On the Impact of Storing Query Frequency History for Search Engine Result Caching

ABSTRACT

Past frequency statistics of queries is among the key signals to decide on the content of the search engine result caches. In this paper, we first investigate the impact of the sample size for frequency statistics for most popular cache eviction strategies in the literature, and show that cache performance improves with larger samples, i.e., by storing the frequencies of all (or, most of) the queries seen by the search engine. Secondly, we show that a recently proposed approach based on Counting Bloom Filters serve well to store the query frequency statistics in a compact a manner, while still providing comparable cache performance to keeping all frequencies in a raw manner.

1 INTRODUCTION

Large-scale search engines extensively employ caching to store pre-fetched and/or pre-computed data items, such as query result pages, postings lists and their intersections, documents, etc., in the main memory. Research on caching addresses various questions each yielding alternative policies, such as which items should be accepted into the cache (admission), which item should be removed from the cache when it is full (eviction), whether an item should be cached even before requested (prefetching) and when the data item in cache should be re-fetched or -computed (refreshing) [barla, fabrizio].

As in many other domains, frequency and recency of past data items (or, requests) are again strong signals to decide on the items to be evicted from the query result caches of search engines. While these two signals are used on their own in the well-known eviction policies Least Frequently Used (LFU) and Least Recently Used (LRU), respectively; they are also combined for tailoring more effective policies, as in the GDSF policy.

In this paper, our contributions are two-fold: First, we investigate the impact of storing a large query frequency history versus just keeping the frequency of the queries that are in the cache. We show that keeping the entire history (i.e., frequency of all seen queries by the search engine) may improve the cache performance (i.e., hit ratios) for various policies that employ frequency as a signal for eviction. Secondly, we adopt a recently proposed storage scheme for exactly this purpose, i.e., storing past request frequencies in a compact manner for caching, to our application domain. The latter scheme, referred to as Tiny here, stores the query frequency history in a more compact way by using both Simple and Counting Bloom Filters []. Our experiments reveal that the storage space for query history can be significantly reduced while cache performance still remains comparable to storing the entire query history.

The savings from memory space usage are important and meaningful in practice because: 1) As the number of queries submitted to search engines has reached to very large numbers (e.g., Google receives XX queries per day), storing an entire history (or, a truncated version restricted for a month or even a week) may have very demanding memory storage requirements. 2) As recent works have shown, it is possible store query results in alternative formats, such as the top-k results' URLs and snippets vs. just document identifiers [], and in the latter case, much larger number of query results can be cached in the same amount of memory space, making even a small saving in the memory space valuable. Therefore, we believe that our findings showing the importance of keeping the full query history and its achievability using compact storage schemes is a valuable contribution. Note that, while our current evaluation only considers traditional result caches (storing results' with URLs, snippets, etc.), we believe that Tiny storage scheme would also be a perfect fit to be used with hybrid result caches (that also has a segment storing some results as documents IDs), and we leave evaluating such an architecture as a future work.

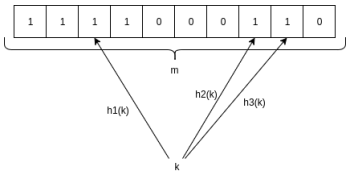


Figure 1: A simple Bloom Filter with m bits and 3 hash functions where k is the key being hashed.

In the following section, we first review the Counting Bloom Filters and then present the Tiny storage scheme as proposed in [**?**] to be used for storing the query frequency history. In Section 3, we describe our simulation framework and in Section 4, we present our results using various well-known cache eviction policies. We review earleir works in Section 5 and provide concluding discussions in Section 6.

2 COUNTING BLOOM FILTERS (CBF) FOR STORING PAST QUERY FREQUENCIES

Bloom filters allow to efficiently query the existence of an item a_i in a set $A = \{a_1, a_2 \cdots a_n\}$ of n elements. As shown in Figure 1, a Bloom filter allocates a vector of m bits that are initially set to 0. When an item a_i from set A is inserted, k distinct hash functions are applied to a_i to obtain the values $h_1(a_i), h_2(a_i), \cdots, h_k(a_i)$ each within the range $[0, m]$, and bits at these positions are set to 1. Consequently, when an item a_i is queried to check whether it is in the set A or not, bits at positions $h_1(a_i), h_2(a_i), \cdots, h_k(a_i)$ are read, and if any of them is set to 0 then a_i has not seen before for sure. Otherwise, we conclude that the item should be in the set, although there is a probability of being wrong. Fortunately, Fan et al. [**?**] show that the probability of obtaining a false positive is low; for instance, it is only 0.9% if m is an order of magnitude larger than n and five hash functions are employed.

