Inferential Time-Decaying Bloom Filters

Jonathan L. Dautrich Jr.
Computer Science and Engineering
University of California, Riverside
dautricj@cs.ucr.edu

Chinya V. Ravishankar
Computer Science and Engineering
University of California, Riverside
ravi@cs.ucr.edu

ABSTRACT

Time-Decaying Bloom Filters are efficient, probabilistic data structures used to answer queries on recently inserted items. As new items are inserted, memory of older items decays. Incorrect query responses incur penalties borne by the application using the filter. Most existing filters may only be tuned to static penalties, and they ignore Bayesian priors and information latent in the filter.

We address these issues in an integrated way by converting existing filters into *inferential* filters. Inferential filters combine latent filter information with Bayesian priors to make query-specific optimal decisions. Our methods are applicable to any Bloom Filter, but we focus on developing *inferential time-decaying filters*, which support new query types and sliding window queries with varying error penalties.

We develop the inferential version of the existing Timing Bloom Filter. Through experiments on real and synthetic datasets, we show that when penalties are query-specific and prior probabilities are known, the inferential Timing Bloom Filter reduces penalties for incorrect responses to sliding-window queries by up to 70%.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models*; G.3 [**Probability and Statistics**]: Miscellaneous

General Terms

Design, Performance

1. INTRODUCTION

Bloom Filters are probabilistic data structures widely used for set membership queries [19]. 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 $\widehat{\mathcal{F}}$, and the set of items inserted into $\mathcal{F}$ is denoted $\{\mathcal{F}\}$.

EDBT/ICDT '13 March 18 - 22 2013, Genoa, Italy

The *Classical Bloom Filter* [3] is used to test whether an item $x \in \{\mathcal{F}\}$. A query should return Pos if $x \in \{\mathcal{F}\}$ and Neg if $x \notin \{\mathcal{F}\}$. Inserted items are never deleted, so filters may become saturated, leading to *false positive* errors where Pos is returned even when $x \notin \{\mathcal{F}\}$.

A *Time-Decaying Bloom Filter* [6, 9, 10, 23, 24], 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 ($I_x < w$). $\widehat{\mathcal{F}}$ only approximates the insertion history, 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 borne by the application using the filter.

Responding accurately to retrospective queries is difficult since information in $\widehat{\mathcal{F}}$ is only approximate. We show that $\widehat{\mathcal{F}}$, though limited, contains valuable information that is ignored by current time-decaying filters. For example, in [6, 18], cells are counters which embed information about insertion age, since they are decremented with each insertion. Yet, these filters are content to return Pos if and only if all the counters set when x was last inserted are non-zero, ignoring the other information in the counts. Even filters that *do* consider exact counts [9] do not provide a clear framework for deciding how to use counter values.

In addition, most filters base decisions on "forward" probabilities, ignoring Bayesian priors. Such filters may yield worse results than when no filter is used at all [16]. 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 $\widehat{\mathcal{F}}$.

Definition 3. A *standard* time-decaying filter uses limited information from $\widehat{\mathcal{F}}$ to answer sliding window queries with Pos or Neg. An *inferential* one 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 ranging from financial costs to response latencies. Standard filters may be tuned to minimize such penalties, so long as they are static and fixed at filter design time. In reality, however, penalties vary by queried item, time, and context. For example, a wrong decision on a high-value item generally costs more than one on a low-value item. Scenarios with query-specific penalties include duplicate detection for items with different values [2], distributed caches with item-specific access times [17], and web crawler caches with pages weighted by importance [13].

For best results, each decision should minimize expected penalty, and must be made dynamically, query-by-query. 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. The filters then use this posterior to compute expected penalties of POS and NEG responses and make minimum-penalty decisions for each query.

In addition to enabling minimum-cost decision strategies, inferential filters enhance flexibility by supporting new types of retrospective queries. For instance, we can identify the most likely insertion age for x by comparing $P(I_x = i|\widehat{\mathcal{F}})$ for various i. We can also aggregate over all i to find the *expected* insertion age. As far as we know, our work is the first to support such queries using Bloom Filters.

1.1 Contributions

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 we can also use our framework to develop a more accurate version of the Classical Bloom Filter as in [16] (see Section 2.4).

We show in detail how to develop an inferential version of the *Timing Bloom Filter* (TBF) [23], 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 [23]. We discuss the standard and inferential BTBF in Sections 3 and 4, respectively.

