Information Processing and Management 48 (2012) 828–840

Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/infoproman

A five-level static cache architecture for web search engines

Rifat Ozcan [a], I. Sengor Altingovde [a,*], B. Barla Cambazoglu [b], Flavio P. Junqueira [b], Özgür Ulusoy [a]

[a] Department of Computer Engineering, Bilkent University, Ankara, Turkey
[b] Yahoo! Research, Diagonal 177, 08018 Barcelona, Spain

ARTICLE INFO

Article history:
Received 11 February 2010
Received in revised form 9 December 2010
Accepted 20 December 2010
Available online 1 February 2011

Keywords:
Web search engines
Static caching
Query processing

ABSTRACT

Caching is a crucial performance component of large-scale web search engines, as it greatly helps reducing average query response times and query processing workloads on backend search clusters. In this paper, we describe a multi-level static cache architecture that stores five different item types: query results, precomputed scores, posting lists, precomputed intersections of posting lists, and documents. Moreover, we propose a greedy heuristic to prioritize items for caching, based on gains computed by using items' past access frequencies, estimated computational costs, and storage overheads. This heuristic takes into account the inter-dependency between individual items when making its caching decisions, i.e., after a particular item is cached, gains of all items that are affected by this decision are updated. Our simulations under realistic assumptions reveal that the proposed heuristic performs better than dividing the entire cache space among particular item types at fixed proportions.

© 2010 Elsevier Ltd. All rights reserved.

1. Introduction

Caching has been a long-studied topic in computer science (Effelsberg & Haerder, 1984; Smith, 1982). Although there have been many works for web caching (Podlipnig & Böszörmenyi, 2003; Wang, 1999), caching in the context of search engines has attracted research attention only over a decade ago (Markatos, 2001). As the volume of queries grows over time, caching techniques present important resource savings and performance advantages to search engines. Query traffic has been growing continuously over the past years, thus making cache elements indispensable for building high-performance web search engines.

The idea behind caching is to copy frequently or recently accessed parts of the data from high-capacity but slow storage devices (e.g., disk) to low-capacity but fast storage devices (e.g., memory). In some caching strategies, this idea is coupled with offline precomputation and storage of certain information. In general, search engine caches can be classified into two categories, according to their capability: static caches (Baeza-Yates & Saint-Jean, 2003; Fagni, Perego, Silvestri, & Orlando, 2006; Markatos, 2001) and dynamic caches (Fagni et al., 2006; Markatos, 2001; Saraiva et al., 2001). Static caches try to capture the access locality of data items. Past data access logs are utilized to decide on the data to be cached. Typically, items that are more frequently accessed in the past are preferred over infrequently accessed items for caching (see Ozcan, Altingovde, & Ulusoy (2008) for other possibilities). Static caches need to be periodically updated, depending on the variation in access frequencies of items. Dynamic caches, on the other hand, try to capture the recency of data access. The data that is more likely to be accessed in the near future remains in the cache. The past research challenge in dynamic caching was to develop novel cache eviction policies (Gan & Suel, 2009). The current challenge is to devise effective policies to keep cache

* Corresponding author.
E-mail addresses: rozcan@cs.bilkent.edu.tr (R. Ozcan), ismaila@cs.bilkent.edu.tr (I. Sengor Altingovde), barla@yahoo-inc.com (B. Barla Cambazoglu), fpj@yahoo-inc.com (F.P. Junqueira), oulusoy@cs.bilkent.edu.tr (Ö. Ulusoy).

0306-4573/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ipm.2010.12.007

entries fresh (Cambazoglu et al., 2010). In this work, our focus is on static caches, which essentially capture long-term patterns of data access.

In a typical search engine, there are five types of data items that are accessed or generated during the search process: query results, precomputed scores, posting lists, precomputed intersections of posting lists, and documents. In practice, it is possible to design a separate, homogenous static cache to hold items of a particular type, for each of the above-mentioned item types. To be brief, throughout the paper, we will refer to these caches as result, score, list, intersection, and document caches. In the literature, there are a number of works that investigate the performance of these caches separately (Fagni et al., 2006; Lempel & Moran, 2003; Markatos, 2001; Tomasic & Garcia-Molina, 1993). There are also several works that combine one or more cache types, forming multi-level caches (Baeza-Yates et al., 2007; Garcia, 2007; Long & Suel, 2005; Marin, Gil-Costa, & Gomez-Pantoja, 2010; Saraiva et al., 2001).

In this paper, we introduce a five-level static cache architecture that brings together all known cache types proposed in literature. We also describe a greedy heuristic for mixed-order caching in this architecture. The proposed heuristic iterates over all possible items that may be cached. At each iteration, the heuristic selects the item with the highest possible gain for caching. The gain of an item is computed by considering its past access frequency as well as estimated processing cost and storage overhead. After an item is selected for caching, gains of all remaining items that are affected by this caching decision are updated. The originality of our work is not only due to the proposed multi-level cache architecture, but also due to the way we model the dependencies between the items in different cache types.

We evaluate the proposed cache architecture through detailed and realistic simulations, in which we model the major costs incurred in query processing. In our simulations, we use a real-life query log and a document collection. Our findings reveal that the proposed multi-level cache architecture and the associated caching heuristic perform better than a policy that is based on dedicating fixed fractions of the entire cache space to different cache types.

The rest of the paper is organized as follows. In Section 2, we provide an overview of query processing in search engines and the involved costs. In Section 3, we describe the proposed multi-level cache architecture and the cost-based, mixed-order static caching heuristic. Section 4 describes the dataset and our experimental setup. Section 5 is devoted to evaluation of the proposed strategy. In Section 6, we review the related work. Section 7 provides the conclusions and point to future research directions.

2. Query processing overview

Web search engines are composed of multiple replicas of large search clusters. Each query is assigned to an individual search cluster, based on the current workload of clusters or based on a hash of the query string. A search cluster is composed of many nodes over which the documents are partitioned. Each node builds and maintains an index over its local document collection. All nodes in the cluster contribute to processing of a query.

The number of nodes in a search cluster is determined based on certain response time constraints that need to be satisfied. The number of cluster replicas is determined based on a constraint on the peak sustainable throughput. In addition, the utilization of search nodes is taken into account (Chowdhury & Pass, 2003).

