Second Chance: A Hybrid Approach for Dynamic Result Caching in Search Engines

I. Sengor Altingovde[1], Rifat Ozcan[1],
B. Barla Cambazoglu[2], and Özgür Ulusoy[1]

[1] Department of Computer Engineering, Bilkent University, Ankara, Turkey
{ismaila,rozcan,oulusoy}@cs.bilkent.edu.tr
[2] Yahoo! Research, Barcelona, Spain
barla@yahoo-inc.com

Abstract. Result caches are vital for efficiency of search engines. In this work, we propose a novel caching strategy in which a dynamic result cache is split into two layers: an HTML cache and a docID cache. The HTML cache in the first layer stores the result pages computed for queries. The docID cache in the second layer stores ids of documents in search results. Experiments under various scenarios show that, in terms of average query processing time, this hybrid caching approach outperforms the traditional approach, which relies only on the HTML cache.

Keywords: Search engines, query processing, result cache.

1 Introduction

Result caching is a crucial mechanism employed in search engines to satisfy low response time and high throughput requirements under high query workloads [2]. Usually, a static result cache is filled by the result pages of queries that were frequent in the past. Additionally, a dynamic result cache is maintained to handle the burst in query traffic. The content of the result cache changes dynamically depending on the query stream. Each time the cache is full, an entry is evicted from the cache based on a certain replacement policy (e.g., LRU). A real life search engine might either split the available cache capacity between static and dynamic caches [3], or involve a sufficiently large dynamic cache that would almost never evict frequent queries, as if they were kept in a static cache.

In design and evaluation of caching strategies, a traditionally used measure is the cache hit rate. Recently, some works have also taken into account the fact that the cost of a cache miss depends on the query, i.e., some queries require more computational resources to be answered [1,4]. These works have shown that it is better to tune a caching strategy according to the query processing cost incurred on the backend system, instead of the achieved hit rate alone.

A typical entry in a dynamic result cache stores the HTML result page[1] generated as an answer to a query. A storage-wise profitable alternative to this

[1] By HTML result page, we mean the textual content such as the URLs and snippets of the documents in the result page [3], but not the visual content in the page.

P. Clough et al. (Eds.): ECIR 2011, LNCS 6611, pp. 510–516, 2011.

Algorithm 1. The second chance caching algorithm

Require: q: query, H: HTML cache, D: docID cache
1: $R_q \leftarrow \emptyset$ ▷ initialize the result set of q
2: **if** $q \notin H$ and $q \notin D$ **then**
3: evaluate q over the backend and obtain R_q ▷ $C_q = C_q^{\text{list}} + C_q^{\text{rank}} + C_q^{\text{doc}} + C_q^{\text{snip}}$
4: insert R_q into H and D
5: **else if** $q \in H$ **then**
6: get R_q from H
7: update statistics of q in both H and D ▷ $C_q = 0$
8: **else if** $q \in D$ **then**
9: get doc ids from D and compute snippets to obtain R_q ▷ $C_q = C_q^{\text{doc}} + C_q^{\text{snip}}$
10: insert R_q into H
11: update statistics of q in D
12: **end if**
13: **return** R_q

is to store only the ids of the documents in the result page (possibly, together with their scores) [3]. Given the same amount of space, a docID cache can store entries for a larger number of queries than the HTML cache, and hence it can yield a higher hit rate. However, since the snippets had to be computed for the matching documents, average query processing times are higher relative to the HTML cache. In this respect, the information provided by the HTML cache is complete and ready-to-serve, whereas it is incomplete and requires further computation (i.e., snippet computations) in case of the docID cache.

In this study, we propose to split a dynamic result cache into two layers, namely HTML and docID caches, and introduce the so-called second chance caching strategy. The basic intuition behind our strategy is that, since a docID result for a query takes significantly less storage space than an HTML result, even if the HTML result is evicted from the cache, the docID result may still remain. The trade-off is simple: we reserve a relatively small space for complete HTML results and store incomplete results for a large number of queries that would, otherwise, be evicted. For those queries stored in the docID cache, we introduce snippet computation overhead. However, for potentially many queries, we avoid the more expensive scoring cost, completely.