In Section 5, we evaluate the standard and inferential BTBF on real and synthetic data streams. We randomly vary error penalties for sliding window queries and compare the total penalties incurred using each BTBF. Our results show that the inferential BTBF improves upon the standard BTBF, reducing penalties when Bayesian priors are known. We discuss related work in Section 6.

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

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

2.1 The Classical Bloom Filter

The Classical Bloom Filter [3] represents the set $\{\mathcal{F}\}$ of all items inserted into the filter ($n = |\{\mathcal{F}\}|$). Each cell is a single bit initialized to 0. We insert x by setting each cell in R_x 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.

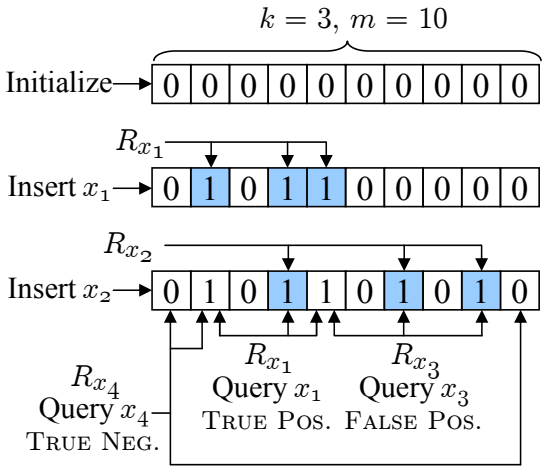

Figure 1: Inserts and queries on a Classical Bloom Filter representing $\{\mathcal{F}\} = \{x_1, x_2\}$.

Figure 1 shows several inserts and possible query outcomes. Cells are never reset to 0, so all cells in R_x remain 1 if $x \in \{\mathcal{F}\}$. Thus, the Classical Bloom Filter has no false negatives. However, 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:

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

2.2 Analytical Approximations

Bloom filter analyses often use approximations. The commonly used expression for P_{FP} approximates Equation 1 by replacing r_x with k, since the k cells touched by x are usually distinct when $m \gg k$. Equation 1 itself assumes that touching each cell in R_x is an independent event, which is not strictly correct [5]. Such approximations are justified for most applications since they greatly simplify analysis and generally have little impact on accuracy [5]. We make such simplifying assumptions throughout the paper when computing posteriors. Our experimental results show that the posteriors are accurate enough to substantially reduce error penalties in most situations.

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, checking every cell is impractical, and most information relevant to x can be obtained from the cells R_x. Thus, for the remainder of the paper, we will use the following notation:

Definition 5. $P(I_x = i|R_x)$, 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.

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
$n = \lvert\{\mathcal{F}\}\rvert$	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	$\lvert R_x\rvert$
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

To turn a standard filter into an inferential one, we must derive an expression for $P(i|R_x)$. $P(i|R_x)$ depends on the filter's contents and on the prior probability that $I_x = i$. Since $P(i|R_x)$ is generally difficult to derive directly, we apply Bayes' theorem:

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

where $P(i)$, $P(R_x|i)$, and $P(R_x)$ are defined as follows:

- $P(I_x = i)$ or $P(i)$: Prior (marginal) probability that exactly i insertions occurred since x was last inserted.
- $P(R_x)$: Prior (marginal) probability that cells R_x have their current contents.
- $P(R_x|I_x = i)$ or $P(R_x|i)$: 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 6. 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:

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

$P(\perp)$ gives the probability that x was not inserted during any of the n insertions thus far, and $P(i), i \neq \perp$ gives the probability that x was inserted, followed by i items distinct from 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 to evaluate $P(\alpha \leq I_x < \beta)$ for $0 \leq \alpha < \beta$, which leads to the following geometric series:

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

2.3.2 Computing Posterior $P(i|R_x)$

To compute $P(i|R_x)$ with Equation 2, we must use $P(R_x)$, which can be hard to compute directly. We can re-write $P(R_x)$ as a marginal sum over $P(i)P(R_x|i)$, giving:

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

To derive $P(i|R_x)$, we still need to find an expression for $P(R_x|i)$ and an efficient way to compute sums over the product $P(i)P(R_x|i)$. Both challenges are filter-specific, so we address them in Section 4.

2.3.3 Retrospective Queries

Inferential time-decaying filters use $P(i|R_x)$ as a building block for constructing responses to retrospective queries. If we compare values of $P(i|R_x)$ for all i, we can determine the highest probability choice i for I_x, which tells us when x was most likely last inserted.

