Analysis of Efficient Cache Admission Policies on Web Search

Erman Yafay
Middle East Technical University
Ankara
erman.yafay@metu.edu.tr

Ismail Sengor Altingovde
Middle East Technical University
Ankara
altingovde@ceng.metu.edu.tr

ABSTRACT

Have the abstract here...

1 INTRODUCTION

Caching tecniques are solutions that generally considered first at hand to boost performance of applications, especially on web. It is the process of keeping the items that are more likely to appear in the future at a faster or a closer store so that these items can be served within shorter times. Hence, accomodating a cache for a system can marginally increase the average system throughput of response with reasonably small costs. Concretely, as the number of items served from the cache increases, the system performance will also increase.

Whenever an item is served from the cache, we call it a cache *hit* and the opposite is called a cache *miss*. Consequently, a common metric that is used to evaluate cache performance is the *hit-rate*. It is the ratio of the cache hits and total number of items served from the system. Therefore, caching algorithms yielding a higher hit-rate is preferable over others. If a caching technique can predict future items and admit them into the cache, it is expect to obtain higher hit-rates.

$$\text{Hit Rate} = \frac{hit}{hit + miss}$$

Various caching methods predict future items by using statistics from the past access patterns. Such well known ones are *Least Recently Used LRU* and *Least Frequently Used LFU* cache eviction policies. As their name implies, if the storage for the cache is full, at each access request for an item, the former evicts the least recently accessed item and admits the recently requested one, whereas the latter evicts the least frequent item. In chapter 2 we give a detailed information of cache eviction and admission policies. It is an important caveat to acknowledge for cache eviction or admission policies that the overhead of making decisions about which item to evict or admit should not overcome the benefit of caching. In other words, using a cache should not be more expensive than using none at all in terms of time complexity.

Cache admission or eviction policies usually require additional data structures to be used in order to keep statistical data about past access patterns. The storage used by these data structures are called the meta-data of the cache. As in time complexity, the storage cost of meta-data of the cache should not surpass the cost of the actual cache. Otherwise, the most amount of space is kept for the meta-data rather than the cache itself. Hence, large meta-data cost can enforce the cache to only store a few items. Moreover, it is important to note that cache size C has the largest impact on hit-rate. Therefore, it is crucial to keep meta-data storage cost to a minimum so that we can have a larger C.

An example of a cache meta-data can be a frequency histogram. *LFU* eviction policy requires to have the knowledge of frequency of the items in order to be able to decide on which item to evict. If the frequencies are kept for all items throughout the whole data access stream then it is called the *Pure LFU (P-LFU)*. However, achieving a such frequency histogram for all items brings a blast on the meta-data size. Therefore, *P-LFU* is not applicable for majority of applications even tough it is shown that it yields the optimal hit-rate for large enough caches [3]. On the contrary, *LRU* does not require any additional meta-data cost, since the cache only needs to know about the order of accesses of items in the cache which can be efficiently implemented using a linked list. However, *LRU* only yields competent hit-rates if the cache is large, as also what we've obtained from our results experimentally.

Karakostas et al. [9] have offered the usage of *Window-LFU (W-LFU)* where they've shown that for Zipf or Zipf-like distributions, keeping the frequency statistics of last accessed $|W|$ items can approximate to *P-LFU*. Therefore, the meta-data storage cost can be reduced marginally compared to *P-LFU*. In Zipf distribution, few items are accessed very frequently, whereas most of the items are accessed a couple of times or even singletons. Zipf or Zipf-like distributions are common in various domains of computer science as well as web search. Formally, the probability of observing an item with rank i is given as with parameter α;

$$Pr\{i\} = \frac{1}{\mathrm{H}_{N,\alpha} i^{\alpha}}$$

where $\mathrm{H}_{N,\alpha}$ is the Nth harmonic number.

Moreover, *W-LFU* better adapts to changes in the access pattern by forgetting old events. However, implementing *W-LFU* can still be costly, since the order of the last $|W|$ accesses have to be known so that when $W + 1$th access occurs the least recent access can be forgotten (decrementing its frequency).

