Cost-Aware Result Caching for Meta-Search Engines

Emre Bakkal
Middle East Technical University
Ankara, Turkey
emre.bakkal@metu.edu.tr

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

Ismail Hakki Toroslu
Middle East Technical University
Ankara, Turkey
toroslu@ceng.metu.edu.tr

ABSTRACT

Our goal in this paper is to design cost-aware result caching approaches for meta-search engines. We introduce different levels of eviction, namely, query-, resource- and entry-level, based on the granularity of the entries to be evicted from the cache when it is full. We also propose a novel entry-level caching approach that is tailored for the meta-search scenario and superior to alternative approaches.

1. INTRODUCTION

The problem of combining search results that are obtained from several different and/or heterogeneous data sources are well studied in the IR and DB literature under various names, such as meta-search, federated-search and data fusion/aggregation (e.g., see [13]). In the context of Web search, a meta-search engine is a tool that forwards a submitted query to component search systems (so-called *resources* hereafter), collects the local top-k results from each of these systems and merges them to obtain a global top-k result. Although we have witnessed the domination of market by general purpose search engines in the last decade, the idea of a meta-search engine can be still useful in the context of specialized domains, such as shopping, healthcare or education. In such specialized domains, the result of a query submitted to a meta-search tool can be either directly presented to end-users, or can be post-processed to be consumed by other applications (e.g., extracting the product information for a price comparison engine in the context of shopping). In either case, the re-use of the results for previously seen queries, typically by constructing a *result cache*, is important not only for the efficiency and scalability of a meta-search system, but also for the availability (i.e., when the resources are temporarily unavailable [4]) and financial effectiveness (i.e., when a resource charges a processing fee per query).

Our goal in this work is to design cost-aware and dynamic result caching approaches to be employed in a meta-search scenario. To this end, as our first contribution, we introduce three different levels of eviction, namely, query-level, resource-level and entry-level, that arise naturally in the meta-search setup and indicate the granularity of the entries to be evicted from the cache when it is full. For each such eviction level, we utilize the well-known traditional and cost-aware eviction policies [11] to identify the actual entries that will be evicted, and hence, end up with several combinations representing alternative caching approaches. To the best of our knowledge, none of the earlier works consider the eviction at different granularities within a cost-aware framework, as we do here. Next, we propose a novel entry-level caching approach that is again cost-aware and exploits the special requirements of the meta-search scenario, i.e., the embarrassingly-parallel nature of processing a given query at each resource. These alternative caching approaches are evaluated using the cache miss-cost metric [11] in a large-scale simulation setup where the impact of various parameters (such as the number of resources, cache size and query cost distribution) is also investigated. Our simulation results show that the highest performance is obtained by using the entry-level caching approaches; and furthermore, our newly proposed approach outperforms both the traditional baselines and other cost-aware competitors.

SIGIR'15, August 9–13, 2015, Santiago, Chile.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3621-5/15/08 ...$15.00
DOI: http://dx.doi.org/10.1145/2766462.2767813.

2. RELATED WORK

For general purpose search engines, result cache is a key component that improves the system efficiency and reduces the load at the backend. In earlier works, query results are cached in terms of the document identifiers [5, 8, 10] or full result pages including the title, URL and snippet of top-k results [9, 5, 2]. For the typical meta-search scenario with non-cooperative resources, the internal documents ids are neither available nor useful, and hence, we assume that the cache stores the results in the latter format.

The caching strategies can be broadly categorized as either static or dynamic [9]. A static cache determines items to be cached based on previous usage statistics and keeps the cache content intact until the next periodic update. In contrast, the dynamic caching strategies employ an eviction strategy to decide on the item to be removed when the cache is full; and they typically rely on the frequency and recency of the cached items, as in the well-known Least Frequently Used (LFU) and Least Recently Used (LRU) policies, respectively. More recently, the cost of generating and/or fetching the cached item is incorporated into the cache eviction policies; and furthermore, employed as a more realistic metric for evaluating the cache performance [11, 6]. Following this line of research, in this study, we tailor cost-aware caching approaches for meta-search engines.

