Online Result Cache Invalidation for Real-time Web Search

Xiao Bai
Yahoo! Research
Barcelona, Spain
xbai@yahoo-inc.com

Flavio P. Junqueira
Yahoo! Research
Barcelona, Spain
fpj@yahoo-inc.com

ABSTRACT

Caches of results are critical components of modern Web search engines, since they enable lower response time to frequent queries and reduce the load to the search engine backend. Results in long-lived cache entries may become stale, however, as search engines continuously update their index to incorporate changes to the Web. Consequently, it is important to provide mechanisms that control the degree of staleness of cached results, ideally enabling the search engine to always return fresh results.

In this paper, we present a new mechanism that identifies and invalidates query results that have become stale in the cache online. The basic idea is to evaluate at query time and against recent changes if cache hits have had their results have changed. For enhancing invalidation efficiency, the generation time of cached queries and their chronological order with respect to the latest index update are used to early prune unaffected queries. We evaluate the proposed approach using documents that change over time and query logs of the Yahoo! search engine. We show that the proposed approach ensures good query results (50% fewer stale results) and high invalidation accuracy (90% fewer unnecessary invalidations) compared to a baseline approach that makes invalidation decisions off-line. More importantly, the proposed approach induces less processing overhead, ensuring an average throughput 73% higher than that of the baseline approach.

1. INTRODUCTION

Large-scale Web search engines rely on one or more cache elements of previously computed results to serve user queries. Frequent queries issued to a search engine are directly served from its result cache, reducing both the average query response latency and the overall workload of the search backend. The cache hit ratio, as shown in the literature, reaches around 50% [2], depending on cache strategies, and even the classical cache strategy ensures a hit ratio of 30% [17]. As such, result caching has been used as an important optimization step in modern Web search engines.

An underlying assumption of using cache in search engines is that the same query, once repeated, always results in the same result. In practice, however, this is not the case. Modern Web search engines update their index periodically to incorporate changes to the Web. As a result, the search result of a query may change accordingly as the corpus of a search engine evolves.

Search engines can update their index in batch mode, incremental mode, or real-time mode, according to the freshness requirements for the search results. In batch mode, when documents are added, modified, or deleted, the search engine produces a new version of the index and rolls out this new version, superseding the previous one. The interval to produce a new version is often related to the size of the document collection, and typically varies from hours to days. Incremental mode is often used when it is desirable to bring down the time to introduce documents into the index to minutes or hours. With incremental indexing, we merge changes to the live index periodically, and each subsequent batch of documents is an increment to the previous index. When a search node finishes processing a new increment, it is used to produce a small index. This small index is then merged back into the live index. With real-time mode, a search node receives a continuous stream of individual documents and introduces them into the index in an online fashion to reduce the latency between fetching a document and making it searchable. The main difference between incremental and real-time modes according to our terminology is whether we merge the index on a per document basis.

It has been recently argued that, if the index is updated incrementally in a search engine using large result cache, the freshness of cached results becomes an issue since stale results potentially hurt user satisfaction [5, 8]. This issue might become even more severe for applications that require real-time index updates. Specifically, for applications like news search and social media search, new content (*e.g.*, news articles and tweets) are generated continuously, resulting in frequent change of query results. Moreover, in such applications, users are likely to issue same queries related to emerging events within short periods, increasing the hit ratio of the cache and thus the risk of serving stale results to users.

To highly benefit from real-time index updates, result caches are ideally managed in such a way that only the queries whose results do not change with respect to index update are served from the cache, and the queries whose

.

results have changed are invalidated from the cache and re-evaluated against the updated index.

A simple approach to invalidate cached queries is to rely on their time-to-live in the cache [8]. A result is considered stale only if its time-to-live has expired. This approach makes its invalidation decisions independently of index update and results in a large number of stale results served to users. Alternatively, another approach, called CIP (Cache Invalidation Predictor) [5], invalidates cached queries whenever there are changes to the index. CIP is document-driven. It first generates for each new (or modified) document a synopsis, composed of a list of terms it contains as well as their TF-IDF (or BM25) scores. It then computes the relevance of this document to every query in the cache. A query is invalidated if its relevance to the new (or modified) document is larger than the least relevant document in its cached result. This approach is effective to ensure the freshness of the served results but is impractical to implement since each index update requires intensive computation that is linear to the cache size. Moreover, the high fraction of unnecessary invalidations increases the workload on the search engine by processing redundant queries against its index.

Contributions. In this paper, we propose *online cache invalidation*, a practical solution to invalidate cached queries with respect to real-time index update. More concretely,

- We propose a query-driven cache invalidation framework. An invalidation decision occurs only when there is a cache hit, which is key to its efficiency since no redundant invalidations occur for queries that are not issued any more. Basically, recent changes occurred in the index are consistently maintained in a subindex to evaluate the queries that lead to cache hits. A query is invalidated if its result from the subindex is more relevant than its cached result.

- We propose a pre-judgment mechanism for early pruning unaffected queries from invalidation, which limits the negative impact of online cache invalidation on query response time. To this end, we maintain the generation time of cached queries and the update time of the posting lists in the index. Only queries that have been inserted in the cache for a long time and whose related inverted lists have changed after their insertion to the cache are evaluated against subindex.

- We evaluate the online invalidation approach in a realistic setting. Different from the evaluation of CIP [5] where a static cache was used against a synthetic workload, we apply a dynamic cache over real document and query streams submitted to the Yahoo! News search engine during several weeks. The experimental evaluation conveys the efficiency of the online invalidation approach. Despite its overhead on search latency upon cache hits and memory usage, it improves the overall throughput by 73% and reduces the redundant invalidations and the stale results by 90% and 50% respectively with well-selected parameters.

Roadmap. The remainder of the paper is organized as follows. Section 2 details the proposed online cache invalidation approach. Section 3 discusses the experimental setup and reports the results. Section 4 surveys related work and Section 5 concludes the work.

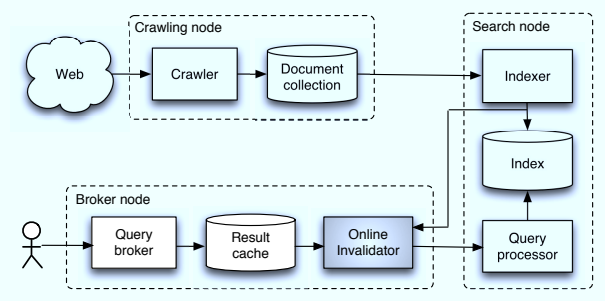


Figure 1: Search system model.