In order to further reduce the meta-data cost of *W-LFU*, Eingizer et al. [7] offers a highly efficient cache admission policy *Tiny LFU* that yields the best performance when augmented with an *LFU* eviction policy. *Tiny LFU* accurately approximates to *W-LFU* by exploiting the fact that data access stream has the properties of Zipf-like distribution and the frequency counting can be effectively approximated with the help of *Bloom Filter*[2] theory and approximation sketches. *Tiny LFU* admission policy achieves a good compromise between changes in distribution and as an approximation of a frequency histogram. However, the proposed method of augmenting *Tiny LFU* with an *LFU* eviction policy may stutter in terms of hit-rate when distribution changes happen frequently. This is due to *LFU* eviction policy cannot adapt to changes in distribution well enough.

This paper contributes the following; we offer an explanation of cache admission and eviction policies and give a general notion algorithm. We offer an implementation of *Tiny LFU* and augment it with various cache eviction policies as well as compare it to *Sliding Window*[6] approach. Lastly we show that *Tiny LFU* can perform better when augmented with an adaptation of *Greedy Dual Size Frequency* cache eviction policy.

Section 2 describes the distinction between admission and eviction policies and gives an algorithm to generalize their usage. Section 3 briefly introduces *Bloom Filters* and approximation sketches. *Tiny LFU* and *Sliding window* are discussed and compared in Section 4. Our experimental setup and *Tiny LFU* customization is explained in Section 5 as well as approximate storage sizes of cache admission and eviction policies under discussion. Section 6 presents our results on AOL query log dataset and finally Section 7 concludes this paper.

2 CACHE ADMISSION AND EVICTION POLICIES

Algorithm 1 Generalized algorithm for cache eviction policies augmented with admission policies.

```
 1: function REQUEST($item$)
 2:     $popularity \leftarrow$ ADMISSION-ADD($item$)
 3:     if $item$ is in cache then
 4:         CHOOSE-NEW-VICTIM()
 5:     else
 6:         if Cache is not full then
 7:             Admit $item$ with new $popularity$
 8:             CHOOSE-NEW-VICTIM()
 9:         else
10:             $victim \leftarrow$ GET-VICTIM()
11:             if $popularity \geq$ ADMISSION-ESTIMATE($victim$) then
12:                 Evict $victim$ admit $item$
13:                 CHOOSE-NEW-VICTIM()
14:             else
15:                 Keep $victim$
16:             end if
17:         end if
18:     end if
19: end function
```

In general cache eviction policy chooses a cache victim when a new access for an item occurs (when the cache is full). For instance, this decision can be based on *LRU* or *LFU* as already discussed earlier. When used without an augmented admission policy, the cache victim is always evicted and newly requested item is always admitted. Conversely, the the newly requested item is only admitted to the cache if its more popular than cache victim according to the admission policy. Algorithm 1 presents a generalized method. In line 2 the admission policy gives a popularity value of newly requested item. This is usually an estimated frequency. Notice that on line 11 the ties are broken in favor of the recent item.

2.1 Eviction Policies Used in Experiments

In this section we briefly describe the eviction policies that we've experimented with.

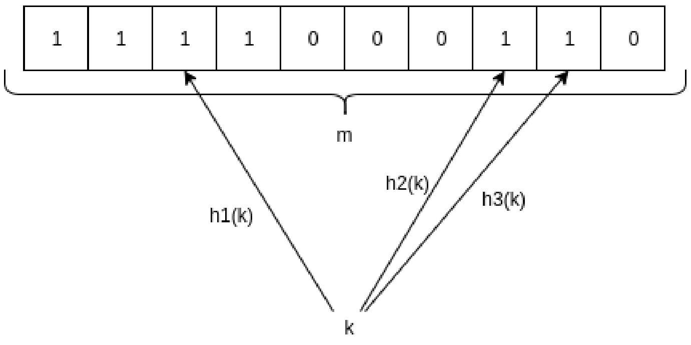

Figure 1: A simple Bloom Filter with m bits and 3 hash functions where k is the key being hashed.