The rest of the paper is as follows. Our caching strategy is presented in Sec. 2. In Sec. 3, we describe a detailed cost model for evaluating this strategy. Section 4 provides experimental results, showing performance improvements under various scenarios. The paper ends with the concluding discussion in Sec. 5.

2 Second Chance Caching Strategy

The main idea of our strategy is to divide the cache into two layers as HTML and docID caches. In Algorithm 1, we provide the basic outline of our caching strategy. Whenever a query q leads to a miss in both the HTML and docID caches, its result is computed at the backend search system and inserted into both caches (lines 3–4). As long as q is found in the HTML cache, its results are

served by this cache and its statistics are updated (lines 6–7). At some point in time, q may become the LRU item. In this case, it is discarded from the HTML cache, but its (incomplete) results still reside in the docID cache. In some sense, this approach gives a second chance to q. If the query is ever repeated soon, it becomes a hit in the docID cache. In this case, the backend system computes only the snippets to create the HTML result page, which is both sent to the user and inserted into the HTML cache (lines 9–11). Note that each case in the algorithm is associated with a cost (C_q), which we will discuss next.

3 Cost Model and Scenarios

Processing of a query q in a search engine has four main steps: (i) fetching posting lists for query terms from disk (C_q^{list}), (ii) decompressing lists and computing the top k results (C_q^{rank}), (iii) fetching k result documents from disk (C_q^{doc}), and (iv) computing snippets for fetched documents (C_q^{snip}). Costs incurred in case of a miss, an HTML cache hit, and a docID cache hit are given in Algorithm 1.

In practice, a search cluster in a commercial search engine is made up of hundreds of nodes that store a part of the inverted index and document collection.[2] This means that steps (i) and (ii) are executed on all nodes in the cluster. Then, the partial results for the query are sent to the broker node, which merges them and computes the final top k document ids. Finally, these documents are accessed to compute snippets (steps (iii) and (iv)) and generate the HTML result page. The cost of transferring and merging partial results is mostly negligible.

Under the above cost model, we further consider two key issues: 1) caching of posting lists as well as documents and 2) assignment of documents to search nodes. Regarding the first issue, it is known that search engines cache posting lists and documents, in addition to search results. In case of a cache miss, steps (i) and (iii) require disk accesses. To this end, we consider two different scenarios for caching the latter two types of data: A "full caching" scenario, where all lists and documents are kept in the memory, and a more conservative "moderate caching" scenario, where only 50% of each item type is cached.

The second issue we consider is the distribution of the snippet computation overhead on the nodes of a search cluster with K nodes. This distribution affects how the costs in steps (iii) and (iv) are computed. We again consider two basic scenarios: In the "random assignment" scenario, we assume that documents in the collection are randomly assigned to cluster nodes, as usual. For $K \gg k$, it is very likely that every document in the top k result resides on a different node. Then, document access and snippet generation take place on each node in parallel and, in effect, the total processing cost is almost equal to the cost of executing steps (iii) and (iv) for only one document (i.e., as if $k = 1$). In the "clustered assignment" case, we assume that documents are assigned to nodes based on, say, topic. In the worst case, all top k documents reside in the same node. Hence, the costs of steps (iii) and (iv) are incurred for all k documents.

[2] The result cache is maintained in the broker node.

4 Experiments

We use a collection of 2.2 million web pages obtained from the ODP web directory (`http://www.dmoz.org`). The index file includes only document ids and term frequencies, and it is compressed with the Elias-δ encoding scheme. We sample from the AOL query log [5] 16.8 million queries, whose all terms appear in our collection. Queries are processed in timestamp order. First 8M queries are used as the training set and the remaining 8.8M queries are used as the test set. Training and test sets include 3.5M and 3.8M unique queries, respectively.

Training queries are used for two purposes. First, frequent queries are used to warm up the HTML and docID caches. Second, for the "moderate caching" scenario, we use these queries to populate the posting list and document caches. To decide on the terms to be cached, we use the popularity/size metric [2], where the former is the term's frequency in the training query log and the latter is the size of its posting list. While caching documents, we process training queries over the collection and select the documents that are most frequent in top 10 search results. In both cases, the items are cached up to the 50% capacity limit.