In contrast to the case of general purpose search engines, caching for meta-search engines is an area that is left unexplored with the exception of a few works. In one of the earliest studies, Chidlovskii et al. propose a semantic cache at the client-side for a meta-search system [4]. Their work also mention the possibility of designing eviction strategies that take into account the cost of the cached entries; however the experimental evaluation only considers typical LRU policy and hit-rate as the efficiency metric. Lee et al. describe a popularity-driven caching strategy for meta-search engines; but again do not consider the notion of cost during caching and evaluation [7], as we do in this paper.

3. CACHING FOR META-SEARCH

We consider a meta-search framework where a broker search system forwards the query to component search systems that may include general purpose search engines as well as the APIs of Web 2.0 platforms, like YouTube or Twitter. The top-k result lists, R_i, from each such resource r_i, are returned to the broker; and merged there to obtain the global query result. We assume that for a given query q, there is an associated cost C_i to obtain the result R_i, which is the elapsed time for query processing at the target resource plus the network transfer time. Remarkably, the same resource can yield different costs for different queries, and the same query can incur different costs from different resources.

For a given query q and assuming there are N different resources r_1 to r_N, we store the vector $R_q : \langle(R_1, C_1), \ldots, (R_N, C_N)\rangle$ in the result cache. We believe it is preferable to store the local results from each resource rather than storing the merged (global) result (as in the earlier works like [4, 7]), as it allows more flexibility to switch to a different result merging algorithm and to apply separate mechanisms to invalidate cached results (e.g., results from more dynamic resources can be assigned a smaller time-to-live value). We consider a dynamic caching setup, as earlier works show that for reasonably large caches, dynamic caching approaches outperform the static counterparts [9].

3.1 Cost-aware eviction strategies

In the following discussion, a cache entry e can denote either the query result vector R_q, or a pair (R_i, C_i) in this vector; *C(e)* denotes the cost of generating this entry, *F(e)* denotes the frequency of this entry (i.e., submission frequency of the corresponding query in the past) and *S(e)* denotes the entry size (in bytes). Using this notation, we summarize the cost-aware strategies that are described in [11] for evicting the entries from a dynamic result cache, as follows:

Least Costly Used (LCU). Evicts the cache entry e with the minimum cost *C(e)*.

Least Frequently and Costly Used (LFCU). Evicts the cache entry that has the minimum *H(e)* value that is computed as $H(e) = C(e) \times F(e)^K$, where $K (> 1)$ is a parameter that aims to emphasize the impact of larger frequencies [11].

Greedy Dual Size (GDS). Evicts the cache entry that has the minimum *H(e)* value that is computed as $H(e) = \frac{C(e)}{S(e)} + L$, where L value serves as an aging factor [3]. The L value is initialized to 0, and every time an entry gets evicted, it is set to the *H(e)* value of the evicted entry.

Greedy Dual Size Frequency (GDSF). Evicts the cache entry that has the minimum *H(e)* value that is computed as $H(e) = \frac{C(e)}{S(e)} \times F(e)^K + L$, where $K$and L values are computed as in the LFCU and GDS eviction strategies, respectively [1]. This strategy combines all four dimensions; namely, size, frequency, recency, and cost of the cached entries, while deciding the entry to be evicted. While the latter three dimensions are more likely to vary, we expect the size of the cached entries to be almost the same in practice; and hence, without loss of generality, all our discussions hereafter assume that the results from each resource take the same amount of space.

3.2 Eviction levels for meta-search

Our choice of storing the entire result vector $R_q : \langle(R_1, C_1), \ldots, (R_N, C_N)\rangle$ in the cache naturally and uniquely allows us to consider different granularities of eviction, as it is possible to evict the entire result vector of a query; or one or more entries from one or more result vectors. In this paper, we define three different caching approaches based on the eviction granularity (level), as follows:

Query-level (QL) eviction: When the cache is full, QL eviction determines a victim query using one of the cost-aware eviction strategies described in Section 3.1, and then evicts the *entire* result vector of the victim query. While doing so, the cost-aware strategies consider the cost of the evicted query, *C(e)*, as $\max(C_1, \ldots, C_n)$ where $(R_i, C_i) \in R_q$, since in meta-search the query is processed at all resources in *parallel*; so the overall latency is the maximum of these individual costs.