2.1.1 Least Frequently Used. Victim suggested is the item that has the least frequency. To prevent a misconception, notice that the frequencies have to be counted with an admission policy. The eviction policy that counts only the items in the cache itself is called the *In-Memory LFU.*

2.1.2 Least Recently Used. Victim suggested is the item that is accessed least recently.

2.1.3 Greedy Dual Size Frequency.

$$\mathcal{H}_{value}(i) = \mathcal{F}_i^{\mathcal{K}} \times \frac{\mathcal{C}_i}{\mathcal{S}_i} + \mathcal{L} \qquad (1)$$

Greedy Dual Size Frequency is proposed by [1] 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 1. $\mathcal{F}_i$ is the frequency of the item, $\mathcal{C}_i$ 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.

3 PRIMER ON BLOOM FILTERS AND APPROXIMATION SKETCHES

Bloom filters allow to query the existence of an item from a set $A = \{a_1, a_2 \cdots a_n\}$ of n elements in a stream $S = [s_1, s_2, \cdots, s_l]$ of length l where each $s_i \in A$. Bloom filters as in Figure 1 allocate a vector of m bits that are initially set to 0. When an item s_i has been retrieved from the stream S, k distinct hash functions are applied to s_i such that obtaining positions $h_1(s_i), h_2(s_i), \cdots, h_k(s_i)$ each within range $[0, m]$ and bits at that positions are set to 1. Consequently, when an item s_i is queried if it has seen before in the stream, bits at positions $h_1(s_i), h_2(s_i), \cdots, h_k(s_i)$ are read and if any of them are set to 0 then s_i has not seen before for sure. The probability of obtaining a false positive is given approximately as at [8];

$$\left(1 - e^{-kn/m}\right)^k$$

3.1 Approximation Sketches

Approximation sketches slightly differ from the Bloom filters in the sense that they allocate m counters rather than m bits. Therefore, rather than querying the previous occurrence of an item, the frequency of the item can be approximated. There are numerous implementation of sketches such as *Count-min sketch* [5] and *Spectral Bloom Filters* [4] where we focus on *Spectral Bloom Filters* with a simpler counting scheme as *Counting Bloom Filter* [8] and *Minimal Increment* [4] operation as in [7].

3.1.1 Spectral Bloom Filter. Basically an *SBF (Spectral Bloom Filter)* has m counters and k hash functions as in bloom filters. Additionally it has two operations namely *Add* and *Estimate* where the *Add* operation is used when a new item is accessed and its frequency should be incremented.*Estimate* is used to obtain frequency approximation of an item. *Estimate* works as follows, after obtaining k values at indexes by applying k hash functions, only the minimum value is returned. Assume with 4 hash functions we obtain values $\{3, 3, 4, 5\}$ thus the result will be 3. *Add* operation works in a similar way with so called *Minimal Increment* method. Considering the same example only the two values of 3's would be incremented. All the logic behind using the minimum elements is that for other elements it is for sure that a collision happened. Therefore, *SBF* tries to minimize the error by manipulating minimum elements.

3.1.2 Saturation of Approximation Sketches. As already mentioned, in a Zipf distribution most of the items are singletons i.e. occurring a single time throughout the most recent accesses of length W. Due to large number of singleton items, an approximation sketch may result in many hash collisions in a brief time, thus resulting in poor frequency approximation for the rest of the data stream. In such sketches it is called that the sketch is saturated [6]. Therefore, to achieve accurate approximations, admission policies that make use of approximation sketches should deal with the saturation problem, especially on Zipf-like distributions.

4 TINY LFU AND SLIDING WINDOW

4.1 Sliding Window Approach

Sliding Window keeps l identical sketches called segments with a total size of W counters where each sketch has $\frac{W}{l}$ counters. l number of sketches are kept in a FIFO queue. At each item access only the sketch at the front is manipulated until l times of accesses are obtained. Afterwards, the sketch at the front is inserted at the end of the queue and a new sketch is fetched and all of its counters are set to 0. Therefore, frequency about least recently accessed $\frac{W}{l}$ items are forgotten. Notice that this method allows the sketch to avoid early saturation, also a sliding window is obtained with the step size of $\frac{W}{l}$.

