A Refreshing Perspective of Search Engine Caching

B. Barla Cambazoglu
Yahoo! Research
Barcelona, Spain
barla@yahoo-inc.com

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

Vassilis Plachouras
Dept. of Informatics, AUEB
Athens - Greece
vplachouras@acm.org

Scott Banachowski
Yahoo! Search
Sunnyvale, CA, USA
sbanacho@yahoo-inc.com

Baoqiu Cui
Yahoo! Search
Sunnyvale, CA, USA
bcui@yahoo-inc.com

Swee Lim
Yahoo! Search
Sunnyvale, CA, USA
sblim@yahoo-inc.com

Bill Bridge
Oracle Corp.
Redwood Shores, CA, USA
bill.bridge@oracle.com

ABSTRACT

Commercial Web search engines have to process user queries over huge Web indexes under tight latency constraints. In practice, to achieve low latency, large result caches are employed and a portion of the query traffic is served using previously computed results. Moreover, search engines need to update their indexes frequently to incorporate changes to the Web. After every index update, however, the content of cache entries may become stale, thus decreasing the freshness of served results. In this work, we first argue that the real problem in today's caching for large-scale search engines is not eviction policies, but the ability to cope with changes to the index, *i.e.*, cache freshness. We then introduce a novel algorithm that uses a time-to-live value to set cache entries to expire and selectively refreshes cached results by issuing refresh queries to back-end search clusters. The algorithm prioritizes the entries to refresh according to a heuristic that combines the frequency of access with the age of an entry in the cache. In addition, for setting the rate at which refresh queries are issued, we present a mechanism that takes into account idle cycles of back-end servers. Evaluation using a real workload shows that our algorithm can achieve hit rate improvements as well as reduction in average hit ages. An implementation of this algorithm is currently in production use at Yahoo!.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; H.3.3 [**Information Search and Retrieval**]: [Search process]; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Performance, Design

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

Keywords

Web search engines, caching, index updates, refresh

1. INTRODUCTION

Search engines are essential services to find the content on the Web. Commercial search engines like Yahoo! have over a hundred billion documents indexed, which map to petabytes of data. Searching through such an enormous amount of data is not trivial, especially when serving a large number of queries concurrently. Thus, search engines rely upon systems comprising large numbers of machines grouped in clusters by functionality, such as index servers, document servers, and caches [6].

Cache components appear in different parts of an engine and in different forms, *e.g.*, result, posting list, and document caches. Herein, we focus on result caches, which store previously computed query results. These caches may be deployed in separate machines, acting as a proxy, or co-exist in the same machine with query processors.[1] At a high level, a search engine receives a query from a user, processes the query over its indexed documents, and returns a small set of relevant results to the user. If a previously computed set of results is cached, the query can be served directly from the cache, eliminating the need to process the query.

Given the high volume of user queries, result caches emerge as crucial performance components to reduce the query traffic to back-end servers and also to reduce the average query processing latency. A well-known observation from the information retrieval literature is that query frequencies follow a power-law distribution [31]. This implies that a few queries have very high frequencies and many appear very infrequently, often just once (singleton queries). As a number of queries repeat often enough, result cache implementations in practice can achieve high hit rates.

Compared to the traditional problem of caching in operating systems, the problem of result caching in search engines is not memory space. For search engines, it is possible to cache millions of result entries on disk and yet improve query response latencies. On average, it takes tens of milliseconds

[1]Result caches are often deployed on caching proxies and brokers of search clusters.

to process a query on a search cluster, and fetching from disk presents a comparable latency, often lower. Furthermore, using disks to store previously computed results provides an opportunity to eliminate the capacity misses encountered in small result caches that use only RAM.

One major drawback of large result caches is freshness. Search engine indexes change frequently due to new batches of crawled documents. Consequently, it is likely that a significant fraction of previously computed results in the cache become stale over time, *i.e.*, some of the top-matching results in the current index are not present in cached entries, thus potentially degrading the quality of results. In fact, we argue that the freshness problem becomes more severe as the cache capacity increases.

A solution to the freshness problem is to *invalidate* entries over time. There are two possible approaches for cache invalidation in this context: *coupled* and *decoupled*. The coupled approach is the one of providing the cache with information about changes to the index. This approach is difficult to realize in practice due to the complexity and computational cost of accurately determining changes to the index and propagating them to the result cache. Such coupled solutions are out of the scope of this work. The decoupled approach, which we adopt in this work, is the one of invalidating cached entries without any concrete knowledge of changes to the index. A simple way to achieve this goal is to use a time-to-live (TTL) value and mark entries as expired once they have been in the cache for longer than TTL. Once entries are invalid, they are eventually either evicted or replaced with new results. Given that the engine is often not processing queries at full capacity, we can leverage idle cycles to re-process queries and *refresh* cache entries. If we use the TTL approach and TTL is long enough, we may be able to populate the cache with fresh entries that will be hits during busy times.

Contributions. In this paper, we present the design of the result cache used in the Yahoo! search engine. This cache introduces the concept of refreshing cache entries and presents a practical algorithm for prioritizing entries to refresh. To the best of our knowledge, we are the first to consider the problem of refreshing result entries in search engine caches. More concretely, our contributions are:

- We propose a mechanism for expiring cache entries based on a time-to-live value and a mechanism for maintaining the cache content fresh by issuing refresh queries to back-end search clusters, depending on availability of idle cycles in those clusters.
- We propose a novel algorithm for prioritizing cache entries to be refreshed based on the access frequency of entries and the age of the cached entry.
- We evaluate the performance of our techniques via simulations over a nine-day query load, containing 130 million queries.
- We provide some statistics from production systems, reporting on our observed benefits in practice.

With the proposed techniques, we are able to obtain two important benefits in production.

- Higher hit rates, which improve the average response time of the search engine.
- Reduced peak query traffic on back-end search clusters, which allows savings in hardware costs.

Outline. Section 2 discusses previous work. In Section 3, we summarize the system model we assume in this work. We present several results in Section 4, motivating the design of a cache with refreshes. Section 5 presents our strategy to select queries to refresh and to determine the rate of refreshes. Experimental results are given in Section 6. In Section 7, we provide details from our experience in production. Section 8 concludes the paper.

2. BACKGROUND

Caching has been studied extensively in the context of operating systems and memory paging [28], databases [10], Web servers and proxies [23], as well as Web search engines [19]. Herein, we do not aim to cover the literature exhaustively. Instead, we focus on results related to the scope of the paper.

