Cost-Aware Strategies for Query Result Caching in Web Search Engines

RIFAT OZCAN, ISMAIL SENGOR ALTINGOVDE, and ÖZGÜR ULUSOY, Bilkent University

Search engines and large-scale IR systems need to cache query results for efficiency and scalability purposes. Static and dynamic caching techniques (as well as their combinations) are employed to effectively cache query results. In this study, we propose cost-aware strategies for static and dynamic caching setups. Our research is motivated by two key observations: (i) query processing costs may significantly vary among different queries, and (ii) the processing cost of a query is not proportional to its popularity (i.e., frequency in the previous logs). The first observation implies that cache misses have different, that is, nonuniform, costs in this context. The latter observation implies that typical caching policies, solely based on query popularity, can not always minimize the total cost. Therefore, we propose to explicitly incorporate the query costs into the caching policies. Simulation results using two large Web crawl datasets and a real query log reveal that the proposed approach improves overall system performance in terms of the average query execution time.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms: Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Query result caching, Web search engines

ACM Reference Format:
Ozcan, R., Altingovde, I. S., and Ulusoy, Ö. 2011. Cost-aware strategies for query result caching in web search engines. ACM Trans. Web 5, 2, Article 9 (May 2011), 25 pages.
DOI = 10.1145/1961659.1961663 http://doi.acm.org/10.1145/1961659.1961663

1. INTRODUCTION

Caching is one of the most crucial mechanisms employed in large-scale information retrieval (IR) systems and Web search engines (SEs) for efficiency and scalability purposes. Search engines essentially cache the query result pages and/or posting lists of terms that appear in the queries. A large-scale search engine would probably cache both types of information at different levels of its architecture (e.g., see Baeza-Yates and Saint-Jean [2003]).

A search engine may employ a static or dynamic cache of such entries, or both [Fagni et al. 2006]. In the static case, the cache is filled with entries as obtained from earlier logs of the Web SE and its content remains intact until the next periodical update. In

This work was supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) by grant numbers 108E008 and 110E135.
A preliminary version of this article appeared in *Proceedings of the 31st European Conference on Information Retrieval*, Lecture Notes in Computer Science, vol. 3478, Springer-Verlag.
Authors' addresses: R. Ozcan, I. S. Altingovde, and Ö. Ulusoy, Computer Engineering Department, Bilkent University, Ankara, 06800, Turkey; email: {rozcan, ismaila, oulusoy}@cs.bilkent.edu.tr.

DOI 10.1145/1961659.1961663 http://doi.acm.org/10.1145/1961659.1961663

the dynamic case, the cache content changes dynamically with respect to the query traffic, as new entries may be inserted and existing entries may be evicted. For both cases, cache entries may be the query results and/or term lists.

In the context of Web SEs, the literature involves several proposals concerning what and how to cache. However, especially for query result caching, the cost of a *miss* is usually disregarded, and all queries are assumed to have the same cost. In this article, we essentially concentrate on the caching of query results and propose cost-aware strategies that explicitly make use of the query costs while determining the cache contents.

Our research is motivated by the following observations: First, queries submitted to a search engine have significantly varying costs in terms of several aspects (e.g., CPU processing time, disk access time, etc.). Thus, it is not realistic to assume that all cache misses would incur the same cost. Second, the frequency of the query is not an indicator of its cost. Thus, caching policies solely based on query popularity may not always lead to optimum performance, and a cost-aware strategy may provide further gains.

In this study, we start by investigating the validity of these observations for our experimental setup. To this end, it is crucial to model the query cost in a realistic and accurate manner. Here, we define query cost as the sum of the actual CPU execution time and the disk access cost, which is computed under a number of different scenarios. The former cost, CPU time, involves decompressing the posting lists, computing the query-document similarities and determining the top-N document identifiers in the final answer set. Obviously, CPU time is independent of the query submission order, that is, can be measured in isolation per query. On the other hand, disk access cost involves fetching the posting lists for query terms from the disk, and depends on the current content of the posting list cache and the order of queries. In this article, we compute the latter cost under three realistic scenarios, where either a quarter, half, or full index is assumed to be cached in memory. The latter option, storing the entire index in memory, is practiced by some industry-scale search engines (e.g., see Dean [2009]). For this case, we only consider CPU time to represent the query cost, as there is no disk access. The former option, caching a relatively smaller fraction of the index, is more viable for medium-scale systems or memory-scarce environments. In this study, we also consider disk access costs under such scenarios while computing the total query processing cost. The other cost factors, namely, network latency and snippet generation costs, are totally left out, assuming that these components would be less dependent on query characteristics and would not be a determining factor in the total cost.

Next, we introduce cost-aware strategies for the static, dynamic, and hybrid caching of query results. For the static caching case, we combine query frequency information with query cost in different ways to generate alternative strategies. For the dynamic case, we again incorporate the cost notion into a typical frequency-based strategy in addition to adapting some cost-aware policies from other domains. In the hybrid caching environment, a number of cost-aware approaches developed for static and dynamic cases are coupled. All these strategies are evaluated in a realistic experimental framework that attempts to imitate the query processing of a search engine, and are shown to improve the total query processing time over a number of test query logs.

The contributions of this article are summarized as follows:

— First, we extend our preliminary work [Altingovde et al. 2009] that have introduced a cost-aware strategy only for the static caching. To this end, we propose a cost-aware counterpart of the static caching method that we have discussed in another

work [Ozcan et al. 2008a]. The latter method takes into account the stability of query frequency in time, and can outperform typical frequency-based caching.
— Second, we introduce two cost-aware caching policies for dynamic query result caching. Furthermore, we adapt several cost-aware policies from other domains.
— Third, we also evaluate the performance of cost-aware methods in a hybrid caching environment (as proposed in Fagni et al. [2006]), in which a certain part of the cache is reserved for static caching and the remaining part is used for the dynamic cache.
— Finally, we experimentally evaluate caching policies using two large Web crawl datasets and real query logs. Our cost function, as discussed above, takes into account both actual CPU processing time that is measured in a realistic setup with list decompression and dynamic pruning, and disk access time computed under several list caching scenarios. Our findings reveal that the cost-aware strategies improve overall system performance in terms of the total query processing time.

The rest of the article is organized as follows: In the next section, we briefly review the related work in the literature. In Section 3, we provide the experimental setup that includes the characteristics of our datasets, query logs and computing resources. Section 4 is devoted to a cost analysis of query processing in a Web SE. The cost-aware static, dynamic, and hybrid caching strategies are discussed in Section 5, and evaluated in Section 6. The experimental results are summarized in Section 7. We conclude and point to future research directions in Section 8.

2. RELATED WORK

Caching is a well-known technique applied in several domains, such as disk page caching in operating systems (OSs) and Web page caching in proxies. Its main goal is to exploit the temporal and spatial localities of the requests in the task at hand. Operating systems attempt to fill the cache with frequently (and/or recently) used disk pages and expect them to be again requested in the near future. The same idea applies to caching in the context of Web SEs; query result pages and/or posting lists of terms that appear in the queries are cached.

The availability of query logs from Web SEs enables research on query characteristics and user behavior on result pages. Markatos [2001] analyzes the query log from the EXCITE search engine and shows that query requests have a temporal locality property, that is, a significant amount of queries submitted previously are submitted again a short time later. This fact shows evidence for the caching potential of query results. That work also compares static vs. dynamic caching approaches for varying cache sizes. In static caching, previous query log history is used to find the most frequent queries submitted and the cache is filled with the results of these queries. A static cache is a read-only cache; that is, no replacement policy is applied. Since the query frequency distribution information becomes obsolete as new queries are submitted, the contents of the static cache must be refreshed at periodical intervals. On the other hand, the dynamic caching strategy starts with an empty cache and fills its entries as new queries arrive. If a cache miss occurs and the cache is full, a victim cache entry is chosen for replacement. There are many cache replacement policies adapted from the literature, such as Least Recently Used (LRU), Least Frequently Used (LFU), etc. The analysis in Markatos [2001] reveals that the static caching strategy performs better when the cache size is small, but dynamic caching becomes preferable when the cache is relatively larger. In a more recent work, Ozcan et al. [2008a] proposes an alternative query selection strategy, which is based on the stability of query frequencies over time intervals instead of using an overall frequency value.

Another recent work [Fagni et al. 2006] proposes a hybrid caching strategy, which involves both static and dynamic caching. The authors divide the available cache space into two parts: One part is reserved for static caching and the remaining part is used for dynamic caching. Their motivation for this approach is based on the fact that static caching exploits the query popularity that lasts for a long time interval and dynamic caching handles the query popularity that arises for a shorter interval (e.g., queries for breaking news, etc.). Experiments with three query logs show that their hybrid strategy, called Static-Dynamic Caching (SDC), achieves better performance than either purely static caching or purely dynamic caching.

While the above works solely consider caching the query result pages, several other works [Baeza-Yates and Saint-Jean 2003; Baeza-Yates et al. 2007a, 2008; Long and Suel 2005; Saraiva et al. 2001] also consider the possibility of caching posting lists. To the best of our knowledge, Saraiva et al.'s [2001] paper is the first work in the literature that mentions a two-level caching idea to combine caching query results and inverted lists. This work also uses index pruning techniques for list caching because the full index contains lists that are too long for some popular or common words. Experiments measure the performance of caching the query results and inverted lists separately, as well as that of caching them together. The results show that two-level caching achieves a 52% higher throughput than caching only inverted lists, and a 36% higher throughput than caching only query results.

Baeza-Yates and Saint-Jean [2003] propose a three-level search index structure using query log distribution. The first level consists of precomputed answers (cached query results) and the second level contains the posting lists of the most frequent query terms in the main memory. The remaining posting lists need to be accessed from the secondary memory, which constitutes the third level in the indexing structure. Since the main memory is shared by cached query results and posting lists, it is important to find the optimal space allocation to achieve the best performance. The authors provide a mathematical formulation of this optimization problem and propose an optimal solution.

