Efficient Web Profiling by Time-Decaying Bloom Filters

Kai Cheng[1], Mizuho Iwaiahra[2], Limin Xiang[1], and Kazuo Ushijima[1]

[1] Faculty of Information Science, Kyushu Sangyo University, Fukuoka, Japan
[2] Graduate School of Informatics, Kyoto University, Kyoto, Japan

Abstract. We propose an efficient data structure, called Time-decaying Bloom Filters (TBF) to maintain time–sensitive hit frequency for the web. Different from the traditional Bloom Filters, a TBF uses an array of counters instead of a bit vector. The value of each counter will decay periodically with time elapsing. We exploit the skewness of web usage and propose a dynamic approach to counter management. In this approach, only "heavy hitters" are monitored by larger counters. Since the majority of web contents are not used that freqeuntly, the proposed approach can improve the space efficiency to a large extent.

1 Introduction

Web usage data contain valuable about the behaviors of web users. In order to improve the quality of services it is critical to maintain various statistics about the underlyng web contents. Particularly, *hit frequency* profile, that is, how often the specific web content is used, can be used in many eapplications, such as web caching, web site reconstruction, and search service personalization. For example, search engines can be tailored by adding hit frequency in the process of search result ranking so that the popularity of web sites, from the clients' perspective, can be used as a guide for less experienced web users [1].

The primary challenge in maintaining such profiles arises in the fact that there are so large volume of web contents to monitor but there is often a quite limited memory space available. The current size of the web is expected to be several billions and popular portal sites often see hundreds of millions requests per day. The challenge also comes from the time-sensitivity of web usage due to the fast change of user interests. Thus recent data more accurate represent the real state of the web usage than past one. It is important to deal with such time-sensitivity of web usage.

The brute-force approach to maintaining hit frequency is to use as many counters as possible for all web sites one can find. This can only be applicable in small–scale web usage profiling, but is unrealistic for large-scale applications, such as busy portal sites, or general-purpose search services due to the limited memory space and CPU time available.

Bloom Filters are space-efficient data structures that maintain a very compact inventory of the underlying data, supporting membership queries over a

given data set [2].The space requirements of Bloom filters fall significantly below the information theoretic lower bounds for error-free data structures. This efficiency is at the cost of a small false positive rate (items not in the set have a small constant probability of being listed as in the set), but have no false negatives (items in the set are always recognized as being in the set). Bloom filters are widely used in practice when storage is at a premium and an occasional false positive is tolerable.

The standard Bloom Filters use a bitmap to hold information about the underlying data set. Initially, all bits in the bitmap is turned off. Then, each item of the given set is hashed into several locations of the bitmap, by using a set of independent hash functions. Bits at these locations are then turned on. In order to find whether an item is a member of the set, one can first compute the locations the item is mapped to. Then checks if all bits at these locations are on. If so, the answer is yes; otherwise answer no. Because bits can be set by other items due to the hash collisions, there may be some false positives.

Standard Bloom Filters, although useful, have several limitations. First, they do not support deletes because simply turning off the corresponding bits may introduce new false negative errors (some bits of other items, although still in the set, may be turned off). Second, they cannot deal with multi–set (set with duplicates of items) to return multiplicities of items. Many applications, e.g., iceberg queries [3], data mining, and data stream management, such multiplicity information is crucial.

Several improvements have been proposed over the original Bloom Filter. In [4], Li, F. et al proposed *Counting Bloom Filters (CBF)*, where a counter has been attached to each bit in the array to count the number of items mapped to that location. Thus, it can support deletions in a set, simply by decrementing the corresponding counters by 1. To maintain the compactness of the structure, these counters were limited to 4 bits, which is shown statistically to be enough to encode the number of items mapped to the same location, based on the maximum occupancy in a probabilistic urn model, even for very large sets.

In [5], Cohen, S. et al proposed a sophisticated implementation for CBF, called, Spectral Bloom Filters (SBF). It introduces efficient schemes for counter updates, namely Minimum Increase (MI), Recurring Minimum (RM) that try to reduce false positive rate. In addition, it develops variable length string access methods for compact representation of the counters, which thus in principle allow counters of arbitrary length. However, to the best of our knowledge, none of the proposed Bloom Filter improvements supports time-sensitive counting of the multiplicity.

