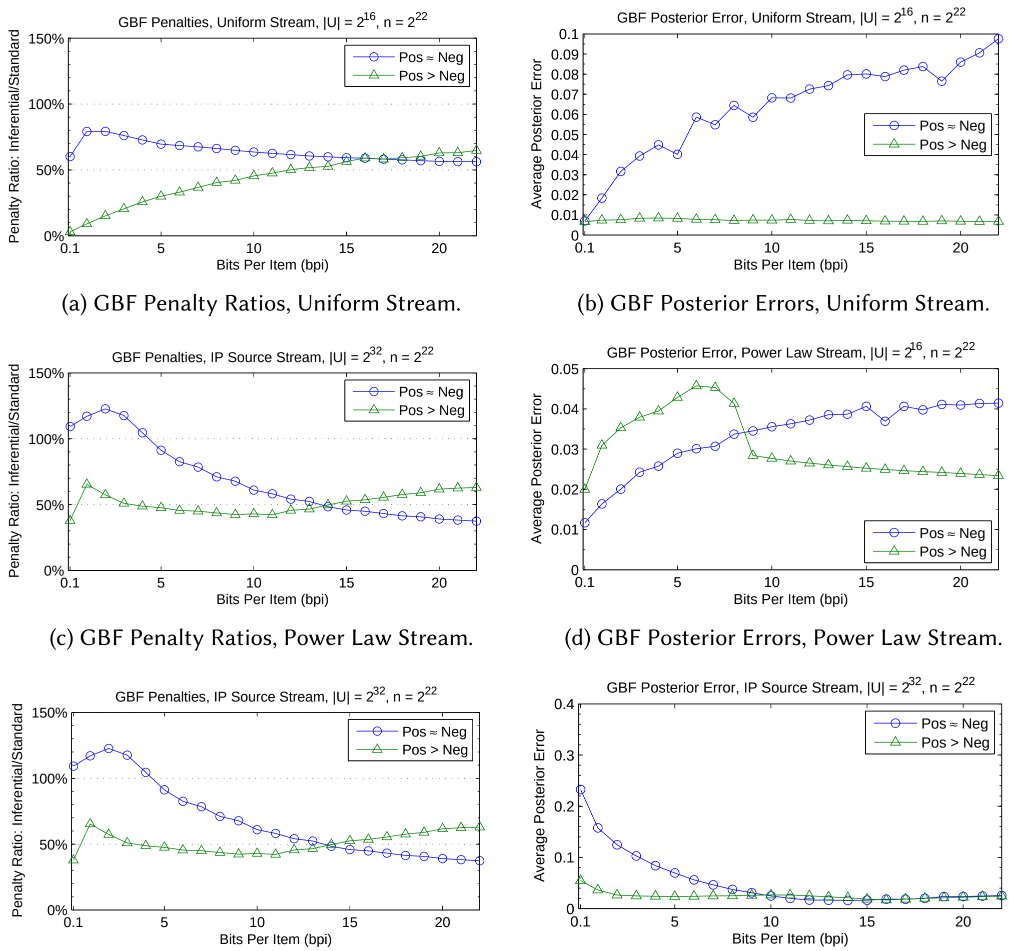
6.4 Streams with Skewed Distributions

For skewed stream item distributions, computing accurate posteriors requires the following:

Assumption 1. *p_x is easy to compute for each x.*

Assumption 2. *p_x is time-invariant for each x.*

If Assumption 1 is violated, priors, and thus posteriors, cannot be computed efficiently. If Assumption 2 is violated, the time-invariant priors yield inaccurate posteriors, increasing penalties.



(a) GBF Penalty Ratios, Uniform Stream.

(b) GBF Posterior Errors, Uniform Stream.

(c) GBF Penalty Ratios, Power Law Stream.

(d) GBF Posterior Errors, Power Law Stream.

(e) GBF Penalty Ratios, IP Source Stream.