More recently, another three-level caching approach is proposed [Long and Suel 2005]. As in Baeza-Yates and Saint-Jean [2003], the first and second levels contain the results and posting lists of the most frequent queries, respectively. The authors propose a third level, namely, an intersection cache, containing common postings of frequently occurring pairs of terms. The intersection cache resides in the secondary memory. Their experimental results show significant performance gains using this three-level caching strategy.

Finally, Baeza-Yates et al. [2007a] compare the two alternatives: caching query results vs. posting lists. Their analysis shows that caching posting lists achieves better hit rates [Baeza-Yates et al. 2007a]. The main reason for this result is that repetitions of terms in queries are more frequent than repetitions of queries. Remarkably, this work also makes use of query costs; however the costs are solely used for obtaining the optimal split of the cache space for storing results and posting lists. In contrast, we propose to employ miss costs in the actual caching policy; that is, while deciding what to store in the cache.

In the literature, the idea of cost-aware caching is applied in some other areas where miss costs are not always uniform [Jeong and Dubois 2003, 2006; Liang et al. 2007]. For instance, in the context of multiprocessor caches, there may be non-uniform miss costs that can be measured in terms of latency, penalty, power, or bandwidth consumption, etc. [Jeong and Dubois 2003, 2006]. Jeong and Dubois [2003] propose several extensions of LRU to make the cache replacement policy cost sensitive. The initial cost function assumes two static miss costs, that is, a low cost (simply 1) and a high cost (experimented with varying values). This work also provides experiments with a

more realistic cost function, which involves miss latency in the second-level cache of a multiprocessor.

Web proxy caching is another area where a cost-aware caching strategy naturally fits. In this case, the cost of a miss would depend on the size of the page missed and its network cost, that is, the number of hops to be traveled to download it. The work of Cao and Irani [1997] introduces GreedyDual-Size (GDS) algorithm, which is a modified version of the Landlord algorithm proposed by Young [2002]. The GDS strategy incorporates locality with cost and size concerns for Web proxy caches. A detailed survey of Web cache replacement strategies involving some function-based strategies is provided by Podlipnig and Böszörményi [2003].

We are aware of only two works in the literature that explicitly incorporate the notion of costs into the caching policies of Web SEs. Garcia [2007] proposes a heterogeneous cache that can store all possible data structures (posting lists, accumulator sets, query results, etc.) to process a query, with each of these entry types associated with a cost function. However, the cost function in that work is solely based on disk access times and must be recomputed for each cache entry after every modification of the cache.

The second study [Gan and Suel 2009] that is simultaneous to our work also proposes cache eviction policies that take the costs of queries into account. In this work, the cost function essentially represents queries' disk access costs and it is estimated by computing the sum of the lengths of the posting lists for the query terms. The authors also experiment with another option, that is, using the length of the shortest list to represent the query cost, and report no major difference in the trends. In our case, instead of predicting the cost, we prefer to use the actual CPU processing time obtained the first time the query is executed, that is, when the query result page is first generated. Our motivation is due to the fact that actual query processing cost is affected by several mechanisms, such as dynamic pruning, list caching, etc. [Baeza-Yates et al. 2007a]; and it may be preferable to use the actual value when available. Furthermore, we use a more realistic simulation of the disk access cost, which takes into account the contents of the list cache under several scenarios.

Note that, there are other differences between our work and that of Gan and Suel [2009]. In this study, we provide several interesting findings regarding the variance of processing times among different queries, and the relationship between a query's processing time and its popularity. In light of our findings, we adapt our previous cost-aware strategy for static caching [Altingovde et al. 2009] to dynamic and static-dynamic caching cases. Gan and Suel's work [2009] also considers some cost-aware techniques for dynamic and hybrid caching. However, they further explore issues like query burstiness and propose cache eviction policies that exploit features other than the query cost.

Finally, a more recent research direction that is orthogonal to all of the above works is investigating the cache freshness problem. In this case, the main concern is not the capacity related problems (as in the eviction policies) but the freshness of the query results that is stored in the cache. To this end, Cambazoglu et al. [2010] proposes a *blind* cache refresh strategy: they assign a fixed time-to-live (TTL) value to each query result in the cache and re-compute the expired queries without verifying whether their result have actually changed or not. They also introduce an eager approach that refreshes expired or about-to-expire queries during the idle times of the search cluster. In contrast, Blanco et al. [2010a, 2010b] attempt to invalidate only those cache items whose results have changed due to incremental index updates. The issue of freshness is not in the scope of our work, and it is left for future investigation.

3. EXPERIMENTAL SETUP

Datasets. In this study, we use two datasets. For the first, we obtained the list of URLs categorized at the Open Directory Project (ODP) Web directory (www.dmoz.org). Among these links, we successfully crawled around 2.2 million pages, which take 37 GBs of disk space in uncompressed HTML format. For the second dataset, we create a subset of the terabyte-order crawl collection provided by Stanford University's WebBase Project [WebBase 2007]. This subset includes approximately 4.3 million pages collected from US government Web sites during the first quarter of 2007. These two datasets will be referred to as "ODP" and "Webbase" datasets, respectively. The datasets are indexed by the Zettair search engine [Zettair 2007] without stemming and stopword removal. We obtained compressed index files of 2.2 GB and 7 GB on disk (including the *term offset information* in the posting lists) for the first and second datasets, respectively.

Query Log. We create a subset of the AOL Query Log [Pass et al. 2006], which contains around 20 million queries of about 650K people for a period of three months. Our subset contains around 700K queries from the first six weeks of the log. Queries submitted in the first three weeks constitute the training set (used to fill the static cache and/or warm up the dynamic cache), whereas queries from the second three weeks are reserved as the test set.

In this article, during both the training and testing stages, the requests for the next page of the results for a query are considered as a single query request, as in Baeza-Yates et al. [2007b]. Another alternative would be to interpret each log entry as <query, result page number> pairs [Fagni et al. 2006]. Since our query log does not contain the result page number information, the latter approach is left as a future work. Accordingly, we presume that a fixed number of N results are cached per query. Since N would be set to a small number in all practical settings, we presume that the actual value of N would not significantly affect the findings in this article. Here, we set N as 30, as earlier works on log analysis reveal that search engine users mostly request only the first few result pages. For instance, Silverstein et al. [1999] report that in 95.7% of queries, users requested up to only three result pages.

Experimental Platform. All experiments are conducted using a computer that includes an Intel Core2 processor running at 2.13 GHz with 2GB RAM. The operating system is Suse Linux.

4. AN ANALYSIS OF THE QUERY PROCESSING COST

4.1 The Setup for Cost Measurement

The underlying motivation for employing result caching in Web SEs (at the server side) is to reduce the burden of query processing. In a typical broker-based distributed environment (e.g., see Cambazoglu [2006]), the cost of query processing would involve several aspects, as shown in Figure 1. The central broker, after consulting its result cache, sends the query to index nodes. Each index node should then fetch the corresponding posting lists to the main memory (if they are not already in the list cache) with the cost C_{DISK}. Next, the postings are processed and partial results are computed with the cost C_{CPU}. More specifically, the CPU cost involves decompressing the posting lists (as they are usually stored in a compressed form), computing a similarity function between the query and the postings, and obtaining the top-N documents as the partial result. Then, each node sends its partial results to the central broker, with the cost C_{NET}, where they are merged. Finally, the central broker

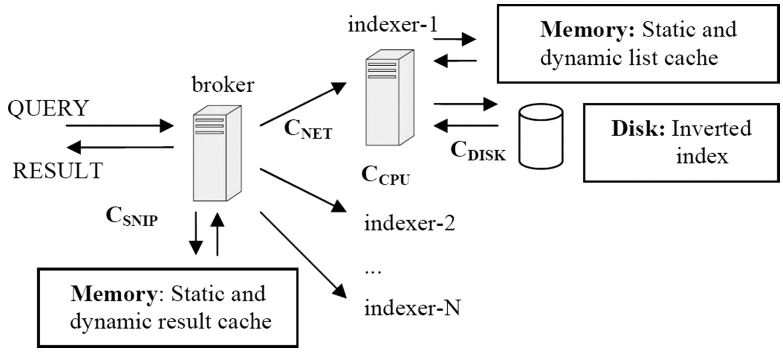

Fig. 1. Query processing in a typical large-scale search engine.

generates the snippets for the query results, with the cost C_{SNIP}, and sends the output page to the user. Thus, the cost of query processing is the sum of all of these costs, that is, $C_{DISK} + C_{CPU} + C_{NET} + C_{SNIP}$.

For the purposes of this article, we consider the sum of the CPU execution time and disk access time (i.e., $C_{DISK} + C_{CPU}$) as the major representative of the overall cost of a query. We justify leaving out the network communication and snippet generation costs as follows: Regarding the issue of network costs, a recent work states that for a distributed system interconnected via a LAN, the network cost would only be a fraction of the query processing cost (e.g., see Table III in Baeza-Yates et al. [2007a]). The snippet generation cost is discarded because its efficiency is investigated in only a few previous studies (e.g., Turpin et al. [2007], Tsegay et al. [2009]), and none of these discusses how the cost of snippet generation compares to other cost components. Furthermore, we envision that the two costs, namely, network communication and snippet generation, are less likely to vary significantly among different queries and neither would be a dominating factor in the query processing cost. This is because, regardless of the size of the posting lists for query terms, only a small and fixed number of results with the highest scores, (typically, top-10 document identifiers) would be transferred through the network. Similarly, only that number of documents would be accessed for snippet generation. Thus, in the rest of this article, we essentially use $C_{CPU} + C_{DISK}$ as the representative of the query processing cost in a search engine.