Time-sensitivity is important since in many traditional and emerging applications, data streams play an increasingly important role, e.g., web tracking and personalization, medical monitoring, sensor databases, and financial monitoring. In most of these applications, the goal is to make decisions based on the statistics or models gathered over the recently observed data elements. For example, one might be interested in gathering statistics about web sites visited by a set of users, say your employers in recent days. Moreover, we would like to maintain

these statistics in a continuous fashion. Thus continuously maintaining statistics about the recent usage while giving away the pass story is a natural requirement.

In this paper, we propose Time-Decaying Bloom Filters (TBF), a novel improvement over the current Bloom Filter variants. We address the challenges in maintaining time-sensitive frequency counts over data streams in general, and web usage data in particular. First, we propose a basic scheme, called basic TBF, for maintaining time-decaying aggregates over data streams. We then propose optimizations to the basic scheme by leveraging skewed distribution of web usage where only a small part of web sites get most hits. The optimized scheme, called exponential TBF, consists of a series of basic TBFs as components, each responsible for a small group of bits in a large counter. Since only a few "large" items will be added in the component TBFs that take care of higher significant bits of large counters, it is quite efficient.

2 Problem Definition

Following [6], we formalize our problem as follows. Given a stream S consists of a sequence of N item occurrences with time-stamps:

$$S = \{(e_1, t_1), (e_2, t_2), \cdots, (e_N, t_N)\}$$

Each item occurrence e_i is drawn from the universe U, i.e., $\forall i, e_i \in U$. Arbitrary repetition of item occurrences in streams is allowed. In the following, we use q, or e to denote an item in general. Let f_e be the frequency count of item e in S up to the current time, weighted by recency of occurrence in an exponentially decaying fashion. Mathematically,

$$f_e = \sum_{\forall t_i (e,t_i) \in S} \lambda^{\lceil \frac{t_{now} - t_i}{T} \rceil} \tag{1}$$

where t_{now} denotes the current time, and λ and T are user-supplied parameters. The parameter $\lambda \in [0, 1]$, called *exponential decaying factor* or simply *decaying factor*, controls the speed of exponential decaying. The parameter $T > 0$ controls the frequency with which the exponential decaying takes place. In other words, it controls the granularity of time-sensitivity. A time period of T time units is referred to as an *epoch*.

Our objective is to supply, at any time instance, an estimate $\hat{f}_e$ of f_e for item e occurring in S, such that

1. $\hat{f}_e \geq f_e$, i.e., $\hat{f}_e$ never undercounts the occurrences of item e
2. Error of the estimate $\hat{f}_e$ is bounded to an allowable level.
3. $O(1)$ time complexity for adding an item into the data structure, and/or answering a query of frequency count for an item.

3 Time-Decaying Bloom Filters

In this section, we describe the basic form of Time-decaying Bloom Filters (TBF). A TBF is a counting Bloom filter (Bloom filter with the bitmap replaced by an array of counters) whose counters are decayed with time elapsing. Given a set of independent hash functions, $h_1, \cdots, h_k$, from objects to $[1..m]$; an array of counters, $C_1, \cdots, C_m$. Let $g(x)$ be an non-increasing real-valued *decay function* [7]. Initially, all counters are reset to 0. When an item, say q, occurs, increment each of the counters $C_{h_1(q)}, \cdots, C_{h_k(q)}$. When T time units elapsed, decay all counters by applying $g(x)$ to them.

Input: Data stream $S = \{< o_1, t_1 >, < o_2, t_2 >, \cdots\}$, where $o_i \in U$,
t_i is the time instance while item o_i occurs
Data: An array of m counters: $< C_1, C_2, \cdots, C_m >$
Output: Estimate of the frequency count for each item $u \in U$
Create:
Iinitalize
for $i \leftarrow 1$ **to** m **do** $C_i \leftarrow 0$;
Increment (on arrival of a new item **q***):*
for $j \leftarrow 1$ **to** k **do** $C_{h_j(q)} += 1$;
Decay (on start of a new epoch):
for $i \leftarrow 1$ **to** m **do** $C_i = \lambda \cdot C_i$;
Query(q)**:**
$\hat{C}(q) \leftarrow (1 - \lambda) \times \mathbf{min}\{C_{h_1(q)}, \ldots C_{h_k(q)}\}$
return $\hat{C}(q)$

Fig. 1: Algorithm *Basic TBF*: Basic Time-Decaying Bloom Filter

Dependent on the form of $g(x)$, the last operation can be complicated. In [7], there is given a abundance of decay functions, e.g., exponential decay, sliding window decay, polynomial decay, polyexponential decay and chordal and polygonal decay. In this paper, we focus on exponential decay, the most commonly used decay function, which constituted the only natural alternative for which simple and efficient algorithms were known [7, 8].