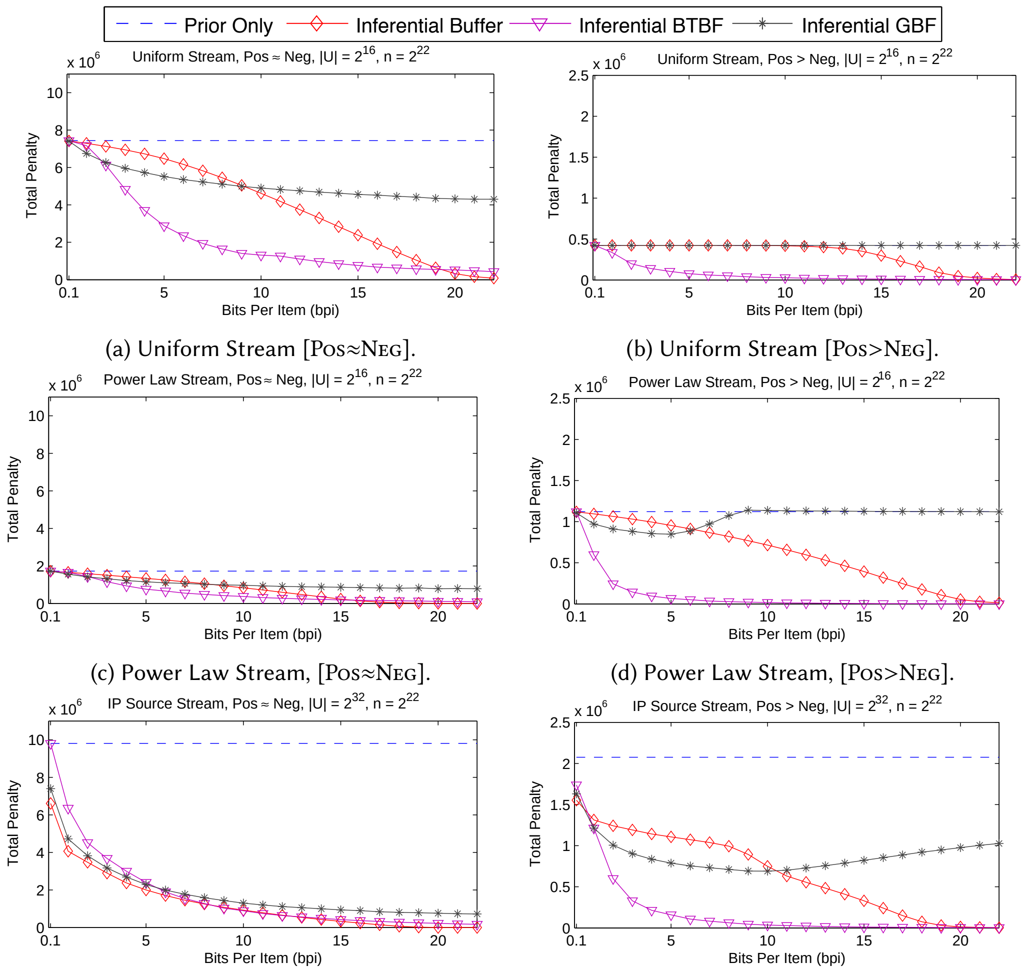
(f) GBF Posterior Errors for IP Source stream.

Fig. 10. GBF performance for various stream types. Lower Y values are better.

We must distinguish between x's *value* and its *rank*. More frequent items (larger p_x) have lower rank. Often, item *ranks* follow a predictable distribution, such as a power law in the case of Zipf-distributed data, while *values* do not. Thus, this property only helps compute p_x if we can infer x's rank from its value, which is often not the case.

6.4.1 Power Law Stream. Our Power Law stream samples items from set $U = \{x \in \mathbb{Z} \mid 1 \leq x \leq 2^{16}\}$ according to the discrete power law distribution $p_x = 1/(x \cdot H_{|U|})$, where $H_{|U|} = \sum_{i=1}^{|U|} 1/i$. Computing p_x is easy since $H_{|U|}$ is fixed, and p_x is time-invariant. Equation 17 gives $D(j)$.