Caching takes advantage of the hierarchical architecture of systems and locality of references in workloads to enable fast access to pre-computed or recently used data. A cache is characterized by its size and the eviction policy used for selecting the entries to be removed when the cache becomes full. Two common online policies are based on evicting the least recently used (LRU) [9], or the least frequently used (LFU) items from the cache. LRU is optimal when the requests are drawn from the LRU stack distance distribution. On the other hand, LFU is optimal when the requests are drawn from a Zipf distribution, which corresponds to the independent reference model (IRM) [2]. While LRU is simple to implement, LFU is more challenging because the running time for a request depends on the cache size, and the correct implementation of LFU requires a complete history of request frequencies. There have been several modifications of LFU and LRU [14, 20, 25] as well as caching policies combining aspects of the two policies [16].

Caching on Web servers and proxies has distinct requirements than memory paging [7]. First, Web pages have variable size and there is a hit only if the whole page is in the cache. Second, fetching Web pages from Web servers and proxies has a variable latency compared to memory paging. Third, the requests are generated by a large number of users rather than a small number of programs.

Aggarwal *et al.* [1] propose a generalization of LRU for handling size-wise heterogeneous objects in Web caches. The size-adjusted LRU (SLRU) evicts the items with the highest cost-to-size ratio, where the cost of an object is inversely proportional to the total number of accesses since its last access. Since SLRU is difficult to implement, the authors propose Pyramidal Selection Scheme, which stores objects in LRUs of similarly sized objects. Hence, when comparing the cost-to-size ratios of objects, it is sufficient to compare the values for the least recent items in each of the LRUs. Tatarinov [29] proposes a policy for Web caches that uses a two-dimensional array of LRU lists. The two dimensions represent the object size and frequency of access. A cached entry is placed in an LRU where all entries have similar size and access frequency. Upon an eviction, the array of LRUs is scanned along the diagonals, starting from the LRU which holds the largest objects with the lowest access frequency. Access frequencies are halved periodically by shifting LRUs in the frequency dimension to prevent cache pollution.

Caching for Web search engines also has particular requirements. The locality of user requests can be exploited by using caching in Web search engines [19, 31]. Early work on caching for information retrieval systems focused on the reduction of the server load by caching data on the client-

side [3], modifying the query evaluation process based on the cached data [13], or improving the quality of results using a set of persistent, optimal queries [24]. Web search engines can cache both query results [11, 17] as well as variable-size posting lists from the inverted index [18, 26, 30]. Baeza-Yates *et al.* [4] investigate the trade-offs between caching query results and posting lists, using static or dynamic mechanisms. Skobeltsyn *et al.* [27] study the use of pruned indexes with the user query stream filtered by a cache. Gan and Suel [12] introduce caching mechanisms that take into account estimations of query processing costs and eviction policies using query features, such as query term frequencies. Ozcan *et al.* [22] also use various query features for static caching of query results.

An important issue that has been studied in the context of caching and in particular Web proxy caching is consistency [21]. Contents of a proxy cache may not correspond to the online version of the cached Web page because the page may have been updated. One simple approach that leads to weak consistency, which does not guarantee that the cached version is up-to-date, is the use of a TTL parameter[2]. Furthermore, caches are most commonly considered passive components, *i.e.*, caches request data only in response to a user request. However, active caches may also anticipate future requests of cached entries by refreshing them without waiting for a corresponding user request. Cohen and Kaplan [8] evaluate a number of refreshing policies for Web caches and find that frequency-based policies significantly outperform recency-based policies, because the vast majority of freshness misses, which can be eliminated, occur on the more popular URLs. This is in agreement with findings of [5]. Kroeger *et al.* [15] also demonstrate using Web proxy server logs that an active cache with refreshing may decrease the latency between the cache and the server significantly.

In the case of Web search engines, the issue of consistency or freshness appears in two different forms. First, the indexed documents may correspond to an older version of the Web pages, due to the delay of the crawling and indexing processes. Second, the documents matched for a cached query may correspond to an older version of the index after an index update. In this work, we focus on the second aspect of the problem. With the exception of Alonso *et al.* [3], which mention cache consistency and the use of TTL, no work on caching for Web search engines addresses issues of index updates and freshness of results.

3. SYSTEM MODEL

Search engines execute essentially three main tasks: crawling, indexing, and query processing. The crawler of a search engine continuously updates its document collection by fetching new or modified documents from the Web and deleting documents that are no longer available. The indexer periodically processes the crawled document collection and generates a new inverted index. Alternatively, an incremental indexer can be used to continuously reflect every change in the collection to the inverted index. Finally, query processors evaluate online user queries over the inverted index.

In the design of a Web search engine, another typical component is a cache of results. A result cache contains a set of entries indexed by queries. Typically, a hash function is used to map queries to entries. Once a new query request arrives, the cache hashes the query string and determines if a corresponding entry is present or not. If a query is not cached, the engine evaluates it and caches the results, evicting another query if necessary. A result cache provides two desirable benefits: it reduces the average latency perceived by users, and it reduces the load on back-end query processors. Such a cache may run on the same machines as query processors or on separate machines. Herein, we assume that the result cache resides on separate machines and that most resources of those machines are available to the cache.

Given current storage technology and configurations of commodity servers, it is possible to have result caches that can accommodate a very large number of result entries (*e.g.*, of the order of millions), thus making them perform as an infinite cache. A consequence of this observation is that, in these caches, replacement policies become less important. With such a large number of entries, hit rates are determined by the number of singleton queries, not the eviction policies.

At the same time, current commercial search engines have an incentive to update their indexes as frequently as possible to increase the degree of freshness of served results. However, as the index evolves, there is a need to invalidate cache entries that contain stale results. One trivial way of achieving freshness is having indexers notify the cache whenever the inverted index is updated, and having the cache invalidate all entries (*i.e.*, flushing). Unfortunately, given the large size of the cache, re-warming the cache may take a long time and, during this period, hit rates remain lower compared to the steady state. Hence, flushing the cache content often severely impacts the hit rate, while not flushing for a long period impacts freshness. In the next section, we present some motivating results for deriving a simple algorithm to invalidate entries, without any feedback from the indexer and without flushing upon every index update.

4. MOTIVATING THE CACHE DESIGN

In this section, we discuss important issues considered during the design of our cache. We initially discuss the query log we used in our experiments, and progressively motivate our design. All data points in the plots are hourly snapshots.