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

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

We can then compute the expected number of insertions since x was last inserted:

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

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 a warm-up, we show how to develop an inferential version of the Classical Bloom Filter (Section 2.1) by computing the posterior $P(x \in \{\mathcal{F}\}|R_x)$. This posterior was first derived for the Classical Bloom Filter in [16], but the analysis there does not apply to time-decaying filters or retrospective queries. 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 [16]. Using this posterior, the inferential Classical Bloom Filter can be adapted to provide optimal responses given item-specific prior probabilities p_x and query-specific error penalties. Since $n = |\{\mathcal{F}\}|$,

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

Let $r_x = |R_x|$, and let c_x be the number of cells (bits) in R_x that are set to 1.

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

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:*

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

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 $c_x = r_x$:

$$P(I_x < n|R_x) = \frac{\sum_{i=0}^{n-1} P(i) \cdot 1}{P(\perp)\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{r_x} + \sum_{i=0}^{n-1} P(i) \cdot 1}$$

$$= \frac{(1 - P(\perp))}{P(\perp) \cdot P_{FP} + (1 - P(\perp))}$$

$$= \frac{1}{1 + \dfrac{P(\perp) \cdot P_{FP}}{1 - P(\perp)}} = \frac{1}{1 + \dfrac{(1-p_x)^n \cdot P_{FP}}{1 - (1-p_x)^n}}. \quad \square$$

For the Classical Bloom Filter it is common to assume that $r_x = k$, $k = (m/n)\ln 2$ and that $P_{FP} = (1 - e^{-kn/m})^k$, as in [10, 16, 18, 23]. Doing so gives $P_{FP} = (1/2)^{(m/n)\ln 2}$, so when $c_x = r_x$, rearranging Theorem 1 and substituting $P(I_x < n) = 1 - (1 - p_x)^n$ gives

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

which is consistent with the probability expressions in [16].

2.5 Expected Number of Distinct Items

The accuracy of our posterior expressions 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 $x \in U$, and is given by

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

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

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

If the item probabilities follow a Zipf-like discrete power law $p_x = 1/(H_{|U|} \cdot x)$, then it is shown in [22] that

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

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

When real-world distributions are hard to model analytically, we can experimentally determine $D(j)$ for certain values of j, and then interpolate for intermediate values. Our experience suggests that piecewise logarithmic interpolation (using straight lines on a log-log plot) generally yields acceptable results. That is, 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:

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

2.6 Minimum-Penalty Decisions

As noted in Section 1, penalties for incorrect responses may be query-specific, so tuning a standard filter to fixed false positive/negative rates is non-optimal. With an inferential filter, we can use posteriors to make better-informed, query-specific choices between returning POS and NEG.

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 $E_{\text{POS}} \le E_{\text{NEG}}$, and NEG otherwise.

3. STANDARD TIMING BLOOM FILTERS

The Timing Bloom Filter (TBF) [23] 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.

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 $G(r_x, c_x, \lambda_x)$

3.1.1 TBF: Insert

To insert item x into a TBF, we proceed 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 7. 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, $\lambda(\theta.T) \geq 1$ by the time any queries are issued. 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.

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.