In this article, all distinct queries are processed using the Zettair search engine in batch mode to obtain the isolated CPU execution time per query. That is, we measured the time to decompress the lists, compute similarities and obtain the identifiers of the top-N documents, where N is set to 30. Since Zettair is executed at a centralized architecture, there is no network cost. To be more accurate, we describe the setup as follows.

— We use Zettair by its default mode, which employs an early pruning strategy that dynamically limits the number of accumulators used for a query. In particular, this dynamic pruning strategy adaptively decides to discard some of the existing accumulators or add new ones (up to a predefined target number of accumulators) as each query term is processed (see Lester et al. [2005] for details). Following the practice in Lester et al. [2005], we also set the target number of accumulators to approximately 1% of the number of documents in the collection, namely, 20K. Employing a dynamic pruning strategy is a crucial choice for the practicality of our proposal, since no real Web SE would make a full evaluation and the CPU execution time clearly depends on the partial evaluation strategy used in the system.

Table I. Disk Parameters for Simulating C_{DISK}

Parameter Name	Value
Seek time	8.5 ms
Rotational delay	4.17 ms
Block read time	4.883 μs
Block size	512 bytes

— All query terms in the log are converted to lower case. The queries are modified to include an additional "AND" conjunct between each term, so that the search engine runs in the "conjunctive" mode. This is the default search mode of the major search engines [de Moura et al. 2005; Ntoulas and Cho 2007]. Stopwords are not eliminated from the queries. No stemming algorithm is applied. Finally, all phrase queries are discarded.

In contrast to C_{CPU}, it is hard to associate a query with an isolated disk access cost, because for a real-life system disk access time depends on the query stream and the posting lists that are buffered and/or cached in the memory at the processing time. Thus, instead of measuring the actual disk access time, we compute this value per query under three different scenarios, where 25%, 50%, or 100% of the index is assumed to be cached. As discussed in the related work section, it is now widely accepted that any large-scale SE involves a reasonably large list cache that accompanies the result cache. Indeed, it is known that some major search engines store all posting lists in the main memory, an approach totally eliminating the cost of disk access [Dean 2009; Strohman and Croft 2007]. Therefore, our scenarios reflect realistic choices for the list cache setup.

For each of these scenarios, query term frequencies are obtained from the training logs and those terms with the highest ratio of term frequency to posting list length are stored in the cache, as proposed in Baeza-Yates et al. [2007a]. That study also reports that a static list cache filled in this manner yields a better hit rate than dynamic approaches (see Figure 8 in Baeza-Yates et al. [2007a]); so our framework only includes the static list cache. For each scenario, we first determine which terms of a given query cause a cache miss, and compute the disk access cost of each such term as the sum of seek time, rotational latency, and block transfer time, which is typical (e.g., see Ramakrishnan and Gehrke [2003]). In this computation, we use the parameters of a moderate disk, as listed in Table I.

To sum up, our framework realistically models the cost of processing a query in terms of the CPU and disk access times. The cost of a query computed by this model is independent of the query load on the system, as it only considers the isolated CPU and disk access times. More specifically, disk access time in this setup depends on the contents of the list cache (and, in some sense, previous queries), but not on the query load.

Before discussing cost-aware result caching strategies, we investigate answers to the following questions: (i) Do the costs of different queries really vary considerably?, (ii) Is there a correlation between the frequency and the cost of the queries? We expect the answer to the first question to be positive; that is, there should be a large variance among query execution times. On the other hand, we expect the answer of the second question to be negative; that is, frequency should not be a trustworthy indicator of processing time, so that cost-aware strategies can further improve solely frequency-based strategies. We explore the answers to these questions in the next section and show that our expectations are justified.

Table II. Characteristics of the Query Log Variants

Query log	Number of distinct queries		Number of all queries	
	Training	Test	Training	Test
ODP query log	253,961	209,636	446,952	362,843
Webbase query log	211,854	175,557	386,179	313,884
Webbase semantically aligned query log	28,794	24,066	45,705	37,506

4.2 Experiments

After a few initial experiments with the datasets and the query log previously described, it turns out that a nontrivial number of queries yields no answer for our datasets. As an additional problem, some of the most frequent queries in the query log appear much less frequently in the dataset, which might bias our experiments. Since previous works in the literature emphasize that the dataset and query log should be compatible to ensure a reliable experimental framework [Webber and Moffat 2005], we first focus on resolving this issue.

The ODP dataset contains pages from a general Web directory that includes several different categories and the AOL log is a general domain search engine log. That is, the content of our dataset matches the query log to a certain extent, although the dataset is, of course, much more restricted than the real AOL collection. Thus, for this case, it would be adequate to simply discard all queries that have no answer in the ODP dataset.

On the other hand, recall that the Webbase dataset includes pages crawled only from the .gov domain. Thus, there seems to be a higher possibility of mismatch between the Webbase dataset and AOL query log. To avoid any bias that might be caused by this situation, we decided to obtain a domain-restricted version of the query log for the Webbase collection. In what follows, we describe the query logs corresponding to the datasets as used in this study.

(1) *ODP Query Log*. We keep all queries in the log that yield non-empty results on the ODP dataset.
(2) *Webbase Query Log*. We keep all queries in the log that yield non-empty results on the Webbase dataset.
(3) *Webbase Semantically Aligned Query Log*. Following a similar approach discussed in a recent work [Tsegay et al. 2007], we first submit all distinct queries in our original query log to *Yahoo!* search engine's "Web search" service [Yahoo! 2009] to get the top-10 results. Next, we only keep those queries that yield at least one result from the .gov domain.

In Table II, we report the number of the remaining queries in each query log. To experimentally justify that these query logs and the corresponding datasets are compatible, we conduct an experiment as follows. We process randomly chosen 5K queries from each of the three query logs in conjunctive mode on the corresponding collection, and record the total number of results per query. Next, we submit the same queries to Yahoo! (using its Web search API), again in conjunctive (default) processing mode. For each case, we also store the number of results per query as returned by the search engine API.

In Figure 2, we represent these 5K queries on a log-log scale plot, where the x-axis is the ratio of the number of results retrieved in our corresponding collection to the collection size, and the y-axis is the same ratio for Yahoo! collection. We assume that the underlying collection of Yahoo! includes around 31.5 billion pages, which is the reported number of results when searching for the term "a" on the Yahoo! Web site.

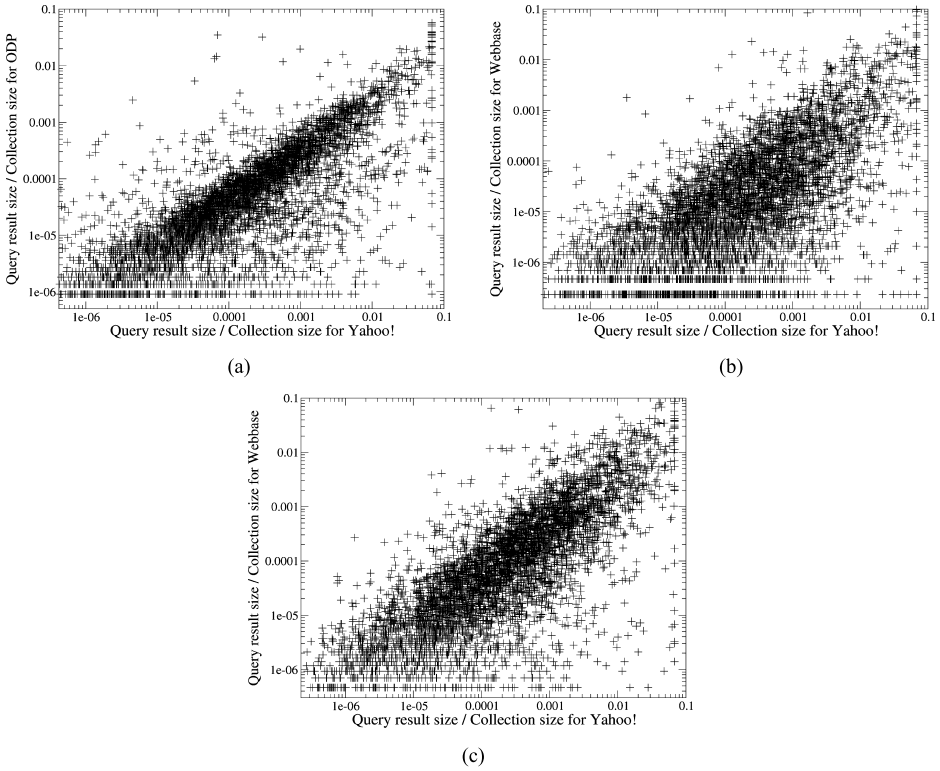

Fig. 2. Correlation of "query result size/collection size" on Yahoo! and (a) ODP, (b) Webbase, and (c) Webbase semantically aligned for the conjunctive processing mode.

The figure reveals that the ratio of the number of results per query in each collection is positively correlated with the ratio in the Yahoo! search engine, that is, yielding correlation coefficients of 0.80, 0.57, and 0.86 for the ODP, Webbase, and Webbase semantically aligned logs, respectively. Thus, we conclude that our collections and query sets are compatible, and experimental evaluations would provide meaningful results.

Next, for each of the query logs in Table II, we obtain the CPU execution time of all distinct queries using the setup described in the previous section. The experiments are repeated four times and the results reveal that the computed CPU costs are stable and can be used as the basis of the following discussions. For instance, for ODP log, we find that the average standard deviation of query execution times is 2 ms. Considering that the average query processing time is 93 ms for this case, we believe that the standard deviation figure (possibly due to system-dependent fluctuations) is reasonable and justifies our claim of the stability of execution times.