In terms of storage, each counter at a segment can be capped to $\log \frac{W}{l}$ since it is the maximum number that an item can occur. However also note that, when doing an estimation for an item, l distinct sketches should estimate and their results should be accumulated. As a result, the estimation time will be costly for large l. Therefore, we can give the total required number of bits of the *Sliding Window* approach as;

$$W \times log_2\left(\frac{W}{l}\right) + log_2 l$$

4.2 Tiny LFU

Tiny LFU contains a single *SBF*. To have the feel of *W-LFU*, *Tiny LFU* has a novel mechanism to keep the sketch fresh i.e. take recency into account called the reset method. *Tiny LFU* keeps a window counter starting from 0 and it is incremented after each access. Whenever the window counter is equal to W, all of the counters in the sketch as well as the window counter are divided by 2. Notice that this operation can be obtained by right shifting bits of the counters by 1. More detail and the justification of the reset method is presented in [7].

Concretely, the most important improvement that *Tiny LFU* brings is its storage optimizations, as its name also refers to. Since, for a cache of size C an item should reside in the cache if it has a larger frequency than $1/C$. Therefore, for a window size of W, each counter can be capped to $log_2 \frac{W}{C}$ bits since it already deserves to be in cache i.e. there is no need to further increment its counter. For instance, if $W/C = 7$ then only 3 bits counters are sufficient.

Another storage optimization of *Tiny LFU* also solves the problem of saturation. Rather than only keeping an *SBF* for all items in a data stream, *Tiny LFU* also contains a Bloom filter which is called the doorkeeper. For an accessed item, *Tiny LFU* first checks its bits in the doorkeeper, and only further increases the full counters if the item exists in the doorkeeper. This way, singleton items are not reserved full counters in the *SBF*. This brings 2 benefits, firstly it slows down or avoids the saturation of the sketch due to the fact that there will be less collisions. Secondly, since the most of the items are singletons, we can reserve less full counters in the *SBF*. Therefore, by adding a W size of bits to doorkeeper, length of the *SBF* where full counters reside can be marginally decreased. Moreover, 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 *SBF* are added and returned. Assuming that a doorkeeper with W bits, and full counters *SBF* with length d such that $d \ll W$, the total number of bits required to allocate for *Tiny LFU* is;

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

5 EXPERIMENTAL SETUP

6 RESULTS

7 CONCLUSION

REFERENCES

[1] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. 2000. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review* 27, 4 (2000), 3–11.

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

[3] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Vol. 1. IEEE, 126–134.

[4] Saar Cohen and Yossi Matias. 2003. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 241–252.

Table 1: For each eviction policy and cache size the best admission policies Hit-Rate is written in bold. For window lfu variants (strawman and tiny) best working window size is shown in the table.