Given x_t, the net value obtained at time instance t after a period of T, the exponentially decayed count value can be recursively defined as

$$y_t = (1 - \lambda)x_t + \lambda \cdot y_{t-1}, \quad \lambda \in [0, 1] \tag{2}$$

λ is called an *exponentailly decaying factor* or simply *decaying factor*. Let $y'_t = y_t/(1 - \lambda)$, we have the following form.

$$y'_t = x_t + \lambda \cdot y'_{t-1} \tag{3}$$

Using the above equation, we can increment a counter directly after the decay function was applied, i.e., $\lambda \cdot y'_{t-1}$. Because y'_t is available, the result of y_t can be obtained by the following equation.

$$y_t = (1 - \lambda) \cdot y'_t \tag{4}$$

3.1 Problems

There are a few problems with the basic TBFs, particularly in the web usage profiling context. First, *it is not economical to use uniform-sized single counter for all items.* The usage of web is well known to be quite biased, with a small fraction of popular sites getting very high hits, while the rest and also the majority are rarely used. As the values of counters vary significantly from one or two hits for the "cold" many to millions and thousands for the "hot" few, it is not suitable to allocate the bits to count each of these items. Our experiments verify this, finding that above 85% of web sites share less than 10% visits from clients, while about 10% of popular sites receive up to 90% accesses [1]. The hit frequencies vary from 1 to 50,000 during a one-month period.

Second,*it is costly to update all counters simultaneously by applying the decay function whenever a new epoch starts.* There are the following difficulties when carrying out the operation.

4 Exploiting Skewed Distrbution of Web Usage

Due to the strong skew of web usage, there are only a few counters with high values and most counters only hold relatively small values. To leverage this feature, we optimize the basic TBF by *extensible representation of large counters.* There are standard Bloom Filters to enable quick membership query. The basic TBF with small counters are used to hold frequency counts for most small items. For those large items, we provide extra counters together with a *lookup table.* When, the small counters in the TBF_1 get overflow, we allocate a new counter from the *free counter pool* for the carried digits. The lookup table keeps the bi-directional links between the overflowed counters and the extra counters. The extra counter can also be allocated additional extra counters to continuously receive overflowed digits. In this way, a counter in TBF_1 can extend to a linked counter list to represent much larger values.

Note that we use a standard Bloom Filter for quickly testing membership of each item in stead of directly looking up the TBF_1 and the associated lookup table, as shown in Figure 4. This is necessary because even all counters in TBF_1 corresponding to an item q are all zero's, q also be recorded in the filter. In this case, it has to follow several links in lookup table to decide whether all higher bits of a counter are zero. This would be costly and should be avoided. The standard Bloom Filter is used for this purpose.

To insert a new item q into the optimized TBF filter, we simply insert the item into TBF_1, incrementing each of its counters $h_{1,i}(i = 1, \cdots, k)$ by 1 unless the counter becomes overflow. If ALL these counters get overflow, and if there is an extra counter for this TBF_1 counter, increment that counter by 1; Otherwise