In Tables 1 and 2, we provide the parameters and cost formulas used, respectively. Decompression and scoring times are experimentally obtained over our dataset. For snippet computation time, we assume a simple method (each query term in the document along with a few neighboring terms are added to the snippet) and set this value to a fraction of the query processing time.

In Table 1, we specify the ratio (S) between the sizes of a result item in the HTML cache and that in the docID cache because all experiments reported below specify the cache capacity in terms of the number of HTML result pages that can fit into the cache. Hence, our findings are valid as long as the ratio among the result item sizes is preserved, regardless of the actual values. Assuming that

Table 1. Parameters

Parameter	Value	Parameter	Value
Size ratio of cache items (S)	64	Number of results per query (k)	10
Disk latency (D_ℓ)	12.7 ms	Decompression per posting (P_d)	100 ns
Disk block read (D_br)	4.9 μs	Ranking per posting (P_r)	200 ns
Disk block size (D_bs)	512 bytes	Snippet computation per byte (P_s)	10 ns

Table 2. Cost formulas (t denotes a term in q, I_t denotes the inverted list of t, and d denotes a document in the query result set R_q)

Cost	Formula	Description
C_q^{list}	$\sum_{t \in q} (D_\ell + (D_\mathrm{br} \times \lvert I_t \rvert / D_\mathrm{bs}))$	Fetching of posting lists from disk
C_q^{rank}	$\sum_{t \in q} (\lvert I_t \rvert \times (P_\mathrm{d} + P_\mathrm{r}))$	Score computations during ranking
C_q^{doc}	$\sum_{d \in R_q} (D_\ell + (D_\mathrm{br} \times \lvert d \rvert / D_\mathrm{bs}))$	Fetching of documents from disk
C_q^{snip}	$\sum_{d \in R_q} (\lvert d \rvert \times P_\mathrm{s})$	Snippet computations

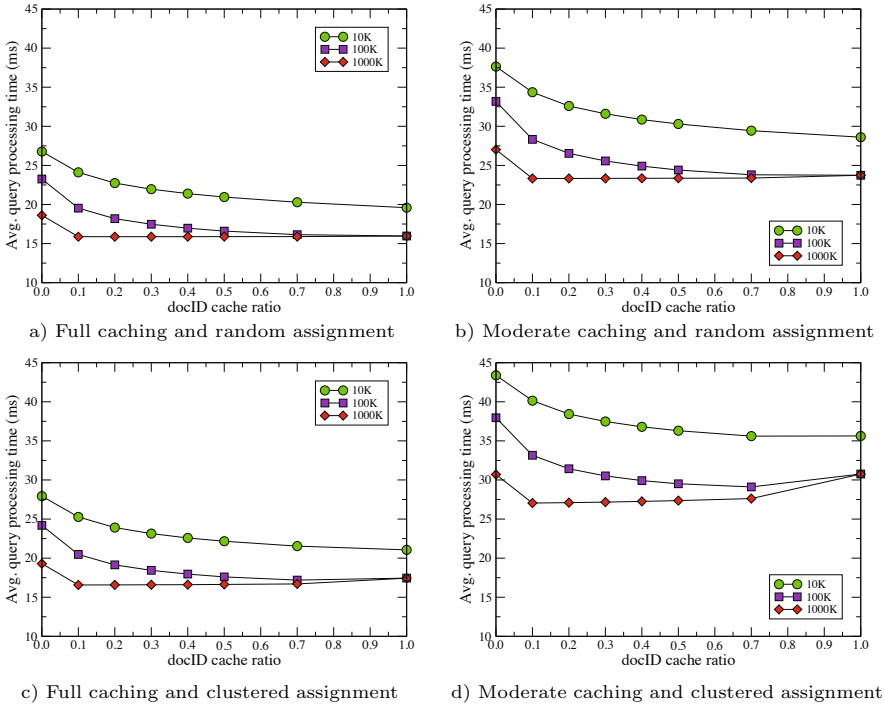

Fig. 1. Performance of the hybrid result cache under different scenarios. In all figures, the docID cache ratio of 0 corresponds to the pure HTML cache, which is our baseline.

a single document id may take around 4 bytes and a result URL and snippet (of 20 terms) may take around 256 bytes, we set this ratio to 64.

