Latency-Aware Strategy for Static List Caching in Flash-based Web Search Engines

Jiancong Tong
Nankai-Baidu Joint Lab
Nankai University
lingfenghx@gmail.com

Gang Wang
Nankai-Baidu Joint Lab
Nankai University
wgzwp@163.com

Xiaoguang Liu
Nankai-Baidu Joint Lab
Nankai University
liuxg@nbjl.nankai.edu.cn

ABSTRACT

Caching is a widely used technique to boost the performance of search engines. Based on the observation that the speed gap between the random access of flash-based solid state drive and its sequential access is much inapparent than that of magnetic hard disk drive, we introduce a new static list caching algorithm which takes the block-level access latency into consideration. The experimental results show that the proposed policy can reduce the average disk access latency per query by up to 14% over the state-of-the-art algorithms in the SSD-based infrastructure. Besides, the results also reveal that our new strategy outperforms other existing algorithms even on HDD-based architecture.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; H.3.4 [**System and Software**]: Performance evaluation—*efficiency and effectiveness*

General Terms

Algorithm, Performance, Experimentation

Keywords

Search Engines, Static caching, Solid State Drive, Latency

1. INTRODUCTION

Caching technologies have been widely employed by Web search engines, as they can reduce the query latency and result in higher query throughput. In a static list cache, the cache typically keeps the posting lists of the most-frequent query terms that are extracted from the previous query log. It is general to treat the static list caching problem as a classic `KNAPSACK` problem [1]: The fixed size cache memory is viewed as a fixed capacity knapsack, while the data to be filled in the cache memory are regarded as commodities that have their own value and volume. For a give term t, its posting list is denoted as $\ell(t)$. Since the volume (the length of $\ell(t)$, counted in bytes, denoted as $|\ell(t)|$) is fixed, estimating its "value" (the I/O cost that can be saved if it is cached) as precisely as possible is crucial to the performance of a caching algorithm.

In the past, the traditional magnetic hard disk drives (HDD) are used as external storage media. Due to the mechanical nature of HDD, the cost of a random read is about two orders of magnitude larger than that of a sequential read [13]. Thus, the access latency caused by a cache miss is dominated by the expensive random seek operation. Consequently, most of the existing static caching techniques are designed and tuned to achieve higher hit ratio, as the lower cache miss ratio indicates less random reads required.

Recently, solid state drive (SSD) has been largely deployed as secondary storage media in Web search engines [6, 7]. Since the internal structure of SSD involves no mechanical operations, the random read is no longer a determinant of the overall access latency. We repeat the read latency test in [13] with an SSD (OCZ Vertex-3 120GB) and a HDD (Seagate Barracuda 500GB, 7200rpm) and get Figure 1. As indicated in Figure 1, different from the case of HDD, the latency (in block level) of a random read in SSD is now comparable to that of a sequential read. Therefore, in an SSD-based infrastructure, caching algorithms should now aim to save not only random reads but also sequential reads. In other words, with SSD as the external storage media for the posting lists, the existing static list caching algorithms may not perform as good as they do with HDD.

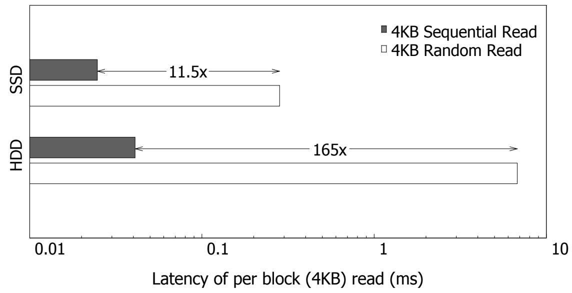

Figure 1: Read access latency on HDD and SSD. Each read fetches a block (4KB). The page cache is by-passed.