Query processing involves a number of steps: issuing the query to search nodes, computing a partial result ranking in all nodes, merging partial rankings to obtain a global top-k result set, computing snippets for the top-k documents, and generating the final result page (see Fig. 1 for the workflow). In this study, we ignore the overheads due to network communication between the nodes and the overhead of the result merging step. These overheads are relatively insignificant. For

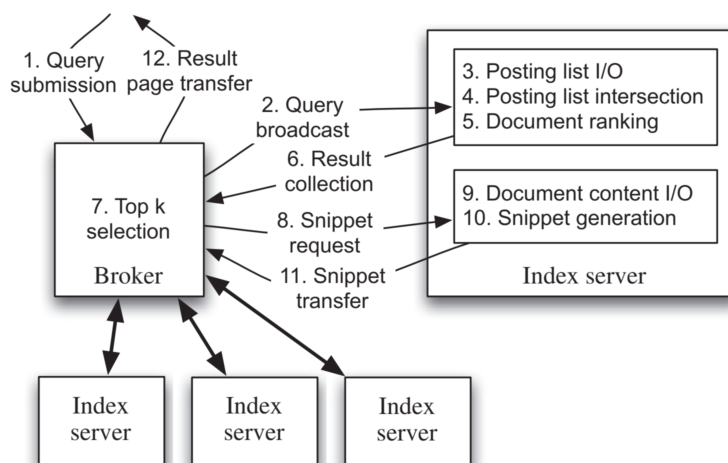

Fig. 1. The basic query processing workflow in web search engines.

instance, the cost of network transfer is estimated to be less than 1 ms in Baeza-Yates et al. (2007), a tiny fraction of the query processing cost, if the nodes are connected through a local area network. The cost of result merging, especially for low k values (e.g., 10), would not be more than a few milliseconds either.

We take into account the following steps, which incur relatively high processing or I/O overhead:

- *Step 1*: For all query terms, fetch the associated posting lists from the disk. This incurs I/O cost, denoted as C_{pl}.
- *Step 2*: Compute relevance scores for the documents in the intersection of fetched posting lists and select the top-k documents with the highest scores. This incurs CPU overhead, denoted as C_{rank}.
- *Step 3*: For the top-k documents identified, fetch the document data from the disk. This incurs I/O overhead, denoted as C_{doc}.
- *Step 4*: For the fetched documents, compute snippets. This incurs CPU overhead, denoted as C_{snip}.

We note that the search architecture we consider in our work slightly differs than those described in the literature in that there is not a single central broker that collects partial results form the index servers and merges them to generate the final query result. Instead, we assume that every search node acts both as a broker and as an index server. A node plays the role of a broker only for a subset of queries, i.e., in a sense, the role of the central broker is distributed over the entire search system. The node that serves as a broker for a particular query can be determined by using the MD5 hash of the query. This node is responsible for gathering and merging the partial results from index servers (including itself). Every search node executes all steps in query processing and accesses (or generates) five types of data items, namely lists, intersections, scores, documents, and results (in this order). Consequently, in our cache architecture, every search node maintains all five types of caches for the data local to itself.

3. Five-level static caching

3.1. Architecture

Herein, we describe a five-level cache architecture for static caching in search engines. In this architecture, the space reserved to each cache type is not individually constrained. Instead, there is a global capacity constraint that applies to all caches, i.e., all caches share the same storage space. Therefore, individual caches can continue to grow as long as their total size is below the global capacity constraint. Each cache stores a different type of (key, value) pair and provides saving for one or more of the query processing costs mentioned in Section 2. In Table 1, we list the five different cache types considered in this work, along with their (key, value) pairs and associated cost savings.

3.2. Cost-based mixed-order caching algorithm

We propose a simple greedy heuristic to fill the caches in the above-mentioned architecture. In this heuristic, five priority queues are maintained, one for each cache type, to prioritize the items of a certain type according to their gains. The heuristic consists of two steps. In the first step, the initial gains of items are computed and they are inserted into their respective priority queues. In the second step, the items with the highest gains are selected and gain updates are performed on the remaining items, in an iterative manner.

3.2.1. Initial gain computation

For each item that is candidate to be cached (e.g., query result, document, term), we compute the potential gain that would be obtained by caching that item, using the statistics in a previous query log. In an earlier work (Baeza-Yates et al., 2007), the gain computation is usually based on the ratio between the access frequency of the item and its size, i.e., the numerator is the observed frequency of requests for the item and the denominator is the space (in bytes) that the item would occupy in the cache. In our cost-based framework (Altingovde, Ozcan, & Ulusoy, 2009), the gain computation also involves a cost component that represents the saving achieved in query processing time (in ms) by caching the item (see Eq. (1)). In our case, for each item type, the corresponding cost saving is computed as shown in the fourth column of Table 1.

$$\text{Gain} = \frac{\text{Cost saving} \times \text{Frequency}}{\text{Size}}. \quad (1)$$

Table 1
Cache types and their cost savings.

Cache type	Key	Value	Cost saving
Result	Query	Snippets of top k documents (output of Step 4)	$C_{pl} + C_{rank} + C_{doc} + C_{snip}$
Score	Query	Relevance scores for top-k documents (output of Step 2)	$C_{pl} + C_{rank}$
Intersection	Set of term ids	Intersection of posting lists (intermediate output of Step 2)	$C_{pl} + C_{rank}$
List	Term id	Posting lists (data fetched in Step 1)	C_{pl}
Document	Document id	Raw document content (data fetched in Step 3)	C_{doc}

Here, an important assumption is that the past access frequencies of items accurately represent their future access frequencies. Although this assumption generally holds, for certain items, smoothing is needed. For instance, previous studies show that the frequency of query terms exhibit little variation in a reasonably long time period, e.g., a few months (Baeza-Yates et al., 2007). Thus, for list caching, the term access frequencies observed in earlier query logs would be a good evidence to rely on. On the other hand, for a given query stream, almost half of the queries are singleton (Baeza-Yates et al., 2007) (i.e., they appear only once in the entire query log), which implies that the past query frequencies may not exactly capture the future frequencies. This is especially true for infrequent queries. In a recent study, it is mentioned that "past queries that have occurred with some high frequency still tend to appear with a high frequency, whereas queries with relatively less frequency may appear even more sparsely, or totally fade away in the future" (Altingovde et al., 2009). A similar discussion about the estimation of future query frequencies for result caching is also provided in Gan and Suel (2009). In the latter work, it is reported that, for queries with a past frequency greater than 20, the future frequencies almost exactly follow the past observations. Thus, it is practical to construct an estimation table by using a separate query log for queries with a frequency value less than 20. In this paper, we follow the same practice, discussed in Section 5.1.

After their initial gains are computed, the items are inserted into their respective item priority queues based on these values. The head of an item priority queue at any time shows (potentially) the most cost-effective item to be cached from that particular cache type. We note that the gains are comparable across all queues since the gain of any item represents the total processing time saving that would be achieved if the item is cached.