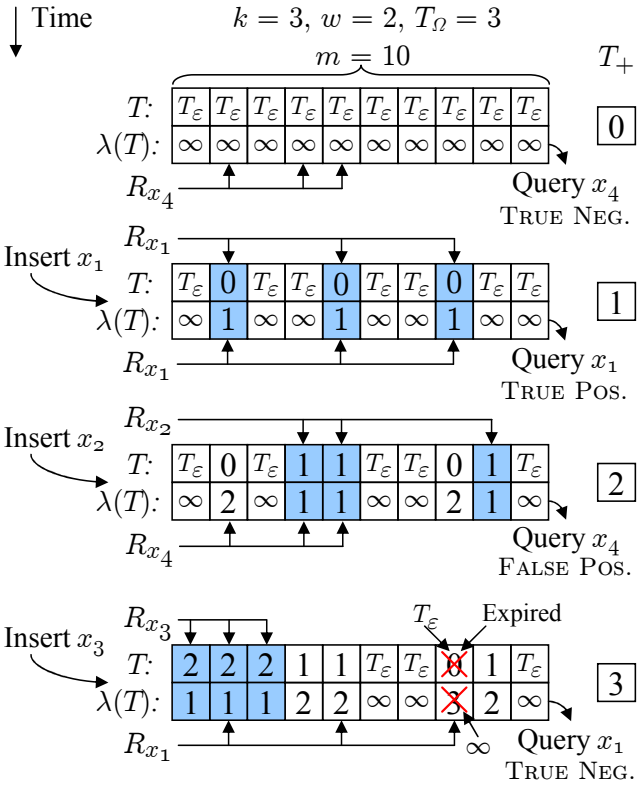

Figure 2: Inserts and queries on a Timing Bloom Filter. Timestamps touched by each insertion highlighted. Ages are relative to the updated T_+.

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 prohibitively expensive. The solution in [23] is to increase T_Ω by an amount we call *padding*.

Definition 8. 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 that we need only check $O(k)$ timers per insertion. Since we already

[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.

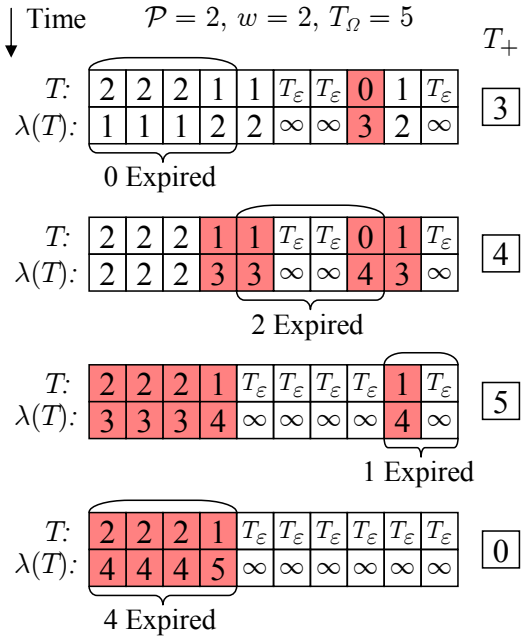

Figure 3: Using padding $\mathcal{P}$ in a Timing Bloom Filter, new insertions omitted for clarity. Newly expired timestamps (highlighted) can remain expired for $\mathcal{P}+1$ insertions before T_+ cycles, so we need only check 4 timers per insertion.

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.

3.2 Block Timing Bloom Filters

A problem with the TBF is that the $T_\Omega+2$ possible timestamp values require it to use $O(\log w)$ bits per timer (*bpt*). For example, the sample TBF in [23] 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 per item in the window of interest, which is excessive. In fact, given 303 bits per item, we could simply use unique hashes for every item in the window, and index them using a hash table. This alternative setup performs at least as well as the TBF, 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 [18]. The BTBF was alluded to, but not developed, in [23].

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. We keep a counter b to record 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.

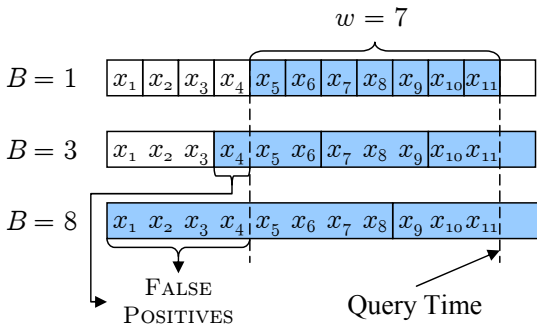

Figure 4: For the BTBF to treat x_5 as in the window, it must treat all items in x_5's block as in the window. Thus, larger blocks yield more false positives.

Definition 7 for $\lambda(\theta.T)$ still holds, but our definition of an expired timestamp becomes more general:

Definition 9. In a BTBF, timestamp $\theta.T$ has *expired* once its age $\lambda(\theta.T) \geq \left\lceil \frac{w-b}{B} \right\rceil + 1$.

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. Since Neg is returned only if at least one timestamp in R_x has expired, 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, respectively, 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 must return 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.