Resource-level (RL) eviction: In this approach, instead of evicting a query result as a whole, we first uniformly partition the cache space for each resource. Then, when the cache is full, the cost-aware eviction strategy determines a victim *(R, C)* pair to be removed from *each* resource partition.

Example. To illustrate the difference between query-level and resource-level approaches, we represent the cache content with the matrix M (for brevity) as shown in Figure 1. In the matrix, each row (column) is a query (resource), respectively; and the entries denote the cost of retrieving a query result from a particular resource. For simplicity, assume that we apply LCU as the cost-aware eviction strategy. In this case, when the cache is full, the QL approach evicts the entire result for q_2, i.e., the second row of the matrix, as $C(q_1)$, $C(q_2)$ and $C(q_3)$ are 45, 24 and 51, respectively. In contrast, the RL eviction approach determines the least-costly entry per resource; and in this case, the entries $M[1, 1]$ (i.e.; the result of q_1 from r_1), $M[2, 3]$ and $M[3, 2]$ are removed. In this case, the cache miss-cost is $1 + 4 + 8 = 13$, assuming q_1, q_2 and q_3 are re-submitted in the near future.

Entry-level (EL) eviction: As a third alternative, we consider each (R_i, C_i) pair as an independent cache entry; and when the cache is full, the eviction policy determines N victim pairs to be removed, so that a new query result vector R can be stored. Note that, in this case, several pairs from a particular query or resource can be removed. For instance, considering the cache contents in Figure 1 and assuming LCU, the entries $M[1, 1]$, $M[3, 1]$ and $M[2, 3]$ should be removed, as their costs lead to minimum total cost; i.e., 7.

Optimal solution for EL eviction. As the astute reader will realize, while the caching approach with entry-level eviction has much lower miss-cost than its competitors, it is not optimal. Indeed, a better solution should take into account the observation that if two pairs (R_i, C_i) and (R_j, C_j) are evicted for a query q; the miss-cost will not be $C_i + C_j$;

$$\text{M}=\begin{array}{c|ccc} & r_1 & r_2 & r_3 \\ q_1 & 1 & 12 & 45 \\ q_2 & 5 & 24 & 4 \\ q_3 & 3 & 8 & 51 \end{array} \quad \begin{array}{l} \textbf{QL+LCU}: M[2,1], M[2,2], M[2,3] \\ \textbf{RL+LCU}: M[1,1], M[3,2], M[2,3] \\ \textbf{EL+LCU}: M[1,1], M[3,1], M[2,3] \end{array}$$

Figure 1: Cache contents shown as a matrix (left) and evicted entries for each eviction level with LCU (right).

but $max(C_i, C_j)$, as R_i and R_j will be retrieved from the resources r_i and r_j in parallel. For example, in Figure 1, the optimal solution (i.e., with the least possible miss-cost) is evicting $M[1,1]$, $M[2,1]$ and $M[2,3]$, as the incurred miss-cost would be $1 + max(5,4) = 6$.

The optimal solution for the entry-level eviction can be computed using dynamic programming, in a similar fashion to that of the well-known 0-1 Knapsack problem. Let's define the matrix A to store the cost of query q using r resources in ascending order of the costs. Then, in each step, we attempt to add a new query to the solution, and while doing so, we compute (and store in a table) the costs when we evict entries for r resources from this new query, where $0 \le r \le N$, and entries for $N - r$ resources from the earlier queries. Hence, the recursive formula for the dynamic programming solution is:

$$d(q,n) = \begin{cases} 0 & \text{if } n = 0, \\ \min\limits_{0 \le r \le N} (d(q-1,r) + A(q,n-r)) & \text{if } n > 0. \end{cases} \tag{1}$$

Obviously, even computing the optimal cost has the computational complexity $O(MN^2)$, where M and N denote the number of queries in the cache and number of resources, respectively. Moreover, there is an additional overhead of updating the costs (using Equation 1) after each cache-miss and subsequent eviction of entries. Based on these run time requirements, it is not affordable to use the optimal solution as a cache management approach in practical systems.