3.2.2. Selection and gain update

This is an iterative step, in which a selection priority queue is maintained to locate the item with the highest expected gain among the item priority queues. This priority queue keeps the current best (head) item of the five item priority queues. Hence, its capacity is fixed to five items. At each iteration, the head of the selection queue is dequeued and permanently added into the cache. A new item of the same type (the new best item in the same item queue) is inserted into the selection queue. The iterations are repeated until the total storage size of the cached items reaches the cache capacity or no item remains in item priority queues.

During this process, caching of an item may affect the frequencies or cost savings of other items. In Fig. 2, we illustrate the dependencies among items of different types. In the figure, a directed edge from type T_i to T_j indicates that, whenever an item of type T_i is cached, the update operation on the edge label should be performed on the related items of type T_j. The update operation may reduce either the frequency or the cost saving. For example, if the result set of a query is cached, the frequencies of the posting lists for the terms appearing in the query are reduced (as in Skobeltsyn, Junqueira, Plachouras, & Baeza-Yates (2008)). In contrast, whenever the posting list of a term is cached, the cost saving for the results of the queries including that term is reduced by the cost of fetching that list, i.e., C_{pl}, for that list. Similar trade-offs exist between different items and cache types.

4. Dataset and setup

As the document collection, we use about 2.2 million web pages crawled from the open directory project[1]. The collection is indexed without stemming and stopword removal. The uncompressed index file, which includes only a document identifier and a term frequency per posting, takes 3.2 GB on disk.

As the query log, we use a subset of the AOL log (Pass, Chowdhury, & Torgeson, 2006), which contains around 20 million queries issued during a period of 12 weeks. Our training query set, which is used to estimate items' access frequencies, includes one million queries from the first 6 weeks. The test query set contains an equal number of queries from the second 6 weeks. We verified the compatibility of the document collection and query log in an earlier work (Ozcan, Altingovde, & Ulusoy, in press).

Query terms are case-folded and sorted in alphabetical order. We also removed punctuation and stopwords. Queries are processed in conjunctive mode, i.e., all query terms appear in documents matching the query. In performance evaluations, we omitted queries that do not match any documents.

Queries in the training set are used to compute the item frequencies for results (equivalently, scores), lists, and intersections. Without loss of generality, we consider intersections only for the term pairs that appear in queries. Access frequencies for documents are obtained from query results.

In our simulation runs, we consider a distributed search cluster where each search node caches all five types of data items. However, while computing costs of data items, we restrict our experiments to a single search node. This choice is reasonable since we neglect network and result merging overheads, as discussed in Section 2. Furthermore, since query processing in a search cluster is embarrassingly parallel, query processing times on a one-node system with Q queries and D documents are comparable to those on a K-node system with $K \times Q$ queries and $K \times D$ documents. In this respect, our choice of a 3.2 GB dataset is not arbitrary, but intentional. We estimate that the data size we use in our simulations roughly corresponds to the size of data that would be indexed by a single node in a typical search cluster. Nevertheless, in what follows, we discuss how the costs associated with the aforementioned four steps in query processing (see Section 2) are computed to

[1] Open directory project, available at http://www.dmoz.org.

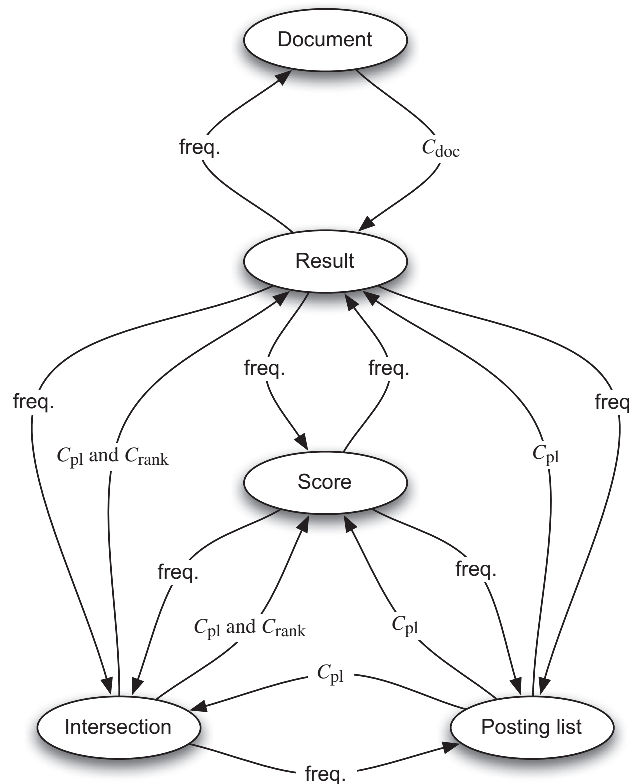

Fig. 2. Update dependencies in the mixed-order caching heuristic. Vertices represent item types. Edge labels show the cost components that are decreased in gain updates.

model the workflow of a node in a realistic manner. Fig. 3 illustrates the workflow used by our simulator and also illustrates the interaction between different cache components.

In a real-life search cluster, the processing cost of a query depends on all nodes, since partial results (i.e., steps 1 and 2) have to be computed at each node. However, under a uniform workload assumption (i.e., no load imbalance), the execution time on a node would almost be the same as the time on other nodes. Thus, in the simulation runs, a node considers its own processing time for the query and assumes that it would not need to wait for other nodes, i.e., partial results from those would also be ready at the same time.[2] Subsequently, in our setup, computation of C_{pl} and C_{rank} values for finding the partial top k results in a single node is a realistic choice.[3]

Once partial results are computed and sent to the broker node for a particular query, this node fetches the documents and generates snippets to create the final result. For this stage of processing (i.e., steps 3 and 4), we again consider the local execution time of each computer. However, during snippet generation, a node itself may not need to take into account the time for creating snippets for all top 10 results. This is because the documents are partitioned into nodes and, for a practically large number of servers in a cluster (e.g., in the order of hundreds (Dean, 2009)), it is highly likely that each document in the final top-10 set will be located in a different node. Thus, we can model the cost of document access and snippet generation steps for only one document, i.e., presumably for the highest ranking document in its partial result set. In other words, we assume that a node would contribute only its top-scoring partial result to the final top 10 results. This is a reasonable assumption, given the high number of nodes in a cluster and the desire of search engines for providing diversity in their top-ranked results.

