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

RIFAT OZCAN, Turgut Ozal University
ISMAIL SENGOR ALTINGOVDE, Middle East Technical University
B. BARLA CAMBAZOGLU, Yahoo Labs, Barcelona
ÖZGÜR ULUSOY, Bilkent University

Web search engines are known to cache the results of previously issued queries. The stored results typically contain the document summaries and some data that is used to construct the final search result page returned to the user. An alternative strategy is to store in the cache only the result document IDs, which take much less space, allowing results of more queries to be cached. These two strategies lead to an interesting trade-off between the hit rate and the average query response latency. In this work, in order to exploit this trade-off, we propose a hybrid result caching strategy where a dynamic result cache is split into two sections: an HTML cache and a docID cache. Moreover, using a realistic cost model, we evaluate the performance of different result prefetching strategies for the proposed hybrid cache and the baseline HTML-only cache. Finally, we propose a machine learning approach to predict singleton queries, which occur only once in the query stream. We show that when the proposed hybrid result caching strategy is coupled with the singleton query predictor, the hit rate is further improved.

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, Experimentation, Performance

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

ACM Reference Format:
Ozcan, R., Altingovde, I. S., Cambazoglu, B. B., and Ulusoy, Ö. 2013. Second chance: A hybrid approach for dynamic result caching and prefetching in search engines. *ACM Trans. Web* 8, 1, Article 3 (December 2013), 22 pages.
DOI:http://dx.doi.org/10.1145/2536777

1. INTRODUCTION

Result caching is an important technique employed in many information retrieval systems. A result cache can serve the precomputed search results for a large number of queries. Hence, this technique helps a retrieval system to satisfy the low response latency and high query processing throughput requirements, especially under high user

A preliminary version of the article appeared as a short paper in *Proceedings of the 33rd European Conference on Information Retrieval* [Altingovde et al. 2011].

This research is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under grant numbers 108E008 and 110E135.

Authors' addresses: R. Ozcan (corresponding author), Turgut Ozal University; email: rifatozcan1981@gmail.com; I. S. Altingovde, Middle East Technical University; B. B. Cambazoglu, Yahoo Labs; Ö. Ulusoy, Bilkent University.

DOI:http://dx.doi.org/10.1145/2536777

query traffic volumes. In result caching, a commonly used approach is to create a static result cache that keeps the result pages associated with queries that frequently appear in the past query logs [Baeza-Yates and Saint-Jean 2003; Baeza-Yates et al. 2008; Markatos 2001]. Another feasible alternative is to employ a dynamic result cache that aims to maintain the recently served query results [Baeza-Yates et al. 2008; Markatos 2001; Saraiva et al. 2001]. In limited-capacity dynamic caches, an eviction policy (e.g., LRU) is used to delete the entries that are not likely to be requested in the near future. A hybrid cache that contains a static and a dynamic part is also shown to be feasible [Fagni et al. 2006]. The performance of result caches are evaluated in the literature by the widely known hit rate metric. Recent studies [Gan and Suel 2009; Ozcan et al. 2011a] show that query processing costs vary significantly, and hence a cost-based metric that considers the query execution times is more realistic.

A result cache entry typically stores an entire HTML result page[1], which is generated as a response to a query. In terms of space consumption, a better alternative is to store only the document IDs of the results in the page, that is, the so-called docID cache [Fagni et al. 2006]. Given the same amount of space, a docID cache can accommodate a larger number of query results compared to an HTML cache. Hence, a docID cache is likely to yield higher cache hit rates than an HTML cache. However, since the snippets are not stored in a docID cache, they have to be recomputed at each request for the cached query results, leading to a slight increase in query response latencies.

In the literature on result caching, the issues of eviction [Gan and Suel 2009], prefetching [Fagni et al. 2006; Lempel and Moran 2003], and admission [Baeza-Yates et al. 2007] are well investigated. In cache eviction, upon a cache miss, a selected entry is deleted from the cache. The objective is to maintain in the cache those entries that are more likely to lead to cache hits in the future. In result prefetching, the basic intuition is to exploit the spatial locality in the result page requests of the users. A request for the ith result page associated with a query provides an evidence that the $(i+1)$th result page will be requested in the near future. Hence, the successive result pages are proactively fetched and cached. In cache admission, given a query, the goal is to identify whether the query should be cached or not. Only the queries that are predicted to repeat in the near future are admitted to the cache. All three techniques are shown to improve the cache hit rate.

In this work, our objective is to further improve the performance of the state-of-the-art result caching techniques by a variety of new optimizations. To this end, we first propose a hybrid dynamic result cache that is composed of two parts: HTML and docID caches. In this hybrid cache, the query results are maintained either in both caches or only in the docID cache, that is, a result entry may be evicted from the HTML cache and yet it may remain in the docID cache. We show that this approach leads to an increase in the hit rates and a decrease in query response latencies compared to an HTML-only dynamic cache. As another contribution, we focus on prefetching of query result pages and introduce a variant of the adaptive prefetching strategy described in Fagni et al. [2006]. The experimental results show that the proposed hybrid cache still outperforms an HTML-only cache when prefetching is employed. Finally, we devise admission policies that decide whether a query result should be stored only in the docID cache or both in docID and HTML caches. In particular, we propose a machine learning approach to identify singleton queries (queries that occur in the query stream only once). The results of queries that are identified as singleton are stored only in the docID cache. The experimental results show that this strategy further increases the

[1]Although we call it an HTML page, in practice, there is no need to store the HTML tags and scripts, which can be added while generating the final result page returned to the user.

hit rate of the HTML cache within our hybrid result caching framework. The contributions of the article are as follows.

— First, we propose a hybrid dynamic result caching strategy which involves an HTML cache and a docID cache.
— Second, we propose a variant of the adaptive prefetching strategy introduced in Fagni et al. [2006] and show that result prefetching couples well with our hybrid caching strategy.
— Third, we introduce a machine learning approach to decide whether the results of a given query should be stored only in the docID cache or both in HTML and docID caches.
— Finally, using a realistic cost model and a real-life query log, we extensively evaluate the performance of the proposed optimizations.

The rest of the article is organized as follows. In Section 2, we briefly review the related work in the literature. In Section 3, we describe the proposed hybrid result caching strategy. We present the details of the proposed prefetching strategy in Section 4. Section 5 is devoted to the machine learning approach used for singleton query detection. The experimental results associated with each contribution are provided in the respective sections. We conclude in Section 6.

2. RELATED WORK

2.1. Search Result Caching

In the context of information retrieval, the caching mechanisms are extensively studied by many works in the literature. Essentially, it is possible to cache query results [Fagni et al. 2006; Gan and Suel 2009; Markatos 2001], a portion of the inverted index [Baeza-Yates et al. 2008; Zhang et al. 2008], or a combination of both [Baeza-Yates and Jonassen 2012; Baeza-Yates and Saint-Jean 2003; Baeza-Yates et al. 2008; Long and Suel 2005; Saraiva et al. 2001]. Furthermore, caching strategies for documents and snippets are also proposed [Ceccarelli et al. 2011; Tsegay et al. 2009; Turpin et al. 2007]. In this work, we focus only on caching of query results.

In an early work, Markatos [2001] compares the static and dynamic caching policies. The static cache content is filled using the frequent queries obtained from a query log. The dynamic cache changes its content based on the query stream, applying a cache eviction policy such as the least recently used (LRU) or least frequently used (LFU) policies (a detailed survey on cache replacement policies is provided in [Podlipnig and Boszormenyi 2003]). Markatos shows that a static cache achieves better hit ratios for small cache capacities; however, a dynamic cache is preferable for large-capacity caches. Fagni et al. [2006] propose a hybrid of static and dynamic caches, called static-dynamic cache (SDC), which outperforms both static-only and dynamic-only caches.

Early studies on query result caching try to improve the hit rate. More recent studies show that the processing time cost of queries significantly vary, and they suggest considering the processing time cost of queries when evaluating the performance of a result cache [Gan and Suel 2009; Ozcan et al. 2011a]. Sazoglu et al. [2013a] take a step beyond and propose a financial cost metric for result caching. In this study, we provide our experimental results in terms of the hit rate and query response latency metrics.