4.1 Query log

In the experiments, we use a query log obtained from the traffic of the Yahoo! Web search engine. The log contains 130,320,176 queries (65,100,647 unique), received during nine consecutive days of operation. Queries are obtained from a subset of cache servers and represent only a portion of the entire Yahoo! query traffic. Since queries are sliced between the cache servers based on their MD5 hashes and the query log is large enough, we assume that the query distribution is similar to that of the entire query traffic.

Fig. 1 shows the query frequency distribution, which follows a power-law distribution, as also observed in some previous work [4]. In our query log, there are 49,679,763 singleton queries, and the most frequent query appears 372,447 times. Fig. 2 shows inter-arrival time of queries, and we observe that some queries repeat within short time intervals. More specifically, 32.1% of consecutive query repetitions are within a minute, and 53.2% within an hour. These two observations motivate, respectively, the use of LFU- and LRU-based heuristics in a refresh algorithm. Fig. 3 shows the hourly query traffic, where we observe periods of low ac-

[2]A cached entry is considered invalid or stale when it has been cached for longer than the time specified by the TTL.

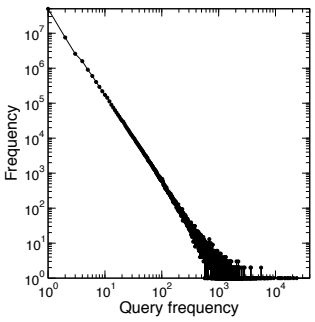

Figure 1: Frequency distribution of queries (log-log scale).

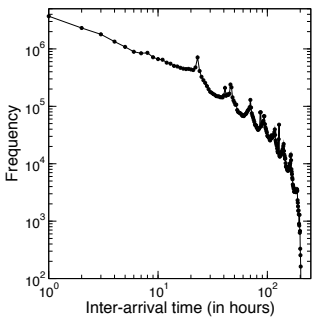

Figure 2: Frequency distribution of inter-arrival times of queries (log-log scale).

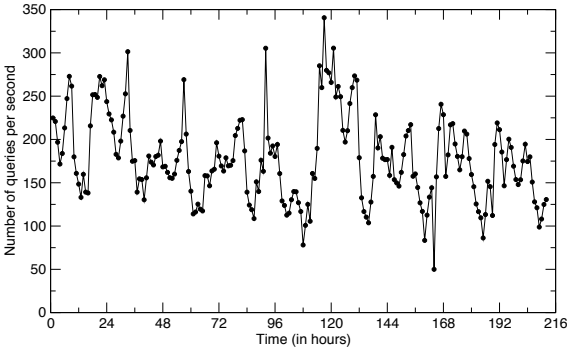

Figure 3: Query traffic.

tivity. In production, as we will discuss, these low-activity periods are used for issuing refresh queries to the back-end clusters. Throughout the motivating experiments in this section, we simulate an LRU-based caching algorithm.

4.2 Cache capacity

In general, there is a correlation between the hit rate and the capacity of a cache, *i.e.*, the possibility of a hit increases as we keep more entries in the cache. Figure 4 shows that hit rate increases with the cache capacity, given in number of entries. Depending on the cache capacity, after an initial warm-up period, hit rates tend to fluctuate within a fixed interval (*e.g.*, [0.25–0.35] for a one-million entry cache). With an infinite cache, we obtain hit rates up to 0.6, almost doubling the hit rate of the cache with one million entries.

Practical result caches in large Web search engines perform approximately as infinite caches because they store a large number of entries using RAM and disk. When equipped with a large enough cache, only singleton queries result in a cache miss (compulsory misses) and have to be evaluated. Note that processing a query may require cycles from hundreds of computers. Consequently, retrieving the results of a query from the disk rather than recomputing it over a huge Web index uses substantially fewer resources.

Having a large result cache improves hit rates. However, it also implies storing the same result for arbitrarily long periods of time. This is not a desirable behavior in a production environment as the freshness of results determines the quality perceived by users (especially for certain query classes such as news queries). In practice, freshness is an issue even for small caches: some popular cache entries are never evicted, leading to poor freshness for such entries.

To assess freshness of results, we use the *average age of a hit*: the difference between the time of the hit and the time of the last update. The rationale for this metric is that a cache entry is more likely to contain stale results if it has been in the cache for a longer time. This is a consequence of periodic index updates, which increase the likelihood over time that the cached results no longer match those computed using the current index. The age of an entry, however, is just an indication, and it does not imply that results have effectively changed. Verifying whether results have changed would require recomputing queries upon index updates, which is unrealistic both in our experimental and production settings.

To illustrate the freshness issue, Fig. 5 displays the variation of the average age of a hit over time. As expected, the freshness problem is more severe with the infinite cache, which never evicts its entries. We see that, after nine days, the average age of a hit on the infinite cache becomes about 5.6 days, which is definitely not acceptable for today's Web search standards. A linear increase is observed in the average age of a hit for fixed-capacity caches as well. Even with the relatively small, one-million-entry cache, the average hit age rises to 18.6 hours after only nine days of operation.

Having an unbounded cache implementation is not trivial in practice. One of the main issues is having a mapping (often a hash map) in memory from queries to entries, and such a mapping ideally should fit completely in RAM. However, it is practical to have a cache that is large enough so that a large fraction of the queries evicted are singletons, and consequently there are no or very little capacity misses. In our analysis, we are interested in the behavior of caches that are large enough to approximate an infinite cache, consequently the following experiments assume an infinite cache.

4.3 Flushing

A naïve solution to the freshness problem is to periodically flush the content of the cache and re-warm it from scratch with new queries. This approach guarantees that all potentially stale results are discarded, and it enforces processing of every future query over the current index at least once. As shown in Fig. 6, reasonable and consistently low average hit ages can be achieved by flushing the cache. With a flushing period of every 16 hours, the average hit age is under 8 hours. We observe that the average hit age always remains below half of the flushing period.

Although flushing improves freshness, it can lead to significant degradation of hit rates due to many compulsory misses. This problem is illustrated in Fig. 7. In general, the hit rate is seen to be somewhat recovered throughout the time interval between two consecutive flushes. However, the sharp drop of hit rate right after flushes is not affordable to a search engine serving thousands of queries per second. Such a sudden and sharp drop of hit rate may lead to a high query traffic to the back-end search clusters, perhaps even exceeding the maximum query processing throughput the clusters can sustain. Such a load spike leads to degradation when evaluating queries (thus deteriorating result quality) or queries may even be discarded without processing.