2. ONLINE CACHE INVALIDATION

2.1 System model

Fig. 1 illustrates the search systems that we assume in this work. Generally, a crawler continuously updates the document collection of the search engine by discovering and fetching new and modified documents from the Web, and deleting documents that are no longer available. A search node is in charge of indexing a subset of documents from the entire document collection, *i.e.,* document-based partition of index, and processing queries over the local index it possesses. When a user issues a query, the query broker inside a broker node first checks if the query result is available in the cache. If it is the case, the result is immediately returned to the user. Otherwise, the query broker forwards the query to the corresponding search nodes, merges the local results from those nodes to form the final result, and add the query and its final result to the cache. The use of cache enables a short average latency for query processing and reduce workload imposed on the backend query processors.

The index, however, evolves in real time as new documents are fetched and old documents are modified or deleted. Cached results consequently become stale, since they no longer reflect the most recent document collection of the search engine. More specifically, a query result is stale if its top-k relevant documents after an index update are different from those before the index update. As a consequence, to keep the freshness of results served from the cache, the search engine needs to effectively and efficiently detect the stale queries and invalidate them from the cache.

In this work, we use a component, called *online invalidator*, to perform invalidations. The online invalidator is inside the broker node. This design choice is the same as that for CIP [5], and we also assume that indexer within each search node is able to communicate with the online invalidator about the latest changes in the index. With online invalidator, an invalidation occurs upon a cache hit and cache hits considered stale at query time are forwarded to the query processors. We detail in the following the mechanisms of online invalidator to invalidate stale query results from the cache.

2.2 Cache invalidator architecture

In this work, a query result is considered stale if its top-k relevant documents differ from those in the cache in either document IDs or in their ranking order [1]. In fact, document additions, modifications or deletions may make a cached result stale:

- When a document is added, if it is more relevant than

the kth document in a query's top-k result, it should enter the result of that query.

- When a document is modified, if it contains more terms after the modification, it may become relevant to a query for which the terms no overlap with the terms of the document. If a document contains fewer terms after the modification, it may become not relevant to a query for which it was previously in the top-k results. In fact, modifications also change the frequency of terms in a document and consequently the ranking of the document even if it remains in the top-k results of some queries.
- When a document is deleted, if it is in the top-k result of a query, it needs to be replaced by other relevant document in the result of that query.

We assume a basic scoring function that solely relies on content-based features like TF-IDF or BM25, to rank the top-k result of each query. A document is relevant to a query only if it contains all the query terms. This is to keep our design and experiments comparable with those in [5]. To efficiently detect the staleness caused by the above reasons and invalidate queries to keep the freshness of results, the online invalidator maintains four data structures (Fig. 2):

- The first data structure (①) maintains, for each deleted document, its document ID d and the timestamp $T(d)$ when it is deleted, in the form of pair $\langle d, T(d)\rangle$. This is necessary to make invalidation decisions upon document deletion.
- The second data structure (②) is a subindex of documents that are added to or modified in the main search index in the search nodes. This index is used to estimate if the top-k result of a query has changed due to the addition or modification of some documents in the system and will be detailed in Section 2.4.
- The third data structure (③) maintains, for each term t in the subindex, the timestamp $T(t)$ corresponding to the latest update of its posting list in the main search index, in the form of pair $\langle t, T(t)\rangle$. This information is used to eliminate unnecessary detection of stale results over the subindex.
- The last data structure (④) maintains, for each added or modified document, its document ID d and the latest timestamp $T(d)$ when it is added or modified, in the form of pair $\langle d, T(d)\rangle$. This information is used to control the size of the subindex.

Fig. 2 illustrates the architecture of the online invalidator. The online invalidation is made upon cache hit. For each cache hit, a triple $\langle q, T(q), R(q)\rangle$ is transmitted to the online invalidator. $T(q)$ indicates when query q is added to the cache and $R(q)$ is the top-k result of q. This triple is first passed to the *pre-judgment* component that estimates the likelihood of a hit result to be fresh, relying on the age of this triple in the cache and the update time of the subindex. If the hit result is likely to be stale, the triple is passed to the *final judgment* component that evaluates the impact of index change on $R(q)$ through the presence of all the top-k documents and the appearance of new relevant documents. A query is forwarded to the backend query processors only if it is considered stale after the final judgment. The invalidation process is detailed in Section 2.3. The synopsis of all the added and modified documents are transmitted to the query broker, but it is impractical and unnecessary to keep all of them in the subindex. Therefore, only a limited number of documents are maintained in the subindex by the *subindex management* component as we will see in Section 2.4.

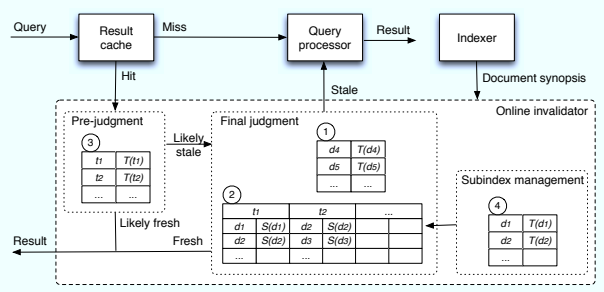


Figure 2: Online cache invalidation architecture.

2.3 Invalidation strategy

As seen in Fig. 2, the online cache invalidation is accomplished in two steps. The pre-judgment makes a preselection of queries and only passes the queries whose results in the cache are likely to be stale to the final judgment based on the following two observations:

- The top-k results of a query are likely to remain the same if the query is resubmitted within short time periods, denoted as δT.
- The top-k results of a query remains the same if not all the posting lists corresponding to the query terms have been updated since its last cache hit.

Algorithm 1 depicts the online invalidation process with its pseudo-code. In the pre-judgment, when there is a hit on query q at time $T_i(q)$, if $T_i(q) - T(q) < \delta T$, the cached set of top-k results $R(q)$, obtained at $T(q)$, is considered fresh and returned to user. This operation has $O(1)$ complexity. Otherwise, for each query term, the update time of its posting list $T(t_j)$ (maintained in ③) is compared to $T_i(q)$. If there exists a term t_j in q, $T(t_j) < T(q)$, $R(q)$ has certainly not changed. This is because a document can impact a query result only if it contains all the query terms, resulting in $T(t_j) > T(q)$ for all t_j in q. Otherwise, the final judgment is performed. This operation has $O(|q|)$ complexity, where $|q|$ is the number of terms q contains.