In the literature, cached query results are assumed to be stored as HTML pages, which could be returned to the users without any further processing. As an alternative, Fagni et al. [2006] mention the possibility of a docID cache that stores only the IDs of documents matching the query. However, in their study, they do not compare the two types of result caches. In a more recent study [Marin et al. 2010], a docID cache, referred to as the top *k* cache, is investigated in a distributed query processing framework, also involving an HTML result cache and a posting list cache. Similarly, in a previous work, we propose a five-level static cache architecture [Ozcan et al. 2012] that contains separate caches for HTML result pages, docID results, posting lists, intersections of posting lists, and documents. In the current study, we focus only on the HTML and docID result caches and propose a hybrid dynamic cache architecture. We are not aware of any prior work that evaluates the performance of a standalone docID cache or a hybrid cache maintaining both HTML results and document IDs.

A recently active research topic is to maintain the freshness of result cache entries. Cambazoglu et al. [2010] analyze the effectiveness of assigning fixed time-to-live (TTL) values to each query result in the cache. In contrast, Alici et al. [2012] propose using query-specific TTL values and show that this strategy outperforms the fixed TTL strategy. Sazoglu et al. [2013b] investigate hybrid strategies for setting TTL values. Other works [Alicia et al. 2011; Blanco et al. 2010a, 2010b; Bortnikov et al. 2011; Jonassen 2012] propose more sophisticated mechanisms. Herein, we do not further discuss these studies, since the cache freshness issue is not in the scope of our work.

2.2. Result Page Prefetching

The main goal in prefetching is to prepare and retrieve the successive result page(s) before they are actually requested by the user submitting the query. In practice, when a request comes for a $\langle Q, i\rangle$ pair (the ith result page of query Q), the Web search engine proactively fetches the F successive result pages associated with the query, that is, $\langle Q, i+1\rangle, \langle Q, i+2\rangle, \ldots, \langle Q, i+F\rangle$.[2] Earlier works show that prefetching significantly improves the cache hit rate [Fagni et al. 2006; Lempel and Moran 2003]. However, this technique incurs additional load on the backend servers due to the overhead of preparing the result pages that are not yet requested. Moreover, in order to make space for the prefetched pages, the result cache may be forced to evict some entries that would potentially lead to a cache hit [Lempel and Moran 2003]. Therefore, prefetching must be handled by taking into account both the additional load on the backend servers and the eviction of entries in the result cache.

In the literature, various improvements are proposed over the basic prefetching mechanism just described. The analysis of Fagni et al. [2006] reveals that whenever a miss occurs for the first page of results, the probability that the user will request the next page (in this case, the second result page) is approximately 0.1. However, if a miss occurs for a page other than the first page of results, then the probability of requesting the next page increases to 0.5. Based on this observation, the authors propose an adaptive prefetching policy. In a nutshell, instead of prefetching a fixed number of successive pages, their method decides on the value of the prefetching factor F according to the requested page number.

The probability driven cache (PDC) policy proposed by Lempel and Moran [2003] exploits the result page browsing behavior of the search engine users while deciding on the cache content. This work does not propose another prefetching strategy but applies the aforementioned basic prefetching strategy on top of PDC. The authors show that prefetching of query results considerably increases the hit rate. However, as they essentially focus on maximizing the hit rate, their evaluations do not take into account the additional load incurred on the backend servers due to prefetching.

In another study, Lempel and Moran [2004] try to find the optimal number of pages to be prefetched in a distributed search architecture using a document-based-partitioned index. They present a formal cost model (time and space complexity) for query processing in this environment. Based on that cost model, they formulate an

[2]F is a parameter referred to as the prefetching factor.

optimization problem (the objective is to minimize the total cost) and propose optimal and approximate solutions. Their goal is to find a constant optimal prefetching factor (F) independent of the currently requested pages.

2.3. Feature-Based Caching

Feature-based caching policies extract certain features to decide which queries should be admitted into the cache or which queries should be evicted. In the literature, Baeza-Yates et al. [2007] propose cache admission policies based on some query features (length and frequency) for their two-segment cache, where one segment is reserved for queries that are considered to be valuable according to an admission policy and the other segment is reserved for the remaining queries. In a later work, Gan and Suel [2009] extract a number of features for each query from the query log and distribute queries into different buckets based on these features. They evict queries starting from the most rarely accessed bucket when the cache is full.

In this article, we also propose a feature-based caching strategy, where the goal is to predict queries that are issued only once (singleton queries). We incorporate this singleton query predictor into our hybrid result caching framework. Although there are several efforts on classifying long and rare queries in the context of search effectiveness [Bailey et al. 2010; Broder et al. 2007], we are not aware of any work that tries to learn singleton queries for caching, as we have done in this work.

3. HYBRID RESULT CACHING

3.1. Motivation

The common practice in query result caching is to store HTML result pages, each typically containing titles, URLs, and snippets of result documents. If one stores only the result document IDs (the docID-only cache), a considerably larger number of query results can fit into the cache, eliminating the processing need for many queries. In this case, however, result snippets have to be generated upon every cache hit, increasing query response times.

Herein, we propose a hybrid result caching strategy, where the available cache space is split into an HTML part and a docID part. The motivation behind this strategy is to exploit the previously-mentioned trade-off between caching more document IDs and the overhead of snippet generation. In the proposed strategy, the document IDs associated with a query may continue to reside in the docID part after the corresponding HTML result page is evicted from the HTML part. If the query is issued again, it can be answered using the result entry in the docID part, incurring an additional snippet generation cost. In practice, this cost is relatively minor compared to the cost of recomputing the query results at the search backend.

3.2. Algorithm

In our hybrid result caching strategy, the cache space is split into an HTML part and a docID part.[3] The query results are cached as follows (Algorithm 1). When a query q is seen for the first time (i.e., a miss in the result cache), the results are computed and added to both HTML and docID parts (Lines 2–4). As long as q remains in the HTML part of the cache, it is answered by this part, and its statistics are updated (Lines 5–7). Suppose that at some point in time, q becomes the LRU item. In this case, it is discarded from the HTML part, but its (incomplete) results still reside in the docID part. In a sense, this approach gives a second chance to q. If the query is repeated soon,

[3]We assume that both the baseline HTML-only cache and the proposed hybrid cache (in both parts) use the LRU cache replacement policy for eviction.

ALGORITHM 1: The Second Chance Caching Algorithm

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

it is likely to be a hit in the docID part (Lines 8–11). In this case, the backend system generates the snippets missing in the result page, which is then returned to the user and also placed in the HTML part.

3.3. Cost Model and Scenarios

Processing a query q in a large-scale search engine involves four major steps: (i) fetching posting lists for the query terms from the disk (C_q^{list}), (ii) decompressing the lists and computing the top k results (C_q^{rank}), (iii) fetching k result documents from the disk (C_q^{doc}), and (iv) computing the snippets (C_q^{snip}). In Algorithm 1, we indicate the costs incurred in our hybrid cache in case of a cache miss, an HTML cache hit, or a docID cache hit.

In practice, a search engine architecture involves many search clusters [Cambazoglu and Baeza-Yates 2011]. Each search cluster includes hundreds of nodes that store a part of the inverted index and document collection. This means that for a given search cluster that handles a query, Steps (i) and (ii) are executed in all nodes of the cluster. The local (partial) results for the query are then sent to a broker node, which merges them and computes the final top k document IDs.[4] Finally, the corresponding documents are accessed to compute the snippets (Steps (iii) and (iv)) and generate the HTML result page, which is returned to the user and stored in the result cache at the broker.

Assuming the previously-mentioned cost model, we further consider two key issues to make our setup as realistic as possible: caching of lists and documents and partitioning of documents among the search nodes. We start with the first issue. Steps (i) and (iii) mentioned before requiring-making disk accesses to fetch the inverted lists and documents into memory, respectively. To reduce or totally avoid the disk access costs incurred due to these two steps, search engines cache posting lists and documents. In this work, we consider separate in-memory inverted list and document caches that can keep a certain fraction of the inverted index and document collection, respectively. In the "no caching" scenario (0%), we assume that all lists and documents reside on the disk. In contrast, the "full caching" scenario (100%) assumes that all lists and documents are kept in the memory. These two extreme cases correspond to search engines

[4]We omit the cost of transferring (and merging) the partial results, as this cost is similar for most queries and forms a relatively small part of the overall cost.

with very low and very high financial budgets. We also consider scenarios where the search system, depending on its financial resources, can afford caching only a certain fraction of lists and documents (10%, 30%, 50%, 70%, or 90%). To decide which lists to cache, we use the popularity/size metric [Baeza-Yates et al. 2008], where the former is the access frequency of a list in a training query log and the latter is the size of the list. While caching documents, we process the training queries over the collection and determine the number of times a document appears in the top 10 results. For both types of items, the items with the highest metric values are cached without exceeding the cache capacity.