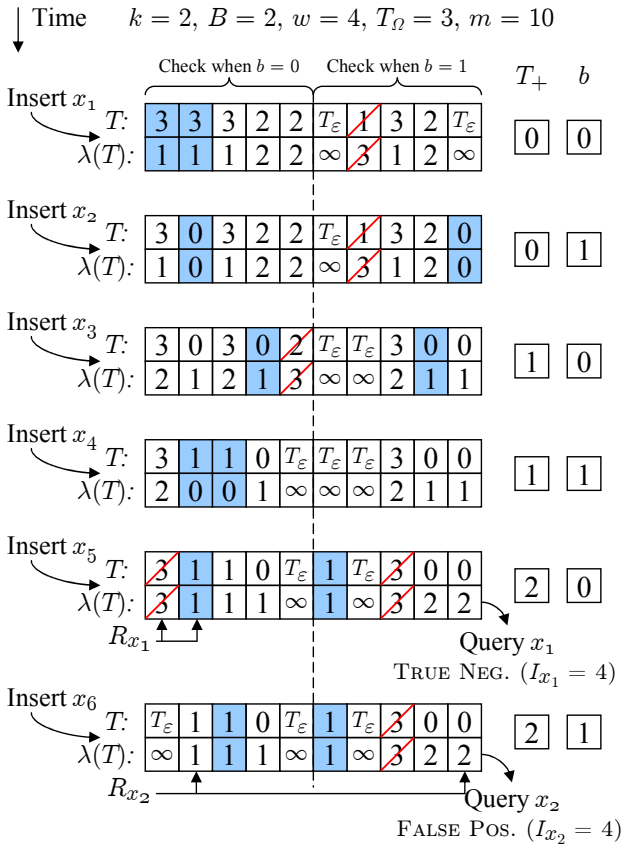

Figure 5: A BTBF with no padding. Timers touched by the last insertion are highlighted. Timestamp T expires once $\lambda(T) \geq 3$, and is shown with a slash until it is changed to T_ε. Since $B = 2$, we need only check half the timers for expiration after each insertion.

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* Block Timing Bloom Filter (BTBF), which is able to return the sliding window posterior probability $\lim_{n\to\infty} P(I_x < w|R_x)$, 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. Due to space limits, we do not derive $E[I_x|R_x]$, nor do we explicitly derive $\lim_{n\to\infty} P(i|R_x)$. For the sake of brevity, we omit the limit notation in the rest of the paper.

Definition 10. Let T_x be the oldest timestamp in R_x, and let λ_x be its age $\lambda(T_x)$, given by

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

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 11. 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 probability $P(i)$ is given by Equation 4. In order to compute posteriors, we must sum over the conditional probability $P(R_x|i)$. This probability varies depending on the relationship between i and λ_x, so to sum $P(R_x|i)$ over various i, 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 use the following function when computing posteriors:

Definition 12. 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. 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 5.1.4).

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

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

The 0 posterior probability that $I_x < w$ when timestamps in R_x have expired reflects the fact that the standard BTBF has no false negatives.

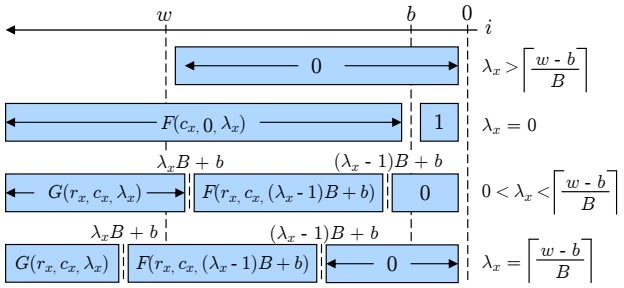

Figure 6: $P(R_x|i)$ for different values of λ_x and i.

4.2 Case $\lambda_x = 0$

LEMMA 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 $F(r_x, 0, b)$. □

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 13. 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.

LEMMA 2. *If $0 < \lambda_x \leq \lceil \frac{w-b}{B} \rceil$, then*

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

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 from the following events:

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*

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

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

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

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

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

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

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

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

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

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

PROOF: If $\lambda_x = \lceil \frac{w-b}{B} \rceil$, then $(\lambda_x-1)B+b < w \leq \lambda_x B+b$. 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

We want to efficiently evaluate the inferential BTBF's posteriors given by Equations 23, 26, and 27. Since m and k are fixed, we can pre-compute the value $(1-1/m)^k$ used in Equation 21. Equation 23 requires $O(\log_2(w \cdot r_x \cdot D(b))) \preceq O(\log_2(w \cdot k \cdot B))$ floating point multiplications to compute its exponents, plus the cost of computing $D(b)$, which depends on the distribution (see Section 2.5). Equation 23 is the most expensive, but is fortunately needed only in the rare case when $\lambda_x = 0$.