BTBF penalty ratios are shown in Figure 9c, and Posterior Errors in Figure 9d. The standard BTBF has no false negatives, so it performs well under condition [Pos>Neg]. Thus, penalty reductions are more pronounced under [Pos≈Neg]. Posterior Errors stay under 0.035 for all *bpi*, so our posterior expressions are again largely accurate. In this case, as for Uniform streams, Posterior Errors are low



(a) Uniform Stream [Pos≈Neg].

(b) Uniform Stream [Pos>Neg].

(c) Power Law Stream, [Pos≈Neg].

(d) Power Law Stream, [Pos>Neg].

(e) IP Source Stream, [Pos≈Neg].

(f) IP Source Stream, [Pos>Neg].

Fig. 11. Total Penalty values for different input stream types and parameters. Lower Y values are better.

for very low *bpi*, where the posterior depends largely on the precisely-known priors, and converges to 0.025 when *bpi* is large.

GBF penalty ratios are shown in Figure 10c, and Posterior Errors in Figure 10d. Under [Pos>Neg], the inferential GBF penalty ratio *increases* for $bpi > 5$, since k reaches our maximum of 30 by the time $bpi = 5$ (see Figure 8). After this point, the most effective choice of $k \leq 30$ for the inferential GBF is $k = 1$, which incurs lower posterior error, but slightly increases the penalty ratio.

Figures 11c and 11d show that the inferential BTBF has lowest penalty until $bpi \approx 16$ for [Pos≈Neg], and $bpi \approx 22$ for [Pos>Neg]. The inferential BTBF's advantage is greater for [Pos≈Neg] since the standard BTBF has no false negatives, and benefits when most queries return Pos.

6.4.2 Source IP Data Stream. The Source IP data stream [4] draws anonymized source addresses from IPv4 packet headers ($|U| = 2^{32}$). Address distribution is complex, so p_x is hard to model analytically (see Assumption 1). We handled this problem by pre-processing the stream items x to be queried, computing p_x based on the observed frequency of address x, and saving (x, p_x) pairs for the queried x.

We sample $D(j)$ for $j \in \{1, 10, 10^2, 10^3, 10^4, 10^5, 2^{18}\}$ over the stream itself. We inserted 2^{18} items during each of 8 sampling trials. We then averaged $D(j)$ values over all 8 trials, and interpolated between averages using Equation 18.

BTBF penalty ratios are shown in Figure 9e, and Posterior Errors in Figure 9f. GBF penalty ratios are in Figure 10e, and Posterior Errors in Figure 10f. The stream is bursty, so p_x is not strictly time-invariant, violating Assumption 2. Thus, priors are not accurate, leading to higher Posterior Error for low *bpi*, where the posterior relies heavily on the prior.

These high errors, combined with the zero false negative rate of the standard BTBF, cause the inferential BTBF to incur *higher* penalties for some low *bpi* under [Pos>Neg]. However, the inferential BTBF still generally reduces penalties for most trials.

Figures 11e and 11f show total penalties for the IP Source stream. Every queried item is re-inserted. For [Pos≈Neg], where data is bursty and w is small, any other prior insertions of x are likely to have been recent. The BTBF spends comparable space on each item in the window, but the GBF and Simple Buffer devote more space to more recent items, so they initially outperform the BTBF. The GBF thus has the potential to outperform the BTBF in such scenarios, where more recent items are more important to remember.

Under [Pos>Neg], w is larger, so an item is more likely to have multiple bursts throughout the window, which the BTBF handles well. The inferential GBF costs increase near $bpi = 9$, where k reaches 30 (Figure 8).

7 RELATED WORK

An extensive survey of various Bloom Filter variants appears in [20], the variants being compared with respect to their performance and generality. Applications of Bloom Filters are discussed in [3].