The second issue that we have to consider is the assignment of documents in an N-node search cluster. This determines how the costs in Steps (iii) and (iv) are computed. Once the broker node for a query determines the top k result IDs for a query ($k \ll N$), we consider three possible document assignment scenarios: "random", "clustered", and "partially clustered". In the random assignment scenario, we assume that the documents are randomly allocated to the nodes. Hence, we can expect that each of the top k documents resides on a different node. In this case, the document access and snippet generation steps take place at each node in parallel. In effect, the total cost becomes approximately equal to the cost of executing Steps (iii) and (iv) for only one document (i.e., as if $k = 1$). In the clustered assignment scenario, we assume that all documents that are relevant to the query are available in a single node. Hence, the cost of Steps (iii) and (iv) is computed for all k documents. Finally, in the partially clustered assignment scenario, we assume that half of the results come from a single node. Hence, we expect that Steps (iii) and (iv) would take time as if they are executed for $k/2$ documents. Throughout the article, unless stated otherwise, we assume random document assignment, since this is a more realistic scenario.

3.4. Experiments

3.4.1. Dataset and Query Log. We use a collection of 2.2 million webpages obtained from the ODP Web directory.[5] The index file includes only the document IDs and term frequencies. After compression (using the Elias-γ encoding scheme [Elias 1975]), the index takes about 750MB on disk.

As the query sample, we use 16.8 million queries from the AOL query log [Pass et al. 2006]. The queries are filtered such that all query terms appear in our collection. We also filter queries that do not match any documents, since there is no result to cache for such queries.[6] To understand the impact of filtering on the characteristics of the original query log, we compared the frequency distribution of queries in the original and filtered query logs. We observed that both logs exhibit similar power law characteristics. Furthermore, temporal locality characteristics (i.e., interarrival times as computed in [Fagni et al. 2006]) of the two logs were found to be quite similar. The final (filtered) query log used in our study contains around 14.9 million queries, of which 5.4 million are unique. In the simulations, the first 8 million queries are used as the training set and the remaining 6.9 million queries are used as the test set. Queries are sorted based on their submission times.

3.4.2. Simulation Setup. In Tables I and II, we list the cost formulas and the parameters used in these formulas, respectively. The decompression and scoring times (per posting) are empirically obtained from our data. We rely on the literature in order to set the parameters related to snippet generation. Turpin et al. measured the generation time for a single snippet as 2.1ms, excluding the disk access latency (see Table 3 in

[5] `http://www.dmoz.org`

[6] We employ conjunctive (AND) query semantics, that is, match documents that include all query terms.

Table I. Cost Formulas

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

Note: t denotes a term in q, I_t denotes the inverted list of t, and d denotes a document in the query result set R_q.

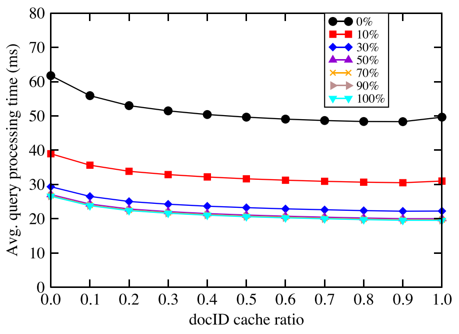
Table II. Parameters

Parameter	Value	Parameter	Value
Size ratio of cache items (S)	64	Number of results per query (k)	10
Disk latency (D_ℓ)	12.7 ms	Decompression per posting (P_{d})	100 ns
Disk block read (D_{br})	4.9 μs	Ranking per posting (P_{r})	200 ns
Disk block size (D_{bs})	512 bytes	Snippet computation per byte (P_{s})	100 ns

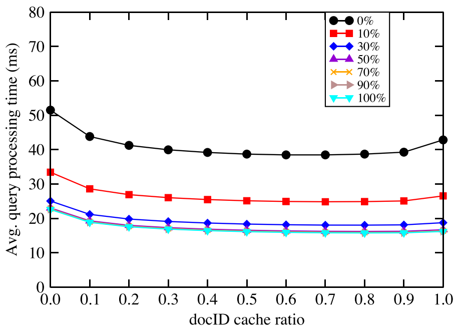
[Turpin et al. 2007]). Similarly, in a recent work, Arroyuelo et al. [2012] report the per-snippet generation time for compressed document text to be within a range from 2.0ms to 11.7ms, assuming different compression techniques (see Table 3 in [Arroyuelo et al. 2012]). In our case, if we assume a snippet generation cost of 100ns/byte or 1,000ns/byte, the snippet of an average-size document (around 16 KB for our collection) can be generated in 1.6ms or 16ms, respectively (excluding the disk access latency). Herein, we report simulation results using both 100ns/byte and 1,000ns/byte, the former representing a state-of-the-art fast snippet generation algorithm and the latter representing a more sophisticated snippet generation algorithm.

In Table II, we specify the ratio (S) between the size of a result entry in the HTML cache and the size of an entry in the docID cache. All experiments reported next specify the cache capacity in terms of the number of HTML result pages that can fit into the cache. Hence, our findings are valid as long as the ratio between the HTML and docID result entry sizes is preserved, regardless of the absolute values. Assuming that a single document ID may take around 4 bytes and a result URL and snippet (containing 20 terms) may take around 256 bytes, we set this ratio to 64. We anticipate that even if the items are compressed, the ratio would be similar. We note that, in this set of experiments, we ignore prefetching of query results and assume that, for each query request, the search engine computes and stores the results for the top k documents upon a cache miss. In Section 4, we elaborate on the issue of prefetching in more detail, both for the baseline HTML-only cache and the proposed hybrid cache. We compare the performance of the hybrid result cache with the HTML-only cache and the docID-only cache. We experiment with three different cache capacities that are representatives of small (10K), medium (100K), and large (500K) caches for our query log.

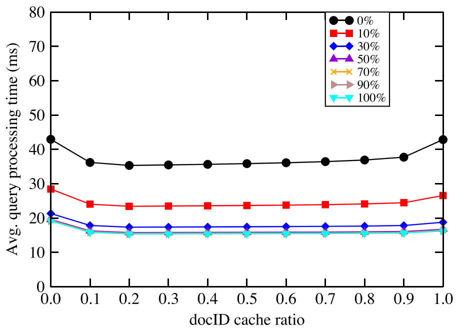
3.4.3. Results. Figure 1 demonstrates the performance of the small, medium, and large hybrid result caches assuming a snippet generation cost of 100ns/byte and for varying levels of list and document caching. According to the figure, the HTML-only cache (docID cache ratio is 0) is always inferior to the docID-only cache (docID cache ratio is 1) and hybrid cache (docID cache ratio is in [0, 1]). The docID-only cache can outperform the hybrid cache only for the small cache scenario and only when more than 30% of the lists and documents are cached (Figure 1(a)). On the other hand, for both medium and large cache scenarios, the proposed hybrid cache achieves the lowest query processing times regardless of the fraction of cached lists and documents. In the case of the medium result cache, the best performance is achieved with docID



(a) Small cache (10K), snippet cost: 100ns/byte.



(b) Medium cache (100K), snippet cost: 100ns/byte.



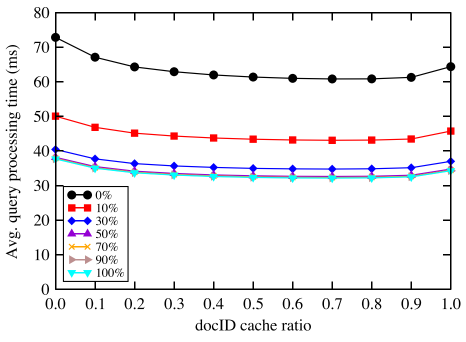
(c) Large cache (500K), snippet cost: 100ns/byte.

Fig. 1. The performance of the hybrid result cache for varying docID cache ratios and varying cache capacities with 100ns/byte snippet computation cost and random document assignment ($k = 1$). Each curve represents a particular percentage of list and document caching.