In this paper, we introduce a new static list caching strategy that takes the block-level access latency into account. The proposed method aims to reduce the overall cost of both the random reads and sequential reads. The experimental results show that the proposed strategy outperforms other state-of-the-art static caching algorithms. To the best of our knowledge, this is the first study to develop a new caching policy specific to SSD-based search engine infrastructure.

2. BACKGROUND

Static caching has been studied extensively in the literatures [1, 2, 3, 5, 8, 9, 10, 11]. Baeza-Yates and Saint-Jean [3] presented a hierarchical memory organization for inverted lists. In [3], the posting lists of the most frequent terms in the previous query log were stored in main memory, acting as static list caching. We will use the policy, referred as *Query-Term-Frequency (QTF)*, as one of the baselines to compare with. Another static caching policy for posting lists was proposed by Baeza-Yates et al. [1]. They proposed an algorithm which caches the posting lists of the terms with the highest values of the ratio $\frac{f_q(t)}{|\ell(t)|}$, while $f_q(t)$ denotes the frequency of a term t in the previous query log. We refer this algorithm as *QTFDF* and use it as another baseline.

Apart from the work described above, some researchers focused on the cost-based static caching algorithms. Ozcan et al. [11] proposed a cost-aware strategy for static query result caching, while a multi-level static cache architecture for various item types was introduced in [9]. The cost models described in these two work are analogous to ours, except that (i) we are dedicated to posting list caching and (ii) we introduce a new cost function which takes the block-level access latency in SSD into consideration.

Work by Wang et al. [13] was the first to investigate the impact of SSD on cache management of Web search engines. They also discussed the impact of the narrower speed gap (compare to that of HDD) between the random access and sequential access in SSD on the existing caching algorithms, which is very similar to our work. However, they did not come up with better caching policies that are appropriate for SSD-based infrastructure.

3. LATENCY-AWARE METHOD

In this paper, the static list caching problem is treated as a `KNAPSACK` problem and solved by a greedy approach like [1]. That is, for any term t, we calculate the benefit $\mathcal{B}(t)$, i.e., the I/O cost that can be saved during the whole query evaluating process if $\ell(t)$ is kept in the cache. Then we sort the terms according to their $\mathcal{B}(t)$ in descending order and fill as many "high-beneficial" terms as possible into the static cache until it is full. The benefit $\mathcal{B}(t)$ is computed by a benefit function (1), with a cost function $\mathcal{C}(t)$ estimating the access cost that will be saved by a single cache hit. Note that $\frac{f_q(t)}{|\ell(t)|}$ in (1) depicts the average number of cache hits yielded by per cached byte of $\ell(t)$.

$$\mathcal{B}(t) = \mathcal{C}(t) \times \frac{f_q(t)}{|\ell(t)|} \tag{1}$$

$$\mathcal{B}(t) = \begin{cases} \frac{f_q(t)}{|\ell(t)|} & , \mathcal{C}(t) = 1 \quad \Leftrightarrow \quad \text{QTFDF} \\ f_q(t) & , \mathcal{C}(t) = |\ell(t)| \quad \Leftrightarrow \quad \text{QTF} \\ \frac{f_q(t)}{|\ell(t)|} \times \mathcal{T}(t) & , \mathcal{C}(t) = \mathcal{T}(t) \quad \Leftrightarrow \quad \text{MECH} \end{cases} \tag{2}$$