We compare the performance of our strategy for varying cache split ratios, which range from the pure HTML cache (the ratio is 0) to the pure docID cache (the ratio is 1) over the test query set. We experiment with three different cache capacities that are representatives of small (10K), medium (100K), and large (1000K) caches for our query log. We express the cache size in terms of the number of HTML result pages that can fit into the cache, e.g., the 1000K cache can store 1000K pages. The largest cache capacity corresponds to about 11% of all test queries and 26% of the unique test queries. As mentioned before, for the random and clustered assignment scenarios, we set k to 1 and 10, respectively.

In Fig. 1, we consider four different combinations of our scenarios. Since our strategy (by definition) improves the hit rate of the baseline HTML cache and the trade-off is in terms of the incurred system costs, we report only query processing times. Our findings are as follows: (i) For all cases, we can identify docID cache ratios that yield better performance than the baseline, i.e., the pure HTML cache. In other words, using a hybrid cache always reduces the average query processing time with respect to the baseline. (ii) For cache sizes of 10K and 100K, it is more efficient to devote the majority (i.e., greater than 70%) or even all of the cache space to the docID cache. This implies that for these

cases, most of the frequent queries cannot stay long enough in the pure HTML cache. For these cache sizes, reductions in average query processing time reach up to 31%. (iii) For the largest cache size, we observe that reserving 10% of the capacity to the docID cache yields the best performance. This indicates that, as the available cache size increases, the gain provided by the docID cache decreases, as it would be possible to store more results in the HTML cache. Still, the relative reductions provided by the second chance caching strategy are 15% and 14% for the full and moderate caching cases with random assignment of documents, respectively. Achieved reductions are slightly lower for the clustered assignment of documents (i.e., 14% and 12% for full and moderate caching, respectively). The setup with the highest query processing cost (i.e., moderate caching with clustered document assignment) yields the lowest reduction (12%).

5 Concluding Discussion

Our strategy provides significant advantages when the result cache capacity is limited. In the experiments, we showed that our strategy yields 15% reduction in query processing time even when the cache is large enough to store 26% of all unique queries in the test set. Clearly, as the result cache gets larger, benefits of our strategy diminish. At one extreme, if the search system has enough resources to cache the results of all non-singleton queries for a reasonably long time, the hybrid cache is rendered useless. However, given the current query workloads of search engines, such a solution may require large amounts of storage, summing up to an infeasible financial value. Our solution, on the other hand, is a compromise for providing better utilization on a limited-memory result cache.

Our caching strategy exploits the fact that the snippet generation cost is a fraction of the total query processing cost. To best of our knowledge, there is no work that explicitly compares the cost of snippet generation to other costs, such as decompression, list intersection, and ranking. Nevertheless, to investigate the sensitivity of our caching strategy to snippet generation cost, we repeated our experiments with higher P_s values (100 ns and 1000 ns per byte). For the full and moderate caching scenarios with random document assignment, the best docID cache ratios still reduce query processing times by 14% and 12%, for the 100 ns case, and by 10% and 5%, for the 1000 ns case, respectively.

This work is a first step for investigating the performance of a hybrid dynamic caching strategy for search engines in a cost-based framework. In the future, we plan to integrate result pre-fetching into our strategy.

References

1. Altingovde, I., Ozcan, R., Ulusoy, O.: A cost-aware strategy for query result caching in web search engines. In: Boughanem, M., Berrut, C., Mothe, J., Soule-Dupuy, C. (eds.) ECIR 2009. LNCS, vol. 5478, pp. 628–636. Springer, Heidelberg (2009)

2. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The impact of caching on search engines. In: Proc. 30th Annual Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval, pp. 183–190 (2007)
3. Fagni, T., Perego, R., Silvestri, F., Orlando, S.: 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)
4. Gan, Q., Suel, T.: Improved techniques for result caching in web search engines. In: Proc. 18th Int'l Conf. on World Wide Web, pp. 431–440 (2009)
5. Pass, G., Chowdhury, A., Torgeson, C.: A picture of search. In: Proc. 1st Int'l Conf. on Scalable Information Systems, p. 1 (2006)