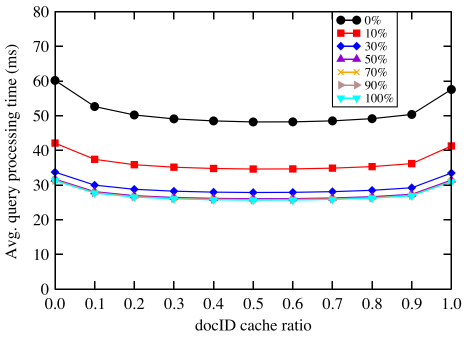
cache ratios between 0.7 and 0.9 (Figure 1(b)). In the case of the large cache, the most efficient hybrid cache configuration reserves 20% of the cache capacity for document IDs and the rest for HTML result pages (Figure 1(c)). In Figure 2, we report similar results assuming a snippet generation cost of 1,000ns/byte. In this case, independent of the result cache capacity, the hybrid cache performs better than both HTML-only and docID-only caches. Moreover, the performance gap of the hybrid cache over the HTML-only and docID-only caches is more emphasized.

These results lead to the following observations. First, as the result cache size gets larger, a larger portion of the hybrid cache should be reserved for the HTML results in order to obtain the best performance. Second, as the snippet generation cost gets higher, our hybrid cache outperforms the docID-only cache with a larger margin. Finally, when larger fractions of lists and documents are cached, the absolute query processing times drop, but the performance gain due to the hybrid cache remains the same. For instance, assuming a snippet generation cost of 1,000ns/byte, the improvements range from 5.5% to 6.2% (for the small cache), from 16.1% to 17.1% (for the medium cache), and from 13.5% to 14.8% (for the large cache).

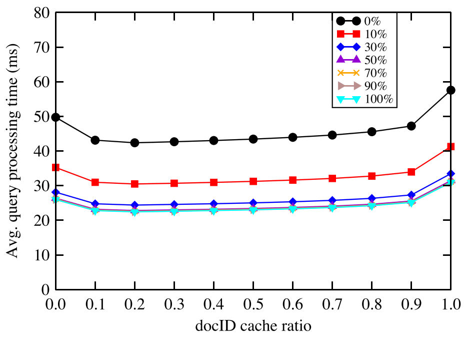
Next, we next analyze the contribution of each query cost component to the overall query processing cost under each cache configuration. In these experiments, we only consider a large result cache that can hold 500K HTML result pages. We simulate low- and moderate-budget search engines that can afford caching 10% and 50% of lists and documents, respectively. Figure 3 presents the distribution of average query processing time across different query cost components for HTML-only, docID-only, and



(a) Small cache (10K), snippet cost: 1,000ns/byte.



(b) Medium cache (100K), snip. cost: 1,000ns/byte.

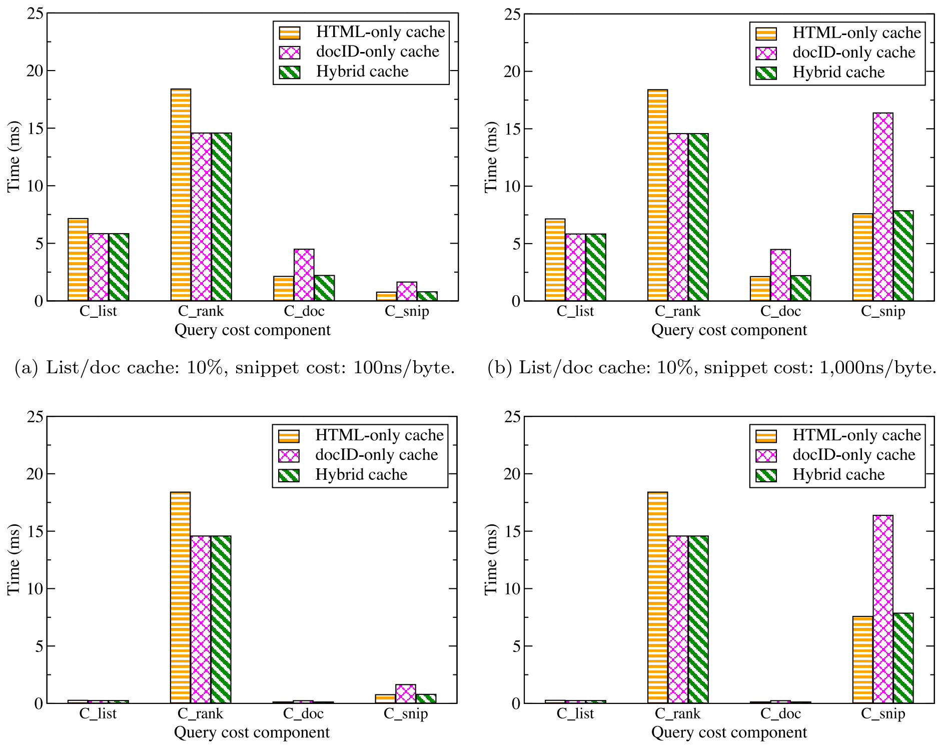


(c) Large cache (500K), snippet cost: 1,000ns/byte.

Fig. 2. The performance of the hybrid result cache for varying docID cache ratios and varying cache capacities with 1,000ns/byte snippet computation cost and random document assignment ($k = 1$). Each curve represents a particular percentage of list and document caching.

hybrid caches (the labels C_list, C_rank, C_doc, and C_snip in the figures correspond to C_q^{list}, C_q^{rank}, C_q^{doc}, and C_q^{snip}, respectively). For the hybrid cache, we assume the best-performing docID cache ratio, where 80% of the cache space is reserved for the HTML cache and 20% is reserved for the docID cache. The first observation drawn from the figure is that the disk access costs for fetching the posting lists and documents constitute a significant fraction of the total query processing cost when only 10% of lists and documents are cached in the memory. On the other hand, when 50% of lists and documents are cached, the overhead of disk accesses becomes negligible. This motivates the use of larger list and document caches in practice (if affordable). Another observation is that the total snippet cost (C_snip) constitutes about 3% of the total query processing cost if we assume a snippet generation cost of 100ns/byte. This cost becomes more pronounced (about 25%–30% of the overall cost) when the snippet generation cost is increased to 1,000ns/byte.

The configuration of the result cache has a major impact on the query cost components. According to Figure 3, the docID-only cache leads to an increase in the percentage of time spent for fetching the documents (C_doc) and snippet generation (C_snip) with respect to the HTML-only cache. The hybrid cache reduces the percent share of list fetching and document ranking with a negligible increase in the percent share of document fetching and snippet generation. This is because the hybrid cache decreases the hit rate of the HTML-only cache by only 1%–2%, that is, extra document access and snippet generation is needed for a small fraction of queries. In short, the proposed



(a) List/doc cache: 10%, snippet cost: 100ns/byte. (b) List/doc cache: 10%, snippet cost: 1,000ns/byte.

(c) List/doc cache: 50%, snippet cost: 100ns/byte. (d) List/doc cache: 50%, snippet cost: 1,000ns/byte.

Fig. 3. Query cost components for HTML-only, docID-only, and hybrid caches.

hybrid cache is a very good compromise between the HTML-only and docID-only caches, since it performs as good as the better performing of these two caches (for each query cost component).

4. RESULT PAGE PREFETCHING

4.1. Motivation

As discussed in Section 2.2, prefetching is an important mechanism that can increase the performance of a result cache. Herein, our goal is to adapt and incorporate the state-of-the-art prefetching mechanisms into the proposed hybrid result caching strategy. Our hybrid cache leads to new opportunities in terms of prefetching, since result pages that are not yet requested can be prefetched into the docID part of the hybrid cache, instead of the HTML part of the cache. This can be a remedy to the biggest disadvantage of prefetching, that is, increased cache space consumption due to the prefetched results. Furthermore, for prefetched results that are stored in the docID part of the cache and are never requested, we avoid the snippet generation cost. In what follows, we discuss how we couple the proposed hybrid cache with various prefetching strategies and provide experimental results using the previously presented cost model.