In Figure 3, we provide the normalized log-log scatter plots that relate the query's frequency in the log and the CPU execution time[1] for randomly selected 10K queries. These plots reveal the answers to the questions raised at the end of the previous section. First, we see that the query execution time covers a wide range, from a fraction

[1]Note that, this experiment considers the scenario where the entire index is assumed to be in the memory and thus C_{DISK} is discarded. The findings for other scenarios, that is, those involving C_{DISK}, are similar and not reported here for brevity.

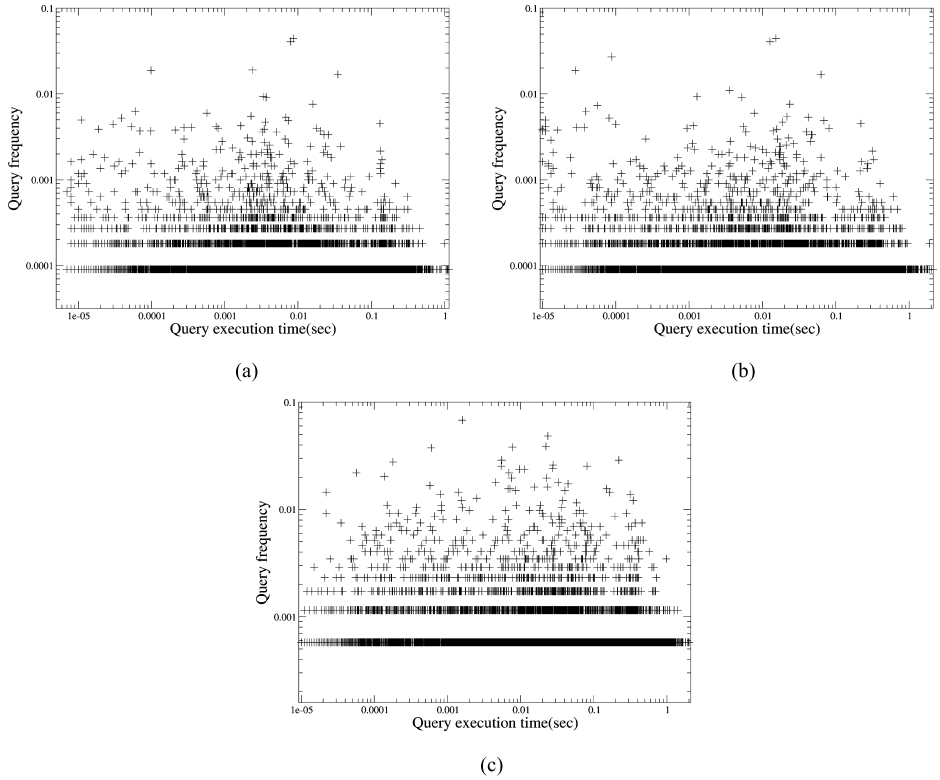

Fig. 3. Normalized log-log scatter plot of the query CPU execution time and query frequency in the (a) ODP, (b) Webbase, and (c) Webbase semantically aligned query log.

of a millisecond to a few thousand milliseconds. Thus, it may be useful to devise cost-aware strategies in the caching mechanisms. Second, we cannot derive a high positive correlation between the query frequency and the processing times. That is, a very frequent query may be cheaper than a less-frequent query. This can be explained by the following arguments: In earlier works, it is stated that query terms and collection terms may not be highly correlated (e.g., a correlation between Chilean crawl data and the query log is found to be only 0.15 [Baeza-Yates and Saint-Jean 2003]), which means that a highly frequent query may return fewer documents, and be cheaper to process. Indeed, in a separate experiment reported below, we observe that this situation also holds for the AOL query log using both the ODP dataset and the Yahoo! search engine.

In Figure 4, we show the normalized log-log scatter plots that relate the query frequency in the log and result-set size, that is, the ratio of number of query results to the size of ODP collection, for randomly selected 10K queries. As can be seen from the figure, the correlation is very low; that is, the correlation coefficient is -0.01. In order to show that the same trend also holds true for a real search engine, we provide the same plot obtained for Yahoo! in Figure 5. Here, we obtain the number of results returned for randomly selected 5K queries using the Yahoo! Web search API. Figure 5 again demonstrates that queries with the same frequency might have a very different number of results, and that there is no positive correlation between query frequency and query result frequency; that is, for this case, the correlation coefficient is only 0.03. Similar findings are also observed for Webbase dataset and corresponding query logs, but not reported here due to space considerations.

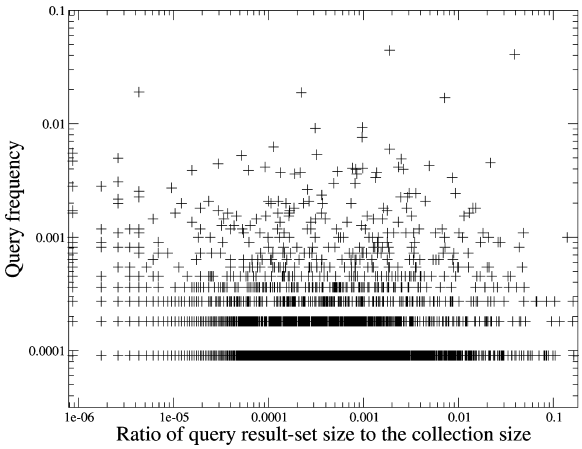

Fig. 4. Normalized log-log scatter plot of the query result-set size and the query frequency in the ODP query log for 10K queries.

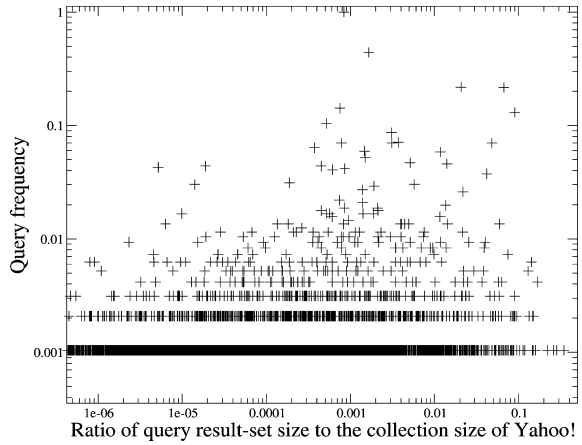

Fig. 5. Normalized log-log scatter plot of the query result-set size in Yahoo! and the query frequency in the query log for 5K queries.

Finally, note that, even for the cases where the above trend does not hold (i.e., the frequent queries return a large number of result documents), the processing time does not necessarily follow the proportion, due to the compression and early stopping (pruning) techniques applied during query processing (see Figure 10 in Baeza-Yates et al. [2007a], for example).

Our findings in this section are encouraging in the following ways: We observe that the query processing costs, and accordingly, the miss costs, are nonuniform and may vary considerably among different queries. Furthermore, this variation is not directly correlated to the query frequency, a feature already employed in current caching strategies. These call for a cost-aware caching strategy, which we discuss next.

5. COST-AWARE STATIC AND DYNAMIC CACHING

In this section, we describe our cost-aware strategies for static, dynamic, and hybrid result caching.

5.1 Cost-Aware Caching Policies for a Static Result Cache

As discussed in the literature [Baeza-Yates et al. 2007a], filling a static cache with a predefined capacity can be reduced to the well-known knapsack problem, where query result pages are the items with certain sizes and values. In our case, we presume that cache capacity is expressed in terms of the number of queries; that is, each query (and its results) is allocated a unit space. Then, the question is how to fill the knapsack with the items that are most valuable. Setting the value of a query to its frequency in previous query logs, that is, filling the cache with the results of the most frequent past queries, is a typical approach employed in SEs. However, as we discuss previously, miss costs of queries are not uniform. Therefore, the improvement promises of such a strategy evaluated in terms of, for example, hit rate, may not translate to actual improvements that can be measured in terms of query processing time or throughput. To remedy this problem, we propose to directly embed miss costs into the static caching policies. In what follows, we specify a number of cost-aware static caching approaches in addition to the traditional frequency-based caching strategy, which serves as a baseline. In these discussions, for a given query q, its cost and frequency are denoted as C_q and F_q,respectively.

Most Frequent (MostFreq). This is the baseline method, which basically fills the static cache with the results of the most frequent queries in the query log. Thus, the value of a cache item is simply determined as follows:

$$Value(q) = F_q.$$

Frequency Then Cost (FreqThenCost). Previous studies show that query frequencies in a log follow a power-law distribution; that is, there exist a few queries with high frequencies and many queries with very low frequencies [Xie and O'Hallaron 2002]. This means that for a reasonably large cache size, the MostFreq strategy should select among a large number of queries with the same – relatively low – frequency value, possibly breaking the ties at random. In the FreqThenCost strategy, we define the value of a query with the pair (F_q, C_q) so that while filling the cache we first choose the results of the queries with the highest frequencies, and from the queries with the same frequency values we choose those with the highest cost. We anticipate that this strategy would be more effective than MostFreq especially for caches with larger capacities, for which more queries with the same frequency value would be considered for caching. In Section 6.1, we provide experimental results that justify our expectation.

Stability Then Cost (StabThenCost). In a recent study, we introduce another feature, namely, query frequency stability (QFS), to determine the value of a query for caching [Ozcan et al. 2008a]. This feature represents the change in a query's popularity during a time interval. The underlying intuition for this feature stems from the fact that in order to be cached, queries should be frequent and remain frequent over a certain time period. The QFS feature is defined as follows:

Assume that query q has the total frequency of f in a training query log that spans a time period T. Consider that this time period is divided into n equal time intervals[2] and query q has the following values of frequency: $F = \{f_1, f_2, \ldots, f_n\}$; one for each T/n time units. Then, the stability of query q is defined by the following formula:

$$QFS_q = \sum_{i=1}^{n} \frac{|f_i - f_\mu|}{f_\mu},$$

where $f_\mu = {}^{f}/_{n}$ is the mean frequency of q during T.