In the simulations, we assume that the node at hand experiences the cost of producing partial results for top 10 documents and then producing the snippets for the highest scoring document (d_{top}). We believe that this setup reflects the costs that would be experienced by each node in a search cluster as close as possible. The cost values associated with each query step is computed using the formulas shown in Table 2. Note that there is a subtle detail in the computation of C_{rank} for an intersection item. For an intersection of two lists I_1 and I_2, the total posting count of a query is computed as $|I_1| + |I_2| - |I_1 \cap I_2|$ since the gain in this case entirely avoids processing of lists $|I_1|$ and $|I_2|$.

[2] In practice, a search engine may enforce an upper bound on the execution time at the nodes so that the execution terminates when this threshold is reached (Cambazoglu et al., 2010).

[3] Note that search engines usually generate 10 results for each result page (Lempel & Moran, 2003). Hence, we essentially focus on generating 10 results at a time.

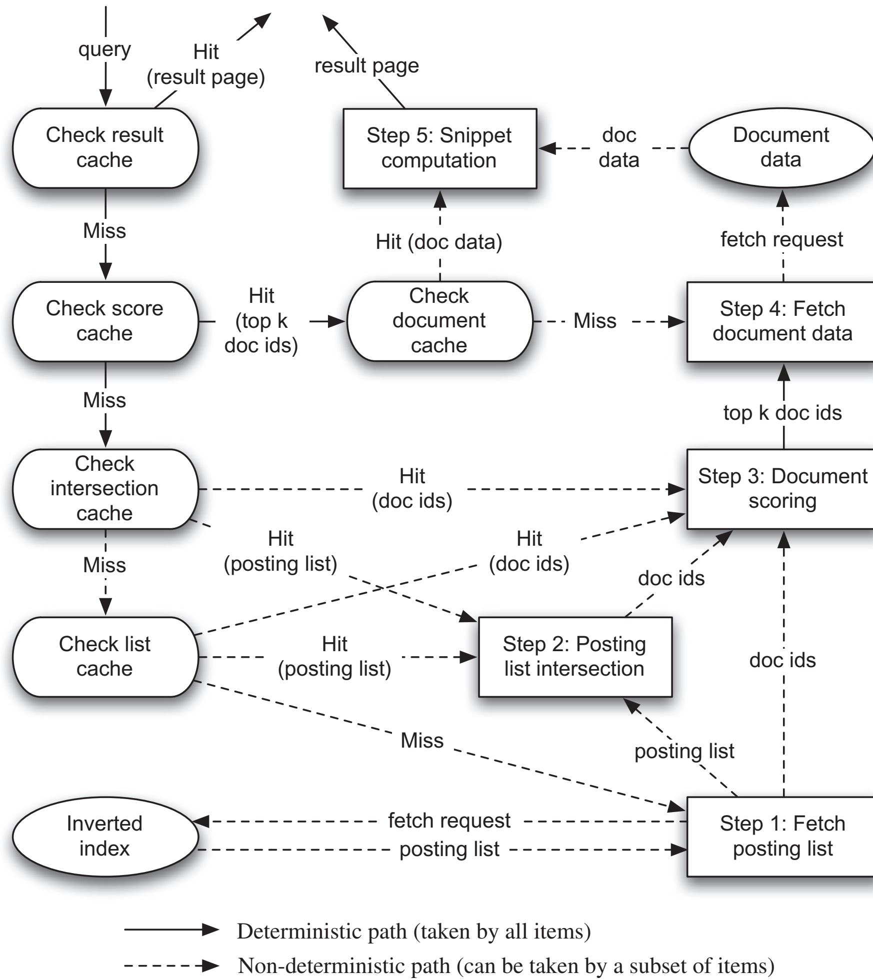

Fig. 3. The workflow used by the simulator in query processing.

Table 2
Cost computations in the cache simulation.

Cost	Notation	Computation
Posting lists access	C_{pl}	$D_{seek} + D_{rotation} + D_{read} \times \left\lceil \frac{\lvert I_i \rvert \times S_p}{D_{block}} \right\rceil$
Ranking	C_{rank}	$CPU_{scoring} \times \sum_{t_i \in q} (\lvert I_i \rvert \times S_p)$
Document access	C_{doc}	$D_{seek} + D_{rotation} + D_{read} \times \left\lceil \frac{\lvert d_{top} \rvert}{D_{block}} \right\rceil$
Snippet generation	C_{snip}	$CPU_{snippet} \times \lvert d \rvert$

Parameters used in simulations are given in Table 3. The default parameters are determined empirically or by consulting the literature. In particular, parameters regarding the disk access times are figures for a modern disk (Ramakrishnan & Gehrke, 2003). Storage parameters are based on typical assumptions in the literature, i.e., a posting size is usually assumed to take 8 bytes (4 bytes for document id and term frequency). The scoring time is computed by running experiments with the publicly available Terrier system (Ounis et al., 2005) on our dataset. Finally, we assume a rather simplistic snippet generation mechanism (highlighting the first appearance of the query words as well as a couple of words surrounding them in a document) and set the snippet computation time to a fraction of the scoring time.

5. Experimental evaluation

In this section, we evaluate the performance of several cache architectures in terms of the total query processing time. In particular, we first compare the performance of each cache type separately. Next, we discuss the performance of some

Table 3
Simulation parameters.

Parameter	Type	Notation	Default value
Result item size	Storage	S_r	512 bytes
Score item size	Storage	S_s	8 bytes
Posting size	Storage	S_p	8 bytes
Disk seek	Disk	D_{seek}	8.5 ms
Rotational latency	Disk	D_{rotation}	4.17 ms
Disk block read	Disk	D_{read}	4.883 ns
Block size	Disk	D_{block}	512 bytes
Cache lookup cost	CPU	$\mathrm{CPU}_{\mathrm{lookup}}$	40 ns
Scoring cost per posting	CPU	$\mathrm{CPU}_{\mathrm{scoring}}$	200 ns
Snippet generation cost per byte	CPU	$\mathrm{CPU}_{\mathrm{snippet}}$	10 ns
Requested query results	Other	k	10

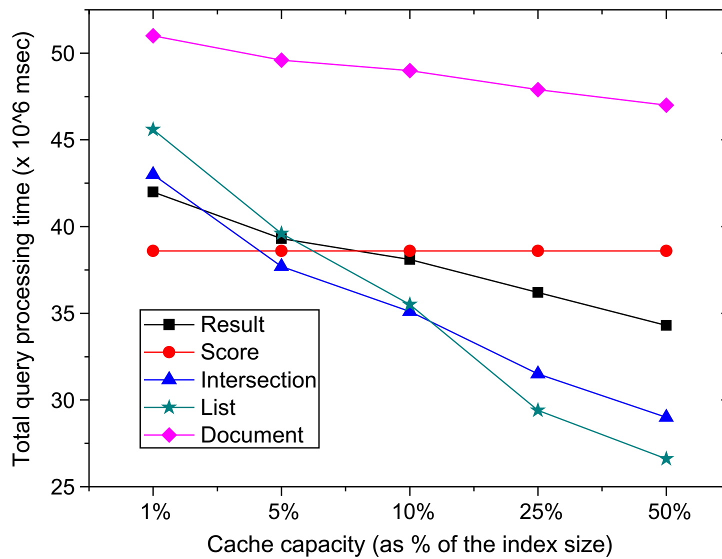

Fig. 4. Performance of one-level cache architectures.

previously proposed two- and three-level cache architectures, where each cache type is reserved a fixed portion of the available cache space. Finally, we evaluate the performance of our five-level cache architecture with mixed-order caching strategy and show that it is superior to others.