4.2. Algorithm

As the baseline prefetching strategies, we evaluate two different alternatives: *fixed* and *adaptive*. In the fixed prefetching strategy, a fixed number of result pages following the

ALGORITHM 2: Fixed Prefetching Strategy Assuming an HTML-Only Result Cache

```
Input: $H$: HTML-only cache, $\langle Q, i\rangle$: requested result page of query $Q$, $F$: prefetching factor.
if $q \in H$ then
    get $\langle Q, i\rangle$ from $H$;
    update statistics of $\langle Q, i\rangle$ in $H$ ▷ $C_Q = 0$;
else
    evaluate $Q$ over the backend and obtain $\langle Q, i\rangle$ ▷ $C_Q = C_Q^{\text{list}} + C_Q^{\text{rank}} + C_{\langle Q,i\rangle}^{\text{doc}} + C_{\langle Q,i\rangle}^{\text{snip}}$;
    insert $\langle Q, i\rangle$ into $H$;
    prefetch $\langle Q, i+1\rangle$ to $\langle Q, i+F\rangle$ into $H$ ▷ $C_P = F \times (C_{\langle Q,i\rangle}^{\text{doc}} + C_{\langle Q,i\rangle}^{\text{snip}})$;
return $\langle Q, i\rangle$;
```

requested result page are prefetched if the requested page is not already cached. More specifically, if the request for result page i leads to a cache miss, the consecutive pages, $i + 1$ through $i + F$, are prefetched, where F is the prefetching factor. In the adaptive strategy [Fagni et al. 2006], the prefetching is performed based on the rank of the requested result page and whether the request leads to a cache hit or a miss. If the first result page is requested by the user and it is not cached, only the second result page is prefetched. If the user asks for the second page, this strategy prefetches several consecutive pages based on the prefetching factor, regardless of whether the requested page exists in the cache or not. Finally if a request comes for pages other than the first or second result pages, prefetching is performed only for the case of a cache miss. The pseudocodes for the fixed and adaptive prefetching strategies are displayed in Algorithms 2 and 3, respectively.

In our work, we adapt the baseline prefetching strategies so that they store the prefetched pages in the docID part of the cache. For the sake of brevity, we show the algorithm (Algorithm 4) only for the modified version of the fixed prefetching strategy, as this should be adequate to convey the general idea. Additionally, we describe a variant of the adaptive prefetching strategy that is tailored for our hybrid cache. In this variant, which we refer to as adaptive+, when the second result page is requested, the prefetched pages are stored in the HTML part of the cache, instead of the docID part. This is because the request of the second result page is a strong indicator of the future requests for the following result pages [Fagni et al. 2006].

We revise the notation used in the previous section for the cost model to be able to represent the costs when prefetching is involved. In particular, we use a new notation for the costs associated with reading a document and snippet generation. In Section 3.3, these costs were defined on a per-query basis. In the case of prefetching, we deal with result page requests. Herein, we denote by $C_{\langle Q,i\rangle}^{\text{doc}}$ the cost of fetching the documents associated with the ith result page of query Q. Similarly, $C_{\langle Q,i\rangle}^{\text{snip}}$ denotes the cost of snippet generation for the ith result page. In addition, we introduce two additional cost parameters: C_Q denotes the cost of processing query Q to generate the result page requested by user, and C_P denotes the cost of prefetching additional result pages.

4.3. Experiments

4.3.1. Simulation Setup. We evaluate the performance of the prefetching strategies by the hit rate and average query processing time metrics. We use the former metric for the sake of comparability with previous studies. For the latter metric, we consider both including the prefetching cost ($C_Q + C_P$) and excluding it (C_Q), because prefetching always increases the workload of the system, but it may not affect the response time if

ALGORITHM 3: Adaptive Prefetching Strategy Assuming an HTML-Only Result Cache

```
Input: $H$: HTML-only cache, $\langle Q, i\rangle$: requested result page of query $Q$, $F$: prefetching factor.
if $i = 1$ then
    if $\langle Q, 1\rangle \in H$ then
        get $\langle Q, 1\rangle$ from $H$ $\triangleright$ $C_Q = 0$;
    else
        prefetch $\langle Q, 1\rangle$ and $\langle Q, 2\rangle$ $\triangleright$ $C_Q = C_Q^{\text{list}} + C_Q^{\text{rank}} + C_{\langle Q,1\rangle}^{\text{doc}} + C_{\langle Q,1\rangle}^{\text{snip}}$ $C_P = C_{\langle Q,2\rangle}^{\text{doc}} + C_{\langle Q,2\rangle}^{\text{snip}}$;
else if $i = 2$ then
    if $\langle Q, 2\rangle \in H$ then
        get $\langle Q, 2\rangle$ from $H$ and prefetch pages $\langle Q, 3\rangle$ through $\langle Q, (F+2)\rangle$ $\triangleright$ $C_Q = 0$
        $C_P = C_Q^{\text{list}} + C_Q^{\text{rank}} + (F \times (C_{\langle Q,i\rangle}^{\text{doc}} + C_{\langle Q,i\rangle}^{\text{snip}}))$;
    else
        prefetch $\langle Q, 2\rangle$ through $\langle Q, (F+1)\rangle$ $\triangleright$ $C_Q = C_Q^{\text{list}} + C_Q^{\text{rank}} + C_{\langle Q,2\rangle}^{\text{doc}} + C_{\langle Q,2\rangle}^{\text{snip}}$
        $C_P = (F-1) \times (C_{\langle Q,i\rangle}^{\text{doc}} + C_{\langle Q,i\rangle}^{\text{snip}})$;
else
    if $\langle Q, i\rangle \in H$ then
        get $\langle Q, i\rangle$ from $H$ $\triangleright$ $C_Q = 0$;
    else
        prefetch $\langle Q, i\rangle$ through $\langle Q, (F+i-1)\rangle$ $\triangleright$ $C_Q = C_Q^{\text{list}} + C_Q^{\text{rank}} + C_{\langle Q,i\rangle}^{\text{doc}} + C_{\langle Q,i\rangle}^{\text{snip}}$
        $C_P = (F-1) \times (C_{\langle Q,i\rangle}^{\text{doc}} + C_{\langle Q,i\rangle}^{\text{snip}})$
return $\langle Q, i\rangle$;
```

ALGORITHM 4: Fixed Prefetching Strategy Assuming a Hybrid Result Cache

```
Input: $H$: HTML part of the cache, $D$: docID part of the cache, $\langle Q, i\rangle$: requested result page of
        query $Q$, $F$: prefetching factor.
if $q \in H$ then
    get $\langle Q, i\rangle$ from $H$;
    update statistics of $\langle Q, i\rangle$ in both $H$ and $D$ $\triangleright$ $C_Q = 0$;
else if $q \in D$ then
    get doc ids from $D$ and compute snippets to obtain $\langle Q, i\rangle$ $\triangleright$ $C_Q = C_{\langle Q,i\rangle}^{\text{doc}} + C_{\langle Q,i\rangle}^{\text{snip}}$;
    insert $\langle Q, i\rangle$ into $H$;
    update statistics of $\langle Q, i\rangle$ in $D$;
else
    evaluate $Q$ over the backend and obtain $\langle Q, i\rangle$ $\triangleright$ $C_Q = C_Q^{\text{list}} + C_Q^{\text{rank}} + C_{\langle Q,i\rangle}^{\text{doc}} + C_{\langle Q,i\rangle}^{\text{snip}}$;
    insert $\langle Q, i\rangle$ into $H$ and $D$;
    prefetch $\langle Q, i+1\rangle$ to $\langle Q, i+F\rangle$ into $D$ $\triangleright$ $C_P = 0$;
return $\langle Q, i\rangle$;
```

it is performed during the idle cycles of the backend search system. In the experiments, we use a large result cache that can store up to 500K HTML result pages.

4.3.2. Results. In Figures 4(a), 4(b), and 4(c), we compare the performance of the fixed and adaptive prefetching strategies for an HTML-only cache. These strategies are proposed in earlier studies, where they were evaluated in terms of the hit rate metric. To