Note that we may return stale results to users with the first pre-judgment that compares the query time to its time in the cache. Returning stale results in this case is a consequence of the decision being independent of the index. Any change to the index during this time interval may change the set of results. The second pre-judgment, however, is accurate in the sense that only the cached results that are indeed fresh are returned to users. This argument comes from the observation that a query result does not change if not all posting lists affecting the query have changed.

In the final judgment, the invalidator first examines if the set $R(q)$ has changed due to the deletion of some documents it contains. In this step, we check if each document in $R(q)$ is in the list of deleted documents (①). If there is such a document, and it has been deleted after $R(q)$ is computed (*i.e.*, $T(d) \geq T(q)$), $R(q)$ is invalidated from the cache and q is forwarded to the backend query processors as a cache miss. Otherwise, query q is processed against the subindex using the same ranking function as the query processors in the search nodes to obtain the top-k' result in the subindex

Algorithm 1 Online cache invalidation process

Input: query q issued at $T_i(q)$ &
its cached top-k result $R(q)$ computed at $T(q)$
Output: Invalidation decision on q

Pre-judgment:
if $T_i(q) - T(q) < \delta T$ **then**
  return invalidation $\leftarrow$ false
for each t in q **do**
  if $T(t) < T(q)$ **then**
    return invalidation $\leftarrow$ false

Final judgment:
for each d in $R(q)$ **do**
  if d is deleted & $T(d) \geq T(q)$ **then**
    return invalidation $\leftarrow$ true
process q against subindex and obtain top-k' result $R'(q)$
for each d' in $R'(q)$ **do**
  if $d' \notin R(q)$ & $Score(q, d') > Score(q, d_k)$ **then**
    return invalidation $\leftarrow$ true
return invalidation $\leftarrow$ false

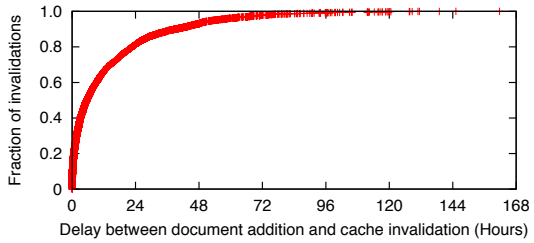
$R'(q)$. If there is a document d' in $R'(q)$ that is not in $R(q)$ and its relevance score to q ($Score(q, d')$) is larger than that of the kth document in $R(q)$ ($Score(q, d_k)$), d' (or another document that is more relevant to q than d') should be included in the top-k results of q instead of the original kth document d_k. $R(q)$ is thus considered stale and invalidated from the cache. Then query q is forwarded to the query processors.

The invalidation due to document deletion in the final judgment is accurate and can be achieved in $O(k)$.The query processing using the subindex in the worst case is linear on the total length of the query related posting lists, *i.e.*, $O(|q| \times L)$, where L is the maximum length of the posting lists. The comparison of $R'(q)$ and $R(q)$ requires at most k^2 operations. The query processing on the subindex thus dominates the time required for making invalidation. However, the invalidation time can be controlled by limiting the number of documents maintained in the subindex. Moreover, as we show in Section 3.3, the lightweight pre-judgment significantly improves search performance by reducing the number of cache hits that require final judgment and thus the query response latency.

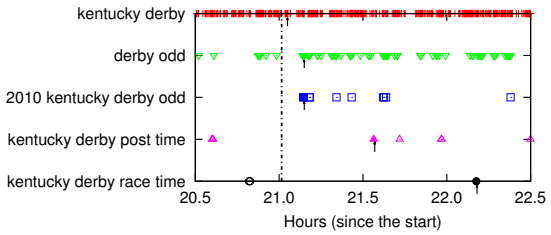
2.4 Subindex management

We describe in this section how we build and maintain the subindex. This is a process running in the backend, independent of query processing. The subindex is built in the same way as the main search index and is also updated in real time when the online invalidator receives a document synopsis from the indexer (Fig. 2). The synopsis of a document is generated as in the CIP work [5] and it consists of pairs of the form $\langle t, S(d) \rangle$, where t is a term in the document d and $S(d)$ is the relevance score of d for t. Since we focus on basic ranking functions (*e.g.* TF-IDF and BM25) to compute the query results, $S(d)$ is also a TF-IDF or BM25 score.

In the subindex, for each term, an inverted posting list is maintained. Each posting in the list of term t corresponds to a document that contains t and its relevance score to t (TF-IDF or BM25). The postings are ranked in descending order of relevance scores. An arriving synopsis can be easily inserted to the relevant posting lists and ranked accordingly using the same mechanisms of the indexer. If a received synopsis corresponds to a document deletion, the document is



(a) Distribution of cache invalidation delay.



(b) Example of cache invalidation delay.

Figure 3: Rationale behind limiting subindex size.

inserted to the list of deleted documents with the timestamp when it is deleted (①). The update time of the corresponding posting lists are updated with this timestamp (③). If a received synopsis corresponds to a document addition or modification, the timestamp when the operation occurs is updated for each term appearing in the document (③).

The subindex cannot grow without bounds. Documents that are less likely to have impact on the results of future queries should be evicted from the subindex to keep its size moderate. One natural choice here is to fix the size of the subindex by keeping only the documents that are recently added or modified. Intuitively, if a document has been added or modified a long time ago, it is likely that the queries that include this document in their top-k results have been processed against the index, and thus have fresh cached results. Fig. 3 illustrates the intuition underlying our design choice. Fig. 3(a) depicts the distribution of necessary time for a query to be invalidated from the cache after a relevant document is added to the search index. (The setup of this experiment is detailed in Section 3.1.). We observe that more than 80% of queries can be timely invalidated if each document is maintained in the subindex for 24 hours. Therefore, we only keep in the subindex the S document that are newly added or modified.