5.1. Performance of single-level cache architectures

In Fig. 4, we show the total query processing time versus cache size (ranging from 1% to 50% of the full index size in bytes), separately for each cache type. As expected, for the smallest cache size (i.e., 1% of the index), the performance of the score cache is the best, and the result cache is the runner-up. In this case, it is not possible to cache all non-tail queries by the result cache. The processing time achieved by the intersection cache is better than that of the list cache, and the document cache is the worst. However, as the cache size grows, the list cache becomes the most efficient choice as the terms are shared among many queries. The intersection cache, which is shared by fewer number of queries, performs better at the beginning, but then becomes inferior to the list cache as the cache size exceeds 10% of the index. For the larger cache sizes, the result cache is better than the score cache, but cannot compete with the list cache, as also discussed in Baeza-Yates et al. (2007). Finally, our findings show that caching only documents is not a feasible choice at all for a search engine, as fetching posting lists are much more expensive than the former.

In the experiments mentioned above, the caching decision for a particular item is given by using the gain function (Eq. (1)), where the frequency of an item is simply set to its past access frequency observed in the training query log. However, the frequency values observed in the training log are not necessarily true indicators of the future frequencies for some of the item types, like result pages (Altingovde et al., 2009). This can be explained by the fact that, for all item types relevant to the query processing process, the access frequencies follow a power-law distribution.[4] Thus, while relatively few items are

[4] This is also verified for our query logs in a separate set of experiments that are not reported here to save space.

repeated many times in the log, majority of the items are asked rarely, or only once. This is especially true for result pages (and scores), as previous works show that almost 50% of the queries in a given query stream are singleton. For such rarely encountered items, future access frequencies may significantly differ than the training frequencies. This may diminish the performance of the static cache.

As a remedy to this problem, following the practice of Gan and Suel (2009), we construct a table to experimentally obtain future frequency values corresponding to a small set of past frequency values, i.e., up to 20 (Gan & Suel, 2009) show that queries with past access frequency higher than 20 achieves almost the same future frequency. To this end, we use another subset of the AOL log different from the training and test logs employed throughout this section. This validation log is split into two parts, representing past and future observations. For each frequency value $f < 20$, we check the future frequency values of those queries that have frequency f in the past. The average of such frequency values is recorded as the future frequency value for f. The same idea is applied for the frequency of query terms (i.e., list items).

In Table 4, to save space, we only report the frequency values obtained for past frequencies smaller than 5, for both queries and query terms. We observe that the frequency values for query terms are more stable in comparison to query frequencies. This suggests that frequency correction is more crucial for query result caching than posting list caching.

In Fig. 5, we show the effect of frequency correction on result caching performance. Here, while computing the gains, we use the estimated future frequency value whenever the training frequency of the query is found to be less than 20. According to the figure, the improvement is higher for medium cache sizes (5% or 10% of the index size). As the cache size grows, the gains become smaller since the majority of result items can fit into the cache (for instance, a static cache with a capacity equal to 50% of the full index size can store more than 80% of all results). In this case, the remaining result pages are for the singleton queries, all of which would have the same estimated future frequency (i.e., 0.15 in Table 4) and the same relative caching order, as before. On the other hand, our experiments using corrected frequencies for list items do not yield any improvement. Hence, in the rest of the experiments, we employ frequency correction only for result items.

5.2. Performance of two-level and three-level cache architectures

In most work in literature, a two-level cache architecture that involves a result and a list cache is considered. An extension of this architecture is three-level caching, where an intersection cache is introduced. Note that, in Long and Suel (2005), the intersection cache is assumed to be on disk, whereas we assume that all caches are in the memory.

In Fig. 6, we demonstrate the performance of two-level caching for various divisions of cache space between the two caches. Our findings confirm those of Baeza-Yates et al. (2007) in that the minimum query processing time is obtained when almost 20% of the cache space is devoted to results and the rest is used for the lists. Note that, while filling the cache, the frequencies of the cached results are reduced from those of the lists, as recommended in Baeza-Yates et al. (2007).

We show the comparison of the best performing two-level cache with two-level mixed-order cache in Fig. 7. It is seen that mixed-order caching considerably improves with the frequency estimation. On the other hand, the performance improvement in the baseline two-level cache due to frequency estimation is minor. Nevertheless, mixed-order two-level cache with frequency estimation achieves as good performance as the baseline cache. Note that, there is no tuning overhead in mixed-order caching and the division ratio of cache space among results and lists is decided by the algorithm adaptively, which is an advantage over the baseline.

In Fig. 8, we show the same plot for three-level caches. Here, the best performance for three-level caches is obtained when 20% of the cache space is reserved for result items, 20% for intersection items, and the remaining 60% for list items. Note that, determining the best division of the cache capacity among these item types requires experimenting with a large number of combinations; and these are not reported here to save space. On the other hand, as before, our mixed-order caching approach achieves the same performance as the best case of the baseline algorithm without any tuning.

5.3. Performance of five-level cache architecture with mixed-order algorithm

In this section, we analyze the performance of our five-level cache architecture. To the best of our knowledge, there is no work in the literature that investigates static caching of these five item types (results, scores, intersections, lists and documents) in a single framework. As in previous sections, we reserve a fixed fraction of the cache space for each item type in the baseline five-level cache. However, as it might be expected, the tuning of the cache space splitting becomes a very tedious job in the five-level architecture.

Table 4
Future frequency values for past frequencies smaller than 5.