In order to use Bloom filters to keep track of the counts, say, to represent the frequency history of a query stream, it is adequate

to allocate a vector of m counters rather than that of m bits. In their work [**?**], Einziger et al. employ a Minimal Increment CBF to store the approximate frequency values of previous queries. The Estimate operation is used to obtain the approximate frequency of a given query, which is the minimum count among the values at the indexes computed by k different hash functions. Similarly, the Add operation computes k different hash values for a given query and increments only the minimal counters at corresponding positions.

- *Pure:* In this approach, the query string and its frequency is plainly stored for each query seen in the stream, regardless of the queries are cached, or not.
- *Sliding Window (SW):* As the Pure strategy may waste a large storage space, earlier works (e.g., [**?**]) proposed to only store the frequency of last W queries. A further improvement over this approach is rather than keeping exact frequencies, again storing the approximate values using sketches that are similar to CBFs [citation?]. In [**?**], a baseline method, SW, is designed based on the latter work, as follows.
  SW approach keeps l identical counting sketches, so-called segments, in a FIFO queue. At each request, i.e., query, only the sketch at the front is manipulated until $\frac{W}{l}$ times of queries are observed. Afterwards, the front sketch is inserted at the tail of the queue and a new sketch is fetched and all of its counters are set to 0. By doing so, the frequency of the least recently accessed $\frac{W}{l}$ items are forgotten. Note that, to estimate the frequency a query, l distinct sketches should be accessed to sum their frequency estimations.
  In terms of the storage requirements, each counter at a segment can be capped to $\log \frac{W}{l}$ since it is the maximum number of times that a query can occur. Thus, the total required number of bits for the *Sliding Window* approach is as in Equation 1 where d is the number of counters per segment;

$$n \times l \times log_2 \left(\frac{W}{l} \right) \tag{1}$$

- *Tiny:* Different from the previous approach, *Tiny* contains a single CBF and a simple bloom filter that is called the *doorkeeper*. The latter is intended to reduce the size of the counters in the CBF: As query streams are known to include a large percentage (i.e., up to 44% []) of singleton queries that appear only once, when a query arrives, first the doorkeeper is checked to see whether it has been seen before, and only for such queries the counters in CBF are increased. Thus, the doorkeeper brings two benefits: Firstly, it slows down the saturation of the CBF as there would be less collisions. Secondly, since singleton queries are not represented via full counters of CBF, the number and/or size of such counters can be smaller.
  Sharing the same motivation with the previous sliding window approach, Tiny employs a mechanism to keep the frequency values fresh; i.e., to reset the counters at certain time points. To this end, it keeps a window counter, starting from 0, that is incremented after each query. Whenever the window counter is equal to W, all of the counters in the CBF as well as the window counter are divided by 2 and the bits in the *doorkeeper* are all set to 0. Notice that this operation can be implemented efficiently by right shifting the bits of the counters by 1. More details and the justification of this reset method are presented in [**?**].
  Obviously, the most important promise of Tiny approach is its storage optimization. Einziger et al. [**?**] argue that for a cache of size C, an entry should reside in the cache if it has a larger frequency than $1/C$. Therefore, for a window size of W queries, each counter can be capped to $log_2 \frac{W}{C}$ bits. Although we consider this choice a greatly simplifying heuristic, experiments in their work as well as ours reported in Section 6 show that it works good in practice.
  Note that, since the doorkeeper can count up to 1, full counters are only necessary to count up to $\frac{W}{C} - 1$. Therefore, when estimating a frequency, estimation from the doorkeeper and CBF are summed. Assuming that a doorkeeper has d bits, and the number of full counters in CBF is n (typically, $n < d$), the total number of bits required to allocate for *Tiny* is given in the following Equation 2:

$$d + n \times log_2 \left(\frac{W}{C} - 1 \right) \tag{2}$$

Table 1: Table of Parameters

Parameter	Value
Average size of a query ($AvgQ$)	16.5 bytes
Size of a pointer (P)	4 bytes
Size of an 32-bit integer (I)	4 bytes
Number of distinct queries (U)	$\approx$ 6.7 millions

3 EXPERIMENTAL SETUP

3.1 Query log and simulation parameters

As the query stream, we use the queries from the AOL query log [Pass et al. 2006] which consists of $\approx$ 17 million queries. A query on average is 16.5 characters. We separate the full log into two splits and use the first one as a training set which contains $\approx$ 10 million queries. The rest of the query log is used as a test set in which we use %10 percent of it to warm-up our cache and we evaluate the hit-rate with the rest of the test set.