Filters including the Standard Bloom Filter [2], the Generalized Bloom Filter [16], and others [17, 19] use single-bit cells. Other filters use multiple bits in each cell to represent counters, as in the Counting Bloom Filter (CBF) [11], timers, as in the TBF [38], or other values [5, 8, 30].

Simple filters [2, 9, 31] only allow items to be inserted, and generally represent static sets. Deletable filters [19, 29, 30] allow items to be deleted as well as inserted, and represent dynamic sets. Decaying filters represent a dynamic set of recently inserted items. As new items are inserted, decaying filters lose their memory of older items.

Deletable filters such as the CBF can function as decaying filters by storing a queue of recent items [35, 36]. When a new item arrives, an old item is removed from the queue and deleted from the filter. Storing the queue requires many bits per item, so such techniques are only practical when a great deal of space is available to the filter.

Common decaying filters use multi-bit counters and insert an item by setting all its touched cells to some maximum value such as a window width. Cells are regularly decremented, with minimum value 0. When the filter is queried, the item is deemed to be in the window if all touched cells have values greater than 0. In [8], cells to decrement are chosen randomly after each insertion, while in [15, 33, 39], all non-zero counters are decremented after each block of inserts. The TBF [38] implicitly decrements cells by assigning each cell a timestamp, and periodically incrementing a *current timestamp*. Posterior expressions for such decaying filters are similar to those of the BTBF.

The work in [14] addresses the false positive problem by applying techniques from Combinatorial Group Testing to create an auxiliary data structure called the EGH filter, and gurantees the absence of false positives as long as the number of inserted items is below a threshold d. Bloom Filters have been used for comparing sets, in applications such as distributed joins [18, 22, 26, 27] and cache management [11]. The work in [21] designs a new data structure called the Invertible Counting Bloom Filter that permits comparison operations, such as set differencing, on multisets.

Current designs ignore much of the information latent in filters. Authors of [15] note that in decaying filters, the *number of timers* with minimum value touched by x affects the posterior probability that x is in the window, but do not derive that probability. Authors of [31] use *specific counter values* in a CBF to derive the posterior probability that x is in a static set. They show that the posterior depends on the product of the counters touched by x, and use it to improve accuracy. Authors of [5] use knowledge of data stream item frequencies to improve accuracy. They build a hierarchy of decaying filters and assign items to filters based on frequency, using more information to store more frequent items.

We introduced our framework for constructing inferential time-decaying filters and an analysis of the Timing Bloom Filter in [7]. The present work significantly extends and generalizes this prior work in numerous ways. First, it discusses the Generalized Bloom Filter (GBF) and presents an Inferential version of the GBF, with detailed analysis and proofs that analyze its performance and inferential properties. Next, it presents a baseline comparison standard for Bloom filters in the form of an idealized Simple Inferential Buffer approach that uses equivalent total storage, to contrast with the Bloom filter approach of dedicating a fixed number of bits to a small hash for each item in the filter. This comparison is important, because when any Bloom filter variant presented outperforms this approach, it is guaranteed to outperform inferential versions of all such list-based techniques. The current work also presents a direct comparison (total penalty vs. bits per item) between the Inferential BTBF, the Inferential GBF, the Inferential Buffer, and a scheme relying only on prior probabilities (0 bits stored per item) as a lower bound on quality. The experiments and implementation presented are greatly expanded, including tables with time per query for each type of filter in microseconds. Finally, this paper presents significantly expanded details on optimal parameter selection (Figure 8).

8 CONCLUSION

We have shown how to turn standard time-decaying filters into inferential filters, using prior probabilities and previously unused information in the filter. We showed how inferential filters can support new types of retrospective queries and adapt to query-specific error penalties on existing sliding window queries. We developed a space-efficient extension of the existing Timing Bloom Filter called the Block Timing Bloom Filter (BTBF), and turned the standard BTBF into an inferential BTBF. We also developed an inferential version of the Generalized Bloom Filter (GBF).