Past query (or term) frequency	Future query frequency	Future query term frequency
1	0.15	0.73
2	0.66	1.46
3	1.53	2.27
4	2.47	3.15
5	3.48	3.93

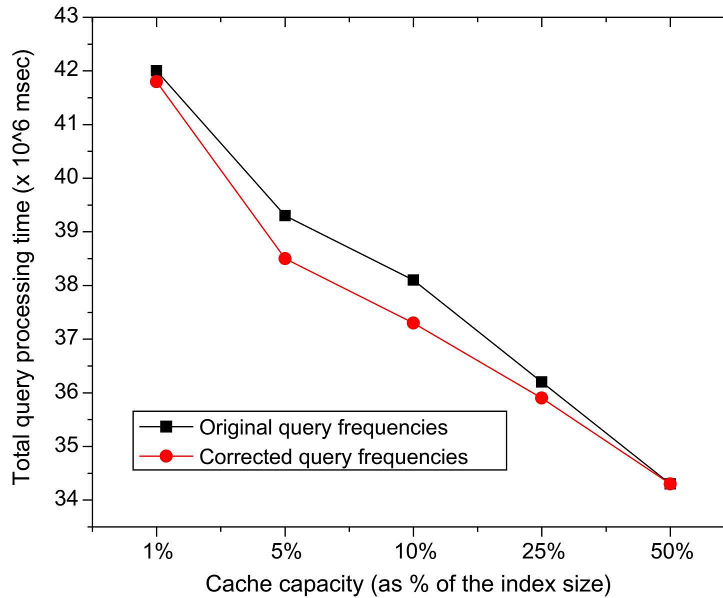

Fig. 5. The effect of frequency correction on the result cache performance.

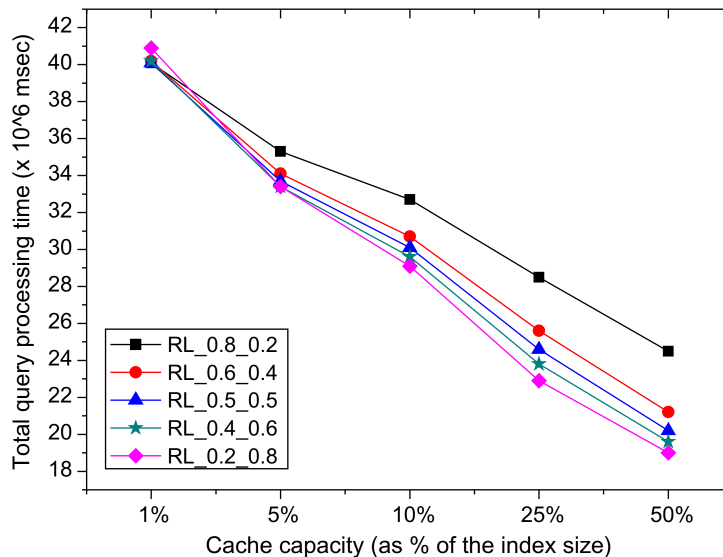

Fig. 6. The performance of two-level caches for varying split ratios of cache space between result (R) and list (L) items.

In the five-level mixed-order caching experiments, we realized that score items turn out to be the most profitable item type according to the greedy selection approach. This is because a score item provides significant cost savings (i.e., all query processing costs, other than the snippet generation cost, are eliminated) while consuming very small storage space (see Table 3). Thus, even for very small cache sizes (e.g., 1% of the index), the majority of the score items are selected for caching before any other item types. This causes a devastating effect on the item types that are related to scores (see Fig. 2). Since almost all score items are brought into the cache, the frequency values of the result items, intersection items, and list items are significantly reduced. This makes documents the only profitable item type to be cached after the scores. Clearly, a cache configuration of mostly scores and documents would yield a poor performance.

As a remedy, we still allow the greedy approach to cache the score items before the others, but do not permit those score items to affect frequencies of related items. Our decision is based on the observation that all possible score items require a tiny fraction of the cache space, i.e., in our case, all score items take about only 1% of the index size. Hence, it is reasonable to keep them in the cache as long as they are not allowed to totally reset frequencies of related items.

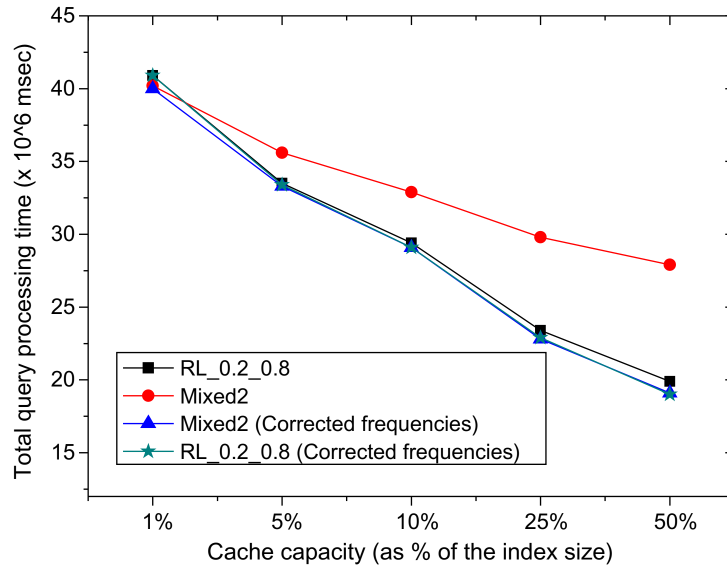

Fig. 7. The comparison of baseline two-level cache with two-level mixed-order cache.

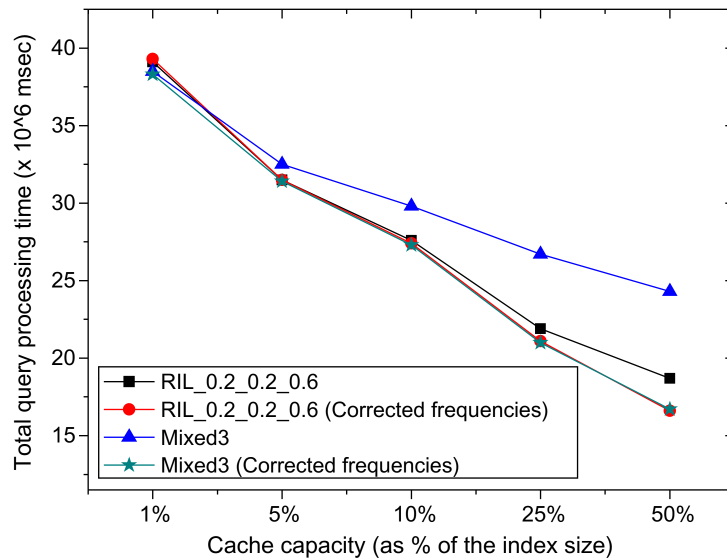

Fig. 8. The comparison of baseline three-level cache with three-level mixed-order cache.