The training set is used only to keep and accumulate query frequencies and does not effect the state of the cache. For the experiments performed with LRU eviction policy; we simply discard the training set since we do not need to keep any history for LRU and the state of the cache only depends on the last accessed C queries where C is the size of the cache. Table 1 summarizes our experimental setup parameters.

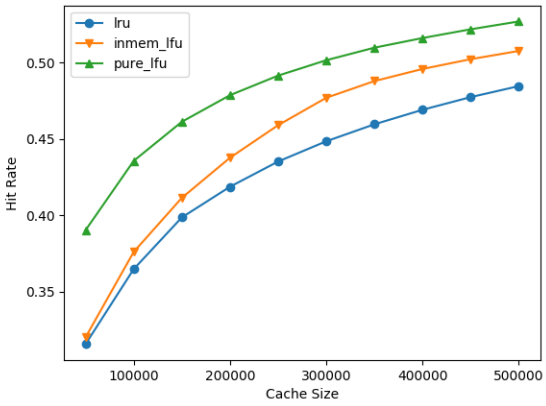
3.2 Results

In the following subsections we first briefly describe single signal caching algorithms and the effect of keeping full query frequency history on the hit-rate. These algorithms are *single signal* in the sense that they are driven either by the frequency or the recency of the queries. Following that, we elaborate on the additional space requirements of keeping frequency history and show the benefits of keeping them compactly.

Table 2: Memory requirement of single signal caching algorithms in bytes where C is the size of the cache.

Algorithm	Memory Cost	Formula (bytes)
LRU	M_{lru}	$C \times (AvgQ + 3P)$
In-memory LFU	M_{lfu}^{inmem}	$C \times (AvgQ + 6P + I)$
Pure LFU	M_{lfu}^{pure}	$M_{lfu}^{inmem} + U \times (AvgQ + I)$

Figure 2: Hit-ratios of single signal caching algorithms



Conversely, caching algorithms that do take both frequency and receny into account are called *multi signal* and not surprisingly, they yield better hit-ratios compared to single ones. We show that these algorithms can also benefit from the usage of frequency history with an accountable space overhead.

3.3 Single Signal Caching

Below we describe two well known caching algorithms and their implementations in our setup. Table 2 shows the worst-case space requirements according to our implementations.

- *Least Recently Used (LRU)* always evicts the least recently accessed item if the cache is full. A doubly-linked list can be used to keep the access order of queries and a hash-table that maps the query strings into pointers to linked-list nodes is sufficient enough to apply any operation at $O(1)$ time.
- *Least Frequently Used (LFU)* always evicts the least frequent item if the cache is full. The variant that does not keep any history and accumulates the frequency of items for only the cached ones is called the *In-memory LFU*. In contrast to the LRU, a doubly-linked list with special frequency nodes is required [Shah et al.] for an efficient implementation. At worst-case, number of frequency nodes are as many as the cached items i.e. each frequency node contains a single cached item. Other LFU variant that keeps the full frequency history of queries without any approximation is called the Pure LFU. It requires an additional hash-table that maps query strings into frequency values.

Figure 1 shows hit-ratio's of single signal caching algorithms with respect to the cache size. Pure LFU marginally improves the hit-ratio compared to Inmemory LFU and LRU, especially for small cache sizes. Since the counters of the Inmemory LFU reside in the frequency nodes, number of counters increase proportional to the cache size. Hence, the improvement of Inmemory LFU over LRU becomes more significant as the cache size gets larger.

Some of the earlier work [gan and suel] shown opposite results to ours in which the LRU offers better hit-ratios than the Inmemory LFU on the same dataset. We've observed that Inmemory LFU is fragile to the changes in frequency distribution over long query streams. Since our cache warm-up (around 700k) is much shorter relative to [gan and suel] (around 5m), Inmemory LFU performs better. When we increased our cache warm-up to also include the training set (around 10m) the hit-ratio dropped significantly and we achieved similar results to [gan and suel].