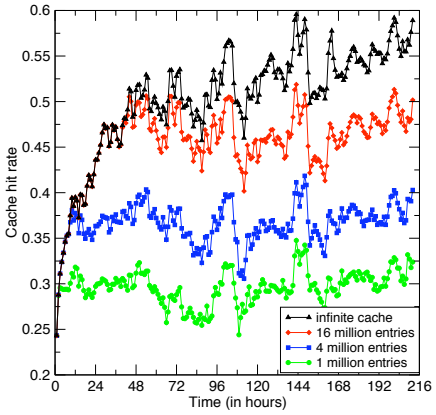

Figure 4: Hit rate with varying cache capacity.

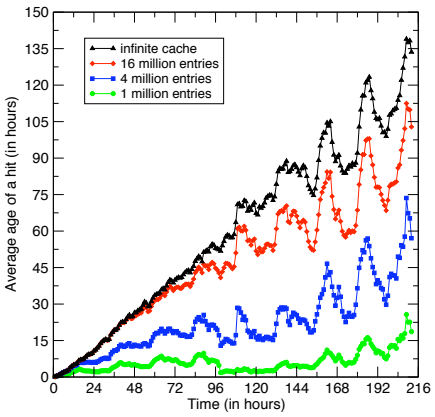

Figure 5: Average age of a hit with varying cache capacity.

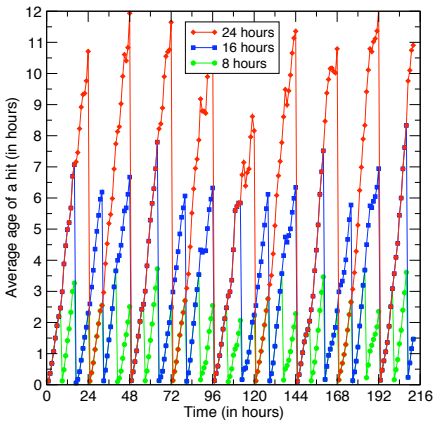

Figure 6: Average age of a hit with different periods of flushing.

4.4 Time-to-live (TTL)

A viable alternative to the freshness problem is to bound the amount of time the search engine is allowed to serve a given entry from the cache by associating a TTL value with each entry. An entry is said to be *expired* if the difference between the current time and the last time the entry is updated is larger than the TTL value. Otherwise, the entry is *fresh*. The content of an expired entry is considered stale. Hence, every hit on an expired entry is treated as a miss, and the corresponding query is forwarded to the back-end clusters. This approach sets an upper-bound on the age of a hit, but does not prevent the search engine from serving stale results, as expiration is not synchronized with updates to the index. Consequently, under this scheme, it becomes crucial to carefully select a value that provides an acceptable degree of freshness. Setting TTL to low values may not have a significant impact on freshness as changes to the Web may take days to be available on search engines.

Although TTL is not an optimal solution to the freshness problem, it is easy to implement and provides a flexible mechanism[3]. Fig. 8 shows variation of the average hit age over time for three different, typical TTL values. After the initial warm-up period, the average age values are seen to fluctuate within fixed intervals. The hit age averages after nine days are 2.1, 4.5, and 7.5 hours for TTLs of length 8, 16, and 24 hours, respectively.

A major drawback of using TTLs is the negative impact of expired entries on hit rate (remember that any request for an expired entry is considered a miss). Fig. 9 shows the variation of hit rate in time for different TTL values (the same TTL value is used for all entries). In general, the observed hit rate is higher than those when flushing and it is more stable. Hit rate still drops with respect to the no-TTL case, but lower average age of entries (topmost curves in Fig. 4 and Fig. 5, respectively) justifies the use of TTL.

4.5 Refreshing

An important enhancement to the TTL scheme is to use idle cycles of the back-end query processors to *refresh* expired cache entries by re-computing the results of cached queries. We assume that query processors are engineered to have a budget of cycles, and these cycles are not used for any other task if not used to process queries. Given the daily and weekly cycles of traffic, there are typically periods of low activity providing some spare capacity.

Adding a mechanism for cache refreshes on top of the TTL mechanism brings two major benefits to a search engine. First, we are able to increase hit rates by reducing the number of misses due to requests on expired entries. An important implication of higher hit rate is lower average latency experienced by users. Second, the number of user queries hitting the back-end search clusters drops, which reduces the amount of back-end hardware used.

A refresh mechanism requires a policy to select entries to refresh and order them. Ideally, we keep a set of entries fresh such that the hit rate is maximized and the average age is minimized, with a given constraint on the number of queries the back-end clusters are able to process per second. There are several possible criteria for selecting the entries to be refreshed, *e.g.*, frequency of the query, recency of the query, cost of processing the query at the back-end, and the probability of a change in the cached results.

In this work, we use the frequency and recency information since they are good indicators of queries that are valuable enough to be refreshed. We do not consider the cost of query processing since this is difficult to estimate in an online fashion. Similarly, we do not incorporate to our techniques the probability that an entry's content is stale, as estimating this probability accurately requires communication with the index servers. In the following section, we propose a novel caching algorithm that combines the TTL and refresh mechanisms, utilizing the recency and frequency information obtained from the past query traffic.

5. REFRESH STRATEGY

In this section, we discuss an important data structure that we use to maintain cached queries and our policies to select queries to refresh (Section 5.1). In Section 5.2, we present the algorithm we use to adjust the refresh rate.

5.1 Selecting queries to refresh

There are several possible policies we can use to decide the order of queries to assign to refresh slots. We chose to give higher priority to "hotter" and "older" queries. That is, queries that appear more frequently (hot queries) and that have been in the cache longer (older queries) have a

[3]In a Web search engine, search clusters may have different requirements and may require adjusting TTL separately.

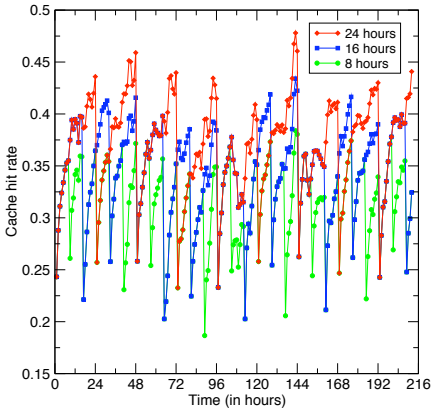

Figure 7: Hit rate with different flushing periods.