We showed that our sliding window posterior expressions for the inferential GBF and BTBF are accurate in practice. We experimentally evaluated the standard and inferential filters, comparing total penalties incurred by each when answering sliding window queries with query-specific penalties. The inferential BTBF generally reduced penalties by 10%–70%. Accurate modeling of filters and item probabilities is important, as poor modeling can cause inferential filters to perform poorly.

We showed that in most cases, the inferential GBF and BTBF outperform the standard GBF and BTBF when false positive/negative costs vary between queries. We also showed that the inferential BTBF outperforms the inferential GBF when the window width is fixed for all queries, and that the inferential BTBF outperforms buffer-based techniques, such as those using Counting Bloom Filters, when storage space is limited.

Future work in this area may include additional modeling, developing inferential versions of other filters, and identifying optimal parameters for inferential filters.

9 ACKNOWLEDGEMENTS

This work was supported by grants N00014-07-C-0311 from the Office of Naval Research, CPS-1330110 and IIS-1527984 by the National Science Foundation, and by the National Physical Science Consortium Graduate Fellowship.

REFERENCES

[1] Kursad Asdemir, Özden Yurtseven, and Moin Yahya. 2008. An Economic Model of Click Fraud in Publisher Networks. *Int. J. Electron. Commerce* 13, 2 (Dec. 2008), 61–90.

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

[3] Andrei Broder and Michael Mitzenmacher. 2003. Survey: Network Applications of Bloom Filters: A Survey. 1 (11 2003).

[4] CAIDA. 2011. The CAIDA UCSD Anonymized Internet Traces 2011 - Equinox Chicago Direction A starting 20110217-125904. Available at http://www.caida.org/data/passive/passive_2011_dataset.xml.

[5] K. Cheng, L. Xiang, and M. Iwaihara. 2005. Time-decaying bloom filters for data streams with skewed distributions. In *Proc. RIDE-SDMA 2005*. 63–69.

[6] K. Christensen, A. Roginsky, and M. Jimeno. 2010. A new analysis of the false positive rate of a Bloom filter. *Inform. Process. Lett.* 110, 21 (2010), 944–949.

[7] Jonathan L Dautrich Jr and Chinya V Ravishankar. 2013. Inferential time-decaying Bloom filters. In *Proceedings of the 16th International Conference on Extending Database Technology*. ACM, 239–250.

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

[9] B. Donnet, B. Baynat, and T. Friedman. 2006. Retouched Bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Proc. CoNEXT*. 13.

[10] M Douglas McIlroy. 1982. Development of a Spelling List. 1 (02 1982), 91 – 99.

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

[12] Lee L. Gremillion. 1982. Designing a Bloom Filter for Differential File Access. *Commun. ACM* 25, 9 (Sept. 1982), 600–604. https://doi.org/10.1145/358628.358632

[13] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. 2010. The dynamic Bloom filters. *Knowledge and Data Engineering, IEEE Transactions on* 22, 1 (2010), 120–133.

[14] Śandor Kiss, Éva Hosszu, János Tapolcai, Lajos Rónyai, and Ori Rottenstreich. 2018. Bloom Filter with a False Positive Free Zone. In *INFOCOM 2017. IEEE International Conference on Computer Communications. IEEE*. IEEE.

[15] G. Koloniari, N. Ntarmos, E. Pitoura, and D. Souravlias. 2011. One is enough: distributed filtering for duplicate elimination. In *Proc. CIKM*. 433–442.

[16] R.P. Laufer, P.B. Velloso, and O.C. Duarte. 2011. A Generalized Bloom Filter to secure distributed network applications. *Computer Networks* (2011).

[17] X. Li, J. Wu, and J. Xu. 2006. Hint-based routing in WSNs using scope decay bloom filters. In *Proc. IWNAS*. IEEE, 8–15.