Greedy solution for EL eviction. As a remedy, we propose a novel greedy algorithm that also takes into account the aforementioned parallel processing effect in meta-search scenario. In a nutshell, the algorithm works as follows: While storing the vector R_q for a query, the entries (R_i, C_i) in the vector are sorted wrt. C_i values. Next, for a pair that is at position p in the sorted list, its C' value is defined as $\frac{C_i}{p}$. Intuitively, this indicates that if we remove the entries up to and including position p; we will have size-p free space and the miss-cost for this query will be C_i; so the cost per space is $\frac{C_i}{p}$. Once these C' values are computed, any of the eviction strategies described in Section 3.1 can be employed; but taking into account the C' values while computing $H(e)$ values. Finally, when an entry (R_p, C_p) at position p is removed (along with all the entries at positions $p' < p$); we reduce the cost of all entries at higher positions by C_p; since the C_p (the current miss-cost for the query q) will be anyway incurred from this point on. For these entries, the C' values are also recomputed with respect to modified cost values. The new algorithm is called Greedy Parallel-Aware Caching Strategy (GPACS).

Example. To illustrate how GPACS operate, consider Figure 2 where each result vector R_q (of Figure 1) is sorted with respect to C_i values and corresponding C' values are shown per entry in the matrix M'. Note that, in practice, these values can be stored along with the actual entries; the matrix view is just for the illustration purposes. We again use LCU to evict entries; and in this case, we evict the entries $M[1,3]$ and $M[2,2]$ that have the lowest total costs (i.e., 1 and 2.5, respectively) and yield a total of 3 empty spaces, as required to store a new query result. Note that, C' for $(R_1, 5)$ is 2.5; as 5 is divided by its position index, which 2.

$$\begin{array}{l} R_{q1} :<(R_3,45),(R_2,12),(R_1,1)> \\ R_{q2} :<(R_2,24),(R_1,5),(R_3,4)> \\ R_{q3} :<(R_3,51),(R_2,8),(R_1,3)> \end{array} \quad M'=\begin{array}{c|ccc} & p_1 & p_2 & p_3 \\ q_1 & 15 & 6 & 1 \\ q_2 & 8 & 2.5 & 4 \\ q_3 & 17 & 4 & 3 \end{array}$$

Figure 2: Re-sorted result vectors (left) and corresponding C' values shown as a matrix (right).

Recall that selecting $M[2,2]$ means that we evict $(R_1, 5)$ pair at position 2 and all pairs in lower positions (i.e., $(R_3, 4)$ at position 1), so that we obtain 2 empty places in the cache. Finally, once $(R_1, 5)$ and $(R_3, 4)$ entries are evicted for q_2, the cost of the remaining $(R_2, 24)$ entry is updated as 19, since evicting this entry will bring an additional cost of 19 from this point on.

4. EVALUATION AND DISCUSSIONS

Setup. While there are public datasets (e.g., from TREC FedWeb Track) to evaluate the effectiveness of meta-search, the number of queries in these datasets are too small (i.e., up to 1,000 queries) to evaluate the performance of a result cache. Therefore, we construct a simulation setup as follows. We use an excerpt from the AOL query log [12] to simulate the repetition patterns of queries. For each query, we assume results from N different resources are retrieved and cached. For each such resource, we define an interval that represents the cost of retrieving results from this resource. Then, for each query q and resource r, we sample a cost value uniformly at random from the interval associated with this resource. We store these (R_i, C_i) values generated per query so that all caching strategies are evaluated under the same conditions.