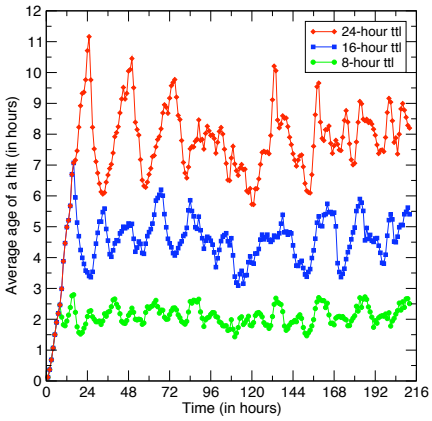

Figure 8: Average age of a hit for different TTL values.

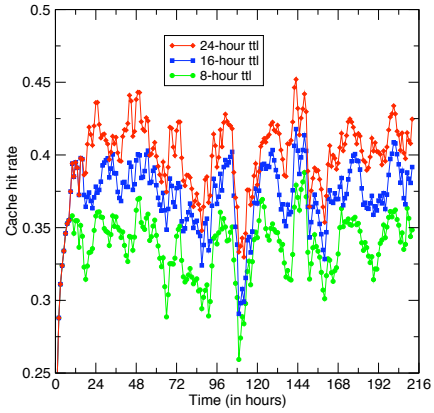

Figure 9: Hit rate for different TTL values.

higher priority for refresh. To keep track of *temperature* and *age*, we use a two-dimensional bucket array, as illustrated in Fig. 10 (each circle corresponds to a bucket containing a linked list of query entries). We use T and A as the number of temperatures and age buckets, respectively.

The hottest temperature and the freshest age are both zero. We initially add a query to bucket $(T-1, 0)$ and increase the age of cached entries by shifting the buckets along the age dimension as times elapses. We determine the interval between age shifts using two input parameters: the number of age buckets and the number of requests that we call *singleton requests*. The number of singleton requests is the number of requests that must be processed to declare a query a singleton. A singleton request is a request to process a query that only appears once in a given time frame. If we select, for example, 24 hours, then we estimate the number of requests using the arrival rate of user queries. We could have alternatively specified this parameter as an amount of time, but we have chosen not to do so due to implementation constraints (*e.g.*, how to keep track of time when the cache is off for maintenance).

We adjust the temperature according to the frequency of occurrence. To avoid unnecessary fluctuations due to variations in the frequency of queries over time, we use the historical average of temperatures as a factor to compute a new average upon a hit. Instead of scanning the bucket array periodically, we update the temperature of cache entries lazily, and we recompute the temperature of a query upon either a hit or a refresh attempt. This is a desirable feature because scanning the bucket array structure periodically and recomputing temperature values can be costly due to the large number of entries. Also, updating the temperature of queries lazily affects neither the hit rate nor the order of refreshes (for the same age, we always update higher temperatures first).

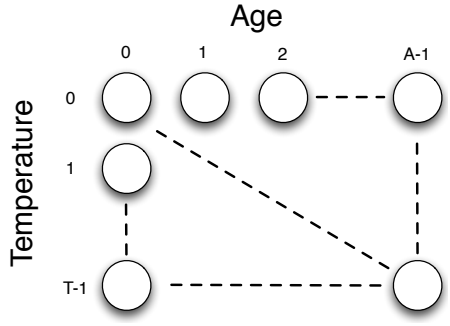

Figure 10: Bucket array.

To select queries to refresh, we use a policy that selects hotter and older queries first. For every temperature τ and age α, we compute the value of $s = (T - \tau) * \alpha$, and order buckets according to decreasing order of the value of s. The policy we use currently in production prioritizes expired queries: it refreshes all eligible expired queries before selecting fresh queries. An important decision in this procedure is how many queries to refresh at a time. We determine the rate of refreshes dynamically, and we discuss it next.

5.2 Refresh-rate adjustment

Based on the analysis of key performance metrics, including latency, CPU, and memory usage on back-end nodes, we have chosen latency as the main guidance in cache refresh rate adjustment. In the production system, latencies of all queries are monitored, and an average latency (*tick latency*) is generated after every *tick* (a time unit that is usually set to 1 or 2 seconds). Depending on the query traffic volume and normal workload, for every cache node, we have an expected latency range, $[L..H]$, where L and H are lower and upper latency bounds, respectively. The system is said to be running under healthy conditions during a time period if all tick latencies are within the expected range.

The basic idea of refresh rate adjustment is simple: after every tick period, we try to reset the target number of queries, $\mathcal{T}$, which includes both user queries and refresh queries, for the next tick period based on tick latency *lat* (Algorithm 1). If *lat* is below L and the total number of queries processed in the previous tick, $\mathcal{Q}$, is greater than or equal to $\mathcal{T}$, we increase $\mathcal{T}$ by δ_1. If the tick latency is higher than H and $\mathcal{Q}$ is less than or equal to $\mathcal{T}$, we decrease $\mathcal{T}$ by δ_2 (both δ_1 and δ_2 are configurable parameters). Otherwise, $\mathcal{T}$ remains unchanged. Usually, an upper limit $\mathcal{M}$ is enforced as a safeguard for $\mathcal{T}$, and δ_2 is set to a number much larger than δ_1 so that refresh rate can drop quickly in response to latency spikes due to load spikes.

Because tick latencies may fluctuate significantly in production, $\mathcal{T}$ might be adjusted too frequently and never reach the expected value. To address this problem, we keep an

Algorithm 1 Adjust target number of queries ($\mathcal{T}$)

```
lat ← getTickLatency()
Q ← getNumQueriesProcessed()
if lat < L and Q ≥ T then
  T ← min(T + δ1, M)
else if lat > H and Q ≤ T then
  T ← max(T − δ2, 0)
else
  T ← T
end if
```

array of tick latencies for the past n ticks, where n is configurable parameter. Instead of using only the last tick latency lat, we use the running average of last n tick latencies in Algorithm 1. A larger value of n leads to a more stable and consistent view of overall latency. Similarly, we keep track of the number of latest consecutive tick latencies higher than H. Once this number reaches a configurable value of m, we decrease $\mathcal{T}$ by δ_2. To be conservative, m can still be set to 1.

6. CACHE EVALUATION

6.1 Simulation parameters

We evaluate the performance of the proposed algorithm using a simulator, actively used in production for tuning purposes. In all experiments, we assume that we have an infinite cache, even though we have verified that caches of 15 million entries (with a TTL of 16 hours) already saturate hit rates (*i.e.*, hit rates do not increase with a larger cache).