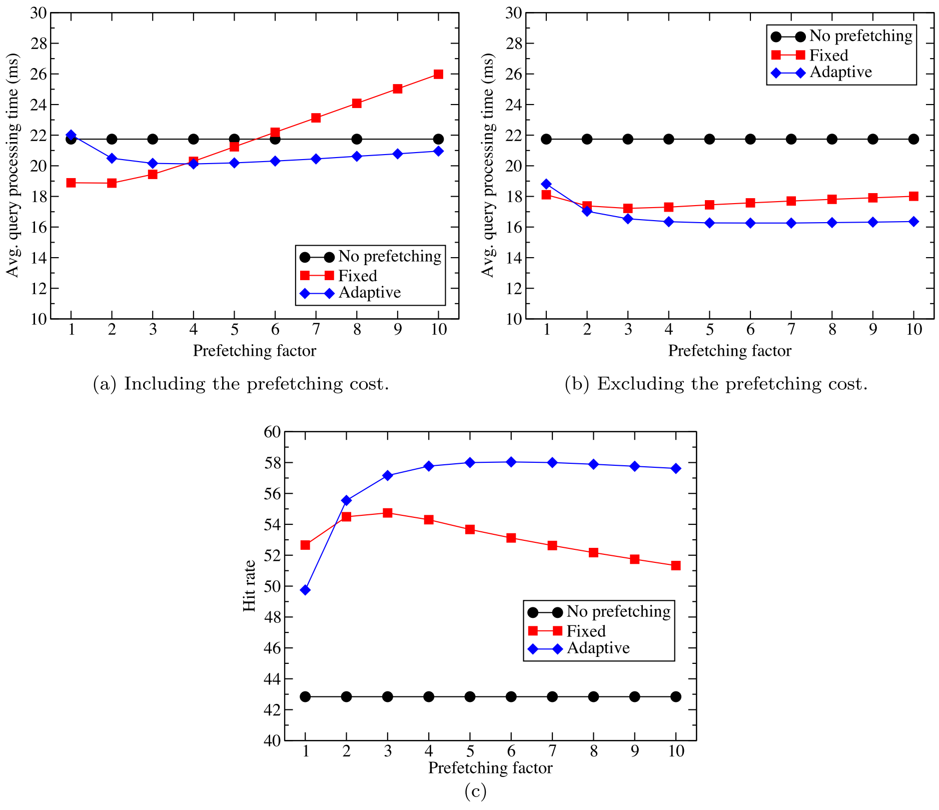


Fig. 4. The performance of the HTML-only cache with the fixed and adaptive prefetching strategies for varying prefetching factor values.

the best of our knowledge, this is the first time they are compared in terms of the query processing cost metric. According to Figures 4(a) and 4(b), both prefetching strategies reduce the average query processing time considerably compared to the case without any prefetching. Interestingly, the adaptive strategy incurs more prefetching overhead than the fixed strategy. If we include the prefetching cost in timings (Figure 4(a)), the fixed strategy performs better than the adaptive strategy. More specifically, the fixed strategy achieves up to 13.2% reduction in the average query processing time while the adaptive strategy reduces the processing time by at most 7.4%. However, if the prefetching cost is excluded (Figure 4(b)), the adaptive strategy performs better than the fixed strategy, as the reduction in the average query processing time goes up to 25.2%. In terms of the hit rate metric, the adaptive strategy achieves better results as well.

The value of the best prefetching factor for a given strategy depends on whether the prefetching cost is included or not and the metric that is being optimized, that is, average query processing time or hit rate. In terms of the query processing time metric, the fixed strategy achieves the best performance for $F = 2$ (including the prefetching cost) and $F = 3$ (excluding the prefetching cost). The adaptive strategy yields the best result when $F = 4$ (including the prefetching cost) and $F = 6$ (excluding the prefetching cost). The fixed and adaptive strategies have the highest hit rates when the prefetching factor is $F = 3$ and $F = 6$, respectively.

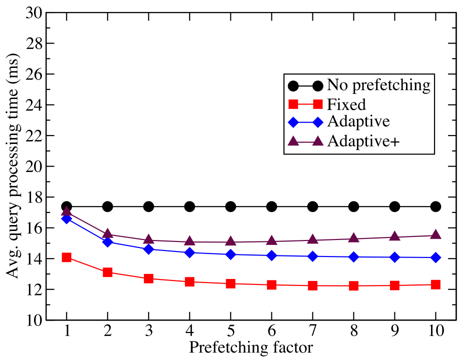
In Figures 5(a), 5(b), and 5(c), we compare the performance of the fixed, adaptive, and adaptive+ prefetching strategies for the hybrid cache, assuming a snippet cost of 100ns/byte. For each prefetching strategy, we evaluate the performance of the hybrid cache by varying the docID cache ratio from 0 to 1 (with a step size of 0.1). The best-performing hybrid cache configuration for a given prefetching strategy is chosen to be the one that yields the lowest average query processing time over a range of prefetching factor values (between 1 and 10). In the figures, we report the results only for these best performing configurations.

The results indicate that when the prefetching strategies are coupled with the hybrid cache, the best-performing cache configuration is different from the best-performing configuration under no result prefetching. Without prefetching, the best hybrid cache configuration is obtained for a docID cache ratio of 0.3.[7] In contrast, when we apply the fixed and adaptive strategies, the best cache configurations are obtained with docID cache ratios of 0.7 and 0.5, respectively. Interestingly, employing the adaptive+ strategy for prefetching does not affect the best performing configuration for the hybrid cache. A possible explanation is the following. For the fixed and adaptive strategies, prefetched pages are always stored in the docID part of the cache, and this requires increasing the ratio of the docID part in the hybrid cache. In contrast, the adaptive+ strategy, by definition, can prefetch results into any of the two cache parts, and thus their relative ratio does not have to be changed. This means that the proposed adaptive+ strategy is robust to the changes in the prefetching factor, that is, once the best cache configuration is deployed to the system, the search engine can vary the prefetching factor or totally ignore prefetching without a need to re-configure the cache.

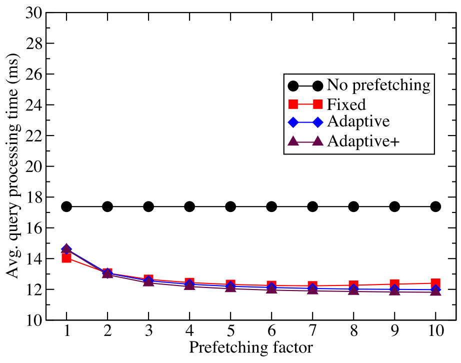
We observe that the fixed strategy is again the most efficient prefetching strategy when the prefetching cost is included (Figure 5(a)). However, when the prefetching cost is excluded (Figure 5(b)), the best results are obtained by the adaptive+ prefetching strategy. In a separate experiment, which is not reported here, we observed that, as the snippet generation cost increases, the performance gap between the adaptive+ and adaptive strategies become larger. For a more sophisticated snippet generation algorithm (assuming 1,000ns/byte for snippet generation), the adaptive+ strategy provides 8.2% reduction in the average query processing time compared to the adaptive strategy.

In Figure 5(c), we compare the hit rates. The hit rates are presented separately for the HTML and docID parts of the hybrid cache. Obviously, for the hybrid cache with no prefetching, the HTML and docID cache hit rates are constant, regardless of the prefetching factor. We see that the HTML cache hit rates for the fixed and adaptive strategies are also constant, as they only prefetch into the docID part of the cache. These latter strategies yield lower HTML cache hit rates compared to the hybrid cache with no prefetching, because they reserve a smaller fraction of the cache space for the HTML cache, as previously discussed. On the other hand, these two strategies considerably improve the docID cache hit rate as the prefetching factor increases. Finally, the adaptive+ strategy increases the hit rates both for HTML and docID parts of the cache, because it prefetches both types of items. In particular, the adaptive+ strategy increases the HTML and docID cache hit rates by 18.6% and 44.1%, respectively, compared to the hybrid cache with no prefetching. Our findings reveal that the proposed adaptive+ strategy suits well to hybrid result caching

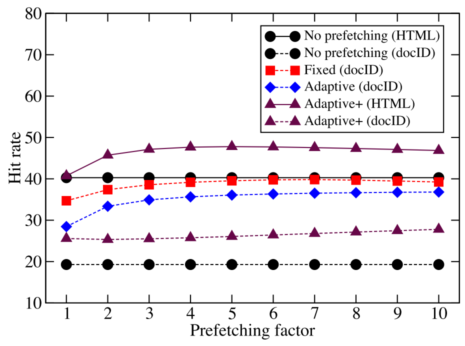
[7]Note that this value is different from the 0.2 value reported for the large cache scenario in the previous section. This is because the requests for different result pages of the same query should be treated separately for the purposes of prefetching.



(a) Including the prefetching cost.



(b) Excluding the prefetching cost.



(c)

Fig. 5. The performance of the hybrid cache with the fixed, adaptive, and adaptive+ prefetching strategies. In (c), we separately provide the hit rates for the HTML and docID parts of the hybrid caches coupled with each prefetching strategy.

and yields promising results in terms of both hit rate and query processing time metrics.