To make the eviction of documents from the subindex efficient, when a document is added to the index, it is added to the list of added and modified documents with the corresponding timestamp (④). When a document is modified, its timestamp in the list is updated. When a document is deleted, it is removed from the list. This list is kept sorted in ascending order of timestamps. Once the size of the subindex becomes larger than the pre-defined size S after an insertion, the document in the head of the list is removed from subindex. This operation does not affect the update time of posting lists (③) and the latter are only updated when the main search index changes.

Limiting the size of subindex may lead to stale results. Consider the example in Fig. 3(b), where a document is added to the search index in the 21st hour of the experiment (vertical line) and each point in a horizontal line represents the occurrence time of a query. The point pointed to by

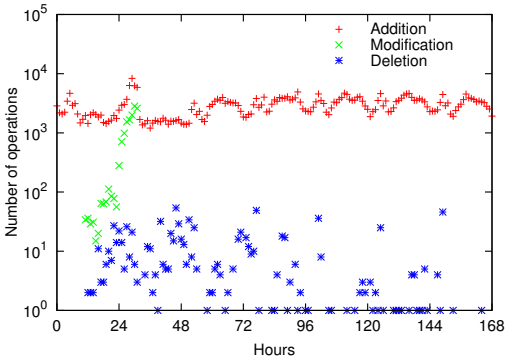


Figure 4: Document dynamics.

arrow on a horizontal line (*i.e.*, the first point on the right of the vertical line) shows the time when a query is invalidated due to the addition of this document. We observe that more frequent queries require less time to incorporate a new document into its result set. As a result, if a query is rarely issued, when a document that impacts its result is removed before a new query occurrence, the cache invalidator is unable to realize the change to the main search index. A solution around this problem is to use a TTL-based invalidation [8] prior to our online invalidation. We discuss further the impact of the size of the subindex in Section 3.2.

3. EVALUATION

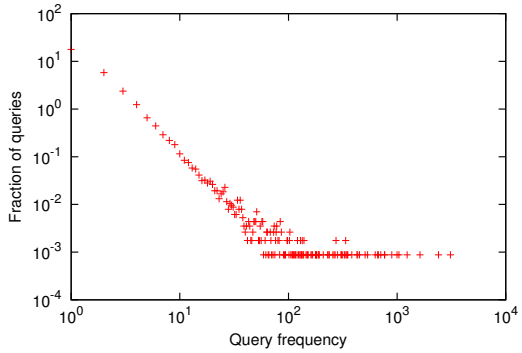
We report in this section the performance of the online cache invalidation with respect to accuracy, efficiency, and its overall impact on the search engine. Section 3.1 details the setup of our experiments. Section 3.2 and Section 3.3 qualitatively and quantitatively evaluate the online cache invalidator through comparison with the state-of-the-art CIP approach [5]. In the experiments, we implement CIP with its best-case setting, *i.e.,* entire synopsis ($\eta = 1$) and score thresholding ($1_s = true$).

3.1 Experimental setup

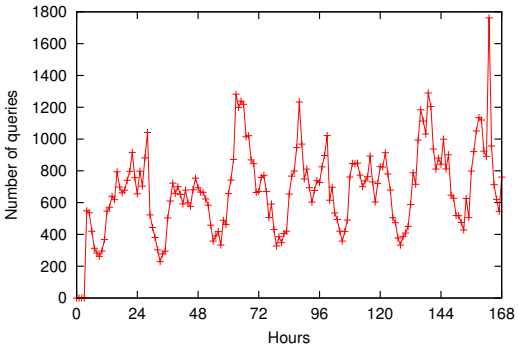
Dataset. In the experiments, we use a sample of the document history and query logs from the Yahoo! News search engine obtained in May 2010. We used the documents sampled from the first 3 weeks of this period to build the search engine index. This collection contains roughly 1.4M unique documents. The documents and queries sampled from the last week of this period are used to emulate the update of index and the arrival of queries.

In the document set, there are 488,441 (97.1%) additions, 13,562 (2.7%) modifications and 881 (0.2%) deletions. The amount of additions dominates the changes to the collection. Fig. 4 depicts the arrival rate of document changes during the experiment. We observe that the arrival rate of document additions remains relatively stable and occasionally presents spikes due to emerging events. Document modification mainly occurs in the first two days, following a weekly recurring pattern that we do not show in this figure. This is due to the periodically re-crawl of existing pages in the index. Document deletion rarely happens and they are spread over time.

The queries we use in the experiments is a sample of queries searching for news articles extracted from the query logs of the Yahoo! search engine. The resulted query set contains 113,943 queries, out of which 34,121 are unique and their frequency follows a power-law distribution (Fig. 5(a)). Fig. 5(b) illustrates the arrival rate of these queries in the search engine. We observe that the arrival of queries follows a periodic pattern and the peak area corresponds to the daytime of each evaluation day.



(a) Query frequency distribution.



(b) Query dynamics.

Figure 5: Query characteristics.

Setup. We build the search system on top of a proprietary platform for vertical search developed in Yahoo!. Two servers with Quad-core 2.4GHz CPU, 48G RAM and four 1.8T disks are used and they are inter-connected through a local area network of 1Gbp/s speed. One server acts as the search node: it hosts the main search index and processes the queries. The index is created and updated in real time so that any change to the document collection is reflected into the index in an online fashion. The other server acts as the query broker node: it is in charge of managing the cache, making invalidation decisions, and forwarding queries to the search node in case of cache misses. The subindex used by the online invalidator is also maintained in the query broker node.

3.2 Qualitative performance

Methodology and metrics. We examine the cache invalidation strategies over a dynamic cache of *unlimited size* in a much finer grained manner than that used in the evaluation of CIP [5]. Neither admission nor eviction is necessary and a query is removed from the cache only upon invalidation. Instead of generating a new index periodically and evaluating the same queries repeatedly against different versions of the index, we preserve the full arrival history of documents and queries, and replay them in parallel according to their original timestamps. Note that we only preserve the relative

Table 1: Impact of results retrieved from subindex.

	CIP	Online		
		top-1	top-5	top-10
Stale ratio	0.0055	0.0089	0.0025	0.0003
False positive ratio	0.0356	0.0048	0.0048	0.0048

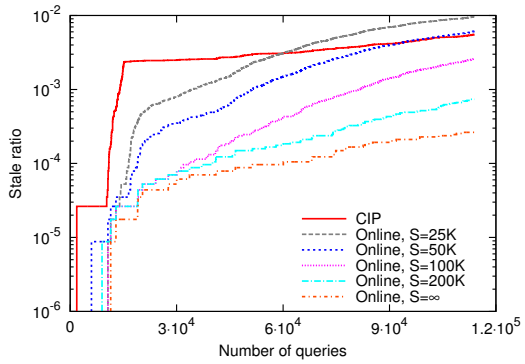
order of arrival. The documents and queries are processed at the maximum speed the system admits.