Parameters. In our simulations, we used 500,000 queries from AOL log [12] in timestamp order. First 300,000 queries are used to warm-up the cache, and the rest are used for evaluation. We assume $N \in \{10, 50\}$, and all cost values are in [0, 1] range. We set the cost intervals for these resources in an ad hoc manner to model the fast, medium-speed and slow resources that exist in a real-life setting (e.g., fast and slow resources have the cost ranges [0, 0.3] and [0.8, 1], respectively). Note that, in practice, a meta-search engine can simply record the retrieval time for each query and resource, to apply the proposed caching approaches. We report simulation results for three different cache sizes, namely, caches that can store 10K, 50K and 100K query results for $N = 10$. We assume the same cache capacity when N is set to 50 (for the sake of fair comparison), hence the number of queries that can be cached drops accordingly, as 2K, 10K and 20K.

Evaluation metric. Since our goal by employing a cache is to reduce the total time cost for the meta-search system, we evaluate the proposed strategies in terms of the cost reduction percentage they achieve with respect to the no-caching case; i.e., when all results have to be retrieved for all 200,000 test queries from the scratch. As discussed before, for a given query q, if only some entries of R_q are located in the cache (i.e., cache hit), the miss-cost incurred for this query is maximum cost among the R_i's that caused a cache-miss.

Results. In Figure 3, we report the performance of query-level (QL), resource-level (RL) and entry-level (EL) caching approaches coupled with six different eviction strategies (i.e., traditional LRU and LFU strategies as well as the cost-

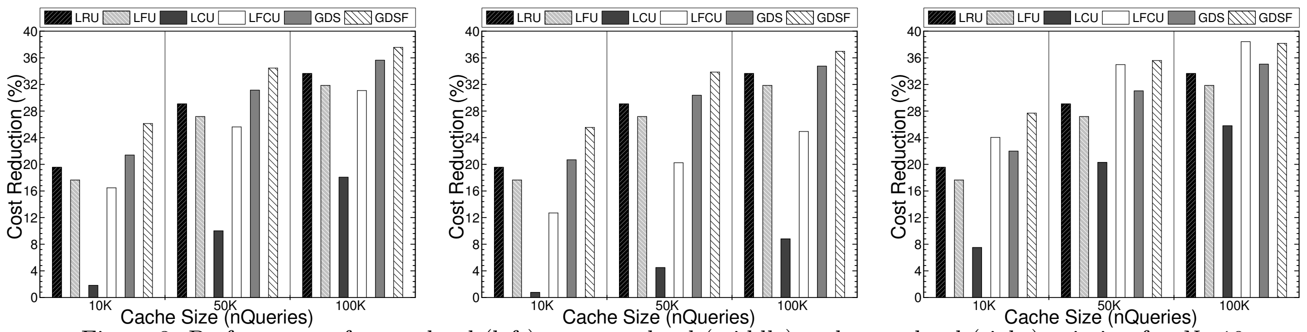

Figure 3: Performance of query-level (left), resource-level (middle) and entry-level (right) eviction for N=10.

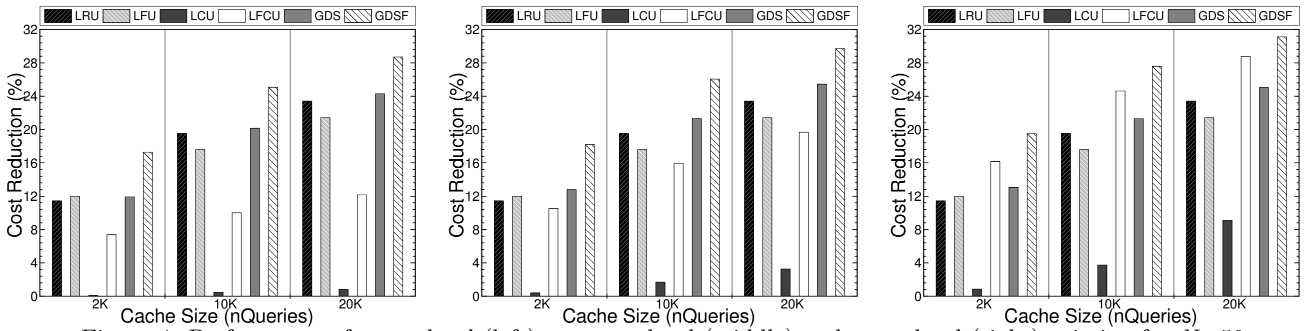

Figure 4: Performance of query-level (left), resource-level (middle) and entry-level (right) eviction for N=50.