In Figures 6(a) and 6(b), we compare the best results observed in the previously presented plots. When the prefetching cost is included (Figure 6(a)), the fixed strategy is the best for both HTML-only and hybrid caches. It is seen that the relative reduction in the average query processing time in case of the hybrid cache (29.6%) is considerably higher than that of the HTML-only cache (13.2%). The hybrid cache with prefetching achieves 35.2% reduction in average query processing time compared to the HTML-only cache with prefetching. We note that, without prefetching, the gain over the HTML-only cache is up to 19.8%. Therefore, we can safely claim that the proposed hybrid cache exploits the prefetching mechanism more effectively than the HTML-only cache. This is due to the fact that the prefetching cost is mostly avoided in the hybrid cache, and hence prefetching more pages does not increase the query processing time as much as it does in the case of the HTML-only cache. When we exclude the prefetching cost (Figure 6(b)), the adaptive strategy achieves the lowest query processing time with 25.2% reduction compared to the HTML-only cache without prefetching. In case of the hybrid cache, the adaptive+ strategy results in the lowest query processing time with 32% reduction. The overall reduction in the average query processing time achieved by the hybrid cache with the adaptive+ strategy over the HTML-only cache with the adaptive strategy is 27.4%.

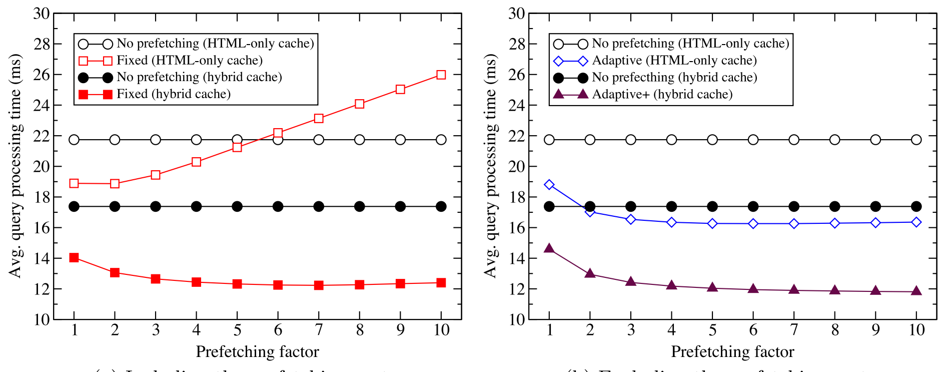


Fig. 6. A comparison of the HTML-only and hybrid caches with no prefetching and best prefetching methods for different prefetching factors.

5. SINGLETON QUERY DETECTION

5.1. Motivation

In the proposed hybrid result caching approach, so far, the docID part of the cache is used to give queries a second chance before they are completely evicted. Queries are admitted to the docID part of the cache independent of their likelihood of repeating in the near future. However, it is known that almost half of the queries in a query log are singletons, that is, they do not repeat in the future [Baeza-Yates et al. 2008]. These singleton queries pollute the cache, leading to eviction of queries that are likely to be a hit, thus decreasing the hit rate. As mentioned in Section 2.3, Baeza-Yates et al. [2007] propose cache admission policies for this problem in order to prevent tail queries from polluting the controlled part of the cache. Herein, we investigate a similar approach, where singleton queries are specially handled within the proposed hybrid caching framework.

5.2. Algorithm

Our basic idea is to detect singleton queries and store them only in the docID part of the cache. Obviously, if we could exactly predict which queries are singletons, the best would be not to cache such queries in either part of the cache. However, in a realistic search setting, it is not possible to make such predictions with a very high accuracy. Our approach avoids polluting the HTML part of the cache and provides some tolerance for misclassification.

Algorithm 5 displays the modified hybrid caching strategy. In the modified algorithm, if the query results are not cached and the query is predicted to be a singleton, then the query results are inserted into the docID part of the cache. Otherwise, they are inserted into both HTML and docID parts of the cache as in the original algorithm. If a query is predicted as a singleton and repeats again (i.e., the query was misclassified as a singleton), the query leads to a hit in the docID part of the cache. In this case, some snippet generation cost is incurred and the results of the query are inserted into the HTML part of the cache as well.

5.3. Experiments

5.3.1. Simulation Setup. To learn a predictive model, we use queries extracted from a past query log as training instances, that is, each $\langle Q, i\rangle$ request in the log forms a separate training instance. The model is then evaluated over the test queries, which are submitted immediately after the training queries. The training and test query

ALGORITHM 5: Second Chance Caching Algorithm with Singleton Query Prediction

```
Input: $q$: query, $H$: HTML cache, $D$: docID cache
$R_q \leftarrow \emptyset$ ▷ initialize the result set of $q$;
if $q \notin H$ and $q \notin D$ then
    evaluate $q$ over the backend and obtain $R_q$ ▷ $C_q = C_q^{\text{list}} + C_q^{\text{rank}} + C_q^{\text{doc}} + C_q^{\text{snip}}$;
    if $q$ is predicted as singleton then
        insert $R_q$ into $D$;
    else
        insert $R_q$ into $H$ and $D$;
...
The rest is the same as Algorithm 1
return $R_q$;
```

Table III. Features Used in Singleton Query Classification

Feature	Description
`resultPageNo`	Requested search result page number
`queryLength`	Number of terms in the query
`avgIDF`	Average inverse document frequency (IDF) of query terms
`maxIDF`	Maximum IDF of query terms
`freq`	Query frequency
`isNav`	Navigational query or not

sets are obtained from the AOL query log (100K training and 100K test queries). An instance can belong to the “singleton” or “not singleton” classes. Since about half of the queries in the training set are singletons, the imbalance between the class sizes is low. To learn a classification model, we use the Weka tool [Hall et al. 2009] (naive Bayesian and J48 decision tree classifiers). Table III provides the list of features used in the model. We first evaluate the performance of our singleton query classifier and then perform experiments with a hybrid cache using this classifier.

5.3.2. Features. The requested search result page number is an important indicator for a singleton query. Silverstein et al. [1999] report that in 95.7% of queries, the users request only the first three search result pages. The query length is also an important indicator, since long queries are generally rare queries in Web search. In addition, we exploit the inverse document frequency values of query terms, because query terms that appear rarely in the Web collection are likely to be typos, which lead to singleton queries. Moreover, queries with low frequencies are more likely to be singletons compared to queries with high frequencies. Finally, we distinguish queries based on whether they are navigational or not, using the techniques described in one of our previous studies [Ozcan et al. 2011b]. In particular, we identify navigational queries using the click data and certain terms (e.g., “www”, “com”, “edu”).

We assume that in a real-life setting, these features can be made available in the broker machine at query time. Result page number and query length features can be obtained directly from the query. Broker node can also store term IDF values. Finally, navigational queries can be identified at query time using only the query terms if no previous click information is available on the broker machine.

5.4. Results

Table IV presents the classification accuracies. According to the table, both classifiers perform better in predicting the singleton class. The decision tree classifier outperforms the naive Bayesian classifier in terms of the F-measure. Since our aim

Table IV. Singleton Query Classification Accuracies

Classifier	Class	Precision	Recall	F-measure
Naive Bayesian	not singleton	0.98	0.54	0.69
	singleton	0.78	0.99	0.87
J48 decision tree	not singleton	0.87	0.70	0.78
	singleton	0.83	0.94	0.88

Table V. Singleton Query Classification Accuracies for “Miss” Queries

Classifier	Class	Precision	Recall	F-measure
Naive Bayes	not singleton	0.37	0.40	0.38
	singleton	0.82	0.80	0.81
J48 decision tree	not singleton	0.42	0.66	0.51
	singleton	0.88	0.73	0.80

is to use this classifier only for queries that lead to a cache miss, it is important to report the accuracy of the classifier for only such queries. In a separate experiment, we evaluate the classifier using 120K miss queries and a relatively large cache. The classification accuracies are reported in Table V. Even though the F-measure drops dramatically for the not singleton class, the values are still above 0.80 for the singleton class.

Next, we evaluate the effect of our singleton query classifier on the cache hit rate. We set the cache capacity to 500K and our hybrid cache reserves 80% of its space for HTML result pages and the remaining 20% for document ids. We use the decision tree classifier, since it performed better in the previous experiments. Figure 7 shows the hit rate comparison between the HTML-only cache, hybrid cache, hybrid cache with machine-learned singleton prediction, and hybrid cache with oracle singleton prediction. According to the figure, our singleton query predictor increases the hit rate of the HTML part of the cache by 9.3% at the expense of a decrease in the hit rate of the docID part of the cache. The hybrid cache with singleton query predictor achieves a higher HTML cache hit rate relative to the original hybrid cache, since some docID cache hits are now converted into HTML cache hits. This also explains the decrease in the docID cache hit rate. An interesting observation is that the increase in the HTML cache hit rate is exactly the same as the decrease in docID cache hit rate. This is because the content of the docID part of the cache does not change, and hence some docID cache hits might turn into HTML cache hits and some HTML cache hits might turn into docID cache hits. As an upper bound, we also report the results for an oracle singleton classifier. In this case, the HTML cache hit rate increases by 16.1%.