3.4 Multi-Signal Caching

- *Greedy Dual Size Frequency-K (GDSF-K):* Greedy Dual Size Frequency is proposed by [**?**] that can offer a good compromise between recency and frequency that is also cost aware. However, for our concerns the cost awareness is ignored. Victim suggested is the item that has the lowest $\mathcal{H}_{value}$ which is given in **??**. $\mathcal{F}_i$ is the frequency of the item, C_i and $\mathcal{S}_i$ are cost and size of page of item respectively (equal to 1). Lastly, $\mathcal{L}$ is the aging factor that is updated to the $\mathcal{H}_{value}$ of the cache victim whenever it is evicted. This way, newly admitted items will obtain a higher $\mathcal{H}_{value}$. Therefore, items that are less recently accessed has a higher chance of being the victim.

4 RESULTS

In Table 1, we report the performance for incorporating the frequency-based admission policy of TinyLFU into a cache using LRU as the eviction policy. Our experiments show that admission policy with the frequency history of entire query stream (i.e., Pure) yields the highest improvement in hit ratio for small caches of up to C=10%. When usign the approximate frequencies (provided by Sliding Window or Tiny) with the admission policy, the hit ratio still improves with respect to the None case, but not as high as in the case of Pure. However, the space usage for the Pure case is very high, 11,264 KB, regardless of the cache size. In contrary, Sliding Window and Tiny achieves gains by using very small storage space; the largest values in Table 1 being 2,639 KB and 495 KB, respectively. These findings show that Tiny yields hit-ratios as high as Slding Window and close to Pure, but employs a significantly smaller storage space.

In Table 2, we report the results when the cache eviction policy is LFU. As cache admission and eviction polices match better, the performance gains of using them together is even larger. Again, Tiny outperforms Sliding Window for all cache sizes with space usage of only about 20% of the former. Note that, in our experiments, we find that LFU usually outperforms LRU in teh corresponding cases, while some earlier works (e.g., []) have shown the reverse again usign the AOL query log. We believe this difference is based on our LFU implementation, as we guarantee that LFU (with and

Table 3: Comparison of single signal caching algorithms where C is the cache size, $\mathcal{M}$ is the memory size in kilobytes, $\mathcal{R}$ is the ratio of memory size to result cache, $\mathcal{H}$ is the hit-ratio, $\mathcal{W}$ is the window size for Tiny in millions, and $\mathcal{B}_w$ is the bit-width of a single full counter for Tiny.

C	LRU		Inmemory LFU		Pure LFU			Tiny LFU				
	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{H}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{H}$	$\mathcal{M}$	$\mathcal{R}$	$\mathcal{W}$	$\mathcal{B}_w$
$100k$	2784		4346		0.436	138976		0.437	37305		10	7
$200k$	5567		8692		0.479	143332		0.479	37989		10	6
$300k$	8350		13038		0.502	147668		0.501	59913		16	6
$400k$	11133		17383		0.516	152013		0.516	64258		16	6
$500k$	13917		21729		0.527	156359		0.526	62745		16	5

w/o admission policy) chooses the least recent cache entry among those with the smallest frequency value.

Finally, Table 3 shows similar result when the eviction policy is GDSF-k (where k is set to 4, experimentally). Note that, this eviction policy both takes into account the frequency and recency; and yields the highest hit ratios even without an admission policy. Therefore, the frequency-based admission policy can still improve it, but less signifcantly. In particular, with Pure frequency storage, the gains in hit ratio still range from 1.9% to 6.6%, while approximate storage approaches seem to be effective only for large cache sizes. Again, Tiny is better than its competitor Sliding Window, and may yield relative improvements of 1.8% and 1.7% for cache sizes set to 10% and 20%, respectively.

5 RELATED WORK

6 CONCLUSION

We revisited the frequency-based admission for search engine result caches and demonstrated that a recently proposed approach, so-called TinyLFU, improves hit ratios for caches with various eviction strategies. Furthermore, TinyLFU acheieves such gains using a very small storage space in comparison to storing the frequency history of the entire stream or of those within a (sliding) window. The latter is an important finding, because the current trend in query result caching is storing not only full query results but also docIDs for some subset of queries [], and for the latter case, even relatively small gains in storage space would allow caching more results, and further increasing the performance. Our work towards designing such hybrid caches with a frequency-based admission policy is underway.