The index is updated in real time once a document is added, modified, or deleted. Once a search engine returns results for a query, we process the same query against the main search index to obtain the ground truth (its ideal set of results).

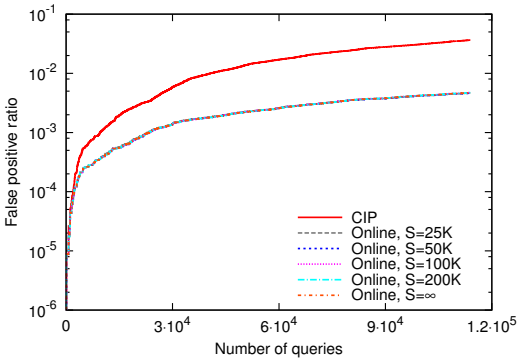
With our online cache invalidation strategy, we execute an invalidation procedure upon every hit. This decision is accurate if a stale query result is correctly invalidated or a fresh query result is returned to the user without forwarding it to the backend query processors (*i.e.*, the search node). A decision may be false positive, indicating that a query is invalidated while its result in the cache is still fresh. A decision may also be false negative, indicating that a query result is considered fresh but it has changed due to updates to the index. A false negative leads to a stale query and hurts result quality. A false positive has no impact on result quality, but it requires forwarding the query to the backend and processing it against the search index, which induces a higher load against the backend nodes. Consequently, we target lower rates of both false negatives and false positives with the invalidation process. After we process each query q, we evaluate whether its top-10 results are stale or the invalidation is a false positive by comparing this result set against the ideal top-10 obtained from the main search index. The *stale ratio* and *false positive ratio* are measured after each query as the cumulative fraction of stale queries and the cumulative redundantly processed queries over all the queries that have been answered.

Impact of subindex size. We start by evaluating the results obtained with different sizes of the subindex in the online invalidator to quantify its impact. In this experiment, we do not use pre-judgment in the online cache invalidation process to assess the stale ratio due to the final judgment step alone.

In the final judgment step, we first use different values of k' to obtain top-k' results from the subindex to make invalidation decisions. Table 1 compares the quality of the online invalidation through the stale ratio and the false positive ratio obtained at the end of the experiment. We observe that as we increase the number of results retrieved from the subindex, the number of stale results served to users drops, which is a consequence of the following. For a given k' ($\leq k$), when all the top-k' results from the subindex are present in the cached top-k results, then the cached top-k result set is considered fresh. The cached result set is not fresh in the case, for example, of a new document ranked in the $k'+1$ position of the subindex being more relevant than the kth result in the cache. A larger value of k' implies that fewer relevant documents are omitted by the online invalidator. Increasing the number of retrieved results from the subindex, however, has negligible impact on the false positive ratio. In the following experiments, we retrieve the top-10 results from subindex in the final judgment step to guarantee the best result quality.



(a) Stale ratio.



(b) False positive ratio.

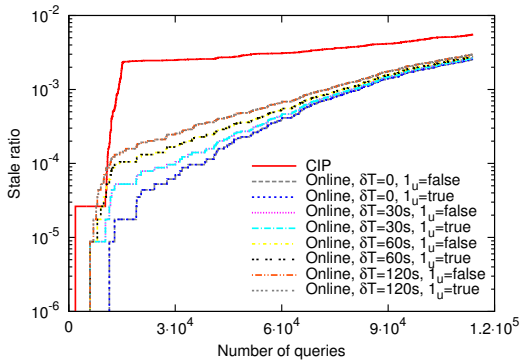
Figure 6: Impact of subindex size on invalidation.

Fig. 6 illustrates the quality of the online invalidation using stale ratio and false positive ratio. The values for each metric are displayed with respect to the number of queries that have been returned. The size of subindex is denoted by S; $S = \infty$ corresponds to the ideal case where all the updates on the search index are maintained in the subindex.

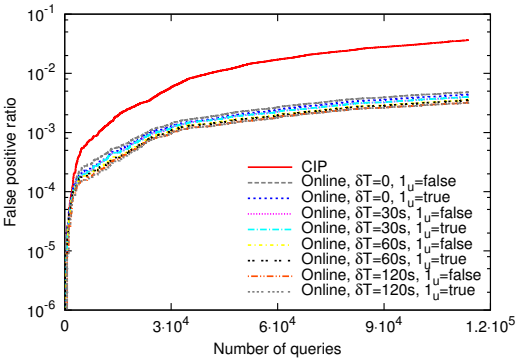
Generally, fewer results are stale if the online invalidation is used (Fig. 6(a)). A small size of the subindex leads to a high stale ratio as we have discussed in Section 2.4. However, even if only 25K documents are maintained in the subindex, accounting for 5% of all the index updates, the stale ratio is still lower than CIP at the early stage of the experiment and is very similar toward the end. Note that the stale ratio is not zero for $S = \infty$, since invalidations are based upon the existence of more relevant documents in the subindex, ignoring order. Further comparison between the top-k in the cache and that in the subindex can ensure a stale ratio of zero, but imposes additional computation and the size of the subindex cannot be unlimited. Note that compared to CIP, our online cache invalidation strategy significantly decreases the false positive ratio (Fig. 6(b)). Only 0.5% of queries are processed redundantly with the online cache invalidation, where it increases to 3.6% with CIP. As we see in Section 3.3, this is key to efficiency.

As we have seen, the size of subindex has almost no impact on the false positive ratio, and the subindex of 100K documents guarantees a fairly good stale ratio. Consequently, we focus on a subindex of 100K documents for the following experiments.

Impact of pre-judgment. The pre-judgment of the online cache invalidation is used to reduce the number of cache hits that require query processing over the subindex. We



(a) Stale ratio.



(b) False positive ratio.

Figure 7: Impact of pre-judgment on invalidation.

first evaluate its impact on search result quality and invalidation accuracy in this section and then its impact on search efficiency in Section 3.3.