Table 1: GPACS vs. best EL competitor (both with GDSF)

N=10			N=50		
Size	EL+GDSF	GPACS+GDSF	Size	EL+GDSF	GPACS+GDSF
100K	38.4%	39.1%	20K	31.1%	31.9%
50K	35.6%	36.2%	10K	26.1%	28.3%
10K	27.7%	28.4%	2K	18.2%	19.9%

aware strategies LCU, LFCU, GDS, and GDSF) for N=10 resources. Our findings are as follows: First, traditional strategies LRU and LFU are inferior to cost-aware strategies, especially, GDS and GDSF, in all cases. In particular, using only cost dimension is ineffective (as LCU is the worst performer in Figure 3) as an eviction strategy, while the GDSF strategy that takes into account all available clues (i.e., the cost, frequency and recency of queries) almost always yields the highest reduction in miss-costs. These findings are in line with those for web search engines [11]. Second, we compare the performances among the caching approaches with different eviction levels. Figure 3 shows that, in general, query-level eviction is better than resource-level eviction; and entry-level eviction outperforms both of the latter. For instance, the reduction in miss-costs is 37.6%, 37.0%, 38.2% for QL, RL and EL eviction approaches with GDSF strategy (for the cache of size 100K), respectively. This indicates that EL eviction is the most effective approach in the meta-search scenario. Figure 4 presents the results for N=50. We see that absolute gains slightly drop as the same cache capacity can now store a smaller number of queries. However, all the trends observed before also hold in Figure 4, implying the robustness of our findings under different settings.

Finally, Table 1 compares the performance of our novel GPACS algorithm to the best-performing case using EL eviction. It turns out that GPACS can yield up to 2% absolute improvements in terms of the cost reduction percentage.

Conclusion and future work. We introduced alternative eviction levels for result caches in a meta-search scenario, and showed that caching approaches with entry-level eviction outperforms those with query- and resource-level eviction. We also proposed a novel entry-level caching algorithm that is specifically tailored for meta-search and can further improve the performance. Our future work involves exploring cache refreshment issues in this setup.

Acknowledgements. This work is partially funded by the Ministry of Science, Industry and Technology of Turkey and by Huawei Inc. under the grant no 0441.STZ.2013-2.

5. REFERENCES

[1] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. *SIGMETRICS Performance Evaluation Review*, 27(4):3–11, Mar. 2000.

[2] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR 2007*, pages 183–190, 2007.

[3] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *USITS 1997*, pages 18–18, 1997.

[4] B. Chidlovskii, C. Roncancio, and M. Schneider. Semantic cache mechanism for heterogeneous web querying. *Computer Networks*, 31(11-16):1347–1360, 1999.

[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 TOIS*, 24(1):51–78, Jan. 2006.

[6] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW 2009*, pages 431–440, 2009.

[7] S. H. Lee, J. S. Hong, and L. Kerschberg. A popularity-driven caching scheme for meta-search engines: An empirical study. In *DEXA 2001*, pages 877–886, 2001.

[8] M. Marin, V. Gil-Costa, and C. Gomez-Pantoja. New caching techniques for web search engines. In *HPDC 2010*, pages 215–226, 2010.

[9] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2), 2001.

[10] R. Ozcan, I. S. Altingovde, B. B. Cambazoglu, F. P. Junqueira, and Özgür Ulusoy. A five-level static cache architecture for web search engines. *IPM*, 48(5):828–840, 2012.

[11] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Transactions on the Web*, 5(2):9:1–9:25, May 2011.

[12] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *INFOSCALE 2006*, 2006.

[13] M. Shokouhi and L. Si. Federated search. *Foundations and Trends in Information Retrieval*, 5(1):1–102, 2011.