Equation 26 requires $O(\log_2 w)$ floating point multiplications, plus the cost of computing $F(c_x, 0, B)$. Since B is fixed, and c_x has only $O(k)$ possible values, we pre-compute and cache the $O(k)$ values of $F(c_x, 0, B)$. We then use the same cached values for Equation 27, which requires only $O(\log_2 B)$ additional multiplications, since $\lambda(x) = \lceil \frac{w-b}{B} \rceil$. Using these techniques, we observed that less time was spent computing probabilities than managing the filter itself.

5. EXPERIMENTS

5.1 Experimental Setup

We have examined two approaches for answering sliding window queries: the standard BTBF, which returns POS or NEG, and the inferential BTBF, which returns the sliding window posterior $P(I_x < w | R_x)$. We now vary false positive/negative error penalties across queries, and test whether using posteriors reduces overall penalties. We compare the techniques using a real-world data stream and two synthetic data streams.

5.1.1 Queries and Error Penalties

We use the same data stream for queries and inserts. As we observe each new item x, we query for x and then insert x, regardless of the query outcome. This model might be used for an expensive multi-level LRU cache, where we only want to make a time-consuming 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 [9, 23], 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 describe in Section 2.6 for deciding whether to return POS or NEG.

5.1.2 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 [18] provides limited guidance. For the BTBF, we fix $bpt = k$ as in [18]. 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 for each trial we tried all k for $1 \leq k \leq 30$, and chose k to minimize total penalty. Thus, penalties measured for each filter are independent of its parameter selection mechanism.

5.1.3 Measuring Performance for Each Filter

Each filter is allowed *bpi* bits per item in the window. Since there are w such items, 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.

Trials with the same stream and w are grouped into an *experiment*, which measures penalties incurred by the standard and inferential BTBF over a range of *bpi* values. For each stream, we ran experiments under two conditions. Condition [POS≈NEG] uses a small enough w to make the numbers of 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, showing the ratio of the penalty for the standard BTBF to that for the inferential BTBF. The choice of w and the actual POS/NEG ratios for each exper-

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

Table 3: Data parameters and characteristics for each experiment/condition. Pos/Neg gives the ratio of queries for items with $I_x < w$ to those with $I_x \geq w$.

iment are given in Table 3. For convenience, we chose w to be a power of 2, but our implementation supports arbitrary integer values for w.

5.1.4 Errors from Approximations

We made several approximations while deriving the posterior $P(I_x < w|R_x)$, so we want to evaluate its accuracy when applied to each dataset. During 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.

We define the *Posterior Error* to be the average absolute difference between a query posterior and its bucket midpoint, given by

$$\frac{1}{n}\sum_{\ell=1}^{20} |f_\ell - M_\ell| \cdot \eta_\ell \tag{28}$$

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.

5.1.5 Implementation

We implemented the filters in Java, and ran each trial using 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. The inferential BTBF caches $O(k)$ static floating-point values to speed up computation in common cases (Section 4.4).

5.2 Uniform Data Stream

The *Uniform* data stream draws items uniformly at random, with replacement, from a set U of 2^{16} integers. $D(j)$ is given by Equation 16, and for each $x \in U$, $p_x = 1/|U|$. Penalty ratios for the standard and inferential BTBF are shown in Figure 7, and Posterior Errors are given in Figure 8. The inferential filter incurs around 80% of the penalties incurred by the standard filter.

At very low *bpi* 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. For large *bpi*, so much filter information is available that most posteriors are close to 0 or 1, and thus differ from their corresponding bin centers by half the bin width, explaining the convergence to 0.025. Thus, our approximations produce noteworthy error only for moderate *bpi*. However, the Posterior Errors for such *bpi* remain under 0.05, indicating largely accurate posterior expressions.

5.3 Streams with Skewed Distributions

If the distribution of stream items is skewed, computing accurate posteriors requires the following assumptions:

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, prior probabilities, and thus posteriors, cannot be computed efficiently. If Assumption 2 is violated, the time-invariant priors yield inaccurate posteriors, and may increase penalties.

5.3.1 Power Law Stream

Our Power Law stream consists of a sequence of items from the set $U = \{x \in \mathbb{Z} \mid 1 \leq x \leq 2^{16}\}$. Items are drawn from U according to a discrete power law distribution with $p_x = 1/(x \cdot H_{|U|})$, where $H_{|U|} = \sum_{i=1}^{|U|} 1/i$ is the $|U|^{\text{th}}$ harmonic number. Computing p_x is easy since $H_{|U|}$ is fixed, and p_x is time-invariant. $D(j)$ is given by Equation 17.