[18] Zhe Li and Kenneth A. Ross. 1995. PERF Join: An Alternative to Two-way Semijoin and Bloomjoin. In *Proceedings of the Fourth International Conference on Information and Knowledge Management (CIKM '95)*. ACM, New York, NY, USA, 137–144. https://doi.org/10.1145/221270.221360

[19] Y. Lu, B. Prabhakar, and F. Bonomi. 2005. Bloom filters: Design innovations and novel applications. In *Proc. Allerton Conference*.

[20] Lailong Luo, Deke Guo, Richard T. B. Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing Bloom Filter: Challenges, Solutions, and Comparisons. *CoRR* abs/1804.04777 (2018). arXiv:1804.04777 http://arxiv.org/abs/1804.04777

[21] L. Luo, D. Guo, J. Wu, O. Rottenstreich, Q. He, Y. Qin, and X. Luo. 2017. Efficient Multiset Synchronization. *IEEE/ACM Transactions on Networking* 25, 2 (April 2017), 1190–1205. https://doi.org/10.1109/TNET.2016.2618006

[22] Lothar F. Mackert and Guy M. Lohman. 1986. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 149–159. http://dl.acm.org/citation.cfm?id=645913.671480

[23] Udi Manber and Sun Wu. 1994. An Algorithm for Approximate Membership Checking with Application to Password Security. *Inf. Process. Lett.* 50, 4 (May 1994), 191–197. https://doi.org/10.1016/0020-0190(94)00032-8

[24] Ahmed Metwally, Dharma Agrawal, and A El Abbadi. 2005. Duplicate detection in click streams. (01 2005), 12–21.

[25] Calvin N. Mooers. 1951. Zatocoding applied to mechanical organization of knowledge. *American Documentation* 2, 1 (1 1951), 20–32. https://doi.org/10.1002/asi.5090020107

[26] J. K. Mullin. 1990. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering* 16, 5 (May 1990), 558–560. https://doi.org/10.1109/32.52778

[27] James K. Mullin. 1993. Estimating the Size of a Relational Join. *Inf. Syst.* 18, 3 (April 1993), 189–196. https://doi.org/10.1016/0306-4379(93)90037-2

[28] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. The PageRank citation ranking: Bringing order to the web. (1999).

[29] C.E. Rothenberg, C.A.B. Macapuna, F.L. Verdi, and M.F. Magalhaes. 2010. The deletable Bloom filter: a new member of the Bloom family. *Communications Letters, IEEE* 14, 6 (2010), 557–559.

[30] O. Rottenstreich, Y. Kanizo, and I. Keslassy. 2012. The variable-increment counting Bloom filter. In *Proc. Infocom.*

[31] O. Rottenstreich and I. Keslassy. 2012. The Bloom Paradox: When not to Use a Bloom Filter?. In *Proc. Infocom.*

[32] S. Saroiu, K.P. Gummadi, R.J. Dunn, S.D. Gribble, and H.M. Levy. 2002. An analysis of internet content delivery systems. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 315–327.

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

[34] S. Tarkoma, C. Rothenberg, and E. Lagerspetz. 2012. Theory and practice of bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE* 99 (2012), 1–25.

[35] X. Wang and H. Shen. 2010. Approximately Detecting Duplicates for Probabilistic Data Streams over Sliding Windows. In *Proc. PAAP.* 263–268.

[36] J. Wei, H. Jiang, K. Zhou, D. Feng, and H. Wang. 2011. Detecting Duplicates over Sliding Windows with RAM-Efficient Detached Counting Bloom Filter Arrays. In *Proc. NAS.* 382–391.

[37] W.B. Wu and C.V. Ravishankar. 2003. The performance of difference coding for sets and relational tables. *Journal of the ACM (JACM)* 50, 5 (2003), 665–693.

[38] L. Zhang and Y. Guan. 2008. Detecting click fraud in pay-per-click streams of online advertising networks. In *Proc. ICDCS.* IEEE, 77–84.

[39] Y. Zhao and J. Wu. 2010. B-SUB: A Practical Bloom-Filter-Based Publish-Subscribe System for Human Networks. In *Proc. ICDCS.* 634–643.