Eviction	**random**			**lru**			**lfu**			**gdsfk**		
Admission	C	HR	M	C	HR	M	C	HR	M	C	HR	M
none	6723	0.169947	131	6723	**0.191673**	183						
	13446	0.2084	262	13446	**0.235173**	367						
	20169	0.232347	393	20169	**0.2602**	551						
	26892	0.24986	505	26892	**0.278367**	735						
	33615	0.266093	656	33615	**0.293233**	919						
	40338	0.280073	787	40338	**0.305033**	1102						
	47061	0.29122	919	47061	**0.314273**	1286						
	53784	0.301453	1050	53784	**0.321947**	1470						
	60507	0.31078	1181	60507	**0.328093**	1654						
	67230	0.31728	1313	67230	**0.333467**	1838						
strawman	6723	0.190513 (16)	354	6723	0.187693 (16)	407	6723	0.200427 (16)	433	6723	**0.200313** (16)	433
	13446	0.216367 (8)	485	13446	0.2173 (8)	590	13446	0.224187 (8)	643	13446	**0.224253** (4)	525
	20169	0.240367 (8)	748	20169	0.24176 (8)	906	20169	**0.24582** (8)	984	20169	0.24583 (8)	984
	26892	0.252547 (4)	748	26892	0.252287 (2)	840	26892	0.25844 (2)	1063	26892	**0.25866** (4)	1063
	33615	0.26694 (4)	951	33615	0.266693 (2)	1058	33615	0.273713 (4)	1345	33615	**0.273713** (4)	1345
	40338	0.280567 (4)	1142	40338	0.276273 (4)	1457	40338	**0.283233** (4)	1615	40338	0.283233 (4)	1615
	47061	0.291813 (4)	1332	47061	0.287313 (4)	1700	47061	0.293653 (4)	1884	47061	**0.293653** (4)	1884
	53784	0.30002 (2)	1273	53784	0.29664 (2)	1693	53784	0.302013 (2)	1903	53784	**0.302013** (2)	1903
	60507	0.308453 (2)	1432	60507	0.300033 (2)	1905	60507	0.308673 (2)	2141	60507	**0.308673** (2)	2141
	67230	0.315227 (2)	1608	67230	0.31246 (2)	2133	67230	**0.316167** (2)	2396	67230	0.316167 (2)	2396
tiny	6723	0.221373 (16)	170	6723	0.20148 (16)	223	6723	**0.222393** (16)	249	6723	0.229373 (16)	249
	13446	0.245493 (8)	295	13446	0.23518 (2)	372	13446	0.248387 (8)	453	13446	**0.25766** (8)	425
	20169	0.265993 (8)	443	20169	0.260047 (2)	558	20169	0.275907 (8)	679	20169	**0.27858** (4)	637
	26892	0.270473 (4)	551	26892	0.278147 (2)	745	26892	0.276967 (4)	866	26892	**0.289247** (4)	850
	33615	0.285753 (4)	689	33615	0.292653 (2)	931	33615	0.292253 (2)	1062	33615	**0.301527** (4)	1062
	40338	0.297587 (4)	827	40338	0.304447 (2)	1117	40338	0.307867 (2)	1275	40338	**0.31112** (4)	1275
	47061	0.30802 (4)	965	47061	0.313427 (2)	1304	47061	0.31812 (2)	1487	47061	**0.31812** (2)	1487
	53784	0.305333 (2)	1070	53784	0.321187 (2)	1490	53784	**0.32506** (2)	1700	53784	0.32506 (2)	1700
	60507	0.31314 (2)	1203	60507	0.32734 (2)	1676	60507	**0.330893** (2)	1913	60507	0.330893 (2)	1913
	67230	0.320467 (2)	1337	67230	0.33244 (2)	1862	67230	**0.33522** (2)	2125	67230	0.33522 (2)	2125
pure	6723	0.2353	3282	6723	0.20608	3335	6723	**0.245173**	3361	6723	0.24424	3361
	13446	0.261127	3414	13446	0.231893	3519	13446	**0.274787**	3571	13446	0.2739	3571
	20169	0.276493	3545	20169	0.25	3702	20169	**0.294327**	3781	20169	0.291067	3781
	26892	0.288467	3676	26892	0.264533	3886	26892	**0.30474**	3991	26892	0.304673	3991
	33615	0.297653	3808	33615	0.272447	4070	33615	**0.312333**	4201	33615	0.312333	4201
	40338	0.306853	3939	40338	0.282847	4254	40338	**0.31914**	4412	40338	0.31914	4412
	47061	0.314273	4070	47061	0.29354	4438	47061	**0.324753**	4622	47061	0.324753	4622
	53784	0.320747	4201	53784	0.298833	4622	53784	**0.32944**	4832	53784	0.32944	4832
	60507	0.326907	4333	60507	0.30684	4805	60507	**0.33366**	5042	60507	0.33366	5042
	67230	0.3316	4464	67230	0.313627	4989	67230	**0.33726**	5252	67230	0.33726	5252

[5] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[6] Xenofontas Dimitropoulos, Marc Stoecklin, Paul Hurley, and Andreas Kind. 2008. The eternal sunshine of the sketch data structure. *Computer Networks* 52, 17 (2008), 3248–3257.

[7] Gil Einziger and Roy Friedman. 2014. Tinylfu: A highly efficient cache admission policy. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on.* IEEE, 146–153.

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

[9] George Karakostas and Dimitrios N Serpanos. 2002. Exploitation of different types of locality for web caches. In *Computers and Communications, 2002. Proceedings. ISCC 2002. Seventh International Symposium on.* IEEE, 207–212.