As we have explained in Section 2.3, the pre-judgment comprises two mechanisms. The first mechanism makes the judgment according to the recency of a query. A cached result is returned to the user if it is issued within the time interval δT. The second mechanism makes the judgment according to the update time of the posting lists corresponding to query terms, for a given query. Fig. 7 compares the performance of the online cache invalidation with each of the mechanisms as well as their combinations. We use 1_u to indicate if the second mechanism of pre-judgment is applied. $\delta T = 0$ means the first mechanism is not applied. Otherwise, the value of δT is measured in seconds.

We observe from Fig. 7(a) that using the first mechanism increases the stale ratio. A larger value of δT gives us a higher stale ratio. The differences are only significant at the early stage of the experiments, though, and become negligible as more queries are processed. We also observe from Fig. 7(a) that the second mechanism has no impact on the stale ratio. This confirms our analysis in Section 2.3 that it ensures only skipping results that indeed have not changed from final judgment.

Fig. 7(b) depicts the evolutions of false positive ratio with different pre-judgment mechanisms. Using larger δT and setting $1_u = true$ reduce the false positive ratio as more queries are eliminated from final judgment. This ensures the false positive ratio is always lower than with CIP.

In the above experiments, we assume that once a query result is added to the cache, it is only removed upon an invalidation. Considering that the CIP approach can be

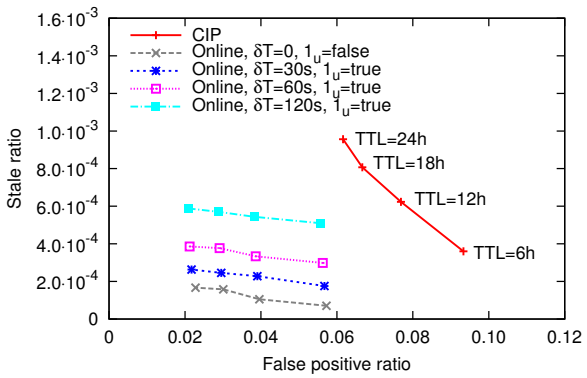


Figure 8: Impact of pre-judgment wrt. different TTL.

augmented with an age-based strategy, we also compare the performance of the online cache invalidation in this context. More specifically, each query in the cache is associated with an expiration period (TTL) and its result is considered stale at the end of this period. The invalidation decision of the online invalidator is only made upon cache hits with respect to TTL.

Fig. 8 illustrates the impact of pre-judgment through the relationship between stale ratio and false positive ratio with respect to various TTL values. The ratios are obtained at the end of each experiment. For the sake of presentation, only the cases using both mechanisms for pre-judgment are shown as the differences on stale ratio and false positive ratio between using and not using the second mechanism are negligible (Fig. 7). In Fig. 8, the points on the same curve from left to right correspond to decreasing TTL values. In all cases, a small value of TTL results in low stale ratio and high false positive ratio, since a large number of queries are invalidated independent of the changes to the index. For instance, when TTL is set to 24 hours, compared to the case where no TTL is used, the stale ratio decreases by 83% for CIP and 86% for online invalidation with $\delta T = 60$ seconds and $1_u = true$, but the false positive ratio increases by 67% for CIP and 500% for online invalidation with the same setting. Compared to CIP, our online invalidation strategy consistently outperforms it in both false positive ratio and stale ratio, with TTL values not smaller than 12 hours. The trade-off between search result quality and search efficiency can be controlled through carefully tuning the value of δT with the online invalidation.

3.3 Quantitative performance

We have seen that the online cache invalidation outperforms the CIP approach in both search result quality (wrt. stale ratio) and invalidation accuracy (wrt. false positive ratio). Since a primary goal of this work is to derive a practical scheme for cache invalidation, we further compare the efficiency of the online invalidation against CIP and discuss their memory consumption in this section.

Methodology and metrics. A key difference between the online invalidation and CIP is the underlying procedure to make invalidations. In the online invalidation approach, we perform invalidation lazily, only upon a cache hit. This invalidation scheme increases the response time in case of a cache hit. It enables fresher results by decreasing the stale ratio as we have seen in Section 3.2. When the index is updated, referring to a document addition, modification or

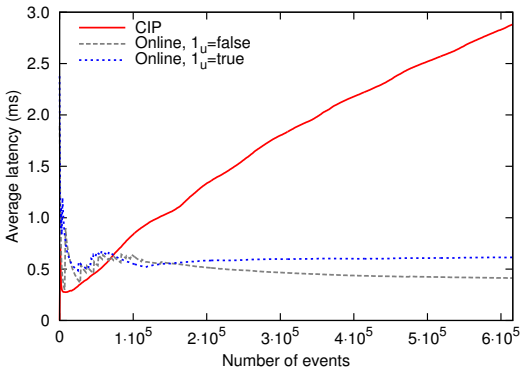


Figure 9: Efficiency of document processing.

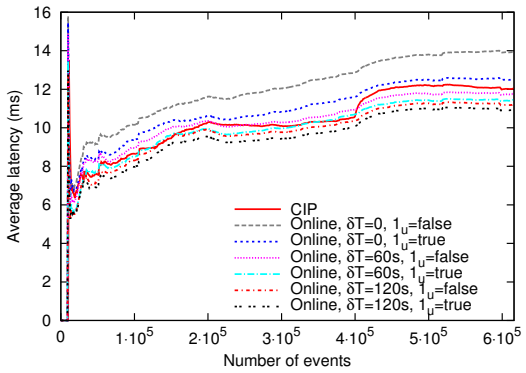


Figure 10: Efficiency of query processing.

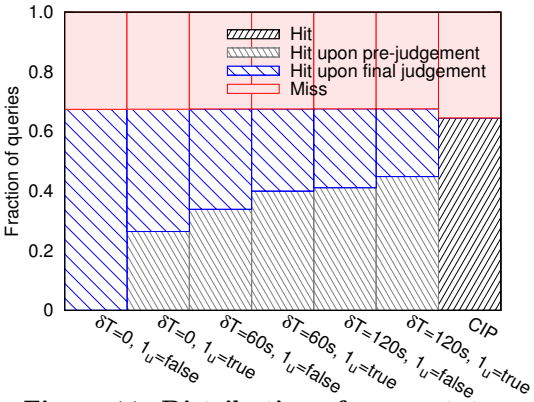


Figure 11: Distribution of query status.

deletion, only the four data structures described in Section 2.2 need to be updated accordingly. In the CIP approach, no additional time is needed upon a cache hit. Once the index is updated, however, it performs invalidations eagerly. This process requires extensive computation on query and document pairs to identify all the cached queries whose results may have changed. In our experiment, we implement the CIP approach with an inverted index of the cached queries to make this process efficient [6]. We measure the efficiency of these approaches through the average latency to process a document synopsis, the average latency to respond to a query and the average throughput of the search engine.