Penalty ratios are shown in Figure 9, and Posterior Errors in Figure 10. The standard BTBF has no false negatives, so it performs well under condition [Pos>Neg]. Thus, penalty reductions are more pronounced under [Pos≈Neg].

Again, we see that our posterior expressions are largely accurate, as the Posterior Errors stay under 0.035 for all *bpi*. In this case, as for Uniform streams, Posterior Errors are low for very low *bpi*, where the posterior depends largely on the precisely-known priors, and converges to 0.025 when *bpi* is large.

5.3.2 Source IP Data Stream

The Source IP data stream [1] is a sequence of anonymized source addresses taken from IPv4 packet headers ($|U| = 2^{32}$). As expected, 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.

Penalty ratios are shown in Figure 11, and Posterior Errors in Figure 12. The stream is bursty, so p_x is not strictly time-invariant, violating Assumption 2. Thus, prior probabilities 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.

6. RELATED WORK

Filters including the Standard Bloom Filter [3], the Generalized Bloom Filter [10], and others [12, 11] use single-bit cells. Other filters use multiple bits in each cell to represent counters, as in the Counting Bloom Filter (CBF) [8], timers, as in the TBF [23], or other values [4, 6, 15].

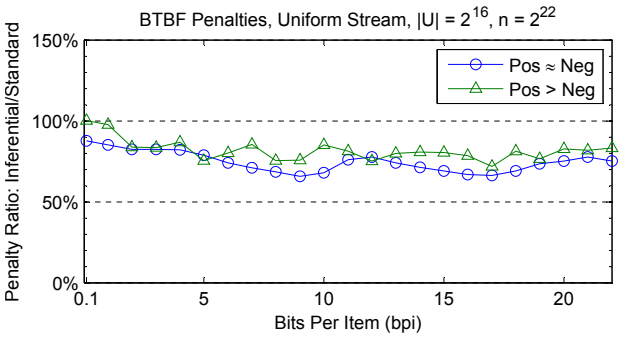

Figure 7: Penalty Ratios, Uniform Stream.

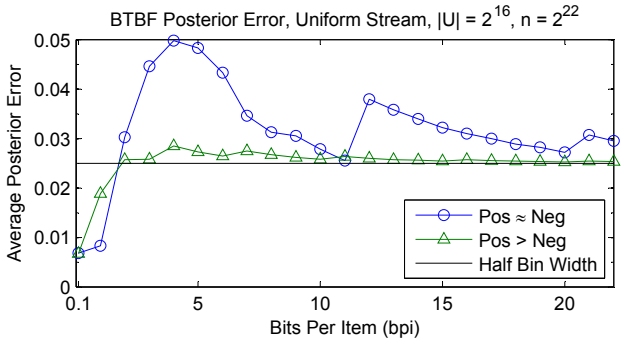

Figure 8: Posterior Errors, Uniform Stream.

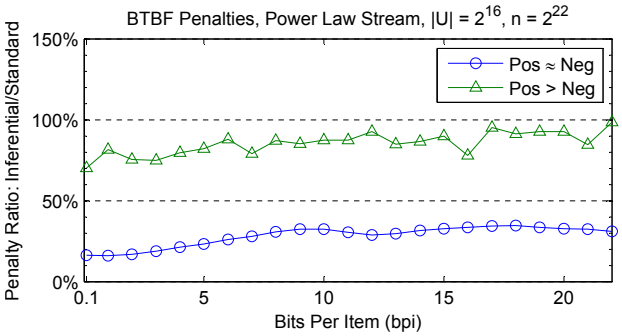

Figure 9: Penalty Ratios, Power Law Stream.

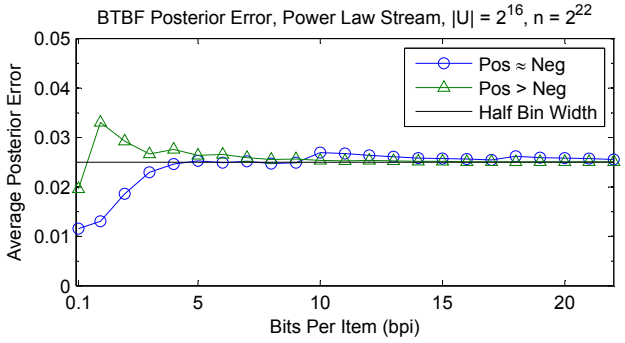

Figure 10: Posterior Errors, Power Law Stream.