As shown in (2), by adopting different cost functions $\mathcal{C}(t)$, all the existing static list caching policies can be modeled by the benefit function. If the access cost of any $\ell(t)$ is simply considered as its length or just a constant, then we get exactly the QTF or QTFDF policy, respectively. Besides these two cases, the access latency can be estimated by a more elaborated formula $\mathcal{T}(t)$, which considers the working principle of the drives as well. In a HDD-based architecture, due to HDD's mechanical nature, the access latency consists of a seek time D_{seek}, a rotational delay D_{rotate} and also the time D_{read} of reading per block data (whose size is D_{block}). Thus, $\mathcal{T}_{hdd}(t)$ can be computed by (3), while D_{seek}, D_{rotate}, D_{read} and D_{block} has the same value as those in [9, 11]. We refer the strategy that applies $\mathcal{T}(t)$ to estimate the access latency as *MECH* (abbreviated for mechanical) and use it as the third baseline. Note that SSD does not carry out the read operation in a mechanical way but a electronical manner, thus, instead of (3), its $\mathcal{T}(t)$ should be computed by (4), where S_{read} represents the time of fetching data of S_{page} size [14][1]. Nevertheless, we still refer this method as MECH in SSD-based setting. As $\mathcal{T}_{ssd}(t)$ is only proportional to $|\ell(t)|$, the items it keeps in the cache are exactly the same as QTF does. Therefore, MECH and QTF will perform identically in SSD-based setting.

$$\mathcal{T}_{hdd}(t) = D_{seek} + D_{rotate} + D_{read} \times \frac{|\ell(t)|}{D_{block}} \tag{3}$$

$$\mathcal{T}_{ssd}(t) = S_{read} \times \frac{|\ell(t)|}{S_{page}} \tag{4}$$

As revealed by Figure 1, however, in SSD-based infrastructure, not each part of $|\ell(t)|$ contributes equally to the access latency of a positing list $\ell(t)$. That is, even though a read operation in SSD involves no mechanical operations (like seek and rotation in HDD), it still can be viewed as a compound of a relative expensive random read and a few low-cost sequential reads. Thus, we develop a new caching strategy (called *BLOCK*) that takes into account the block-level latency gap between random read and sequential read. The $\mathcal{C}(t)$ is now represented by (5), which measures the latency of accessing $\ell(t)$ in terms of the numer of random reads.

$$\mathcal{C}_{block}(t) = \overbrace{1}^{\text{1 random read}} + \underbrace{(\overbrace{\left\lceil \frac{|\ell(t)|}{B_s} \right\rceil - 1}^{\text{\# of Sequential read}}) \div \gamma}_{\text{\# of equivalent random read}} \tag{5}$$

Here γ is the ratio of the random read latency to the sequential read latency of a certain drive (γ_{ssd}=11.5 and γ_{hdd}=165, see Figure 1), and B_s equals to the block size in the system. With $C_{block}(t)$, we can now obtain a more precise benefit $\mathcal{B}(t)$ for each term t in SSD-based infrastructure. We also expect that such estimation will work for HDD-based system, and this is conformed by experimental results reported in Section 4.2.

4. PERFORMANCE EVALUATION

4.1 Experimental Setup

To test our method, we extract 5 million web pages[2] from the GOV2 document set to generate the inverted index. The index consists of only the document identifiers (no frequency or positional information are included), with a size of 4.9GB.

Our query trace is extracted from a large log of queries issued by about 650,000 AOL users over three months[3]. After removing the *followup* queries (queries requesting successive page of results) [5] and eliminating the queries containing terms that do not appear in the document set, the log contains 14.4 million queries. We draw a random sample of 10

[1]Usually, S_{read} and S_{page} are considered as 50 μs and 2KB.

[2]In our testbed, the GOV2 collection is evenly distributed to 5 servers by URL hashing. We only use the first subset since we focus on the caching techniques in a centralized system.

[3]The queries are all de-identified and anonymized, so no personal info would be revealed from the query trace.

Table 1: Detailed experimental results on SSD and HDD.