[2]In this article, we assume one day as the time interval.

In this previous study, it is shown that using the QFS feature for static caching yields better hit rates than solely using the frequency feature. Here, we combine this feature with the query cost and define the value of a query with the pair (QFS_q, C_q). That is, while filling the cache, queries are first selected based on their QFS value and then their associated cost values.

Frequency and Cost (FC_K). For a query q with cost C_q and frequency F_q, the expected value of the query q can be simply computed as the product of these two figures, that is, C_q x F_q. That is, we expect that the query would be as frequent in future as in past logs, and caching its result would provide the gain as expressed by this formula. During the simulations reported in Section 6, we observe that this expectation may not hold in a linearly proportional manner; that is, queries that occur with some high frequency still tend to appear with a high frequency, whereas the queries with a relatively lower frequency may appear even more sparsely, or totally fade away, in future queries. A similar observation is also discussed in Gan and Suel [2009]. For this reason, we use a slightly modified version of the formula that is biased to emphasize higher frequency values and depreciate lower ones, as shown:

$$Value(q) = C_q x F_q^K, \text{where } K > 1.$$

5.2 Cost-Aware Caching Policies for a Dynamic Result Cache

Although static caching is an effective approach for exploiting long-term popular queries, it may miss short-term popular queries submitted to a Web SE during a short time interval. Dynamic caching handles this case by updating its content as the query stream changes. Different from static caching, a dynamic cache does not require a previous query log. It can start with an empty cache and it fills its entries as new queries are submitted to a Web SE. If the result page for a submitted query is not found in the cache, the query is executed and its result is stored in the cache. When the cache is full, a victim cache entry is chosen based on the underlying cache replacement policy. A notable number of cache replacement policies are proposed in the literature (e.g., see Podlipnig and Böszörményi [2003] for Web caches). In the following, we first describe two well-known strategies, namely, least recently used and least frequently used, to serve as a baseline. Then, we introduce two cost-aware dynamic caching strategies in addition to adapting two other approaches from the literature to the result caching domain.

Least Recently Used (LRU). This well-known strategy chooses the least recently referenced/used cache item as the victim for eviction.

Least Frequently Used (LFU). In this policy, each cached entry has a frequency value that shows how many times this entry is requested. The cache item with the lowest frequency value is replaced when the cache is full. This strategy is called "in-cache LFU" in Podlipnig and Böszörményi [2003].

Least Costly Used (LCU). This is the most basic cost-aware replacement policy introduced in this article. Each cached item has an associated cost, as described in Section 4.1. This method chooses the least costly cached query result as the victim.

Least Frequently and Costly Used (LFCU_K). This policy is the dynamic version of the static cost-aware policy that we previously propose [Altingovde et al. 2009]. We employ the same formula specified for the FC_K strategy in Section 5.1.

Table III. Hybrid Cache Configurations

Hybrid Cache Configuration	Static Cache Strategy	Dynamic Cache Strategy
Non cost-aware	MostFreq	LRU
Only static cache is cost-aware	FC_K	LRU
Both static and dynamic caches are cost-aware	FC_K	LFCU_K
	FC_K	GDS
	FC_K	GDSF_K

Greedy Dual Size (GDS). This method maintains a so-called H-value for each cached item [Cao and Irani 1997]. For a given query q with an associated cost C_q and result page size S_q, the H-value is computed as follows:

$$H\_value(q) = \frac{C_q}{S_q} + L$$

In this formula, L is an aging factor that is initialized to zero at the beginning. This policy chooses the cache item with the smallest H-value. Then, the value of L is set to the evicted item's H-value. When a query result is requested again, its H-value is recalculated since the value of L might have changed. The size component in the formula can be ignored as all result pages are assumed to use the same amount of space, as we discuss before.

Greedy Dual Size Frequency (GDSF_K). This method is a slightly modified version of the GDS replacement policy [Arlitt et al. 2000]. In this case, the frequency of cache items is also taken into account. The corresponding *H-value* formula is presented below.

$$H\_value(q) = F_q^K x \frac{C_q}{S_q} + L$$

In this strategy, the frequency of the cache items are also kept and updated after each request. As discussed in Section 5.1, again we favor the higher frequencies by adding an exponent K (>1) to the frequency component. Note that with this extension, the formula also resembles the generalized form of GDSF as proposed in Cherkasova and Ciardo [2001]. However, that work proposes to add weighting parameters for both frequency and size components while setting the cost component to 1. In our case, we have to keep the cost, C_q, and apply weighting only for the frequency values.

5.3 Cost-Aware Caching Policies for a Hybrid Result Cache

The hybrid result caching strategy proposed in Fagni et al. [2006] involves both static and dynamic caching and it outperforms its purely static and purely dynamic counterparts. We employ the hybrid caching framework in order to see the effect of cost-awareness in such a state-of-the-art query result caching environment.

In this cache configuration, total cache size is divided into two parts, for static and dynamic caches. The fractions of the cache space reserved for each kind of cache is based on the underlying query log for the best performance. The static cache part is filled based on a training query log. Then, the dynamic cache part is warmed up by submitting the remaining training queries into this part of the cache. Later, cache performance is evaluated using the disjoint test set. Table III shows the hybrid cache configurations experimented with in this work. We essentially consider three types of cache configurations, based on whether a cost-aware strategy is employed in each cache part. The baseline case does not involve the notion of cost at all; the static and

dynamic caches employ traditional MostFreq and LRU strategies, respectively [Fagni et al. 2006]. In the second case, only the static portion of the hybrid cache can be cost-aware; the static part employs the FC_K strategy and the dynamic part is still based on LRU. In the third configuration, both the static and dynamic portions might be cost-aware. For this case, three different combinations are experimented with. All three combinations use FC_K for the static cache part, but they use three different cost-aware dynamic caching policies, namely, LFCU_K, GDS, and GDSF_K.

6. EXPERIMENTS

In this section, we provide a simulation-based evaluation of the aforementioned cost-aware policies for static, dynamic, and hybrid caching environments. As described in Section 3, the query log is split into training and test sets, and the former is used to fill the static caches and/or warm up the dynamic caches, whereas the latter is used to evaluate the performance. Cache size is specified in terms of the number of the queries. Remarkably, we don't measure the cache hit rate (due to nonuniform miss costs) but use the total query processing time for evaluation. For the cache hits, we assume that the processing time is negligible; that is, the cost is 0. To simulate the cache misses, we use the query processing cost, $C_{CPU} + C_{DISK}$, which is also employed in the training stage. That is, for all distinct queries in the log, we store the actual CPU execution time (C_{CPU}) per query that is reported by the Zettair SE. As mentioned before, CPU cost measurements are repeated four times and found to be stable; that is, no fluctuations are observed. Furthermore, for each list cache scenario, namely, caching 25%, 50%, and 100% of the index, we compute the simulated disk access time, C_{DISK}, per query. Whenever a cache miss occurs, the cost of this query is retrieved as the sum of C_{CPU} and C_{DISK} for the given list cache scenario, and added to the total query processing time.

6.1 Simulation Results for Static Caching

In this section, we essentially compare the four strategies, namely MostFreq, FreqThenCost, StabThenCost, and FC_K, for static caching. In Figures 6(a), 6(b), and 6(c), we provide the total query execution times using these strategies for the ODP log when 25%, 50%, and 100% of the index is cached, respectively. For the sake of brevity, the corresponding experimental results for the Webbase and Webbase semantically aligned logs are given in Figure 7 only for the case where the entire index is cached.

We also provide the potential gains for the optimal cost-aware static caching strategy (OptimalCost), where the test queries are assumed to be known beforehand. Since we know the future frequency of the training queries, we fill the cache with the results of the queries that would yield the highest gain, that is, frequency times cost. Clearly, this is only reported to illustrate how far the proposed strategies are away from the optimal.

In all experiments, cost-aware strategies (FreqThenCost, StabThenCost and FC_K) reduce the overall execution time with respect to the baseline, that is, MostFreq. We observe that the improvement gets higher as higher percentages of the index are cached. This is important because large scale SEs tend to cache most of the index in memory. The best-performing strategy, FC_K (where K is set to 2.5 in light of the findings reported in Altingovde et al. [2009]), yields up to a 3% reduction in total time for varying cache sizes. It is also remarkable that the gains for the optimal cache (denoted as OptimalCost) are much higher, which implies that it is possible to further improve the value function employed in the cost-aware strategies. This is left for future investigation.