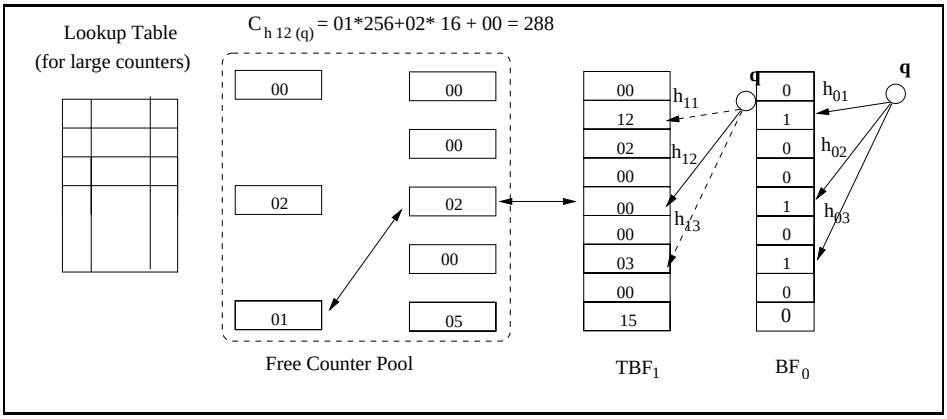

Fig. 2: *Optimized TBF*: Time-Decaying Bloom Filters allow dynamically allocating extra counters from the free counter pool represent much larger values (suppose all counters are of 4 bit length)

allocate an extra counters for the TBF_1 counter. Repeat the process if a extra counter gets overflow too.

To query for the frequency count of an item, say, q, we first check if q is recorded in BF_0. If not, return 0; Otherwise, we check the TBF_1 and the lookup table to constructing the whole counter for q from higher significant bits to lower ones. In Figure 4, q is recorded in the first classic BF, and there are 2 extra counters, holding 01 and 02 respectively. Assuming all are 4-bit counters, the frequency count for q by the TBF_1 counter $C_{h_{12}(q)}$ is 256*1+2*16 + 0 = 288.

The exponential decaying operations take place whenever a epoch starts. We have to update each of TBF_1 counters and the associated extra counters. First, collecting all bits of the whole counter (bits of TBF_1 counters themselves as well as bits in the extra counters), say the "new" counter is C_i. Then compute $\lambda \cdot C_i$. Then, allocate lower 4 bits to the same counter in TBF_1. The left may be written back to one or more extra counters, depending on the decaying results. If no value to write back, the extra counters will be returned to the free counters list.

4.1 Analysis

According to Section 2 the data stream S has N item occurences. Let n be the number of distinct items in S. Then, a standard Bloom Filter with a bitmap of m bits, k independent hash functions, has a error rate

$$E_r = (1 - e^{-kn/m})^k, \tag{5}$$

which is minimized when $k = \ln 2 \cdot (\frac{m}{n})$.

CLAIM 1 *For any item $e \in S$, $\hat{f}_e \leq f_e$. That is, the time-decayed frequency counts estimated by TBF are no more than the real time-decayed frequency counts.*

Proof: Since each counter was set at least by one item, the at any time instance, the accumulated count values are no less than real frequency counts. TBF and Equation 1 are decayed at the same speed and at the same time instance (same T), thus $\hat{f}_e \geq f_e$.

Input: Data stream $S = \{< o_1, t_1 >, < o_2, t_2 >, \cdots\}$, where $o_i \in U$,
t_i is the time instance while item o_i occurs
Data: A standard Bloom Filter, $\text{BF}_0(m, k, n)$
A basic TBF, TBF_1 with k hash functions, m small counters
A free counter pool with $m/2$ free counters of the same size as TBF_1
A lookup table LT for extra counters
Output: Estimate of the frequency count for each item $u \in U$
Create:
Iinitalize
for $i \leftarrow 1$ **to** m **do** $C_i \leftarrow 0$; Reset lookup table LT
Increment (on arrival of a new item **q***):*
for $j \leftarrow 1$ **to** k **do**
  if $C_{h_j(q)}$ *overflow* **then**
    Request a new counter $C_{new}$from free counter pool
    Reqgister C_{new} and the associated $C_{h_j(q)}$ in LT
    $C_{new} += 1$
  end
  $C_{h_j(q)} += 1$
end
Decay (on start of a new epoch):
for $i \leftarrow 1$ **to** m **do**
  Assemble all linked counters in to C_i $C_i = \lambda \cdot C_i$
  Project C_i into sub–counters, release higher significant 0-valued counters (return to the free counter pool)
end
Query(q)**:**
$\hat{C}(q) \leftarrow (1 - \lambda) \times \mathbf{min}\{C_{h_1(q)}, \ldots C_{h_k(q)}\}$
return $\hat{C}(q)$

Fig. 3: Algorithm *Optimized TBF*: Time-Decaying Bloom Filter with extensible counters

5 Experiment Evalution

We did experiment against both synthetic data and real we proxy log data. Figure 5 shows the data used in the experiments. The x-axes of the plots are number of items; the y-axes are occurrence frequencies for the related items. The real data set(left) exhibits strong bias, further analysis verified that it follows a Zipfian distribution. The synthetic data is generated randomly. Distribution of the frequency counts fairly plat.