We use 8, 16, and 24 hours for the TTL values (the `TTL` parameter). To determine the number of refreshes that can be realized at a given time, we adopt the following strategy. At every second, the simulator measures the difference between a given threshold for the peak throughput sustainable by the back-end and the incoming query rate. For example, if the back-end is able to process at most 100 queries per second and the number of queries received in the last second is 75, then 100−75=25 cache entries are refreshed, *i.e.*, 25 refresh queries are issued to the back-end. If the difference is not positive, then the back-end is assumed to have no idle cycles and no refreshes are performed within that second. In our simulations, we try 100, 150, and 200 query/s as the threshold values for the peak sustainable throughput (the `PST` parameter). Finally, we try different age values for which we would be willing to perform a refresh. For example, if this value is set to 8 hours, only the entries that are older than 8 hours are considered candidates for a refresh. In our experiments, the minimum refresh age (the `MRA` parameter) depends on the `TTL` parameter and equals to `TTL`/2, `TTL`/4, or `TTL`/8.

6.2 Baseline algorithms

As the baseline for a performance comparison, we initially tried a simple refresh algorithm, which seeks in the LRU queue the entries stored longer than `MRA`. Each time there is a possibility (idle cycles) for refreshes, this algorithm initiates a new scan starting from the head of the LRU queue, traversing towards the tail of the queue until enough entries are found for refreshing. We observed, however, three issues with this approach. First, as the LRU queue grows (remember that we have an infinite cache), some entries are never refreshed because each time the algorithm starts from the head and never reaches the entries near the tail, *i.e.*, there is starvation. Second, we observed that traversing the entire LRU queue is too costly in practice (in our simulations, in the worst case, about 65 million entries had to be traversed under a second). Third, most entries right after the head are unnecessarily scanned many times since they are either fresh or have been just scanned.

To prevent the above-mentioned problems, we modified the algorithm slightly. In the new algorithm, herein referred to as *cyclic refresh*, a scan continues from where the previous scan terminated. If the tail of the queue is reached, the scan is restarted from the head of the queue. As the baseline, we use this cyclic refresh algorithm and a TTL-based algorithm with no refreshes (referred to as *no refresh* in the plots). It should be noted that there are other simple heuristics that may be used as a baseline (*e.g.*, refreshing the most stale entry first). However, most of them are not practical enough to be implemented in a real-life search engine with an infinite cache. Therefore, we do not consider them here.

6.3 Comparison

Hit rates for different policies are shown in Fig. 11 (`TTL=16`, `PST=150`, `MRA=TTL/4`). In general, the proposed algorithm, referred to as *age-temperature*, achieves higher hit rates, and the no-refresh policy performs the worst, as expected. The average hit rates computed over nine days are 0.403, 0.392, and 0.372, for the age-temperature, cyclic refresh, and no-refresh policies, respectively. Table 1 displays the hit rates achieved by different policies and parameters.

Fig. 12 shows the variation of the average hit age in time under different policies (`TTL=16`, `PST=150`, `MRA=TTL/4`). An interesting observation is that the cyclic refresh algorithm occasionally results in higher hit ages than the no-refresh policy. This is a bit counter-intuitive as refreshing is expected to decrease the hit age. The reason is that the cyclic refresh algorithm reduces the possibility of hits on expired entries, but it is not fast enough to refresh frequent queries more often. It spends most of its limited refresh budget for refreshing entries that are not frequent (especially, the singleton queries). The no-refresh policy, however, by letting the entries expire achieves a reduction in the number of hits on entries with large age values. The mean average hit age, computed over nine days, is very close for no-refresh and cyclic refresh policies (around 4.5 hours). The age-temperature algorithm performs much better due to better selection of entries to refresh. The mean average age is around 2.5 hours. Table 2 displays a comparison of average hit age for various parameters. In remaining experiments, we use the age-temperature algorithm.

6.4 Impact of peak sustainable throughput

The peak throughput of the back-end is an important parameter that affects the impact of refreshing. As the value of `PST` increases, we have more opportunity to perform refreshes. Note that the number of refresh queries also depends on the current hit rate of the cache. If the current cache hit rate is high, we have more free slots to use for submitting refresh queries to the back-end.

Fig. 13 shows the increase in hit rate as we increase the `PST` value (`TTL=16`, `MRA=TTL/4`). Every increase of 50 query/s in `PST` brings about 1% increase in hit rate, on average. Greater

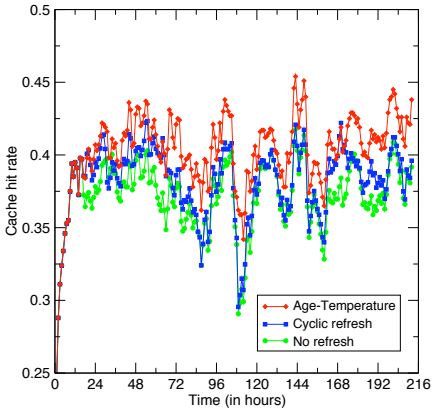

Figure 11: Hit rate with different policies.

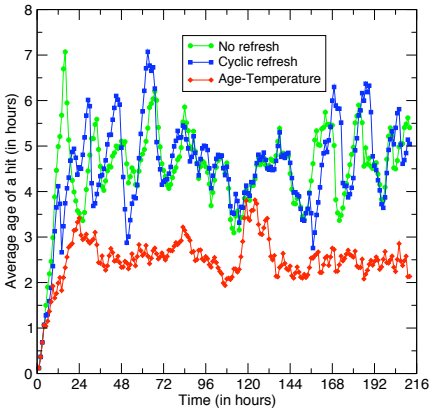

Figure 12: Hit age with different policies.

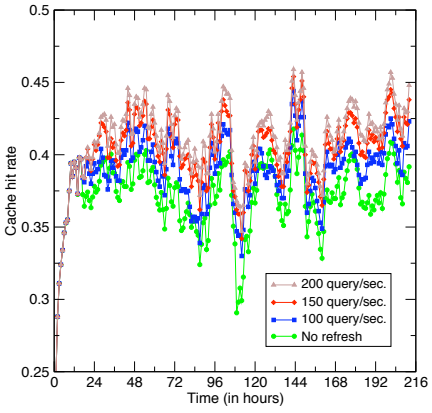

Figure 13: Hit rate with varying peak sustainable throughput.