Document processing efficiency. Fig. 9 shows the average latency for processing an index update (*i.e.*, a document synopsis) as a function of the number of incoming events, including both queries from users and document synopses from indexers. The reason for considering both kinds of events is that the latency of CIP depends on the number of queries in the cache while the latency of online invalidation depends on the number of documents in the subindex.

We observe that the latency of invalidations with CIP increases linearly with the number of arriving events. With CIP, each index update requires scanning all the queries in the cache that contain at least one term in the document, and consequently latency increases with the cache size. On average, 2.8 milliseconds are required after 600K events are issued to the query broker. In contrast, with the same number of events, only 0.6 milliseconds are enough to make the necessary processing according to an index update with online invalidation. The latency for online invalidation converges and remains stable after 200K events. Once the size of subindex reaches its limit (100K in this experiment), the latency for updating the subindex with a document does not vary much. This figure also shows the latency when the second mechanism of pre-judgment is not used. As the invalidator does not need to maintain the update time of each posting list in the subindex (*i.e.*, data structure ③ in Section 2.2), the latency is about 30% less than if it is used.

Search efficiency. Fig. 10 compares the average latency for answering a query with online invalidation against that of CIP. Since this latency depends on the size of the subindex with online invalidation, we also show its value as a function of the number of incoming events. We observe that if no pre-judgment is used ($\delta T = 0$ and $1_u = false$), the latency with online invalidation is up to 20% more than that with CIP. Pre-judgment, however, enables a significantly drop of the query processing time by reducing the number of hits that need final judgment. For instance, with online invalidation, $\delta T = 60$ seconds and $1_u = true$, the average latency for processing a query is about 5% less compared to CIP. Increasing the value of δT further reduces latency.

Fig. 11 depicts the fraction of queries that are served from the cache after pre-judgment (cache hit upon pre-judgment), served from the cache after final judgment (cache hit upon final judgment) and processed against the main search index (cache miss). We observe that online invalidation produces fewer cache misses than CIP, since there are fewer false positives (Fig. 7(b)). Moreover, the use of pre-judgment significantly reduces the number of queries that require final judgment. In online invalidation with $\delta T = 60$ seconds and $1_u = true$, only 41% of cache hits need final judgment to make the invalidation decision. These observations explain the efficiency of online invalidation for query processing.

Throughput. As we have explained in Section 2.1 (Fig. 1), the cache invalidator resides in the query broker node. Therefore, the query broker node is in charge of both processing queries and making invalidation decisions. We finally measure the overall performance of the online invalidation approach through the average throughput of the query broker node. In other words, we are interested in the number of events, including incoming queries and document synopsis that the query broker node can handle per second.

Fig. 12 reveals that the online invalidation consistently outperforms CIP by coping with more events per second. Moreover, the difference in throughput increases as more events arrive in the system. This is due to the computationally intensive invalidation steps of CIP (Fig. 9). The query

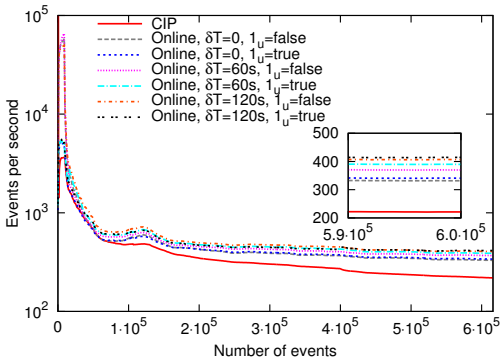


Figure 12: Throughput of query broker.

broker node running the online invalidation with $\delta T = 60$ seconds and $1_u = true$ can processes more than 73% events than that running CIP after 600K events arrive.

Memory footprint. The online cache invalidator maintains four data structures to make invalidation decisions (Section 2.2). In our experiments, the subindex of 100K documents maintains 214K inverted lists with 19.4M postings. As a document ID is 16 bytes and a relevance score is 4 bytes, the subindex requires 388M memory. Maintaining the update time of each posting list (*i.e.*, data structure ③ in Section 2.2) requires 856K memory as a timestamp is 4 bytes. Maintaining the timestamp for each document in the subindex (④) requires 2M memory. The memory consumption for deleted documents (①) is negligible compared to other data structures due to the limited number of deleted documents (Fig. 4). With CIP, the index of cached queries consumes 2.2M with 16K per term list of queries where the average length of a query is 20 bytes. Online cache invalidation requires much more memory than CIP to maintain its subindex. Yet, this accounts for less than 1% of 48G RAM of a modern server. To convey the scalability of the online invalidation, we conduct a separate experiment that replays the same documents and one time more queries as in the previous experiments, resulting in a cache with twice the original size. With a subindex of 100K documents, the stale ratio and false positive ratio remain stable, *i.e.*, 0.0024 and 0.0037 respectively with respect to 0.0025 and 0.0047 in the original setting (Fig. 6). This experiment confirms that the online invalidation is practical to implement in a search engine as increasing the size of cache does not require larger subindex to fit in memory.

Our experimental evaluation shows that with carefully selected parameters, *e.g.*, $\delta T = 60$ seconds and $1_u = true$, the proposed online cache invalidation ensures higher result quality (wrt. 50% lower stale ratio) and makes more accurate invalidation (wrt. 90% lower false positive ratio) than the state-of-the-art CIP approach. The online invalidation is also practical to implement in a Web search engine, given its moderate memory use and efficiency, which guarantees high throughput.

4. RELATED WORK

Caching has been widely used in Web search engines to shorten the average query response time and reduce the workload on search servers. Generally, both search results [4, 11, 12, 17] and posting lists [16, 19] can be cached with different trade-offs [3]. A seminal work on result caching in search engines has been the one of Markatos [17] comparing several basic cache replacement policies. Followup work has addressed issues including eviction [11, 17, 18], admission [4] and pre-fetching [12, 13, 15], to optimize either hit ratio or computational cost.