In Fig. 9, we provide the comparison of our five-level mixed-order caching approach with the best results obtained for the baseline two-level and three-level caches, proposed in literature. We also obtain a five-level baseline cache by experimentally determining the fraction of the cache space that should be devoted to each item type. For this latter case, the best results are obtained when 18% of cache space is reserved for result items, 2% for score items, 15% for intersection items, 60% for list items, and 5% for document items. Our findings reveal that mixed-order caching outperforms the baselines at all cache sizes. In particular, our approach yields up to 18.4% and 9.1% reduction in total query processing time against the two-level and three-level caches, respectively. Furthermore, the highest improvement against the baseline five-level cache is 4.8%. The improvements are statistically significant at 0.05 significance level.

6. Related work

Caching is a well-explored topic in many contexts, such as operating systems (Smith, 1982), databases (Effelsberg & Haerder, 1984), and web servers (Podlipnig & Böszörmenyi, 2003). An early work on caching related to the IR systems is

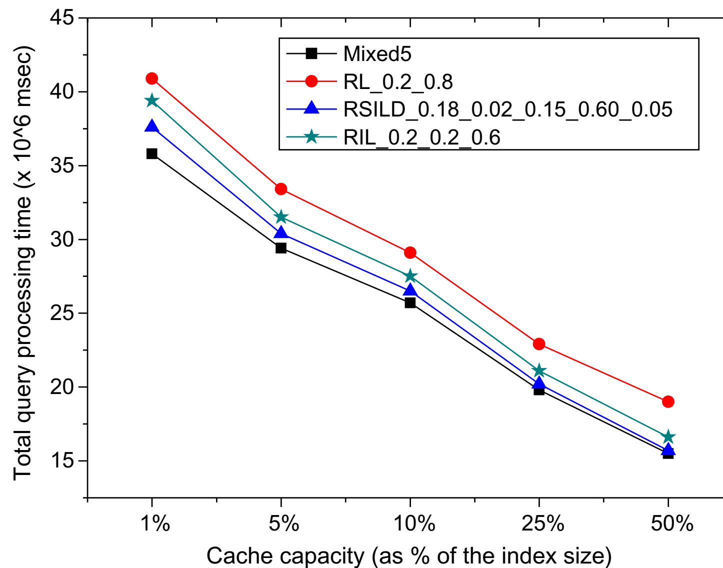

Fig. 9. The comparison of baseline two-, three-, and five-level caches with five-level mixed-order cache.

by (Simpson & Alonso, 1987). The caching discussed in that work (also, in Alonso, Barbara, & Garcia-Molina (1990)), however, is about client-side caches that try to reduce the overhead of getting the data through the network.

In Table 5, we classify the previous studies that essentially focus on caching techniques for large scale IR systems and search engines. We group those works in terms of the cache component focused on and also whether the caching strategy employed is static or dynamic. Although many researchers have investigated the performance of individual cache types or their combinations up to three different cache types, we are not aware of any works that consider all possible cache components in a static caching framework.

An early proposal that discusses posting list caching in an information retrieval system is the work of Tomasic and Garcia-Molina (1993). The authors discuss caching in the context of a shared-nothing IR system, using LRU as the policy in their dynamic cache replacement algorithm. Jonsson, Franklin, and Srivastava (1998) discuss replacement strategies in the context of query refinement under a single user assumption. Zhang, Long, and Suel (2008) evaluate several posting list caching strategies and study the performance benefits of combining list compression and caching.

To our knowledge (Markatos, 2001) is the first to discuss query result caching in search engines. His work shows locality of queries in search engine query logs, which is also shown in Xie and O'Hallaron (2002), and provides a comparison between static and dynamic result caching approaches. Lempel and Moran (2003) introduce a probabilistic caching algorithm for a result cache, taking into account results for the follow-up queries, i.e., the queries generated by clicking the next button. The main purpose of their work is to accurately estimate, using offline statistics from a query log, how many "next pages" should be cached upon a query. Ozcan et al. (2008) discuss a number of selection strategies to fill a static result cache.

Table 5
Classification of earlier works on caching in web search engines.

Type	Static	Dynamic
Result	Markatos (2001), Baeza-Yates and Saint-Jean (2003), Fagni et al. (2006), Ozcan et al. (2008)	Markatos (2001), Saraiva et al. (2001), Lempel and Moran (2003), Long and Suel (2005), Fagni et al. (2006), Gan and Suel (2009), Puppin et al. (2010), Cambazoglu et al. (2010)
Score	–	Fagni et al. (2006)
Intersection	–	Long and Suel (2005)
List	Baeza-Yates and Saint-Jean (2003)	Tomasic and Garcia-Molina (1993), Jonsson et al. (1998), Saraiva et al. (2001), Long and Suel (2005), Zhang, Long, and Suel (2008)
Document	–	–

Saraiva et al. (2001) describe a two-level cache architecture that combines result and list caching and compare it with caching only results or lists. Increase in query processing throughput is achieved by a practical implementation that employs LRU as the replacement policy. A similar idea is also discussed in Baeza-Yates and Saint-Jean (2003). However, that work considers static caching while the previous is for dynamic caching.

Long and Suel (2005) propose caching of previously computed intersections of posting lists. Then a three-level dynamic cache architecture that combines result, list, and intersection caches is described. Garcia (2007) proposes a dynamic cache architecture that stores multiple item types based on their disk access costs.

Fagni et al. (2006) discuss a hybrid architecture, referred to as static-dynamic cache, which combines static and dynamic result caches. The superiority of this hybrid cache to purely static or dynamic caches has been shown by experiments. The idea of having a score cache (called the DocID cache in the work) is also mentioned in this work.

The extensive study of Baeza-Yates et al. (2007) covers many issues in static and dynamic result/list caching. The authors propose a simple heuristic that takes into account the storage size of lists when making caching decisions. Gan and Suel (2009) introduce dynamic caching mechanisms that take into account also the execution costs of queries. Concurrently, Altingovde et al. (2009) propose similar cost-aware strategies for static result caching. Puppin, Perego, Silvestri, and Baeza-Yates (2010) proposes a novel incremental cache architecture for collection selection architectures. Cambazoglu et al. (2010) introduces the staleness problem in large result caches in web search engines, where inverted indexes are frequently updated.

In a concurrent work, which is very related to ours, Marin et al. (2010) propose a hierarchical cache architecture using different levels of cache. This work differs from ours in that the dependency between the caches is not explicitly modeled as we do by means of a greedy heuristic. Moreover, their architecture does not contain a document cache, but uses a location cache, which is not needed in our architecture. Finally, the broker node and search nodes are assigned certain cache types, whereas in our architecture there is no central broker and hence all caches are located in search nodes.