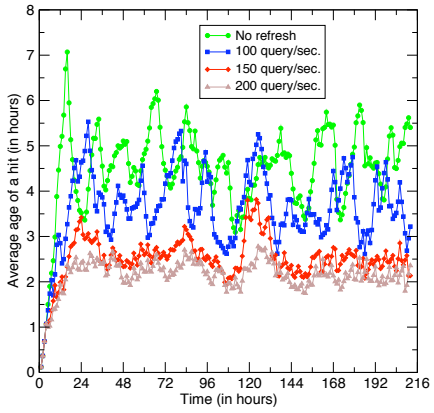

Figure 14: Hit age with varying peak sustainable throughput.

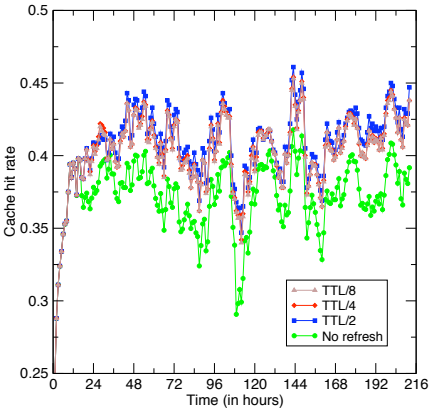

Figure 15: Hit rate as minimum refresh age varies.

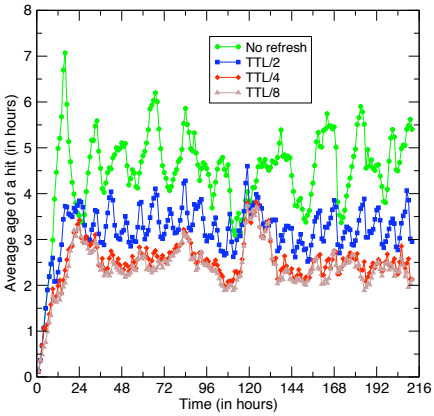

Figure 16: Hit age as minimum refresh age varies.

improvements are achieved in average hit age with increasing PST. The mean average age for no refresh is around 4.5 hours while the age-temperature algorithm achieves 3.6, 2.5, and 2.1 hours for a PST of 100, 150, and 200, respectively. In general, the mean average hit age can be halved with respect to the no-refresh policy (see Table 2).

6.5 Impact of minimum refresh age

Another important parameter is the minimum refresh age, which sets a bound on the freshness of entries that we will be willing to refresh. Setting this to a lower value has the advantage that more entries will be refreshed, which may be feasible if we have enough idle cycles in the back-end. If there are not enough idle cycles, however, then it might not be possible to refresh older entries before they expire, which induces more misses. Validating this claim, Fig. 15 shows that hit rates are similar with varying MRA values (TTL=16, PST=150) and average hit rates slightly decrease as MRA increases (see Table 1). Savings in the average hit age saturate (Fig. 16) as MRA changes from TTL/4 to TTL/8.

7. PRODUCTION EXPERIENCE

The cache elements we discuss in Section 5 have been implemented and deployed, and this implementation is currently in production use in the Yahoo! search engine. We discuss below the performance of our implementation in production, thus providing some evidence that our design is practical, and discuss briefly the issue of *degradation*.

7.1 Performance in production

Figs. 17 and 18 show hit rate and hit age for production nodes, using an implementation of the age-temperature algorithm. The plots have been extracted from our monitoring system. To generate these plots, we have turned off refreshes in n nodes ($n > 1$) and compare against n others that receive equivalent traffic (they are part of the same cluster). Each figure shows the corresponding metric over a period of 3 days for these cache nodes. We can observe in these graphs the exact time when we turn refreshes off. For hit rate (Fig. 17), top curves correspond to nodes with the refresh on, and the absolute difference is higher than 10% at several points. The average hit rate with and without refreshes is 49.2% and 41.0%, respectively, and the absolute difference is 8.1%.

For hit age (Fig. 18), the bottom curves correspond to cache nodes with refreshes off. The TTL we use for these nodes is 18 hours and the average hit age with and without refreshes is 411.8 and 362.3 minutes, respectively. The difference is 49.5 minutes, which is less than 5% of the TTL. The average hit age is higher with refreshes due to the extra hits (hits correspond to queries that have been in the cache for some time).

Table 1: Hit rates averaged over the entire query log

				Cyclic Refresh			Age-Temperature		
TTL	MRA	No Refresh	Flushing	PST=100	PST=150	PST=200	PST=100	PST=150	PST=200
8	TTL/2	0.337	0.311	0.338	0.343	0.35	0.352	0.373	0.388
	TTL/4	0.337	0.311	0.338	0.343	0.349	0.352	0.372	0.381
	TTL/8	0.337	0.311	0.338	0.343	0.348	0.352	0.372	0.381
16	TTL/2	0.372	0.345	0.374	0.382	0.395	0.389	0.407	0.41
	TTL/4	0.372	0.345	0.374	0.382	0.392	0.389	0.403	0.41
	TTL/8	0.372	0.345	0.374	0.381	0.39	0.389	0.402	0.407
24	TTL/2	0.398	0.369	0.401	0.409	0.424	0.417	0.426	0.427
	TTL/4	0.398	0.369	0.401	0.409	0.42	0.416	0.426	0.428
	TTL/8	0.398	0.369	0.401	0.41	0.422	0.416	0.424	0.427

Table 2: Hit ages averaged over the entire query log

				Cyclic Refresh			Age-Temperature		
TTL	MRA	No Refresh	Flushing	PST=100	PST=150	PST=200	PST=100	PST=150	PST=200
8	TTL/2	2.079	1.36	2.086	2.11	2.141	1.954	1.595	1.484
	TTL/4	2.079	1.36	2.077	2.097	2.13	1.953	1.448	1.122
	TTL/8	2.079	1.36	2.079	2.112	2.129	1.953	1.447	1.089
16	TTL/2	4.488	3.121	4.513	4.592	4.686	3.773	3.209	3.11
	TTL/4	4.488	3.121	4.504	4.511	4.611	3.632	2.508	2.147
	TTL/8	4.488	3.121	4.484	4.492	4.593	3.627	2.389	1.915
24	TTL/2	7.51	5.332	7.557	7.5	7.225	5.791	5.092	5.047
	TTL/4	7.51	5.332	7.556	7.415	7.286	5.329	3.826	3.524
	TTL/8	7.51	5.332	7.517	7.454	7.317	5.322	3.525	3.009

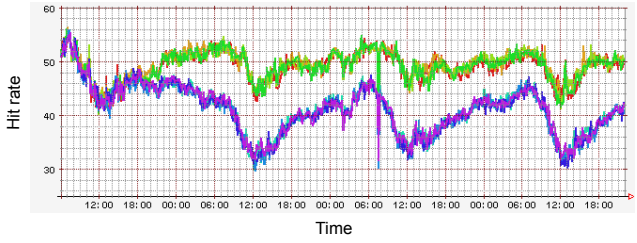

Figure 17: Hit rate in production (%).