This body of work is based on a common assumption that the cache has limited capacity and thus queries in the cache have to be well selected to ensure good performance. However, modern search engines might store their cache entries on disk, enabling a very large cache [8]. In such cases, maintaining the freshness of cached results becomes very important.

A search engine index is not static and is continuously updated according to document additions, modifications, and deletions on the Web that are gradually captured by crawlers. Such changes may lead to inconsistencies between results in the cache and results computed on an updated index. Basically, there are three ways to update a search index [14]: in-place update, index merging, and complete re-build. The complete re-build approach periodically builds a new index with all the available documents and replaces the live index once the new index is ready. The index merging approach (incremental indexing) [10, 5], generates delta indexes with the changes regarding to the previous index and merges them to the live index within short periods. The in-place update approach (real-time indexing) directly performs changes to the live index in an online manner to make the new documents searchable shortly after being crawled. We focus in our work on real-time indexing as it ensures best freshness of the search index that is important for applications like news and social media search.

One way to keep the freshness of query results is to anticipate the future occurrence of cached queries and refresh them before they are issued to the search engine again. In [9], several cache refreshment policies are proposed based on either recency or frequency of cached queries. This work confirms the findings in [7] that frequency-based policies significantly improve the fraction of fresh hits compared to recency-based policies. A more recent work that follows this research line was proposed in [8]. In this work, the frequency of access is combined with the age of an entry in the cache to selectively refresh the cached results, and it is shown to achieve higher hit rate compared to recency-based refresh.

Another way to keep the freshness of query results is to simply invalidate results that have changed from the cache without refreshing. This is mainly motivated by the fact that issuing queries to the back end query processors without explicit user requests may incur unnecessary computational cost. An existing approach to invalidate cached results in search engines is CIP [5]. CIP invalidates queries upon updates to the index. Once a document d is added to or modified in the index, its relevance to every query in the cache is computed and queries for which top-k results are less relevant than d are invalidated. This approach is computationally expensive since the cost for computing the relevance is linear with the size of the cache. To make the invalidation more efficient, in the work of Bortnikov *et al.* [6], cached queries are organized as an index of query terms, and a document update is processed as a query against this index to identify the queries to invalidate. Another approach that reduces the computational cost of CIP was proposed by Alici *et al.* [1]. The basic idea of this work is to maintain and compare the generation time of query results against

the update time of the search index to decide if query results need to be invalidated. Despite being efficient, this approach degrades the freshness of the served results due to the approximation made at invalidation time, and requires up to 20 times more backend communication compared to CIP. In our work, we make cache invalidation at query time as in the work of Alici *et al.*. Our online cache invalidation strategy achieves better invalidation accuracy compared to CIP without inducing a high communication overhead to the backend as the approach of Alici *et al.*. In fact, the same amount of backend communication as CIP is enough for making our online cache invalidation.

5. CONCLUSION

Real-time indexing in Web search engines makes it challenging to keep the freshness of query results served from their cache. In this work, we propose a practical approach that relies on a subindex of recent changes to the search index to invalidate the stale cached queries. A pre-judgment based on generation time of cached queries and update time of search index is applied before the evaluation of queries against the subindex to improve the efficiency of invalidation. Through a realistic evaluation on top of a search engine implementation, we show that our approach ensures higher result quality and invalidation accuracy compared to CIP, a previous approach to invalidation, as it results in lower stale ratio and lower false positive ratio. We show that our approach is efficient, which is an important step towards a practical implementation in commercial search engines.

Our approach in this paper relied upon term-based ranking functions that require periodic updates of global statistics to ensure the accuracy of invalidation. We leave for future work the investigation of cache invalidation approaches that use advanced ranking functions leveraging link-based features like PageRank.

6. REFERENCES

[1] S. Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and O. Ulusoy. Timestamp-based result cache invalidation for web search engines. In *Proceedings of the 34th international ACM SIGIR conference on research and development in Information Retrieval*, pages 973–982, 2011.

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

[3] R. Baeza-Yates, A. Gionis, F. P. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. Design trade-offs for search engine caching. *ACM Trans. Web*, 2:1–28, October 2008.

[4] R. Baeza-Yates, F. Junqueira, V. Plachouras, and H. F. Witschel. Admission policies for caches of search engine results. In *Proceedings of the 14th international conference on string processing and information retrieval*, pages 74–85, 2007.

[5] R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza. Caching search engine results over incremental indices. In *Proceedings of the 19th international conference on World Wide Web*, pages 1065–1066, 2010.

[6] E. Bortnikov, R. Lempel, and K. Vornovitsky. Caching for realtime search. In *Proceedings of the 33rd European conference on advances in information retrieval*, pages 104–116, 2011.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenke. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 126–134, 1999.

[8] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge. A refreshing perspective of search engine caching. In *Proceedings of the 19th international conference on World Wide Web*, pages 181–190, 2010.

[9] E. Cohen and H. Kaplan. Refreshment policies for web content caches. *Comput. Netw.*, 38:795–808, April 2002.

[10] D. Cutting and J. Pedersen. Optimization for dynamic inverted index maintenance. In *Proceedings of the 13th annual international ACM SIGIR conference on research and development in information retrieval*, pages 405–411, 1990.

[11] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *Proceedings of the 18th international conference on World Wide Web*, pages 431–440, 2009.

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

[13] R. Lempel and S. Moran. Optimizing result prefetching in web search engines with segmented indices. *ACM Trans. Internet Technol.*, 4:31–59, February 2004.

[14] N. Lester, J. Zobel, and H. E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *Proceedings of the 27th Australasian conference on computer science*, volume 26, pages 15–23, 2004.

[15] H. Li, W.-C. Lee, A. Sivasubramaniam, and C. L. Giles. A hybrid cache and prefetch mechanism for scientific literature search engines. In *Proceedings of the 7th international conference on Web engineering*, pages 121–136, 2007.

[16] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th international conference on World Wide Web*, pages 257–266, 2005.

[17] E. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[18] P. C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. 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*, pages 51–58, 2001.

[19] Y. Tsegay, A. Turpin, and J. Zobel. Dynamic index pruning for effective caching. In *Proceedings of the sixteenth ACM conference on information and knowledge management*, pages 987–990, 2007.