	Policy	**SSD**							**HDD**						
		Cache Size (GB)							**Cache Size(GB)**						
		0.5	1.0	1.5	2.0	2.5	3.0	3.5	0.5	1.0	1.5	2.0	2.5	3.0	3.5
Average I/O latency (ms)	*QTFDF*	15.87	7.99	4.01	2.31	1.47	1.13	0.99	24.80	12.43	6.34	3.38	1.93	1.36	1.11
	QTF	7.40	3.84	2.41	1.72	1.30	1.12	1.00	17.29	10.79	7.09	4.95	3.46	2.50	1.79
	MECH	7.40	3.84	2.41	1.72	1.30	1.12	1.00	15.42	7.96	4.51	2.89	1.79	1.31	1.09
	BLOCK	7.38	3.82	2.24	1.48	1.18	1.01	0.95	16.17	8.11	4.40	2.61	1.71	1.25	1.08
Term hit ratio (%)	*QTFDF*	64.9	82.6	90.3	94.6	96.9	98.2	98.8	64.9	82.6	90.3	94.6	96.9	98.2	98.8
	QTF	29.4	46.1	59.5	70.7	80.5	87.8	93.4	29.4	46.1	59.5	70.7	80.5	87.8	93.4
	MECH	29.4	46.1	59.5	70.7	80.5	87.8	93.4	64.1	81.3	89.4	94.2	96.8	98.2	98.9
	BLOCK	34.5	54.6	70.0	82.0	91.1	96.2	98.3	48.5	71.7	84.6	91.6	95.7	97.8	98.8
Byte hit ratio (%)	*QTFDF*	54.6	77.4	88.9	94.9	98.0	99.2	99.8	54.6	77.4	88.9	94.9	98.0	99.2	99.8
	QTF	83.3	92.8	96.8	98.7	99.5	99.9	99.9	83.3	92.8	96.8	98.7	99.5	99.9	99.9
	MECH	83.3	92.8	96.8	98.7	99.5	99.9	99.9	77.9	89.2	94.8	97.6	99.0	99.7	99.9
	BLOCK	80.0	90.9	95.7	98.0	99.2	99.8	99.9	80.1	91.0	95.7	98.0	99.2	99.8	99.9

million queries for our experiments. There are 5,753,735 distinct queries and 397,447 distinct query terms in the sampled query set, while the average query length is 3.3. As expected, both the distributions of query frequency and term frequency of the sampled query set follow power-law distributions, with skew-factor $\alpha = 0.43$ and 1.63, respectively.

We then conduct a number of experiments by replaying the sampled query log over the generated index. We employ SvS algorithm [4] as the posting list intersection algorithm to evaluate the queries. We divide the 10 million queries into two parts: 90% of queries are used as training data to obtain reliable statistics such as term frequency, while the rest 10% are used as testing data. Since the index involved in the testing query set only amounts to 3.897 GB, thus the largest cache size was set as 3.5 GB.

The experiments are conducted on a PC running Centos Linux version 6.0, with an Intel i7 930 processor at 2.8 GHz and 12 GB of main memory. The SSD and HDD used are the same as those in Figure 1. In order to minimize the effectiveness of the OS buffer, the page cache is by-passed[4]. The block size in the system is set as 4KB [13]. We report the results in terms of *term hit ratio*, *byte hit ratio*[5] (metric that used in [12]), as well as the *wall-clock time*.

4.2 Experimental Results

We now present the results from the experimental evaluation of different static caching algorithms. First, we load all the index involved into main memory and execute the in-memory evaluation, which provides us with the elapsed time of the intersection operations solely. Second, for each static caching algorithm, the queries are evaluated with a static cache module and the total query response time is recorded. Then, by subtracting the intersecting time from the overall run time, we get the accurate disk access latency during the query processing. The wall-clock times reported in this section are all disk access latencies.

Figure 2 depicts the average access latency of four caching algorithm in the SSD-based infrastructure. In this figure, we can see that BLOCK method outperforms other policies under all the cache size settings, while QTFDF policy has the longest access latency. More detailed numerical results can be found in the left part of Table 1, where the term hit ratio and byte hit ratio are also presented.