7. Conclusion

In this work, we have presented a hybrid static cache architecture that brought together different types of caches that are independently managed in web search engines. We also proposed a greedy heuristic for prioritization of different data items for placement in the cache. This heuristic has taken into account the inter-dependencies between processing costs and access frequencies of data items. The proposed approach is shown to provide efficiency improvements compared to architectures in which caches are manipulated independent of each other. In particular, cost-based mixed-order cache yields up to 18.4% and 9.1% reduction in total query processing time against the two-level and three-level caches, respectively. Furthermore, the highest improvement against the baseline five-level cache is 4.8%.

There are three possible extensions to our work. First, our work considers a tightly coupled cache architecture, in which all cache types are stored on a single search node. In practice, the cache may be partitioned across different nodes (e.g., result and score caches on a proxy node and the rest on search cluster nodes). This kind of a loosely coupled architecture may be worth investigation as cost updates will be very different. Second, the inter-dependency between different caches need to be investigated for the dynamic caching setting. Although this has already taken some attention (Garcia, 2007; Marin et al., 2010), it is still not clear how cost updates can be performed on-the-fly and what data structures are needed to perform these updates efficiently. A third research direction is to consider our hybrid cache architecture in a setting where cache entries are selectively refreshed.

Acknowledgements

This work is partially supported by the COAST project (ICT-248036), funded by the European Community. This work is also partially supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) by the Grant number 108E008.

References

Alonso, R., Barbara, D., & Garcia-Molina, H. (1990). Data caching issues in an information retrieval system. *ACM Transactions on Database Systems, 15*(3), 359–384.

Altingovde, I. S., Ozcan, R., & Ulusoy, Ö. (2009). A cost-aware strategy for query result caching in web search engines. In *Proceedings of the 31st European conference on information retrieval* (pp. 628–636).

Baeza-Yates, R., & Saint-Jean, F. (2003). A three level search engine index based in query log distribution. In *Proceedings of the 10th international symposium on string processing and information retrieval* (pp. 56–65).

Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., & Silvestri, F. (2007). The impact of caching on search engines. In *Proceedings of the 30th annual international ACM SIGIR conference on research and development in information retrieval* (pp. 183–190).

Cambazoglu, B. B., Junqueira, F. P., Plachouras, V., Banachowski, S., Cui, B., Lim, S., et al. (2010). A refreshing perspective of search engine caching. In *Proceedings of the 19th international conference on world wide web* (pp. 181–190).

Chowdhury, A., & Pass, G. (2003). Operational requirements for scalable search systems. In *Proceedings of the 12th international conference on information and knowledge management* (pp. 435–442).

Dean, J. (2009). Challenges in building large-scale information retrieval systems (invited talk). In *Proceedings of the 2nd ACM international conference on web search and data mining* (p. 1).

Effelsberg, W., & Haerder, T. (1984). Principles of database buffer management. *ACM Transactions on Database Systems, 9*(4), 560–595.

Fagni, T., Perego, R., Silvestri, F., & Orlando, S. (2006). Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems, 24*(1), 51–78.

Gan, Q., & Suel, T. (2009). Improved techniques for result caching in web search engines. In *Proceedings of the 18th international conference on world wide web* (pp. 431–440).

Garcia, S. (2007). Search engine optimisation using past queries. PhD Thesis. RMIT University.

Jonsson, B. T., Franklin, M. J., & Srivastava, D. (1998). Interaction of query evaluation and buffer management for information retrieval. In *Proceedings of the 1998 ACM SIGMOD international conference on management of data* (pp. 118–129).

Lempel, R., & Moran, S. (2003). Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th international conference on world wide web* (pp. 19–28).

Long, X., & Suel, T. (2005). Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international conference on world wide web* (pp. 257–266).

Marin, M., Gil-Costa, V., & Gomez-Pantoja, C. (2010). New caching techniques for web search engines. In *Proceedings of the 19th ACM international symposium on high performance distributed computing* (pp. 215–226).

Markatos, E. P. (2001). On caching search engine query results. *Computing Communications, 24*(2), 137–143.

Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., & Johnson, D. (2005). Terrier information retrieval platform. In *Proceedings of the 27th European conference on information retrieval* (pp. 517–519).

Ozcan, R., Altingovde, I. S., & Ulusoy, Ö. (2008). Static query result caching revisited. In *Proceeding of the 17th international conference on world wide web* (pp. 1169–1170).

Ozcan, R., Altingovde, I.S., & Ulusoy, Ö. (in press). Cost-aware strategies for query result caching in web search engines. *ACM Transactions on the Web*.

Pass, G., Chowdhury, A., & Torgeson, C. (2006). A picture of search. In *Proceedings of the 1st international conference on scalable information systems*.

Podlipnig, S., & Böszörmenyi, L. (2003). A survey of web cache replacement strategies. *ACM Computing Surveys, 35*(4), 374–398.

Puppin, D., Perego, R., Silvestri, F., & Baeza-Yates, R. (2010). Tuning the capacity of search engines: Load-driven routing and incremental caching to reduce and balance the load. *ACM Transactions on Information Systems, 28*(2), 1–36.

Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). McGraw-Hill.

Saraiva, P. C., de Moura, E. S., Ziviani, N., Meira, W., Fonseca, R., & Ribeiro-Neto, B. (2001). Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th annual international ACM SIGIR conference on research and development in information retrieval* (pp. 51–58).

Simpson, P., & Alonso, R. (1987). Data caching in information retrieval systems. In *Proceedings of the 10th annual international ACM SIGIR conference on research and development in information retrieval* (pp. 296–305).

Skobeltsyn, G., Junqueira, F., Plachouras, V., & Baeza-Yates, R. (2008). ResIn: A combination of results caching and index pruning for high-performance web search engines. In *Proceedings of the 31st international ACM SIGIR conference on research and development in information retrieval* (pp. 131–138).

Smith, A. J. (1982). Cache memories. *ACM Computing Surveys, 14*(3), 473–530.

Tomasic, A., & Garcia-Molina, H. (1993). Caching and database scaling in distributed shared-nothing information retrieval systems. *ACM SIGMOD Record, 22*(2), 129–138.

Wang, J. (1999). A survey of web caching schemes for the Internet. *ACM SIGCOMM Computer Communication Review, 29*(5), 36–46.

Xie, Y., & O'Hallaron, D. (2002). Locality in search engine queries and its implications for caching. In *Proceedings of the 6th European conference on research and advanced technology for digital libraries* (pp. 1238–1247).

Zhang, J., Long, X., & Suel, T. (2008). Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international world wide web conference* (pp. 387–396).