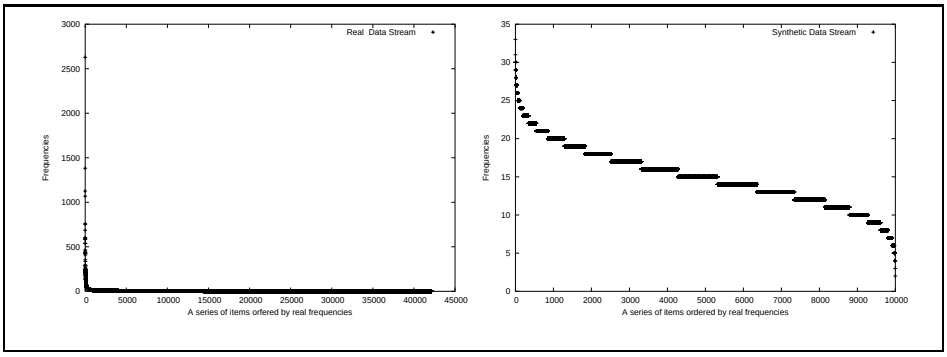

Fig. 4: Web usage data: real data set shows strong skew, while the synthetic data set is average

We tested time and space complexity for basic TBF with 16 bit counters, and optimized TBF with 4-bit counters. The parameters for both basic TBF and the TBF_1 in the optimized TBF are the same. The results show that optimized TBF outperforms basic in space in a factor of 3. The creation time of basic TBF is 10% faster than optimized TBF, but the query time seems to be the same level.

6 Concluding Remarks

In this paper, we have presented succinct data structures and algorithms for efficient summarizing the hit frequency profiles. We focused on two aspects. First, as the recent accesses of a web page is more important than past hits, we have to deal with time-sensitivity when maintaining usage profiles for it. Second, web usage exhibit strong skew with a small fraction of web sites accounting for most hits. So we do not need allocate larger counters for all items to be counted. Instead, we allocate small counters for the first round, then for those necessary to extend, extra counters can be allocated and linked to the origin ones. We develop a novel data structure, called Time-decaying Bloom Filters, TBF, to deal with the above issues. TBFs are built on Bloom Filters and its variants. A commonly-used time–decaying scheme, exponential decay [7] is used.

References

1. Kambayashi, Y., Cheng, K.: Capacity bound-free web warehouse. In Stonebraker, M., Gray, J., Dewitt, D., eds.: Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR 2003. (2003) 47–57
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13** (1970) 422–426
3. Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.D.: Computing iceberg queries efficiently. In: Proceedings of the Twenty-fourth International Conference on Very Large Databases. (1998) 299–310

4. Fan, L., Cao, P., Almeida, J., Broder, A.Z.: Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. In: Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 1998. (1998) 254–265
5. Cohen, S., Matias, Y.: Spectral Bloom Filters. In: Proceedings of the 2003 ACM SIGMOD international conference on Management of data. (2003) 241–252
6. Manjhi, A., Shkapenyuk, V., Dhamdhere, K., Olston, C.: Finding (recently) frequent items in distributed data streams. In: ICDE 2005. (2005) to appear.
7. Cohen, E., Strauss, M.: Maintaining time-decaying stream aggregates. In: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. (2003) 223–233
8. Cohen, E., Kaplan, H.: Spatially-decaying aggregation over a network: Model and algorithms. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. (2004) ???–???
9. Cheng, K., Kambayashi, Y.: A semantic model for hypertext data caching. In: Proceedings of the 21st International Conference on Conceptual Modeling , ER 2002. (2002) 276–290
10. Fischer, M., Salzberg, S.: Finding a majority among n votes: Solution to problem 81-5. Journal of Algorithms **3** (1982) 376–379
11. Gibbons, P., Matias, Y.: New sampling-based summary statistics for improving approximate query answers. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, SIGMOD 1998. (1998) 331–342
12. Gibbons, P., Matias, Y.: Synopsis structures for massive data sets. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, A (1999)