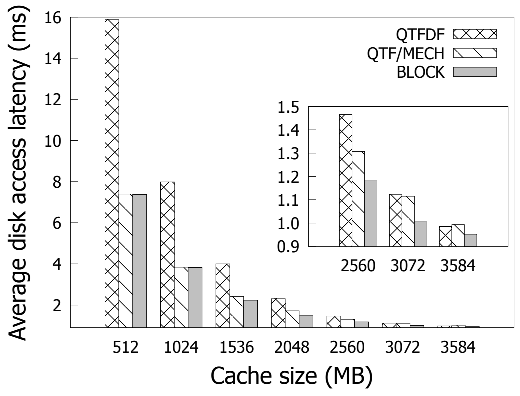

Figure 2: Average access latency (per query) on SSD

Table 2: Number of lists kept in static cache on SSD platform (counted in thousands)

Policy	**Cache Size (GB)**						
	0.5	1.0	1.5	2.0	2.5	3.0	3.5
QTFDF	262	313	336	350	359	367	373
QTF	0.3	0.8	1.8	3.7	7.4	15	33
MECH	0.3	0.8	1.8	3.7	7.4	15	33
BLOCK	25	54	136	181	240	396	342

As shown in Table 1, QTFDF has the highest term hit ratio among all the existing static cache algorithms. This echoes the results in [1]. However, it has the poorest byte hit ratio as well as the disk access latency. In contrast, QTF (also MECH[6]) achieves the best byte hit ratio, though its term hit ratio is markedly worse than other algorithms. Table 2 shows the number of lists that are loaded into the static cache memory. Each numerical value in this table is counted in thousands. We can see that, QTF keeps the

[4]using O_DIRECT flag in open() system call.

[5]Note that the 1 million testing queries require around 5.76 TB data in total during the query processing, provided no index is cached. Thus, a slight difference in byte hit ratio means a substantial difference in the amount of I/O data.

[6]Since MECH has exactly the same performance as QTF on SSD, we do not distinguish them in the following discussions, as long as in the context of SSD-based infrastructure.

fewest lists in the static cache, which explains why it has the lowest term hit ratio. In addition, it can be inferred that, with QTF policy, the lists in the cache are relatively longer than those of its counterparts. Thus, the average size of the posting lists that are fetched when cache misses occur is relatively smaller (which is confirmed by Figure 3). This accounts for the highest byte hit ratio it gets. In the same way, it is easy to understand why QTFDF ends up with a much lower byte hit ratio while achieving a rather higher term hit ratio.

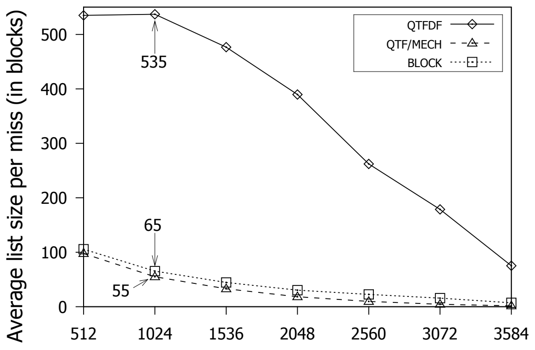

Figure 3: Average list size (in terms of blocks) of all cache misses on SSD platform

The reason that BLOCK has the shortest average disk access latency can be inferred from Figure 3. Take 1024 M-B cache memory for instance, a cache miss under BLOCK method would save about 535-65=470 sequential reads compared with QTFDF, which is equivalent to around 41 random reads on SSD. Even though the **term miss ratio** of QTFDF is around one third (0.174/0.454, actually) over that of BLOCK, it can not make up for the 41 extra random reads saved during each cache miss by BLOCK. On the other hand, although QTF results in smaller number of disk accesses (55 blocks per cache miss in average), the slight superiority is outweighed by the random accesses saved by BLOCK due to its lower term miss ratio ($0.539 * [\frac{55-1}{11.5} + 1] > 0.454 * [\frac{65-1}{11.5} + 1]$). Therefore, these explain why BLOCK yields a shorter access latency than other baselines. Furthermore, the results indicate that higher term hit ratio or byte hit ratio do not necessary lead to lower access latency, which echoes the conclusion in [13].