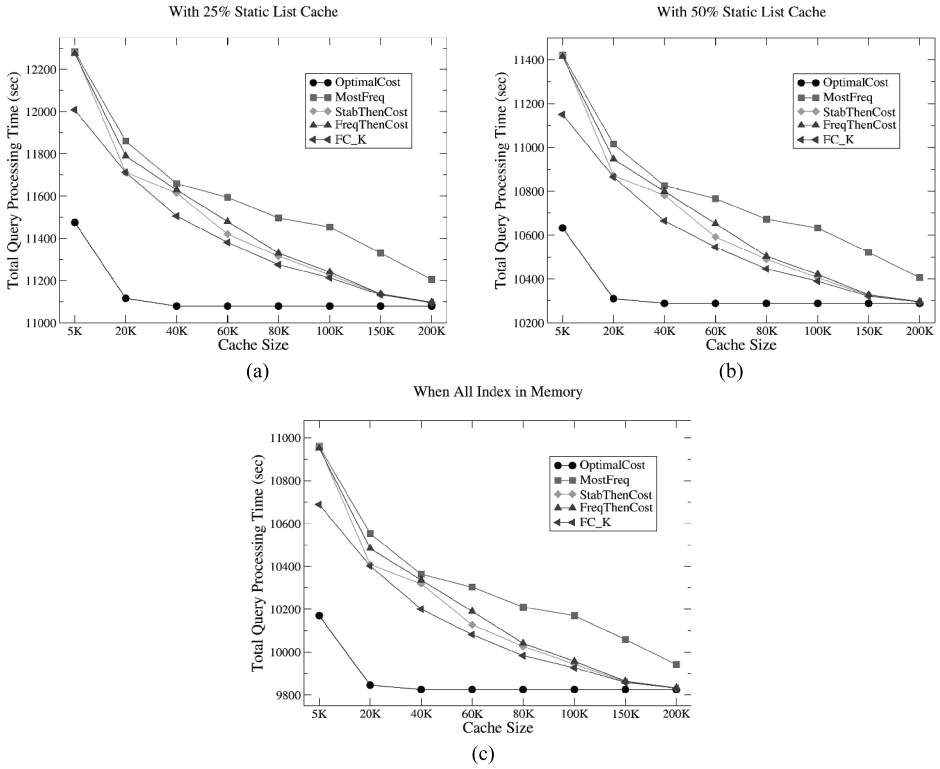

Fig. 6. Total query processing times (in seconds) obtained using different static caching strategies for the ODP log when (a) 25%, (b) 50%, and (c) 100% of the index is cached.

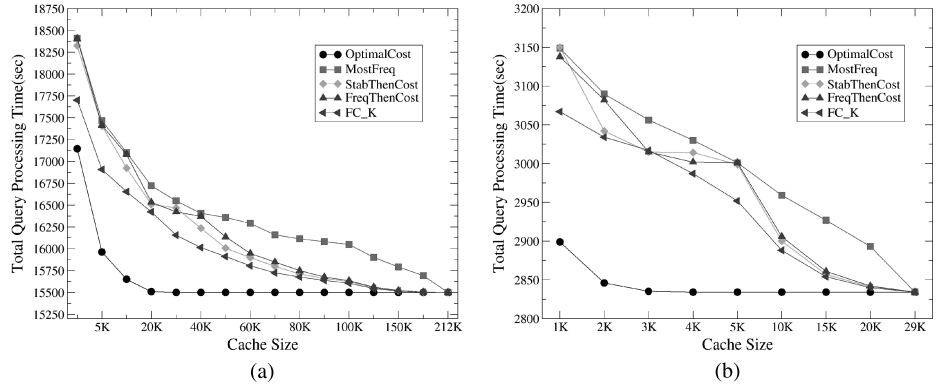

Fig. 7. Total query processing times (in seconds) obtained using different static caching strategies for the (a) Webbase and (b) Webbase semantically aligned logs when 100% of the index is cached.

6.2 Simulation Results for Dynamic Caching

In this section, we experiment with the dynamic caching approaches mentioned in Section 5.2. Note that, LRU and LFU strategies do not use cost values in the replacement policies, while all the other strategies are cost-aware. As a lower bound, we also show the performance of the infinite-sized dynamic cache (INFINITE), for

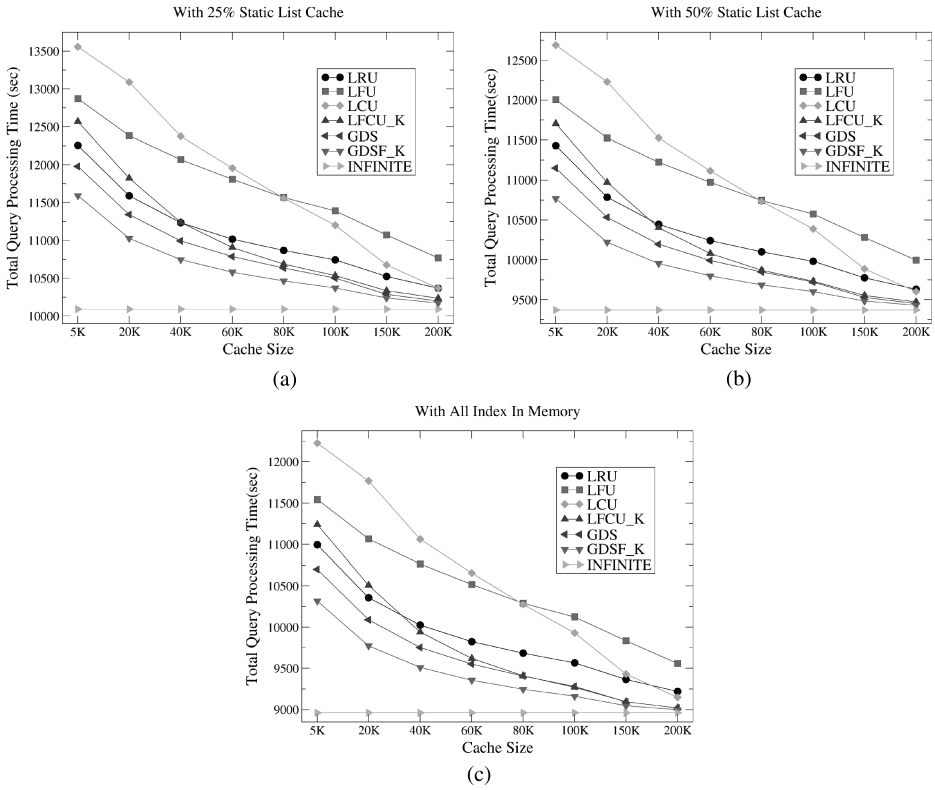

Fig. 8. Total query processing times (in seconds) obtained using different dynamic caching strategies for the ODP log when (a) 25%, (b) 50%, and (c) 100% of the index is cached.

which no replacement policy is necessary. In Figure 8, we display the total query execution times obtained using different dynamic caching strategies for the ODP log when 25%, 50%, and 100% of the index is cached. The results of the same experiment for the Webbase and Webbase semantically aligned logs are given in Figure 9 only for the case where 100% of the index is cached. The other cases, namely caching 25% and 50% of the index, yield similar results and are not reported here.

In all experiments, the trends are very similar. As in the case of static caching, the improvements are more emphasized as the percentage of index that is cached in the memory increases. Therefore, in the following, we only discuss the case for the ODP log when 100% of the index is cached. First, we see that the cost-aware version of LFU, which is LFCU_K(with $K = 2$, tuned by only using the training log), outperforms LFU in all reported cache sizes. The reductions in total query processing times reach up to 8.6%, 9.4%, and 7.4% for the ODP, Webbase, and Webbase semantically aligned logs, respectively. Although LRU is slightly better than LFCU_K for small cache sizes, the cost-aware strategy performs better for medium and large cache sizes. It is seen that the GDSF_K policy (again with the best-performing value of K tuned by only using the training log, namely, 2.5) is the best strategy among all policies for all cache sizes and all three logs. We achieved up to 6.2%, 7%, and 5.9% reductions in total query processing time compared to the LRU cache for the ODP, Webbase, and Webbase semantically aligned logs, respectively.

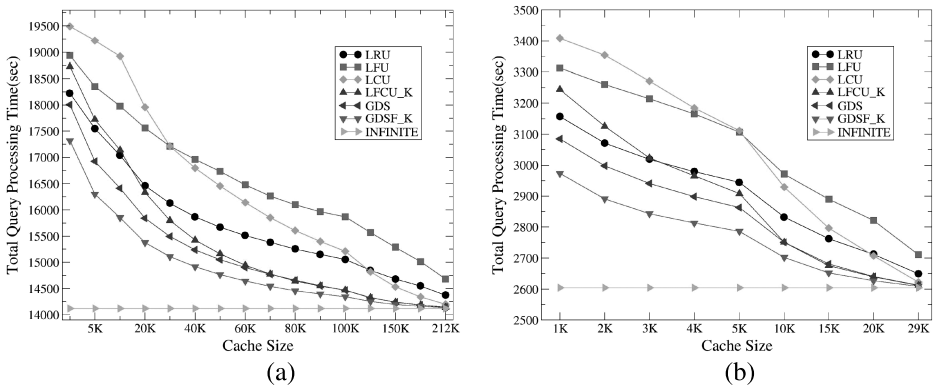

Fig. 9. Total query processing times (in seconds) obtained using different dynamic caching strategies for the (a) Webbase and (b) Webbase semantically aligned logs when 100% of the index is cached.

6.3 Simulation Results for Hybrid Caching

In this section, we experimentally evaluate the hybrid caching approaches mentioned in Section 5.3. In Figure 10, we provide the total query execution times using different hybrid caching strategies for the ODP log when 25%, 50%, and 100% of the index is cached. As before, we report the results for the Webbase and Webbase semantically aligned logs only for the latter scenario, in Figures 11(a) and 11(b), respectively. The fraction parameter for dividing the cache is tuned experimentally using the training query log. We observe that using a split parameter of 50% yields the best performance for the majority of the cases among the five different hybrid cache configurations given in Table III. So, in all of the experiments reported in this article, we equally divide the cache space between static and dynamic caches.

We see that cost-aware policies also improve performance in the hybrid caching environment. For brevity, we discuss the reduction percentages for the ODP log for the scenario where 100% of the index is cached. The other cases exhibit similar trends with smaller gains.

To start with, we observe that it is possible to obtain improvements even when only the static portion of the cache is cost-aware. In Figures 10 and 11, the static_FC_K_dynamic_LRU case outperforms the baseline, especially for the larger cache sizes. For this case, we achieve up to 2.6%, 3.5%, and 2.6% reductions in total query processing time for the ODP, Webbase, and Webbase semantically aligned logs, respectively. Furthermore, if the dynamic portion also employs a cost-aware cache eviction policy, reductions in total query processing time are more emphasized, especially for the GDSF_K policy (i.e., up to 3.8%, 4.9%, and 3.5% reductions for the ODP, Webbase, and Webbase semantically aligned logs, respectively).