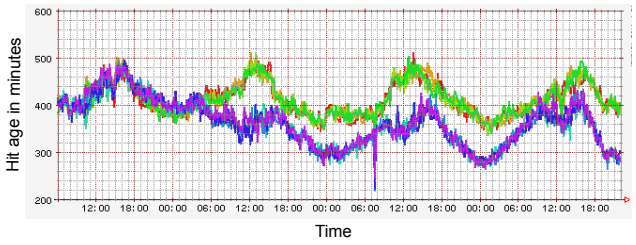

Figure 18: Hit age in production.

Compared to our simulation results, we observe the following. First, the hit rate difference between refreshing and not refreshing is between 7% and 10% in production and at most 5% in our simulations. In production, we have been able to fine-tune our caches to obtain higher hit rates. Second, hit ages are higher when refreshing in production due to more conservative refreshing. We have not felt so far a need to boost it to improve our performance. However, as our simulation results show, it is possible to reduce the average hit age by refreshing more aggressively. It is also important to observe that the refresh rate is adjusted according to performance and the plots reflect this adjustment (Section 5.2).

7.2 Degradation and TTL

In a production environment, back-end nodes may experience workload spikes due to various unpredictable reasons. When this happens, a degradation logic kicks in on back-end nodes, which return degraded (or partially degraded) search results to cache nodes. Whether degraded search results should be cached and how long they should be cached then becomes an issue. If the percentage of degraded results is high, not caching degraded results means that the hit rate will be very low and workload sent to the back-end cannot be reduced quickly, which, in turn, results in more degraded results. On the other hand, caching degraded results means that we are going to serve users with lower-quality results as long as the cache entries are not expired.

With the help of cache refresh, it is easier to find a good solution to this problem. Based on the degree of degradation, we can find suitable TTLs for cache entries such that results with a lower degree of degradation receive a longer TTL, whereas results with a higher degree of degradation receive a much shorter TTL. This way results with a high degree of degradation are given a higher refresh priority. In general, we can cache and serve results with acceptable quality for a longer time while reducing the pressure on back-end nodes to help them recover.

8. CONCLUSIONS

We treat the result caching problem for Web search engines in a novel way. We reformulate the setting in which we

examine result caching, and argue that eviction policies are less important in our setting as we can obtain in production large caches by using disk space to store query results. The problem we have investigated instead is keeping cached results consistent with the search engine's index while sustaining a high hit rate. We show that flushing the cache is not efficient and propose a TTL-based strategy to expire cache entries. Using a real workload, we show that result caching with a TTL parameter improves the average hit age with a loss in hit rate, and also, that actively refreshing cached entries is able to improve both hit rate and age. To improve hit rate and freshness, we introduce a refresh mechanism. The refresh mechanism uses two dimensions corresponding to access frequency and age of cached entries to efficiently select the entries for refreshing. Our simulation results show that it is able to achieve higher hit rate and average hit age than an LRU cache with refreshes. Results from our production setting also confirm our claimed improvement to hit rate as we are able to obtain 7% to 10% more hits compared to not refreshing.

9. REFERENCES

[1] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the World Wide Web. *IEEE Trans. Knowledge and Data Eng.*, 11:94–107, 1999.

[2] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *J. ACM*, 18(1):80–93, 1971.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Syst.*, 15(3):359–384, 1990.

[4] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *30th SIGIR Conference*, pages 183–190, 2007.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *INFOCOM '99*, pages 126–134, 1999.

[6] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[7] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *1st USITS*, 1997.

[8] E. Cohen and H. Kaplan. Refreshment policies for web content caches. In *INFOCOM '01*, pages 1398–1406, 2001.

[9] P. J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970.

[10] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[11] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, 2006.

[12] Q. Gan and T. Suel. Improved techniques for result caching in Web search engines. In *18th WWW Conference*, pages 431–440, 2009.

[13] B. T. Jónsson, M. J. Franklin, and D. Srivastava. Interaction of query evaluation and buffer management for information retrieval. *SIGMOD Rec.*, 27(2):118–129, 1998.

[14] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[15] T. M. Kroeger, D. D. E. Long, and J. C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *1st USITS*, 1997.

[16] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comp.*, 50(12):1352–1361, 2001.

[17] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *12th WWW Conference*, pages 19–28, 2003.

[18] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *14th WWW Conference*, pages 257–266, 2005.

[19] E. P. Markatos. On caching search engine query results. *Comp. Commun.*, 24(2):137–143, 2001.

[20] N. Megiddo and D. S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *Computer*, 37(4):58–65, 2004.

[21] S. V. Nagaraj. *Web Caching And Its Applications (Kluwer International Series in Engineering and Computer Science)*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[22] R. Ozcan, I. S. Altingovde, and O. Ulusoy. Static query result caching revisited. In *17th WWW Conference*, pages 1169–1170, 2008.

[23] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.

[24] V. V. Raghavan and H. Sever. On the reuse of past optimal queries. In *18th SIGIR Conference*, pages 344–350, 1995.

[25] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. *SIGMETRICS Perform. Eval. Rev.*, 18(1):134–142, 1990.

[26] P. C. Saraiva, E. S. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto. Rank-preserving two-level caching for scalable search engines. In *24th SIGIR Conference*, pages 51–58, 2001.

[27] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates. ResIn: A combination of results caching and index pruning for high-performance Web search engines. In *31st SIGIR Conference*, pages 131–138, 2008.

[28] A. J. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[29] I. Tatarinov. An Efficient LFU-Like Policy for Web Caches. Technical Report NDSU-CSORTR-98-01, Computer Science Department, North Dakota State University, Wahpeton, ND, 1998.

[30] A. Tomasic and H. Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. *SIGMOD Rec.*, 22(2):129–138, 1993.

[31] Y. Xie and D. R. O'Hallaron. Locality in search engine queries and its implications for caching. In *INFOCOM*, pages 1238–1247, 2002.