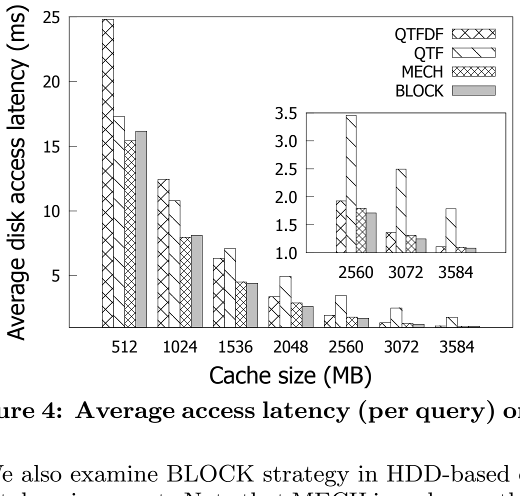

Figure 4: Average access latency (per query) on HD-D

We also examine BLOCK strategy in HDD-based experimental environment. Note that MECH is no longer the same as QTF in this context. Figure 4 illustrates the average disk access latency of the 1 million queries in HDD-based platform, while the term hit ratio and byte hit ratio are presented in the right part of Table 1[7]. Figure 4 shows that our new strategy also beats all the state-of-the-art algorithms (even MECH) under HDD-based infrastructure, though by a small margin. The other details, like the number of lists kept in the static cache by each policy and the reason why BLOCK outperforms other algorithms, are almost the same as those of SSD and are omitted due to the space constraint.

5. CONCLUSIONS

In this paper, we investigate the speed gap between random access and sequential access on both SSD and HDD, and then propose a block-level latency-aware static policy for inverted list caching. Our results show that the new strategy can reduce the disk access latency during query processing in both SSD and HDD based search engine infrastructures. The results also demonstrate that neither term hit ratio, nor byte hit ratio, are reliable reflections of the actual query latency, no matter the search engine infrastructure is HDD or SSD based. In other words, the wall-clock time is the only reliable metric to measure the efficiency of a cache policy.

6. ACKNOWLEDGMENTS

We would like to thank Jianguo Wang, Hao Li and Fan Zhang for their helpful discussions and comments. This work is partially supported by NSFC of China (60903028, 61070014) and Key Projects in the Tianjin Science & Technology Pillar Program (11ZCKFGX01100).

7. REFERENCES

[1] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, 2007.
[2] R. A. Baeza-Yates and S. Jonassen. Modeling static caching in web search engines. In *ECIR*, 2012.
[3] R. A. Baeza-Yates and F. Saint-Jean. A three level search engine index based in query log distribution. In *SPIRE*, 2003.
[4] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, 2001.
[5] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.
[6] R. Li, C. Li, W. Xiao, H. Jin, H. He, X. Gu, K. Wen, and Z. Xu. An efficient ssd-based hybrid storage architecture for large-scale search engines. In *ICPP*, 2012.
[7] R. Ma. Baidu distributed database. In *SACC*, 2010.
[8] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.
[9] R. Ozcan, I. S. Altingovde, B. B. Cambazoglu, F. P. Junqueira, and Ö. Ulusoy. A five-level static cache architecture for web search engine. *Inf. Process. Manage.*, 48(5):828–840, 2012.
[10] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Static query result caching revisited. In *WWW*, 2008.
[11] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Trans. Web*, 5(2):1–25, 2011.
[12] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. In *CIKM*, 2007.
[13] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. The impact of solid state drive on search engine cache management. In *SIGIR*, 2013.
[14] G. Xie, G. Xu, G. Wang, X. Liu, R. Cao, and Y. Gao. hubi: An optimized hybrid mapping scheme for nand flash-based ssds. In *IEEE ICESS*, 2011.

[7]Note that QTF performs identically on SSD and HDD, in terms of term hit ratio and byte hit ratio. So does QTFDF.