In Figure 12, we compare the best performing strategies for static and dynamic caching (namely, FC_K, and GDSF_K) to the hybrid caching that combines both strategies. Our findings confirm previous observations that result caching significantly improves system performance. For instance, even the smallest static cache configuration (including only 5K queries) yields a 14% drop in total query processing time. We also show that hybrid caching is superior to purely static and dynamic caching for smaller cache sizes, whereas it provides comparable performance to dynamic caching for larger cache sizes.

The performance of dynamic caching strategies may suffer from the use of concurrency control mechanisms in a parallel query processing environment. Fagni et al. [2006] argue that such *cache-access concurrency mechanisms* can cause a relatively

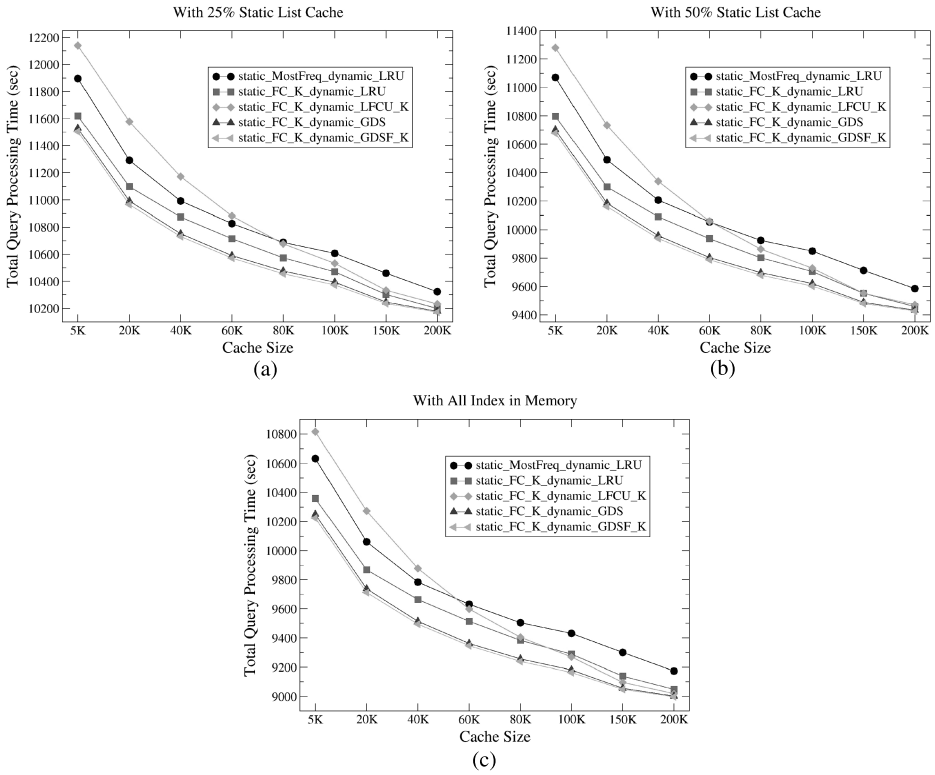

Fig. 10. Total query processing times (in seconds) obtained using different hybrid caching strategies for the ODP log when (a) 25%, (b) 50%, and (c) 100% of the index is cached.

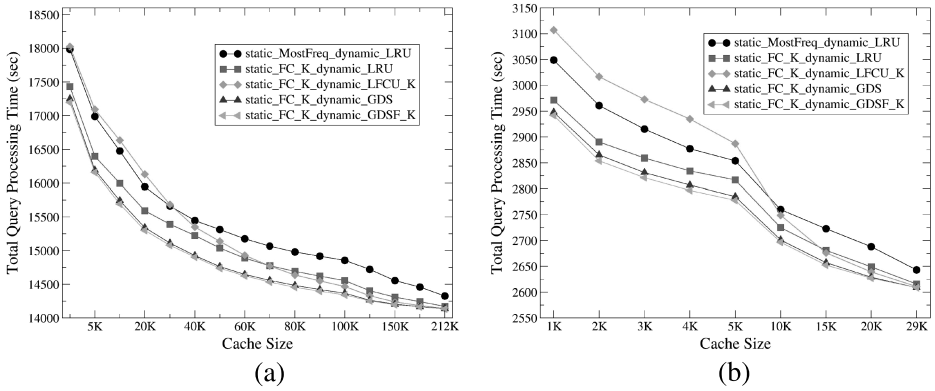

Fig. 11. Total query processing times (in seconds) obtained using different hybrid caching strategies for the (a) Webbase and (b) Webbase semantically aligned logs when 100% of the index is cached.

higher overhead for fully dynamic caching strategies when compared to hybrid strategies, which include a static (i.e., read-only) part. Note that, since the performance gap between the static and dynamic/hybrid strategies is rather large (e.g., see Figure 12), it is less probable that this access overhead would make any difference in relative performance of these strategies. For the purposes of this article, we anticipate that the

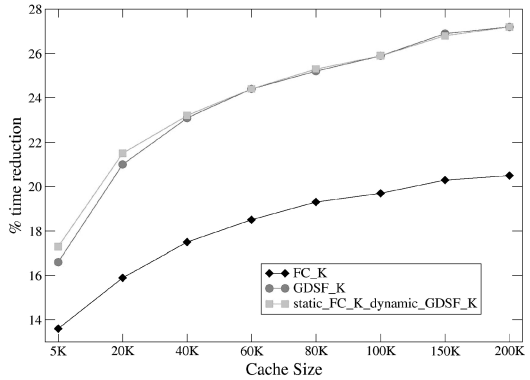

Fig. 12. Percentages of time reduction due to caching using best static, dynamic, and hybrid approaches for the ODP log when 100% of the index is cached.

cost of cache-access mechanisms would not be that significant in comparison to other query processing cost components, and so we do not explicitly consider cache-access overhead in the experiments.

7. SUMMARY OF THE RESULTS AND ADDITIONAL EXPERIMENTS

In this article, we justify the necessity of cost-based caching strategies by demonstrating that query costs are not uniform and may considerably vary among the queries submitted to a SE. Thus, cost-aware caching policies can further improve performance in terms of the total query execution time for static, dynamic, or hybrid caching of the query results.

The experimental results justify our expectations and reveal that it is possible to achieve moderate but significant reductions in total query processing time by employing cost-aware policies in each caching mode. In the static caching mode, the reductions are up to around 3% (in comparison to the classical baseline, that is, caching the most frequent queries). In the dynamic caching mode, the reductions are more emphasized and reach up to around 6% in comparison to the traditional LRU strategy. Finally, up to around 4% improvement is achieved in a hybrid, static-dynamic, caching mode. Thus, for all cases, the cost-aware strategies improve the state-of-the-art baselines.

Note that, in all simulation results reported in Section 6, we measure CPU execution times when the queries are processed in the conjunctive mode (i.e., "AND" mode, as mentioned in Section 4.1). In a set of additional experiments, we also measure the query execution times in the disjunctive ("OR") mode. Figure 13 provides the simulation result for dynamic caching with the ODP log when the entire index is assumed to be cached. The other cache types and query logs are not discussed to save space. Notably, the overall query processing times in Figure 13 are longer than those of the corresponding case with the conjunctive processing, as expected (please compare with the plot in Figure 8(c)). We also see that all trends are the same, except the LCU strategy performs slightly better in the conjunctive mode.

As a final experiment, we analyze the average response time of queries under different query loads for several caching methods. In this simulation, we assume that the time between each query submission follows an exponential distribution as shown in previous works (e.g., Cacheda and Vina [2001]). In particular, we vary the mean query inter-arrival time between 50 ms and 500 ms, corresponding to high- and low-workload scenarios, respectively. We also assume that the search system involves two processors. In Figure 14, we provide average response time figures for dynamic caching

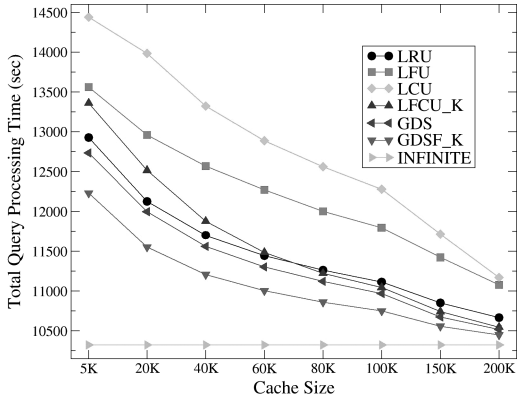

Fig. 13. Total query processing times (in seconds) obtained using different dynamic caching strategies for the ODP log when queries are processed in the disjunctive processing mode.

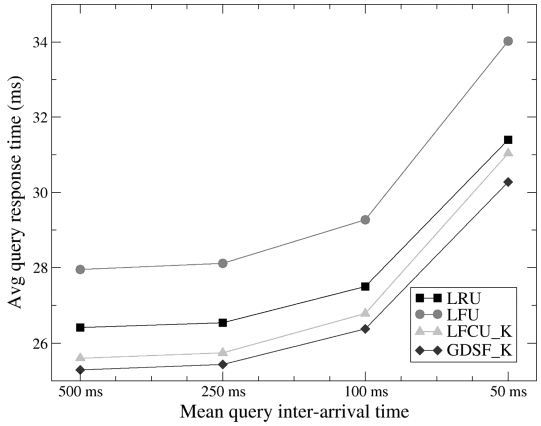

Fig. 14. Average query response time obtained using different caching strategies for various query workloads (simulated by the different mean query inter-arrival times) of the ODP log.

strategies (namely LFU, LRU, LFCU_K and GDSF_K), and only for the case where all index is stored in memory, for the sake of simplicity. Our findings reveal that cost-aware strategies also improve the average response time under different load scenarios. Note that, our results presented here are not conclusive, and we leave an in-depth investigation of response time related issues as a future work.