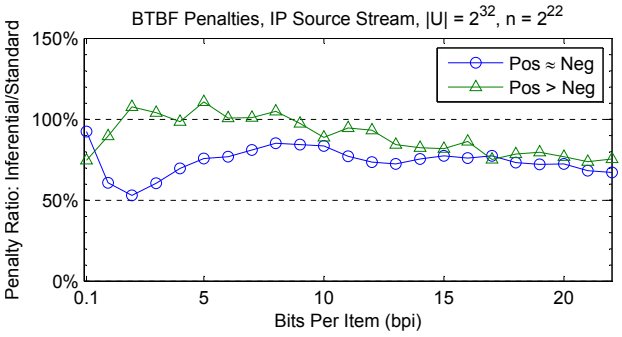

Figure 11: Penalty Ratios, IP Source Stream.

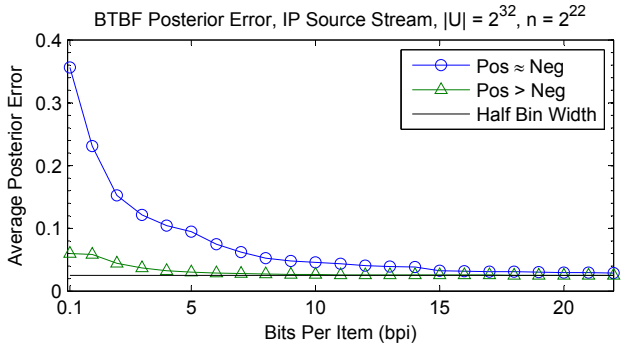

Figure 12: Posterior Errors for IP Source stream.

Simple filters [3, 16, 7] only allow items to be inserted, and generally represent static sets. Deletable filters [12, 14, 15] allow items to be deleted as well as inserted, and represent dynamic sets. Decaying filters represent a dynamic set of recently inserted items. As time goes on and 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 [20, 21]. 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 [6], cells to decrement are chosen randomly after each insertion, while in [9, 18, 24], all non-zero counters are decremented after each block of inserts. The TBF [23] 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.

Filters commonly contain unused information. Authors of [9] 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 [16] 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 [4] use knowledge of data stream item frequencies to improve accuracy. They construct a hierarchy of decaying

filters and assign items to filters based on frequency, using more information to store more frequent items.

7. 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 showed that our sliding window posterior expressions for the inferential BTBF are accurate in practice. We experimentally evaluated the standard and inferential BTBF, 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. Future work in this area may include additional modeling, developing inferential versions of other filters, and identifying optimal parameters for inferential filters.

8. ACKNOWLEDGEMENTS

This work was supported by grant N00014-07-C-0311 from the Office of Naval Research and by the National Physical Science Consortium Graduate Fellowship. We would like to thank the anonymous reviewers for their detailed feedback.

9. REFERENCES

[1] 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`.

[2] K. Asdemir, O. Yurtseven, and M. Yahya. An economic model of click fraud in publisher networks. *Int. J. Electron. Commerce*, 13(2):61–90, Dec. 2008.

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

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

[5] K. Christensen, A. Roginsky, and M. Jimeno. A new analysis of the false positive rate of a bloom filter. *Information Processing Letters*, 110(21):944–949, 2010.

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

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

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

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

[10] R. Laufer, P. Velloso, and O. Duarte. A generalized bloom filter to secure distributed network applications. *Computer Networks*, 2011.

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

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

[13] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

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

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

[16] O. Rottenstreich and I. Keslassy. The bloom paradox: When not to use a bloom filter? In *Proc. Infocom*, 2012.

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

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

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

[20] X. Wang and H. Shen. Approximately detecting duplicates for probabilistic data streams over sliding windows. In *Proc. PAAP*, pages 263–268, 2010.

[21] J. Wei, H. Jiang, K. Zhou, D. Feng, and H. Wang. Detecting duplicates over sliding windows with ram-efficient detached counting bloom filter arrays. In *Proc. NAS*, pages 382–391, 2011.

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

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

[24] Y. Zhao and J. Wu. B-sub: A practical bloom-filter-based publish-subscribe system for human networks. In *Proc. ICDCS*, pages 634–643, 2010.