We note that the singleton query classifier slightly reduces the average query processing time for the hybrid cache, but this improvement is not significant. This is due to the fact that the gain in this case is transforming docID cache hits into HTML cache hits for some queries (since HTML cache hits increase and docID cache hits decrease). This gain is equal to the snippet computation cost for these queries, since these queries are now served from the HTML part of the cache instead of the docID part of the cache, and there is no need for snippet generation (i.e., there is no document access cost since we assume that documents are fully cached in the memory). However, this cost reduction is not significant if we assume a snippet generation cost of 100ns/byte (P_{S} in Table II). If the search engine employs a more sophisticated snippet generation algorithm, we observe a larger reduction. For example, if we assume a 1,000ns/byte cost, then the reduction in query processing time becomes 2.4% and 6.2% for $k=1$ and $k=10$ cases, respectively. Overall, the performance of our classifier appears to be promising,

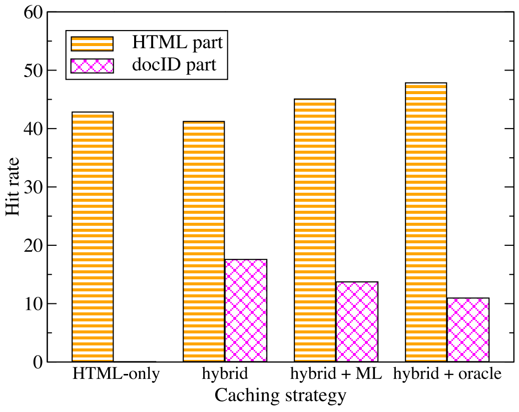


Fig. 7. Hit rates for various caching strategies.

given that the gains for an oracle classifier reach up to 4.1% and 10%, for $k = 1$ and $k = 10$ cases, respectively.

6. CONCLUSION

We introduced a hybrid result cache composed of two parts: an HTML cache, which stores the fully prepared search result pages, and a docID cache, which stores only the IDs of the best-matching k documents. For this hybrid cache, we proposed a novel caching algorithm. We investigated the performance of this cache with and without prefetching, also coupling it with a singleton query predictor. The experiments conducted on a real-life query log demonstrated that the proposed techniques considerably improve the performance of a baseline cache that stores only HTML result pages.

Our current work does not consider the issue of cache freshness. In practice, due to the dynamic behavior of the Web, query results in the cache may become stale after some time. As a future extension, we plan to couple various cache refreshing/invalidation mechanisms (such as those discussed at the end of Section 2.1) with the hybrid cache architecture proposed in our work. Another interesting direction is to investigate the performance of the singleton prediction model as the query stream characteristics change in time and develop online prediction models.

REFERENCES

Alici, S., Altingovde, I. S., Ozcan, R., Cambazoglu, B. B., and Ulusoy, O. 2011. Timestamp-based result cache invalidation for Web search engines. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 973–982.

Alici, S., Altingovde, I. S., Ozcan, R., Cambazoglu, B. B., and Ulusoy, O. 2012. Adaptive time-to-live strategies for query result caching in Web search engines. In *Proceedings of the 34th European Conference Advances in Information Retrieval*. 401–412.

Altingovde, I. S., Ozcan, R., Cambazoglu, B. B., and Ulusoy, O. 2011. Second chance: A hybrid approach for dynamic result caching in search engines. In *Proceedings of the 33rd European Conference on Advances in Information Retrieval*. 510–516.

Arroyuelo, D., González, S., Marin, M., Oyarzún, M., and Suel, T. 2012. To index or not to index: Time-space trade-offs in search engines with positional ranking functions. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 255–264.

Baeza-Yates, R. and Jonassen, S. 2012. Modeling static caching in Web search engines. In *Proceedings of the 34th European Conference on Advances in Information Retrieval*. 436–446.

Baeza-Yates, R. and Saint-Jean, F. 2003. A three level search engine index based in query log distribution. In *Proceedings of the 10th International Conference on String Processing and Information Retrieval*. 56–65.

Baeza-Yates, R., Junqueira, F., Plachouras, V., and Witschel, H. F. 2007. Admission policies for caches of search engine results. In *Proceedings of the 14th International Conference on String Processing and Information Retrieval*. 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.

Bailey, P., White, R. W., Liu, H., and Kumaran, G. 2010. Mining historic query trails to label long and rare search engine queries. *ACM Trans. Web 4,* 4, 15:1–15:27.

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*. 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*. 82–89.

Bortnikov, E., Lempel, R., and Vornovitsky, K. 2011. Caching for realtime search. In *Proceedings of the 33rd European Conference on Advances in Information Retrieval*. 104–116.

Broder, A. Z., Fontoura, M., Gabrilovich, E., Joshi, A., Josifovski, V., and Zhang, T. 2007. Robust classification of rare queries using Web knowledge. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 231–238.

Cambazoglu, B. B. and Baeza-Yates, R. 2011. Scalability challenges in Web search engines. In *Advanced Topics in Information Retrieval*, M. Melucci, R. Baeza-Yates, and W. B. Croft Eds., The Information Retrieval Series, vol. 33. Springer, Berlin Heidelberg, 27–50.

Cambazoglu, B. B., Junqueira, F., 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*. 181–190.

Ceccarelli, D., Lucchese, C., Orlando, S., Perego, R., and Silvestri, F. 2011. Caching query-biased snippets for efficient retrieval. In *Proceedings of the 14th International Conference on Extending Database Technology*. 93–104.

Elias, P. 1975. Universal codeword sets and the representation of the integers. *IEEE Trans. Inf. Theory 21*, 194–203.

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. Inform. 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*. 431–440.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. 2009. The WEKA data mining software: An update. *SIGKDD Explor. 11,* 1.

Jonassen, S., Cambazoglu, B. B., and Silvestri, F. 2012. Prefetching query results and its impact on search engines. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 631–640.

Lempel, R. and Moran, S. 2003. Predictive caching and prefetching of query results in search engines. In *Proceedings of the 12th International Conference on World Wide Web*. 19–28.

Lempel, R. and Moran, S. 2004. Optimizing result prefetching in Web search engines with segmented indices. *ACM Trans. Int. Technol. 4,* 1, 31–59.

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*. 257–266.

Marin, M., Gil-Costa, V., and Gomez-Pantoja, C. 2010. New caching techniques for Web search engines. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 215–226.

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

Ozcan, R., Altingovde, I. S., and Ulusoy, O. 2011a. Cost-aware strategies for query result caching in Web search engines. *ACM Trans. Web 5,* 2, 9:1–9:25.

Ozcan, R., Altingovde, I. S., and Ulusoy, O. 2011b. Exploiting navigational queries for result presentation and caching in Web search engines. *J. Am. Soc. Inf. Sci. Technol. 62,* 4, 714–726.

Ozcan, R., Altingovde, I. S., Cambazoglu, B. B., Junqueira, F. P., and Ulusoy, O. 2012. A five-level static cache architecture for Web search engines. *Inf. Process. Manage. 48,* 5, 828–840.

Pass, G., Chowdhury, A., and Torgeson, C. 2006. A picture of search. In *Proceedings of the 1st International Conference on Scalable Information Systems*.

Podlipnig, S. and Boszormenyi, L. 2003. A survey of Web cache replacement strategies. *ACM Comput. Surv. 35,* 4, 374–398.

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*. 51–58.

Sazoglu, F. B., Cambazoglu, B. B., Ozcan, R., Altingovde, I. S., and Ulusoy, O. 2013a. A financial cost metric for result caching. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 873–876.

Sazoglu, F. B., Cambazoglu, B. B., Ozcan, R., Altingovde, I. S., and Ulusoy, O. 2013b. Strategies for setting time-to-live values in result caches. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*. 1881–1884.

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.

Tsegay, Y., Puglisi, S. J., Turpin, A., and Zobel, J. 2009. Document compaction for efficient query biased snippet generation. In *Proceedings of the 31th European Conference on Advances in Information Retrieval*. 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*. 127–134.

Zhang, J., Long, X., and Suel, T. 2008. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web*. 387–396.

Received August 2012; revised July 2013; accepted October 2013