8. CONCLUSION

In this article, we propose cost-aware caching strategies for the query result caching in Web SEs, and evaluate it for static, dynamic, and hybrid caching cases. We observe considerable reductions in total query processing time (i.e., sum of the CPU execution and disk access times) for all three caching modes. For static caching, we extend our preliminary work [Altingovde et al. 2009] by incorporating cost-awareness into the static caching policy introduced in Ozcan et al. [2008a]. For dynamic caching, we propose two cost-aware policies, namely LCU and LFCU_K, and show that especially the latter strategy achieves better results than its non-cost-aware counterpart and the traditional LRU strategy. We also show that cost-aware policies such as GDS and

GDSF_K, as employed in other domains, perform well in query result caching. Finally, we analyze the performance of the cost-aware policies in a hybrid caching setup such that one portion of the cache is reserved for static caching and the other portion for dynamic caching. We experiment with several different alternatives in this setup and show that if both static and dynamic portions of the cache follow a cost-aware caching policy, performance improvement is highest.

As a future work, we plan to evaluate the cost-aware strategies in a more complex architecture that may involve several other mechanisms, for example, an intersection cache. We also envision that processing and caching requirements may differ for different types of queries (e.g., navigational queries may have different requirements than informational queries, as discussed in Ozcan et al. [2008b]). In our future studies, we plan to extend the cost-aware strategies to take into account the characteristics of such query types.

ACKNOWLEDGMENTS

The authors are thankful to the anonymous reviewers for their helpful feedback. We thank Rana Nelson for proofreading.

REFERENCES

ALTINGOVDE, I. S., OZCAN, R., AND ULUSOY, Ö. 2009. A cost-aware strategy for query result caching in Web search engines. In *Proceedings of 31st European Conference on Information Retrieval*, Lecture Notes in Computer Science, vol. 5478. Springer-Verlag, 628–636.

ARLITT, M. F., CHERKASOVA, L., DILLEY, J., FRIEDRICH, R. J., AND JIN, T. Y. 2000. Evaluating content management techniques for Web proxy caches. *ACM SIGMETRICS Perform. Eval. Rev. 27*, 4, 3–11.

BAEZA-YATES, R. AND SAINT-JEAN, F. 2003. A three level search engine index based in query log distribution. In *Proceedings of 10th International Symposium on String Processing and Information Retrieval*. Lecture Notes in Computer Science, vol. 2857. Springer-Verlag, 56–65.

BAEZA-YATES, R., GIONIS, A., JUNQUEIRA, F., MURDOCK, V., PLACHOURAS, V., AND SILVESTRI, F. 2007a. The impact of caching on search engines. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, 183–190.

BAEZA-YATES, R., JUNQUEIRA, F., PLACHOURAS, V., AND WITSCHEL, H. F. 2007b. Admission policies for caches of search engine results. In *Proceedings of 14th International Symposium on String Processing and Information Retrieval*, Lecture Notes in Computer Science, vol. 4726. Springer-Verlag, 74–85.

BAEZA-YATES, R., GIONIS, A., JUNQUEIRA, F., MURDOCK, V., PLACHOURAS, V., AND SILVESTRI, F. 2008. Design trade-offs for search engine caching. *ACM Trans. Web* 2, 4, 1–28.

BLANCO, R., BORTNIKOV, E., JUNQUEIRA, F., LEMPEL, R., TELLOLI, L., AND ZARAGOZA, H. 2010a. Caching search engine results over incremental indices. In *Proceedings of the 19th International Conference on World Wide Web*. ACM, New York, 1065–1066.

BLANCO, R., BORTNIKOV, E., JUNQUEIRA, F., LEMPEL, R., TELLOLI, L., AND ZARAGOZA, H. 2010b. Caching search engine results over incremental indices. In *Proceedings of the 33rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, 82–89.

CACHEDA, F. AND VINA, A. 2001. Experiencies retrieving information in the World Wide Web. In *Proceedings of the 6th IEEE Symposium on Computers and Communications*. 72–79.

CAMBAZOGLU, B. B. 2006. Models and algorithms for parallel text retrieval. Ph.D. dissertation, Bilkent University, Ankara, Turkey.

CAMBAZOGLU, B. B., JUNQUEIRA, F. P, PLACHOURAS, V., BANACHOWSKI, S., CUI, B., LIM, S., AND BRIDGE, B. 2010. A refreshing perspective of search engine caching. In *Proceedings of the 19th International Conference on World Wide Web*. ACM, New York, 181–190.

CAO, P. AND IRANI, S. 1997. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*. USENIX Association, Berkeley, CA, 18–18.

CHERKASOVA, L. AND CIARDO, G. 2001. Role of aging, frequency and size in Web caching replacement strategies. In *Proceedings of the 2001 Conference on High Performance Computing and Networking (HPCN'01)*. Lecture Notes in Computer Science, vol. 2110. Springer-Verlag, 114–123.

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* ACM, New York.

DE MOURA, E. S., DOS SANTOS, C. F., FERNANDES, D. R., SILVA, A. S., CALADO, P., AND NASCIMENTO, M. A. 2005. Improving Web search efficiency via a locality based static pruning method. In *Proceedings of the 14th International Conference on World Wide Web*, ACM, New York, 235–244.

FAGNI, T., PEREGO, R., SILVESTRI, F., AND ORLANDO, S. 2006. 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.

GAN, Q. AND SUEL, T. 2009. Improved techniques for result caching in web search engines. In *Proceedings of the 18th International Conference on World Wide Web*. ACM, New York, 431–440.

GARCIA, S. 2007. Search engine optimisation using past queries. Ph.D. dissertation. RMIT University.

JEONG, J. AND DUBOIS, M. 2003. Cost-sensitive cache replacement algorithms. In *Proceedings of 9th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 327–337.

JEONG, J. AND DUBOIS, M. 2006. Cache replacement algorithms with nonuniform miss costs. *IEEE Trans. Comput.* 55, 4, 353–365.

LIANG, S., CHEN, K., JIANG, S., AND ZHANG, X. 2007. Cost-aware caching algorithms for distributed storage servers. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*. 373–387.

LESTER, N., MOFFAT, A., WEBBER, W, ZOBEL, J. 2005. Space-limited ranked query evaluation using adaptive pruning. In *Proceedings of 6th International Conference on Web Information Systems Engineering*. Lecture Notes in Computer Science, vol. 3806. Springer-Verlag, 470–477.

LONG, X. AND 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*. ACM, New York, 257–266.

MARKATOS, E. P. 2001. On caching search engine query results. *Comput. Comm. 24*, 2, 137–143.

NTOULAS, A. AND CHO, J. 2007. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, 191–198.

OZCAN, R., ALTINGOVDE, I. S., AND ULUSOY, Ö. 2008a. Static query result caching revisited. In *Proceedings of the 17th International Conference on World Wide Web*. ACM, New York, 1169–1170.

OZCAN, R., ALTINGOVDE, I. S., AND ULUSOY, Ö. 2008b. Utilization of navigational queries for result presentation and caching in search engines. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM)*. ACM, New York, 1499–1500.

PASS, G., CHOWDHURY, A., AND TORGESON, C. 2006. A picture of search. In *Proceedings of the 1st international Conference on Scalable information Systems*. ACM, New York, 1.

PODLIPNIG, S. AND BÖSZÖRMÉNYI, L. 2003. A survey of Web cache replacement strategies. *ACM Comput. Surv. 35*, 4, 374–398.

RAMAKRISHNAN, R. AND GEHRKE, J. 2003. *Database Management Systems*. 3rd Ed., McGraw-Hill.

SARAIVA, P. C., SILVA DE MOURA, E., ZIVIANI, N., MEIRA, W., FONSECA, R., AND RIBERIO-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*. ACM, New York, 51–58.

SILVERSTEIN, C., MARAIS, H., HENZINGER, M., AND MORICZ, M. 1999. Analysis of a very large web search engine query log. *SIGIR Forum 33*, 1, 6–12.

STROHMAN, T. AND CROFT, W.B. 2007. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, 175–182.

TSEGAY, Y., TURPIN, A., AND ZOBEL, J. 2007. Dynamic index pruning for effective caching. In *Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management (CIKM)*. ACM, New York, 987–990.

TSEGAY, Y., PUGLISI, S.J., TURPIN, A., AND ZOBEL, J. 2009. Document compaction for efficient query biased snippet generation. In *Proceedings of the 31st European Conference on Information Retrieval*. Lecture Notes In Computer Science, vol. 5478. Springer-Verlag, 509–520.

TURPIN, A., TSEGAY, Y., HAWKING, D., AND WILLIAMS, H.E. 2007. Fast generation of result snippets in web search. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, New York, 127–134.

WEBBASE. 2007. Stanford University WebBase Project. www-diglib.stanford.edu/~testbed/doc2/WebBase.

WEBBER, W. AND MOFFAT, A. 2005. In search of reliable retrieval experiments. In *Proceedings of the 10th Australasian Document Computing Symposium*. 26–33.

XIE, Y. AND O'HALLARON, D. 2002. Locality in search engine queries and its implications for caching. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communication Societies*. IEEE Computer Society, 1238–1247.

YAHOO! 2009. http://developer.yahoo.com/search/web/V1/webSearch.html.

YOUNG, N.E. 2002. On-line file caching. *Algorithmica 33*, 3, 371–383.

ZETTAIR. 2007. The zettair search engine. http://www.seg.rmit.edu.au/zettair/.

Received November 2009; revised June 2010; accepted August 2010